Teoría y Práctica con EJB 2.1

ÍNDICE DE CONTENIDOS

1. Introducción a las aplicaciones distribuidas 8
1.1. Evolución de los sistemas distribuidos 9
1.1.1. Sistemas Cliente/Servidor 12
1.1.2. Sistemas Distribuidos 14
1.2. Tecnologías actuales para aplicaciones distribuidas 15
1.3. La plataforma J2EE 17
2. Tecnologías de objetos distribuidos en Java 20
2.1. Objetos distribuidos: la base de los EJB 20
2.1.1. Java RMI 21
2.1.1.1. Arquitectura RMI 21
2.1.1.2. Ventajas e inconvenientes 24
2.1.2. RMI-IIOP 27
2.1.2.1. Arquitectura RMI-IIOP 27
2.1.2.2. Ventajas e Inconvenientes 29
2.1.3. Java Naming and Directory Interface (JNDI) 30
2.1.3.1. Sistemas de nombrado y directorios 30
2.1.3.1.1. Conceptos de servicio de nombrado 30
2.1.3.1.2. Conceptos de servicio de directorio 33
2.1.3.2. Arquitectura JNDI 35
2.1.3.2.1. Paquete Naming 37
2.1.3.2.2. Paquete Directory 41
3. Tecnología de componentes en Java 44
3.1. Paradigma de componentes 44
3.1.1. Programación basada en componentes 44
3.1.2. Componentes software 47
3.1.3. JavaBeans 49
3.1.4. JavaBeans vs Enterprise JavaBeans 51
3.2. Arquitectura de componentes distribuidos Java: EJB 52
3.2.1. Aplicaciones con EJB 53
3.2.2. Roles en la tecnología EJB 55
3.2.2.1. Proveedor de beans 56
3.2.2.2. Ensamblador de aplicaciones 57
3.2.2.3. Responsable del despliegue de EJB 57
3.2.2.4. Administrador del sistema 58
3.2.2.5. Proveedor del contenedor y servidor de aplicaciones 59
3.2.2.6. Proveedor de herramientas 59
3.2.3. Contenedor de EJB 59
3.2.3.1. El contenedor EJB como middleware implícito 61
3.2.4. J2EE y JNDI: descubriendo componentes en su entorno 63
3.2.5. Fases de desarrollo de un EJB 65
3.2.6. Estructura de un EJB: la parte servidora 66
3.2.6.1. Clase del bean 68
3.2.6.2. EL objeto EJB 70
3.2.6.2.1. Interfaz Remoto 71
3.2.6.3. El objeto Home 74
3.2.6.3.1. Interfaz Home 75
3.2.6.3.2. Interfaz Local 77

3.2.6.4. Descriptores de despliegue 79
3.2.6.4.1. Fichero ejb-jar-xml 81
3.2.6.4.2. Fichero web.xml 82
3.2.6.5. Fichero específicos del proveedor 84
3.2.6.6. Fichero de empaquetamiento Ejb-jar 84
3.2.6.7. Ficheros de empaquetamiento del cliente 85
3.2.6.8. Ficheros de empaquetamiento de la aplicación 86
3.2.7. Llamando a los EJB´s: la parte cliente 88
3.2.7.1. Buscando una referencia 89
3.2.7.1.1. Creando el contexto inicial 90
3.2.7.1.2. Buscando el objeto Home 92
4. JBoss aplication server: ejecutando la aplicación 95
4.1. Ejemplo Hola Mundo!! 96
4.1.1. Introducción 96
4.1.2. Que se hace en el ejemplo, que no se hace y ¿por qué? 97
4.1.3. Requisitos para ejecutar el ejemplo 99
4.1.4. Preparando el entorno de ejecución 100
4.1.4.1. Arrancando el JBoss 100
4.1.4.2. Preparando la herramienta Ant 102
4.1.5. Preparando los ficheros 107
4.1.5.1. Interfaz remoto 107
4.1.5.2. Interfaz home 108
4.1.5.3. La clase del bean 109
4.1.5.4. Descriptores de despliegue 112
4.1.5.5. Cliente stand-alone 117
4.1.5.6. Cliente web 121
4.1.6. Compilando los fuentes 126
4.1.7. Ficheros específicos del servidor JBoss 127
4.1.8. Empaquetando el EJB 131
4.1.9. Empaquetando el cliente 132
4.1.10. Empaquetando la aplicación WEB 133
4.1.11. Ensamblando el EAR 135
4.1.12. Desplegando la aplicación 137
4.1.13. Ejecutando Hola Mundo!! 140
4.1.13.1. Ejecutando el cliente stand-alone 141
4.1.13.2. Ejecutando el cliente web 142
5. Profundizando en la arquitectura EJB 144
5.1. Tipos de componentes EJB 144
5.2. Servicios de los beans gestionados por el contenedor 145
5.2.1. Gestión de recursos 146
5.2.2. Persistencia 147
5.2.3. Transacciones 148
5.2.4. Concurrencia 151
5.2.5. Seguridad 152
5.3. Servicios del contenedor en JBoss 155
6. Beans de sesión 157
6.1. Bean de sesión con o sin estado? 158
6.2. Beans de sesión sin estado 161
6.2.1. Fundamentos de los beans de sesión sin estado 161
6.2.2. Ejemplos de beans sin estado 163

6.3. Beans de sesión con estado 163
6.3.1. Fundamentos de los beans de sesión con estado 163
6.3.2. Pooling y pasivación en beans con estado 165
6.3.3. Pasivación y serialización en beans con estado 166
6.3.4. Ciclo de vida de una bean de sesion con estado 168
6.3.5. Ejemplo de bean de sesión con estado: una BD literaria 170
6.3.5.1. Introducción 170
6.3.5.2. Requisitos para ejecutar el ejemplo 173
6.3.5.3. Configurando el servidor JBoss 174
6.3.5.4. Preparando el fichero jboss-build.xml de Ant 178
6.3.5.5. Preparando los ficheros 180
6.3.5.5.1. Interfaz remoto 180
6.3.5.5.2. Interfaz home 182
6.3.5.5.3. La clase del bean 183
6.3.5.5.4. Descriptores de despliegue estándar 186
6.3.5.5.5. Ficheros específicos del servidor JBoss 188
6.3.5.5.6. Cliente stand-alone 189
6.3.5.5.7. Fichero application.xml 192
6.3.5.6. Aprovechando al máximo Ant 193
6.3.5.7. Ejecutando Base de Datos Literaria. 194
7. Beans de entidad 197
7.1. Persistencia de objetos 199
7.2. Mapeo Objeto-Relacional 201
7.3. Arquitectura de los beans de entidad 203
7.3.1. Profundizando en el ciclo de vida de los beans de entidad 203
7.3.2. Interacción con la base de datos 207
7.3.3. Llamadas entre beans 209
7.3.4. Otras características de los beans de entidad 212
7.3.4.1. Finders en beans de entidad 212
7.3.4.2. Transacciones en beans de entidad 214
7.3.4.3. Pooling en beans de entidad 215
7.3.5. El objeto EJB Context 216
7.4. Beans BMP (Bean Managed Persistence) 221
7.4.1. Arquitectura BMP 222
7.4.2. Ejemplo BMP: gestión de cuentas corrientes en un banco 223
7.4.2.1. Introducción 223
7.4.2.2. Requisitos para ejecutar el ejemplo: la base de datos 226
7.4.2.3. Preparando los scripts de base de datos 231
7.4.2.4. Preparando el fichero jboss-build.xml de Ant 232
7.4.2.5. Preparando los ficheros 235
7.4.2.5.1. Interfaces Remoto y Local 236
7.4.2.5.2. Interfaces Home y Local Home 238
7.4.2.5.3. La clase primary key 241
7.4.2.5.4. Las clases de los beans de sesión y entidad 243
7.4.2.5.5. Descriptores de despliegue estándar 261
7.4.2.5.6. Ficheros específicos del servidor JBoss 266
7.4.2.5.7. Cliente stand-alone 268
7.4.2.5.8. Fichero application.xml 270
7.4.2.6. Utilizando Ant 271
7.4.2.7. Poblando la base de datos 271

7.4.2.8. Ejecutando el Gestor de Cuentas 273
7.5. Beans CMP (Container Managed Persistence) 276
7.5.1. Arquitectura CMP 277
7.5.2. Descriptor de despliegue ejb-jar.xml con CMP 280
7.5.3. El motor CMP2 en JBoss 283
7.5.3.1. Descriptor de despliegue JBoss para CMP 283
7.5.3.2. Configuración del mapping en JBoss 285
7.5.3.3. Profundizando en la configuración de CMP 289
7.5.4. EJB-QL 291
7.5.5. El método ejbSelect( ) 293
7.5.6. Ejemplo CMP: gestión de cuentas con CMP 294
7.5.6.1. Introducción 294
7.5.6.2. Requisitos para ejecutar el ejemplo 295
7.5.6.3. Preparando el fichero jboss-build.xml de Ant 296
7.5.6.4. Preparando los ficheros 298
7.5.6.4.1. Interfaces Remoto y Local 299
7.5.6.4.2. Interfaces Home y Local Home 301
7.5.6.4.3. La clase primary key 303
7.5.6.4.4. Las clases de los beans de sesión y entidad 304
7.5.6.4.5. Descriptores de despliegue estándar 314
7.5.6.4.6. Ficheros específicos del servidor JBoss 319
7.5.6.4.7. Cliente stand-alone 323
7.5.6.4.8. Fichero application.xml 324
7.5.6.5. Utilizando Ant 325
7.5.6.6. Poblando la base de datos 325
7.5.6.7. Ejecutando el Gestor de Cuentas 326
7.6. BMP o CMP? 329
7.7. Heurística y consejos prácticos en EJB 331
7.7.1. Beans de sesión con estado o sin estado. 331
7.7.2. Recubrir los beans de entidad con un bean de sesion 332
7.7.3. Interfaces locales o remotos 333
7.7.4. Gestión eficaz de transacciones 334
7.7.5. Afinando el rendimiento en CMP 336

ÍNDICE DE FIGURAS

Figura 1: Entorno J2EE......19
Figura 2: Roles implicados en la especificación EJB......56
Figura 3: Composición del componente EJB......67
Figura 4: Estructura del fichero descriptor de despliegue ejb-jar.xml......82
Figura 5: Estructura del fichero descriptor de despliegue web.xml......83
Figura 6: Estructura del fichero EAR......87
Figura 7: Diagrama de secuencia para JNDI......93
Figura 8: Estructura de directorios de trabajo......103
Figura 9: Namespaces para JNDI......140
Figura 10: Pasivación de un bean de sesión con estado......168
Figura 11: Ciclo de vida de un bean de sesión con estado......169
Figura 12: Pasos en la creación de un bean de sesión......204
Figura 13: Pasos en la creación de un bean de entidad......206
Figura 14: Colaboración entre beans de entidad y sesión......212
Figura 15: Diagrama de objetos para BMP......222
Figura 16: Diagrama de objetos para CMP......279
Figura 17: Estructura del fichero descriptor de despliegue jbosscmp-jdbc.xml......284

Motivaciones

Con el presente libro se pretende adentrar al lector en el mundo de las aplicaciones distribuidas y mas concretamente en la arquitectura de componentes distribuidos EJB (Enterprise JavaBeans). El lector aprenderá a desarrollar y utilizar componentes de negocio de tipo sesión con y sin estado (Session Beans) y a gestionar la persistencia mediante componentes de entidad (Entity Beans). Se abordarán las dos perspectivas existentes para gestionar dicha persistencia mediante estos componentes: persistencia gestionada por el componente (BMP) y persistencia gestionada por el contenedor (CMP).

Por otro lado, se enseñarán tanto las mejores estrategias de desarrollo de este tipo de componentes software como el uso de las herramientas necesarias para darle soporte. El lector entrará en el mundo de los servidores de aplicaciones de la mano del servidor JBoss™ Application Server 4.0, aprenderá como se gestiona el empaquetado y el despliegue de una aplicación empresarial basada en los estándares J2EE (Java 2 Enterprise Edition), como se importan/exportan archivos EJB JAR, WAR o EAR, como se gestiona el mapping para la persistencia de los datos en base de datos o la utilización de herramientas de desarrollo como Ant 1.6.2 de la Apache Software Foundation.

En el transcurso de este libro se enseñará como programar Enterprise JavaBeans y cómo instalarlos en un contenedor de EJB. Por lo tanto, uno de los objetivos de este libro es demostrar lo fácil que resulta escribir componentes Java del lado el servidor usando el modelo de componentes Enterprise JavaBeans. Para ello se ha estructurado el libro a modo de tutorial, de forma que cada concepto introducido será acompañado de un ejemplo práctico que lo ilustre para facilitar su comprensión.

Durante este capítulo se verá cuál es la situación tecnológica que va a permitir abordar el desarrollo de aplicaciones distribuidas. En primer lugar se verá una introducción a este tipo de sistemas, para posteriormente estudiar su evolución a largo de los últimos años. Finalmente, se verá un estudio pormenorizado de todas las tecnologías actuales que aporta la plataforma de desarrollo Java 2 Enterprise Edition para el desarrollo de aplicaciones distribuidas, centrándose en la especificación 2.1 de Enterprise JavaBeans.

1. Introducción a las aplicaciones distribuidas

El comienzo del milenio, la era de la conectividad, ha traído consigo un cambio en la forma en la que se desarrollan las aplicaciones informáticas. Las empresas encuentran más apropiado el uso de aplicaciones distribuidas en lugar de las aplicaciones basadas en grandes *mainframes* o las aplicaciones *standalone* que hasta el momento venían utilizando. Y esto es así, entre otras cosas, porque las aplicaciones distribuidas permiten que el sistema de información se adapte completamente a la estructura de la empresa. Si existen varios nodos en donde se distribuye el trabajo de la empresa, ¿por qué no hacer que el sistema de información de la empresa se distribuya de igual manera?

Esto permite:

- Que cada localidad (o elemento estructural de la empresa, por ejemplo, una sucursal de un banco) tenga asignado su parte de funcionalidad del Sistema de Información que, de una forma cooperativa, interactuará con las demás partes para conseguir la funcionalidad general.
- Una mejora en la eficiencia del sistema, ya que cada localidad posee los datos que utiliza más a menudo.
- Que el resto del sistema siga funcionando aun cuando una localidad deje de funcionar o falle.

Pero no todo son ventajas. Las aplicaciones distribuidas son más difíciles de comprender, implementar, depurar y mantener. Por ello se necesitan herramientas potentes que ayuden a los programadores y diseñadores a obtener aplicaciones distribuidas que exploten todo lo posible las ventajas y oculten la dificultad intrínseca de este tipo de aplicaciones.

A la hora de desarrollar e implantar cualquier sistema de información (y, por ende, cualquier aplicación distribuida) se necesitan una serie de herramientas y soportes, tanto desde el punto de vista físico como el lógico:

1. Hardware. Soporte físico, como ordenadores, líneas de conexión y protocolos de bajo nivel: físico, enlace, etcétera.
2. Sistemas Operativos de Red. Proveen de un entorno de operación que abstrae a los programas de los detalles de la máquina específica, o de otros programas que se estén ejecutando en la máquina. Ofrecen, además servicios de conectividad que permiten a los programas, entre otras cosas, acceder a recursos remotos (nodos de procesamiento, impresoras, etcétera) de la misma forma que si fueran locales. En definitiva, hacen que los programas se centren única y exclusivamente de la tarea para la que fueron pensados, sin necesidad de tener en cuenta la naturaleza distribuida del sistema.
3. Protocolos estándar de alto nivel. En entornos donde se requiere que dos o más ordenadores puedan comunicarse para llevar a cabo la tarea, se deben definir una serie de protocolos que estandaricen el modo en el que las aplicaciones existentes en los diferentes ordenadores se comuniquen, independientemente de la aplicación específica.
4. Modelos conceptuales. Permiten establecer un marco lógico que sirve de guía para el desarrollo de las aplicaciones. Por ejemplo, el modelo Cliente/Servidor permite distribuir la funcionalidad de una aplicación en términos de proveedores y demandantes de servicios.
5. Tecnologías. Suponen una materialización de un modelo conceptual. Distintas tecnologías pueden aplicar de manera diferente los conceptos de un mismo modelo conceptual.
6. Lenguajes, entornos de desarrollo y de ejecución. Con los distintos lenguajes y entornos de desarrollo y ejecución se construyen y se implantan definitivamente las distintas aplicaciones que forman el sistema. Éstos darán soporte a distintas tecnologías.

1.1. Evolución de los sistemas distribuidos

El desarrollo y abaratamiento del hardware, la ampliación de las empresas y la demanda de aplicaciones cada vez más complejas y adaptadas a la estructura de la empresa llevó a que se tuvieran que idear nuevas formas de organizar los sistemas de

información de las empresas. A medida que el hardware se fue desarrollando, la demanda de aplicaciones de gestión automatizada de información fue creciendo.

Cuando se necesitaron sistemas de información que se fueran ajustando a las necesidades de las empresas, la única solución que podían aportar los primeros sistemas, debido en gran parte al elevado coste del hardware, era una configuración en la que un único equipo o *mainframe*, relativamente grande y adaptado a las necesidades de la empresa, gestionaba todo el sistema de información. Los distintos puntos en los que se requería el acceso a esa información eran conectados al gran *mainframe* a través de líneas de comunicación utilizando terminales, también llamadas *terminales tontas* o *green screens*, cuyo única funcionalidad era la de mostrar caracteres en la pantalla y enviar la información del usuario en forma de pulsaciones del teclado.

Estos terminales eran mucho más baratos que el gran *mainframe*, y por tanto, una empresa podía distribuir un número relativamente grande de éstos en distintos puntos estratégicos a un coste no muy elevado. Dos causas llevaron a que esta organización fuera evolucionando: primero, el hardware se hizo cada vez más barato, por lo que en los puntos de acceso de información se podían colocar, a un menor coste, ordenadores con una cada vez más grande capacidad de procesamiento, que quedaba ampliamente desaprovechada al utilizarlos éstos como terminales; sin embargo, la causa más importante era la falta de flexibilidad y escalabilidad de la solución basada en *mainframe*.

En primer lugar, todo el código del sistema junto con sus datos residía en el ordenador principal de la empresa. Esto hacía que, para empresas medianamente grandes, el sistema de información fuera un largo y, a la vez, en muchos casos, incomprensible programa al que los programadores se tenían que enfrentar a la hora de realizar alguna modificación o actualización.

En segundo lugar, el sistema quedaba muy poco escalable, y esto era por varias razones: todos los terminales se conectaban a un mismo ordenador, quedando éste limitado incluso por el número de interfaces físicos para conexión de terminales de que dispusiera; todos los terminales requerían del servicio del *mainframe* de forma

más o menos simultánea, lo que hacía que, al ir añadiendo más terminales, el servicio de las peticiones de cada una de ellas se hacía más lento.

Ampliaciones tan básicas como la inclusión de un segundo *mainframe* por cuestiones de ampliación de la capacidad del sistema o por la apertura de una segunda sucursal, llevaron a que surgieran cuestiones bastante importantes y que no estaban resueltas hasta el momento, como por ejemplo ¿Cómo reorganizar la aplicación monolítica existente si en vez de un *mainframe* se tienen dos?, ¿Cómo se redistribuyen los datos de estas aplicaciones?, ¿Cómo se consigue que ambos ordenadores trabajen juntos y se distribuyan el trabajo para aprovechar realmente ambas capacidades de procesamiento?

De forma paralela, en la parte de los usuarios, equipos cada vez más potentes podían ser instalados, lo que llevaba a la pregunta razonable: ¿Cómo se puede conseguir que la capacidad de procesamiento de los ordenadores de los usuarios sea aportada a la del sistema global?

En definitiva, ¿Cómo distribuir la funcionalidad, los datos y los recursos del sistema de información de la empresa de una forma que sea escalable y flexible?. Esta pregunta se ha ido refinado en los últimos años donde la capacidad de procesamiento de los ordenadores de sobremesa dotados de conexiones a redes y de Sistemas Operativos de Red ha sobrepasado la de aquellos *mainframes*, y con el paso del tiempo se ha convertido en otras preguntas como:

- ¿Cómo conseguir sistemas abiertos (en el sentido de que permitan la inclusión de herramientas y subsistemas de terceros de forma sencilla) que permitan integrar los sistemas antiguos?
- ¿Cómo aprovechar los recursos proporcionados por los ordenadores y los Sistemas Operativos de Red, incluso en sistemas heterogéneos con distintas plataformas hardware y diferentes Sistemas Operativos?
- ¿Cómo conseguir que la aplicación distribuya la funcionalidad entre todos los equipos de manera que maximice la productividad, capacidad de procesamiento y utilización de recursos y que a la vez permita un diseño

modular, reutilizable, basado en objetos, portable e independiente de la plataforma?

- ¿Cómo integrar los sistemas de información de la empresa con Internet?

La respuesta a estas preguntas vino a través de la inclusión de un modelo conceptual que separaba los roles de *cliente* y *servidor* para la consecución de una funcionalidad total.

Este es el modelo Cliente/Servidor, que se explicará a continuación y en el que se verá cómo la tendencia actual adopta a Internet y a las Intranets como soporte para el desarrollo de aplicaciones distribuidas.

1.1.1. Sistemas Cliente/Servidor

El modelo Cliente/Servidor divide la funcionalidad de una aplicación en torno a dos roles muy bien definidos: el cliente y el servidor.

De modo abstracto, el servidor ofrece una serie de servicios que pueden ser utilizados por los clientes para completar la funcionalidad de la aplicación. Una interacción básica Cliente/Servidor implica a un cliente que inicia una petición de algún servicio a un servidor, posiblemente incluyendo algunos parámetros que modifiquen el comportamiento del servidor. El servidor entonces realiza la función especificada por el cliente, devolviendo los posibles resultados que el servicio genera.

Esta abstracción permite desarrollar la aplicación en torno a las abstracciones de descomposición modular que proporcionan los servicios. En la práctica, los clientes y servidores se implementan como procesos que se están ejecutando en máquinas conectadas a una red. La infraestructura que se encarga de conectar de alguna manera los procesos cliente y servidor es llamada *middleware* (se podría decir que es la parte "/" de Cliente/Servidor).

El término *middleware* engloba a todos los elementos que hacen posible esta comunicación, desde las líneas físicas de comunicación hasta los protocolos de alto nivel. De hecho se considera que el soporte estándar de desarrollo de aplicaciones

distribuidas es Internet (e Intranet a nivel de empresas), por lo que la infraestructura *middleware* está cubierta hasta el nivel de transporte de la arquitectura OSI, e incluso en algunos casos hasta el de aplicación.

Por otro lado, típicamente cualquier aplicación software puede dividirse en distintas capas de la siguiente manera:

- Una capa de presentación, ya sea en base a un GUI (interfaz gráfico de usuario) o mediante una terminal de caracteres, que implementa la interacción con el usuario y realiza ciertas comprobaciones sencillas como podía ser la validez de fechas.
- Una capa de lógica de negocio o funcionalidad, que implementa la funcionalidad de la aplicación. Esta lógica de negocio se encarga de realizar todos los procesos que son requeridos por el sistema de información.
- Una capa de lógica de datos, que establece cómo los datos de la aplicación son estructurados, ya sea a través de bases de datos SQL con tablas, o a través de ficheros de disco.

Cualquiera de estas capas de la aplicación puede ser distribuida. Por ejemplo, en un sistema con presentación distribuida, cada cliente dispone de su lógica de presentación, y en el servidor se guarda tanto la lógica del negocio como la lógica de datos y los datos mismos. Esta configuración es conocida como una configuración con *clientes livianos* y *servidores gordos* (*thin clients* y *fat servers*). El interfaz entre la lógica de presentación y la de negocio asegura que aunque esta última cambie, los clientes podrán seguir accediendo sin cambio alguno.

Otra opción de distribución podría ser si se elige la distribución de los datos, los clientes poseen ahora, además de la lógica de presentación, la del negocio y la de datos. Los datos son distribuidos sobre servidores de manera que se aprovechen al máximo los recursos y características de cada uno. Los clientes se convierten entonces en *fat clients*, que acceden a servidores que se convierten sólo en repositorios de datos. Esta distribución es algo menos flexible, ya que un cambio en la aplicación implica un cambio en los clientes.

Una última opción mucho más flexible sería la de distribuir la lógica de negocio de la aplicación. La distribución de la funcionalidad entre los distintos clientes permite que se aproveche la capacidad de procesamiento de los clientes para participar de forma cooperativa en la consecución de la funcionalidad total de la aplicación. Cada parte de la funcionalidad se podría distribuir en un procesador específico acompañado de la lógica de datos y los datos que necesite para llevarla a cabo.

Por lo tanto, una configuración arquitectónica basada en capas puede producir una serie de ventajas a la hora de desarrollar aplicaciones distribuidas. Los clientes se pueden construir en base a unos servicios encapsulados en los procesos que implementan la lógica de la aplicación. De esta forma, los clientes se independizan y son más inmunes a cambios tanto en la lógica como en los datos. La funcionalidad que los clientes implementan resulta ser sencilla y en consecuencia los posibles cambios serán muy superficiales. Además, varios clientes pueden reutilizar servicios estándar definidos en niveles intermedios y la aplicación, al estar dividida en partes más pequeñas, hace que el proceso de distribución de funcionalidad en los procesadores más adecuados sea muy flexible.

1.1.2. Sistemas Distribuidos

Los sistemas distribuidos son el último paso en la computación Cliente/Servidor. En vez de diferenciar entre las distintas partes de la aplicación, los sistemas distribuidos ofrecen toda la funcionalidad en forma de *objetos*, con un significado muy en la línea del término *Objeto* de la programación Orientada a Objetos.

No existen los roles explícitos de cliente y servidor, sino que toda la funcionalidad está ahí para ser utilizada. Los procesos que componen la aplicación y que se están ejecutando en las distintas máquinas de la red son clientes y servidores y cooperan para conseguir la funcionalidad total de la aplicación.

Esto da la máxima flexibilidad. El mundo de los sistemas distribuidos es un mundo de *entidades pares* (peer-to-peer), esto es, elementos de procesamiento o *nodos* con distintas disponibilidades de recursos, distinta capacidad de almacenamiento, distintos

requerimientos, que cooperan ofreciendo servicios en forma de objetos y requiriendo otros servicios de otros objetos implementados en otros nodos de la red.

En general, un sistema distribuido es un sistema Cliente/Servidor multinivel con un número potencialmente grande de entidades que pueden desempeñar roles de clientes y servidores según sus necesidades. El hecho de ofrecer una serie de servicios en forma de objetos hace que el diseño y desarrollo se haga en base a interfaces bien definidos que facilitan y apoyan la modularidad y reutilización, a la vez que permiten un diseño mucho más flexible.

Los sistemas distribuidos ofrecen, por lo general, un conjunto de servicios añadidos, como el servicio de directorio, que permite localizar servicios por nombre, gestión de transacciones, etcétera.

1.2. Tecnologías actuales para aplicaciones distribuidas

A continuación se analizarán las distintas tecnologías de desarrollo de aplicaciones distribuidas cliente/servidor que actualmente se utilizan. Sin embargo, antes de hacer ese estudio, se enumerarán las características que se consideran deseables a una tecnología de desarrollo de aplicaciones distribuidas:

- Independencia del hardware y Sistema Operativo subyacente específico sobre el que las tareas asignadas a un nodo se ejecutan, es decir, que permita la actualización de un nodo a un mejor hardware o a distinto Sistema Operativo sin que la aplicación deba ser rediseñada ni recodificada, ni, en el mejor de los casos, recompilada.
- Independencia de localización de la funcionalidad. Se busca que una determinada parte de la funcionalidad de la aplicación pueda ser trasladada hacia otro nodo de sin que la aplicación tenga que ser modificada (salvo, si acaso, el uso de una herramienta sencilla de reconfiguración automática que indique la nueva localización del recurso). Este punto y el anterior aseguran que la aplicación diseñada resistirá a una reestructuración total del sistema informático.

- Independencia del lenguaje en el que estén especificadas las distintas partes de funcionalidad que componen la aplicación. Esto permitiría poder utilizar los lenguajes que más se adecuen a cada parte de la funcionalidad: C/C++ para las aplicaciones críticas en tiempo o Java/C# para la interfaz con el usuario, etcétera.
- Integración sencilla con el sistema de información anteriormente existente, así como de sistemas de terceros, por ejemplo, bases de datos comerciales (Oracle, Sybase, Informix, etcétera.). Construir lo que se conoce con el nombre de Sistemas Abiertos.
- Facilitar la instalación de actualizaciones, mejoras, nuevas prestaciones o nuevas versiones de la aplicación. Esto es lo que se conoce como Gestión de la Configuración.
- Escalabilidad, o lo que es lo mismo, conseguir que el rendimiento de la aplicación guarde una relación directa con la cantidad de recursos que se le asignen.
- Que permita modularidad, reutilización, extensibilidad, programación en base a interfaces, etcétera. En definitiva, características presentes en aplicaciones más sencillas que no implican distribución a través de los lenguajes Orientados a Objetos tradicionales.

Sin embargo, todas estas consideraciones hacen que la utilización de ciertas tecnologías (como EJB´s, RMI, RMI-IIOP o JSP´s) quede restringida a un determinado lenguaje de programación. En concreto, dicho lenguaje es Java. Hay que destacar por tanto, que será la distribución del lenguaje Java denominado Plataforma Java 2 Enterprise Edition (J2EE) la que mediante estas tecnologías brinde todas las facilidades que se utilizan en la mayor parte de las aplicaciones distribuidas desarrolladas actualmente.

Otras alternativas que permiten el desarrollo de aplicaciones de objetos distribuidos fuera del ámbito de lenguaje Java, podrían ser la arquitectura CORBA de OMG, que aporta un alto grado de interoperabilidad entre distintas tecnologías o COM+/Remoting/.NET de Microsoft.

1.3. La plataforma J2EE

Hasta hace poco tiempo las empresas que deseaban desarrollar verdaderos sistemas empresariales tenían que construir sus propios entornos de trabajo para mantener complejas cuestiones como la persistencia (a menudo incluyendo bastante código SQL), transacciones, seguridad y asegurándose así que sus sistemas eran escalables.

Este trabajo era bastante costoso en tiempo y esfuerzo, por lo que resulta una evolución natural el desarrollo de servidores que mantengan todo este entramado de cuestiones para las aplicaciones, de forma que se acelera su desarrollo y se asegura la escalabilidad y el máximo rendimiento. Estos servidores son los denominados servidores de aplicaciones.

El interés no solo se centra en asegurar una gestión eficaz de los procesos de negocio mediante la implantación de sistemas de información, se busca cada vez más la integración y escalabilidad de estos sistemas, de forma que sea factible la integración de datos procedentes de fuentes heterogéneas en información útil para las necesidades estratégicas de la empresa.

Por lo tanto, la plataforma Java 2 Enterprise Edition (J2EE) surge como una especificación que pretende proporcionar un estándar para los servidores de aplicaciones. La meta de J2EE es definir un estándar de funcionalidad que ayude a la nueva generación de sistemas de información, y por tanto a incrementar la competitividad de las empresas que la adopten.

El entorno de trabajo oficial Java para el desarrollo de aplicaciones a nivel empresarial es la Java 2 Enterprise Edition. Esta permite desarrollar aplicaciones distribuidas basadas en un gran espectro de nuevas tecnologías en continua evolución, a la vez que simplifica su proceso al ofrecer un modelo de aplicación basado en componentes software. Obviamente existen otras soluciones pero J2EE tiene el respaldo de empresas como Sun, IBM, Oracle, BEA y una multitud de otros vendedores de servidores de aplicaciones.

La plataforma J2EE ha sido diseñada para proporcionar soporte, tanto desde el lado cliente como desde el lado servidor, para el desarrollo de aplicaciones distribuidas multi-capa. Por lo tanto la J2EE proporciona una plataforma para el desarrollo de aplicaciones a nivel empresarial, con un soporte completo de API´s (interfaces de programación de aplicaciones) para implementar código y con total garantía de portabilidad entre distintos servidores de aplicaciones.

La J2EE proporciona por tanto: conectividad con bases de datos y otros sistemas de empresa, componentes de negocio seguros, transaccionales y (posiblemente) persistentes, infraestructura orientada a mensajería, componentes y Servicios Web, protocolos de comunicaciones e interoperabilidad

Por otra parte, determina una clara división del código dividiéndolo entre capas de presentación, lógica de negocio y datos. Este es el modelo conocido como Model-View-Controler o MVC. Una arquitectura distribuida en capas o multi-capa, como es la de MVC, es aquella en la que la parte cliente se encarga de mantener el interfaz gráfico de usuario, mientras que existen una serie de componentes intermedios en el sistema que se encargan de implementar la lógica de la aplicación. Por último, hay un último nivel que se encarga del la lógica de datos, típicamente un Sistema Gestor de Base de Datos. En el momento en el que los componentes de este último nivel se conviertan en clientes de otros componentes, se convierte en una estructura multi-capa.

Esta configuración permite que los clientes se construyan en base a unos servicios encapsulados en los procesos que implementan la lógica de la aplicación, y por lo tanto son más inmunes a cambios tanto en la lógica como en los datos. Aún así, la funcionalidad que los clientes implementan es tan sencilla que los cambios son muy superficiales. Varios clientes pueden reutilizar servicios estándar definidos en el nivel intermedio. La aplicación, al estar dividida en partes más pequeñas, hace que el proceso de distribución de funcionalidad en los procesadores más adecuados sea muy flexible.

Además, la J2EE proporciona una serie de estándares, en forma de servicios e interfaces de programación ubicados en las capas intermedias de la arquitectura, que

proporcionan los servicios al cliente. Estos estándares permiten la integración de los distintos sistemas de información de la empresa con el resto de la aplicación. Entre ellos destacan: JavaServer Pages, Servlets, Enterprise JavaBeans (EJB´s), JDBC o RMI-IIOP. La siguiente figura muestra el entorno tecnológico que ofrece la plataforma J2EE.

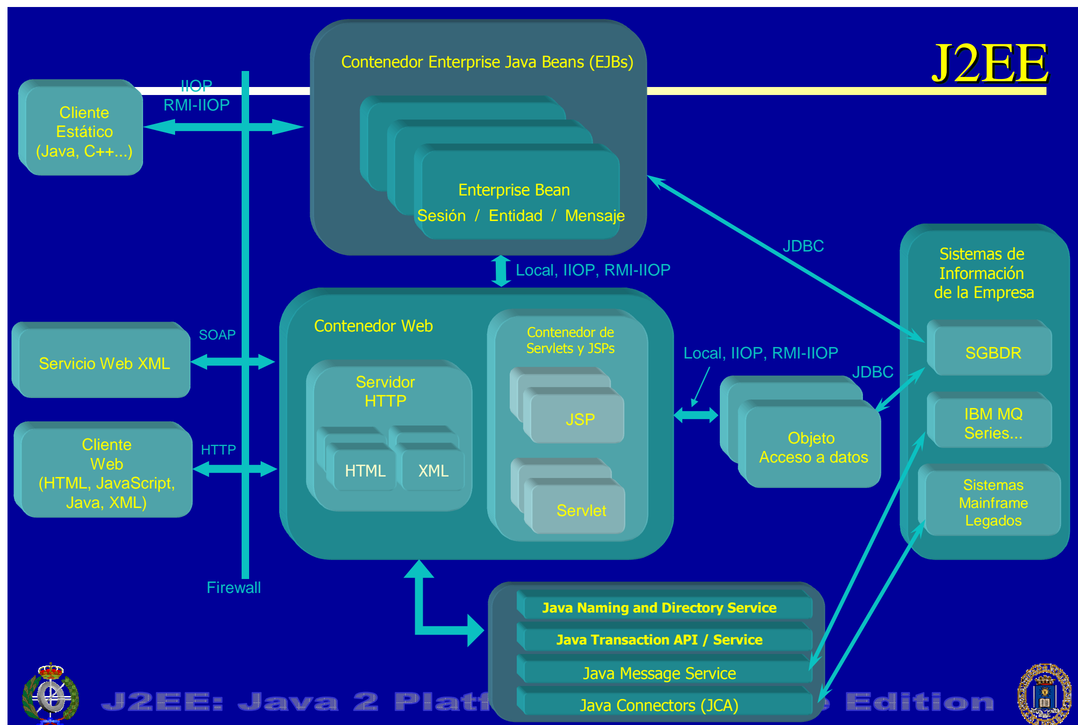


Figura 1: Entorno J2EE

2. Tecnologías de objetos distribuidos en Java

El lenguaje de programación Java, como se ha comentado anteriormente, ofrece una serie de alternativas para la programación de aplicaciones distribuidas utilizando objetos distribuidos como pueden ser RMI o RMI-IIOP. El objetivo que se persigue es que el desarrollo de aplicaciones distribuidas se facilite en extremo, gracias a las abstracciones que estas tecnologías ofrecen al desarrollador.

2.1. Objetos distribuidos: la base de los EJB

Con los objetos distribuidos se llega al máximo nivel de abstracción. Los objetos distribuidos son importantes porque permiten dividir una aplicación a través de una red. Las ventajas de hacer aplicaciones distribuidas han sido ya comentadas, sin embargo, esta división de las aplicaciones trae consigo ciertas complicaciones a los desarrolladores que deben ser tenidas en cuenta.

Por lo tanto, el objetivo es ofrecer al programador un entorno en el que las llamadas a métodos de objetos distribuidos o remotos (residentes en otros elementos de procesamiento de la red o en otros procesos dentro del mismo ordenador) se realicen de la misma manera que las llamadas a métodos de objetos locales (residentes en el mismo espacio de direcciones que el llamante), todo ello, como su propio nombre indica, proporcionando un entorno orientado a objetos con polimorfismo, enlace dinámico, desarrollo basado en interfaces, meta-información, independencia del lenguaje de desarrollo, de la plataforma hardware/Sistema Operativo y de la localización física de los objetos.

A continuación se detallarán en profundidad las tecnologías que aporta la J2EE para el desarrollo de objetos distribuidos. Sin embargo, antes de continuar cabe destacar que la tecnología de componentes distribuidos Enterprise JavaBeans (EJB´s), que es el objetivo de estudio fundamental del presente libro, se asienta en las tecnologías de objetos distribuidos que se presentan en este capítulo. Es por esta razón por lo que

resulta imprescindible un conocimiento detallado de cómo funcionan los objetos distribuidos antes de entrar en el mundo de los componentes Java.

2.1.1. Java RMI

RMI o Remote Method Invocation permite a objetos Java llamar a métodos de otros objetos que están ejecutándose en otras máquinas como si fueran llamadas a objetos definidos localmente por la aplicación. Esta funcionalidad se aporta desde JDK 1.1 (Java Development Kit, version 1.1) a través de las clases java.rmi.* y java.rmi.server.*.

2.1.1.1. Arquitectura RMI

En toda la literatura sobre objetos distribuidos, el proceso por el que un objeto puede invocar métodos en un objeto remoto se divide en dos fases: la obtención de una referencia al objeto remoto y la invocación del método propiamente dicha. A continuación se describen más detalladamente cada una de estas fases.

En primer lugar, hay que decir que se pueden obtener referencias de objetos a los que previamente se les haya declarado como *remotos*, esto es, que ofrecen la posibilidad de ser llamados remotamente. La definición de este tipo de objetos se apoya en los *interfaces* de Java para definir el conjunto de métodos a los que un objeto remoto puede responder.

Este interfaz debe cumplir una serie de requisitos: debe extender el interface *java.rmi.Remote* (que no contiene ningún método adicional) y todos sus métodos deben ser declarados como posibles lanzadores de la excepción.

Cuando se desarrolle un objeto que implemente este interfaz (servidor), si se quiere que los distintos clientes lo localicen, hay que darle un nombre y anotarlo en el registro. El registro RMI (*rmiregistry*) es una asociación entre nombres de objetos servidores de interfaces y referencias a esos objetos.

Cualquier cliente se puede conectar a un registro RMI (que no es más que un servicio basado en *sockets* TCP y un puerto bien conocido, implementado como una aplicación que puede estar ejecutándose en distintas máquinas, teniendo por tanto un registro distribuido) y obtener las referencias a los objetos que desee, dado su nombre. *java.rmi* ofrece un conjunto de clases que permiten conectar y obtener referencias de un registro RMI . Estas clases se podrían denominar como el *servicio de nombres de RMI*, y son *Registry*, *LocateRegistry* y *Naming*:

- El interface *Registry* permite modificar las entradas en un registro, listar su contenido. El método *bind()* o *rebind()* se utiliza para crear una asociación nombre-objeto remoto; el método *lookup()* devuelve la referencia del objeto asociado a un nombre, etcétera.
- La clase *LocateRegistry* ayuda a localizar y crear registros RMI . El método estático *getRegistry()* ayuda a obtener un registro de un *host* específico. El método estático *createRegistry()* crea un nuevo registro.
- La clase *Naming* permite obtener referencias a objetos remotos y actualizar el registro (igual que el interfaz *Registry*), pero esta vez utilizando sintaxis URL: *rmi://host:puerto/nombre*, donde *host* es la dirección del equipo a donde se pretende conectar, y *nombre* el nombre del objeto remoto al que se está referenciando.

Como requisito adicional, cualquier clase que implemente un interfaz remoto, debe heredar (directa o indirectamente) de la clase *java.rmi.server.UnicastRemoteObject*.

Pero aún falta un elemento muy importante. Antes se dijo que, una vez obtenida una referencia, el cliente podía invocar métodos en el objeto como si fuera local. Sin embargo éste no lo es. Entonces, ¿Quién se encarga de traducir las llamadas locales a llamadas en el objeto remoto, posiblemente a través de una red? La respuesta está en el par de clases *Stub* (o Proxy) y *Skeleton*.

Para cada clase remota, se deben crear otras dos que están íntimamente relacionadas con ella: en terminología RMI , la clase *Stub* y la clase *Skeleton*. Estas clases son generadas automáticamente por la herramienta *rmic*, disponible en las distribuciones del JDK (Java Development Kit).

La primera de ellas contiene la funcionalidad encargada de transformar las llamadas locales en llamadas al objeto remoto. Para ello, dispone de métodos para empaquetar los argumentos del método (*argument marshaling*), conectarse con el objeto remoto y enviarle la llamada con los argumentos empaquetados. Al otro lado, es decir, en la parte del objeto remoto, debe existir otra clase (*Skeleton*) que haga justo lo contrario, esto es, que decodifique las llamadas y los argumentos (*unmarshaling*) y las traduzca a llamadas al objeto remoto real, recoja sus argumentos y devuelva el resultado al invocador. De hecho, las referencias que retorna el registro RMI son referencias a objetos *Skeleton* capaces de decodificar las llamadas.

Los objetos que son parámetros de las llamadas remotas se envían por valor. Se necesita por lo tanto un interfaz que permita convertir cualquier objeto en una secuencia de bytes que pueda ser transmitida por la red y que, al otro lado, pueda ser convertida de nuevo en objeto. Para esto se ha definido el interfaz denominado *Object Serialization*.

De esta forma cualquier objeto que vaya a ser serializado, debe declararse como implementador del interfaz *java.io.Serializable*, el cual no contiene ningún método.
Pero, ¿Dónde están las instancias de clases *Stub* y *Skeleton*? La instancia de la última está en el mismo espacio de direcciones que el objeto servidor remoto. La primera debe ejecutarse en el espacio del que efectúa la invocación. Lo cual lleva a otra pregunta: ¿Cómo llega la clase *Stub* al espacio de direcciones del invocador? Nótese que el invocador efectúa su petición de objetos que implementan cierto interfaz, luego no es consciente de la clase de los objetos que de hecho implementa ese interfaz. Esto requiere que las clases *Stub* se descarguen de forma dinámica o *bajo demanda*. De ahí que haya que definir, un *security manager* que monitoriza la carga de clases y *Stubs*.

De igual manera, existe una clase *RMIClassLoader* que es la que en última instancia carga las clases y los *Stubs*. Esta clase se comporta de forma distinta dependiendo de si el cliente es un *applet* o una aplicación normal:

- Si el cliente es un *applet*, las peticiones de carga de clases pasan por el *browser* que lo contiene y se traducen a peticiones HTTP al servidor Web convencional, que debe tener disponibles las clases que el cliente pedirá.
- Si el cliente es una aplicación normal Java, las peticiones se realizan a través de *sockets* conectando con el registro RMI que sirvió las referencias.

De una manera resumida, los pasos que se llevan a cabo para que un objeto pueda invocar métodos en un objeto remoto son los siguientes:

1. El cliente, a través de un registro RMI , obtiene una referencia a un objeto de la clase *Skeleton* que implementa el interfaz remoto requerido.
2. Automáticamente se inicia la descarga de la clase *Stub* correspondiente a la clase del objeto del que se recibe la referencia.
3. Se crea un objeto de la clase *Stub* descargada que actúa como objeto local (proxy) correspondiente al objeto remoto.
4. Las llamadas locales se realizan sobre el objeto creado de la clase *Stub*. En cada invocación, se empaquetan los argumentos y se realiza la invocación sobre el objeto *Skeleton* remoto. Esta llamada es decodificada y se traduce en invocaciones de los métodos del objeto que realmente implementa el interfaz remoto. Los resultados son enviados en sentido contrario.

2.1.1.2. Ventajas e inconvenientes

Como ya se comentó en la introducción, con RMI se ingresa en el mundo de los Objetos Distribuidos. Por lo tanto, resulta claro que la programación de la aplicación distribuida se simplifica en gran medida al poder invocar de forma transparente métodos en objetos remotos como si fueran locales, con una mínima configuración, descargando, opcionalmente, tanto los *applets* como los *Stubs* de forma dinámica (o en demanda).
Sin embargo, existen otras alternativas a este nivel conceptual que ofrecen, en ciertos aspectos, unas características más completas o una perspectiva diferente. Un ejemplo claro es CORBA o RMI-IIOP.

No obstante, RMI ofrece las siguientes ventajas:

- El desarrollo de aplicaciones pequeñas es rápido y sencillo, y no supone ningún *shock* para los programadores Java. De hecho, esta tecnología es muy válida para aplicaciones pequeñas de Intranets corporativas que utilizan un ambiente *sólo Java*.
- Evitan la utilización de IDL.
- La exposición de objetos remotos se hace a través de interfaces, lo que facilita la modularidad, extensibilidad, reutilización, etcétera, además de utilizar las características de la programación Orientada a Objetos como el polimorfismo y el enlace dinámico de una manera natural.
- La invocación de métodos en objetos remotos es transparente al programador e independiente de la localización de esos objetos. Lo único que se necesita es conocer el nombre de los objetos que forman parte del sistema y dónde están localizados (en qué registro RMI).
- Los objetos pueden ser identificados por nombres y se puede utilizar un sistema de nombrado basado en URL´s, mucho más sencillo y cómodo.
- El registro RMI es fácilmente configurable y permite que varios de ellos actúen en colaboración. Además, se pueden registrar objetos remotos u objetos locales en el mismo registro, permitiendo muchas configuraciones distintas, que permiten aprovechar las características de cada equipo conectado a la red.
- Se soporta la descarga automática de clases y de *stubs* que hacen las veces de *actuadores a distancia*.
- La reconfiguración de los clientes es relativamente sencilla y resistente al cambio de localización de servicios y servidores. Esto se pueden conseguir de una forma muy sencilla enviando en las páginas HTML como parámetros a los *applets* que contienen dónde tienen que buscar un registro en concreto, o una lista de registros, que puede ser modificada cada vez que el cliente pide el *applet* a través de su *browser*.
- Favorece los clientes pequeños ó livianos (*thin clients*) y la estandarización de interfaces de usuario a través de *applets* y el *browser* universal (lo que se ha venido en llamar Network Computer).

En cuanto a los inconvenientes, si se compara esta tecnología con las características expuestas al comienzo del capítulo como óptimas o definitivamente deseables para

facilitar el desarrollo de aplicaciones distribuidas, se presentan los siguientes inconvenientes:

- No se provee un mecanismo de meta-información y auto-descripción. Los clientes no pueden *despertar* en un ambiente que se describa a sí mismo. Por ejemplo, que indiquen qué registros hay disponibles y cuáles son sus direcciones, que permita obtener los interfaces existentes en cada registro (no sólo los nombres de los interfaces, sino, además, qué métodos soportan, qué argumentos, etcétera). Por ejemplo, CORBA, si provee esta funcionalidad a través del ORB (Object Request Broker).
- Sólo se soporta un lenguaje, Java, lo que hace difícil la integración de software ya construído (*legacy systems*). El sistema queda ligado a su implementación utilizando Java y ligado, como RMI, a las interioridades de este lenguaje (sistema de tipos, serialización, etcétera). Por ello, no posee un interfaz estándar de comunicación a través del cable, y no es compatible con el estándar CORBA: IIOP (Internet Inter-ORB Protocol). Sin embargo, la aparición de RMI-IIOP se soluciona ésta ultima cuestión, como se verá más adelante.
- Los objetos son pasados por valor, lo cual hace que el sistema no sea escalable a medida que el tamaño de los objetos aumenta (el sistema se hace más ineficiente en tiempo). En el momento en el que la invocación pase como parámetro una colección, todos los objetos que contiene tienen que ser serializados y enviados por el cable. Contra esto se debe argumentar que un buen diseño tendrá en cuenta estas restricciones para alcanzar la mayor eficiencia posible.
- No existe protocolo de paso por *firewall*. En la mayoría de las organizaciones, un *firewall* corta las comunicaciones que no pasan por el puerto 80 (HTTP), 21 (FTP) y 25 (Mail) (estos puertos por regla general). No hay un estándar que permita que las peticiones RMI pasen a través de estos puertos, y que, además, permitan a las aplicaciones que tradicionalmente están en estas direcciones (servidores Web, *browsers*, etcétera) funcionar sin dificultad.
- No favorece la programación totalmente distribuida, sino que los roles Cliente y Servidor deben ser bien establecidos al principio. El hecho de que los objetos se pasen por valor hace que los subsiguientes métodos invocados en

esos objetos se realicen de forma local al que los recibió. Esto hace que el sistema no sea (o no se acerque a ser) *peer-to-peer*, pero en el sentido amplio de que un cliente pueda hacer en cierto momento las veces de servidor sin ningún truco extraño. Sin embargo, CORBA ofrece la posibilidad de *call-backs*, que hacen que los objetos cliente se puedan comportar como servidor en un momento dado.

2.1.2. RMI-IIOP

Una de las tecnologías fundamentales que aporta la J2EE, como alternativa para la programación de aplicaciones utilizando objetos distribuidos, es la de RMI-IIOP.

RMI-IIOP permite escribir objetos distribuidos en Java, facilitando su comunicación con otros objetos en memoria compartida, a través de distintas máquinas virtuales Java y a través de redes IP. Esta tecnología solventa algunos de los problemas fundamentales de los que adolece la tecnología RMI, vista anteriormente, pero sin llegar a solucionar todos los que se han planteado.

Antes de empezar a estudiar las características específicas de la tecnología RMI-IIOP sería conveniente destacar que las similitudes entre ésta y RMI son claras. Como se ha dicho anteriormente, RMI-IIOP surge como una evolución natural de RMI que adopta gran parte de su filosofía y lo único que persigue es solucionar algunos inconvenientes presentes en RMI y que se verán más adelante. Por lo tanto se verán a continuación una introducción a RMI-IIOP, sus diferencias fundamentales respecto a la programación utilizando RMI, para finalizar con un estudio acerca de las ventajas e inconvenientes, también respecto a RMI.

2.1.2.1. Arquitectura RMI-IIOP

Los protocolos de objetos distribuidos definen el formado de mensajes de red enviados entre distintos espacios de direccionamiento o máquinas virtuales. En la actualidad la mayoría de los servidores de aplicaciones soportan el IIOP (Internet Inter-ORB Protocol) de CORBA como protocolo de objetos distribuido.

RMI sobre IIOP (Internet Inter ORB Protocol) combina la facilidad de programación que ofrece RMI con la interoperabilidad que ofrece CORBA. El protocolo IIOP conecta productos CORBA de diferentes vendedores, asegurando la interoperabilidad a través de ellos. De esta forma un cliente RMI puede interactuar con un servidor RMI o con un servidor RMI-IIOP. Un cliente RMI-IIOP podría interactuar con un servidor RMI-IIOP como es lógico o con un servidor RMI. Y finalmente un cliente CORBA podría interactuar con un servidor RMI-IIOP o con un servidor CORBA.

Un ejemplo claro de la versatilidad que se alcanza con RMI-IIOP se aprecia a continuación. Supóngase que se diseña un nuevo objeto distribuido usando la interfaz Java RMI-IIOP. En este caso, se puede generar el IDL (Inteface Definition Language) de ese objeto con la herramienta *rmic*. Partiendo de este IDL se puede implementar un objeto CORBA en C++, por ejemplo. Este objeto C++ es un objeto CORBA puro que puede ser llamado por un objeto CORBA cliente o por un objeto RMI-IIOP cliente. A su vez, para el cliente RMI-IIOP este objeto CORBA C++ aparece como un objeto RMI-IIOP puro porque esta definido por una interfaz Java RMI-IIOP.

Igualmente, si un objeto es implementado con RMI-IIOP, el objeto aparece como un objeto CORBA ante un objeto CORBA cliente porque el objeto CORBA cliente accede a él a través de su IDL.

Se verán ahora cuales son la diferencias fundamentales entre la programación utilizando RMI y RMI-IIOP.

Como se ha comentado anteriormente, RMI-IIOP sigue básicamente la misma filosofía de programación que RMI, por lo que la diferencias fundamentales entre ambas tecnologías se podrían resumir de la siguiente manera:

- Respecto a los paquetes de clases que se utilizan en la implementación de los objetos distribuidos en RMI-IIOP, ahora se utilizan los pertenecientes en el paquete *javax.rmi* en vez de las clases del paquete *java.rmi*.
- La clase a distribuir ya no extiende *UnicastRemoteObject*, sino que extiende *PortableRemoteObject*.

- Los objetos ya no se registran con *Naming.bind* (“Nombre”, objeto) sino que se utiliza *InitialContext.rebind* (“Nombre”, objeto).
- La generación de los *stubs* y los *skeletons* se realiza utilizando *rmic –iiop* y no *rmi* únicamente.
- El servidor de nombres en RMI se denomina *rmiregistry* y ahora en RMI-IIOP el servidor de nombre es *tnameserv*.

Básicamente, son éstas las únicas diferencias existentes entre RMI y RMI-IIOP, por lo que se aprecia que hacer un cambio de tecnología desde RMI a RMI-IIOP permitiendo la interoperabilidad entre distinto lenguajes resulta bastante simple.

2.1.2.2. Ventajas e Inconvenientes

En principio, al ser RMI-IIOP una evolución de RMI se puede deducir que todas las ventajas de RMI-IIOP son las mismas que las que se encontraban en el apartado de RMI. Por lo tanto, no se enumerarán dichas ventajas expuestas ya con anterioridad, aunque si se enumerarán las ventajas que aporta RMI-IIOP como resultado de la solución de los inconvenientes encontrados en RMI:

- RMI-IIOP es el protocolo estándar para la tecnología de componentes Enterprise JavaBeans.
- RMI-IIOP permite mayor interoperabilidad. Uno de los principales inconvenientes de RMI es que solo soporta el lenguaje de programación Java, lo que hace difícil su integración en sistemas software ya construidos. Sin embargo con la aparición de RMI-IIOP este problema se solventa y se permite que cualquier cliente, escrito en cualquier lenguaje de programación, pueda interactuar con cualquier servidor, también escrito en cualquier lenguaje. De esta forma se consigue una tecnología mucho más *abierta* y que facilita la integración de software ya construido.

Por otro lado, para el resto de inconvenientes presentes en RMI, RMI-IIOP no aporta una solución valida. Solo CORBA consigue solucionar problemas como la meta-información y auto-descripción, o el intercambio de roles entre cliente y servidor que hacen posible la utilización de las denominadas llamadas *callbacks* sin necesidad de

recurrir a trucos de programación extraños. Sin embargo CORBA es un estándar que sale fuera del ámbito de la plataforma Java J2EE.

En cuanto a los inconvenientes propios de la tecnología RMI-IIOP, hay que decir que no dispone de garbage collector distribuido, activación de objetos y descarga dinámica de clases.

2.1.3. Java Naming and Directory Interface (JNDI)

El interfaz Java Naming and Directory Interface (JNDI) o interfaz Java de nombres y directorios es una API Java considerada como una extensión estándar de la plataforma J2EE, que proporciona funcionalidades de nombrado y directorio a las aplicaciones escritas usando Java. Está API está definida para ser independiente de cualquier implementación de servicio de directorio.

De esta forma, con JNDI se puede acceder a una gran variedad de directorios (nuevos o ya desarrollados) de una forma común. Se puede acceder a sistemas de nombrado y directorios como LDAP, NetWare, DNS, NIS, sistemas de ficheros, etc.

2.1.3.1. Sistemas de nombrado y directorios

Esta sección ofrece una rápida introducción a los conceptos fundamentales de los sistemas de nombrado y directorios antes de entrar en lo que es realmente la arquitectura JNDI. A continuación se describirán los conceptos que maneja cada uno de estos servicios cuando se utilizan en sistemas distribuidos entre distintos ordenadores.

2.1.3.1.1. Conceptos de servicio de nombrado

Cuando se utilizan sistemas distribuidos entre distintos ordenadores es necesario recurrir a ciertas facilidades como el servicio de nombrado. Este servicio permite conocer qué nombres están asociados con qué objetos y cómo se localizan los objetos basándose en sus nombres.

Cuando se usa cualquier programa o sistema de ordenador, siempre se está accediendo a algún objeto. Por ejemplo, cuando usamos un sistema de correo electrónico, debemos proporcionar el nombre del destinatario al que queremos enviar el correo o si queremos acceder a un fichero del sistema de ficheros, debemos suministrar su nombre.

Por lo tanto, un servicio de nombres permite localizar un objeto a partir su nombre. La función primaria de este servicio será la asociar nombres amigables para las personas con objetos, como direcciones, identificadores, u objetos típicamente usados por programas de ordenador. Por ejemplo, el Internet Domain Name System (DNS) asocia nombres de máquinas (como www.fi.upm.es) a direcciones IP (como 192.9.48.5). Un sistema de ficheros asocia un nombre de fichero (por ejemplo, c:\bin\autoexec.bat) a un manejador de fichero que el programa pueda usar para acceder a los contenidos del fichero. Estos dos ejemplos ilustran bien el amplio rango de escalas en las que existen servicios de nombrado, desde nombrar un objeto en Internet hasta nombrar un objeto en el sistema local de ficheros.

Un sistema de nombrado proporciona un servicio de nombrado a sus clientes y les permite realizar ciertas operaciones relacionadas sobre él. A ese servicio de nombrado se accede a través del propio interfaz del sistema de nombrado. Por su parte, un espacio de nombres es el conjunto de nombres presente en un sistema de nombrado.

Es decir que el DNS, por ejemplo, ofrece por una parte un servicio de nombres que asocia el nombre de una máquina a una dirección IP y el espacio de nombres DNS contendrá todos los nombres de dominios DNS y sus correspondientes entradas en forma de direcciones IP.

Nombres

Para localizar un objeto en un sistema de nombrado, suministramos el nombre del objeto. El sistema de nombrado determina la sintaxis que deben seguir los nombres conocida como convención de nombrado.

Por ejemplo, el nombre de path UNIX, “/usr/alejo”, nombra un fichero “alejo” en el directorio “usr”, que está localizado en el directorio raíz del sistema de ficheros. La convención de nombrado DNS llama a los componentes en el nombre DNS para ordenarlos de derecha a izquierda y están delimitados por puntos ("."). Así el nombre DNS “fi.upm.es” nombra una entrada DNS con el nombre “fi”, relativo a la entrada DNS “upm.es”. O finalmente, la convención de nombrado de Lightweight Directory Access Protocol (LDAP) ordena sus componentes, formados por pares atributo=valor, de derecha a izquierda, delimitados por comas (",").

Uniones

La asociación de un nombre con un objeto se llama una unión. Por ejemplo, un nombre de fichero está unido a un fichero. Por su parte el DNS contiene uniones que asocian nombres de máquinas a direcciones IP y un nombre LDAP estará unido a una entrada LDAP.

Referencias

Dependiendo del servicio de nombres, algunos objetos no pueden almacenarse directamente, es decir, no se puede situar una copia del objeto dentro del servicio de nombrado. En su lugar, deben ser almacenados por referencia; es decir, un puntero o una referencia al objeto se almacenan dentro del servicio de nombrado. Una referencia es información sobre cómo acceder a un objeto. Típicamente, es una representación mucho más compacta que puede ser usada para comunicarse con el objeto, mientras que el propio objeto podría contener más información de estado.

Una referencia es una representación mucho más compacta de información sobre un objeto, y puede usarse para obtener información adicional. Por ejemplo, un objeto impresora, podría contener el estado de la impresora, como su cola actual y la cantidad de papel que le queda. Por otro lado, una referencia al objeto impresora, sólo podría contener información sobre cómo llegar hasta ella, como el nombre de su servidor de impresión y el protocolo de impresión.

Contexto

Un contexto es un conjunto de uniones nombre-objeto. Cada contexto tiene una convención de nombrado asociada. Un contexto proporciona una operación de localización (o resolución) que devuelve el objeto y podría proporcionar operaciones como aquellas para unir nombres, desunir nombres y listar uniones de nombres. Un nombre en un objeto contexto puede unirse a otro objeto contexto (llamado un subcontexto) que tenga la misma convención de nombrado.

Por ejemplo, un directorio de ficheros, como "/usr", en el sistema de ficheros UNIX es un contexto. Un directorio de ficheros nombrado en relación a otro directorio de ficheros es un subcontexto (algunos usuarios UNIX se refieren a esto como un subdirectorio). Es decir, en un directorio de fichero "/usr/bin", el directorio "bin" es un subcontexto de "usr".

2.1.3.1.2. Conceptos de servicio de directorio

Muchos servicios de nombrado se amplían con un servicio de directorio. Un servicio de directorio asocia nombres con objetos y también permite a dichos objetos tener atributos. Así, no solo podemos localizar un objeto por su nombre sino que también podemos obtener sus atributos o buscar el objeto basándonos en sus atributos.

Un servicio de directorio puede usarse para almacenar una gran cantidad de información que puede ser usada por los clientes de ese servicio. Un servicio directorio trabaja con objetos directorio que contienen atributos que describen los objetos que representa. Así pues, un objeto directorio se puede usar, por ejemplo, para representar una impresora, que tuviera como atributos su velocidad, su resolución y color o un usuario podría estar representado por un objeto directorio que tenga como atributos la dirección e-mail del mismo, su N.I.F. e información de sus multas de tráfico.

Los directorios normalmente ordenan sus objetos en árboles. Por ejemplo, LDAP ordena todos sus objetos directorio en un árbol, llamado Directory Information Tree (DIT). En este DIT, un objeto organización, por ejemplo, podría contener objetos departamentos que a su vez podrían contener objetos personas. Cuando los objetos directorios se ordenan de esta forma, juegan el papel de contextos de nombres además

del de contenedores de atributos. De esta forma se consigue combinar el Servicios de Nombres y el Servicio de Directorios.

Atributos

Como se ha comentado anteriormente, un objeto directorio puede tener atributos. Un atributo tiene un identificador de atributo y un conjunto de valores de atributo.

Un identificador de atributo es un "token" o palabra reservada que identifica un atributo independientemente de sus valores. Por ejemplo, dos cuentas de ordenador diferentes podrían tener un atributo "mail", donde "mail" es el identificador del atributo. Por su parte, un valor de atributo es el contenido del propio atributo. La dirección e-mail, por ejemplo, podría tener un identificador de atributo de "mail" y el valor de atributo de "alejandro.barrera@somewhere.com".

Directorios y servicios de directorio

Un servicio de directorio es un servicio que proporciona operaciones para crear, añadir, eliminar y modificar los atributos asociados con los objetos de un directorio. A este servicio se accede a través de su propio interfaz. Se denominará directorio a un conjunto de objetos directorios relacionados.

Hay muchos ejemplos de servicios de directorios posibles. El Novell Directory Service (NDS) es un servicio de directorio de Novel que proporciona información sobre muchos servicios de red, como los servicios de ficheros e impresión. Network Information Service (NIS) es un servicio de directorio disponible en el sistema operativo Solaris para almacenar información relacionada con el sistema, como la que relaciona máquinas, redes, impresoras y usuarios. El Netscape Directory es un servicio de directorio de propósito general basado en el estándar de Internet LDAP.

Filtros de Búsqueda

Podemos localizar un objeto directorio suministrando su nombre al servicio de directorio. De forma alternativa, muchos directorios, como aquellos basados en LDAP, soportan nociones de búsquedas. De esta forma cuando buscamos, podemos suministrar una consulta que consiste en una expresión lógica en la que especificamos

los atributos que el objeto u objetos deben tener. A esta consulta se le llama filtro de búsqueda. Este estilo de búsqueda algunas veces es llamado búsqueda basada en contenido.
El servicio de directorio busca y devuelve los objetos que cumplan el filtro de búsqueda. Por ejemplo, podemos pedirle al servicio de directorio que busque todos los usuarios que tengan el atributo "edad" mayor de 40 años.

2.1.3.2. Arquitectura JNDI

La arquitectura JNDI consiste en una API de acceso y un proveedor de servicios de nombrado o Service Provider Interface (SPI). Así pues, por un lado las aplicaciones Java usarán la API JNDI para acceder a esa gran variedad de servicios de nombres y directorios, mientras que por otro, el proveedor de servicios permitirá conectar de forma transparente una gran variedad de servicios de nombres y directorios.

Por lo tanto, para usar JNDI, debemos tener las clases pertenecientes a la API JNDI, y uno o más proveedores de servicios. Por ejemplo, las últimas distribuciones de Java 2 SDK incluyen los correspondientes proveedores de servicios para servicios de nombres y directorios como los siguientes:

- Protocolo Lightweight Directory Access Protocol (LDAP).
- Servicio de nombres Common Object Services (COS) para la arquitectura Common Object Request Broker Architecture (CORBA).
- Registro de nombres para Java RMI (rmiregistry) y para Java RMI-IIOP (tnameserv).

Figura xx: Arquitectura JNDI

La API JNDI está dividida en los siguientes cinco paquetes:

- javax.naming: Este paquete contiene las clases e interfaces para acceder a servicios de nombrado.
- javax.naming.directory: Este paquete extiende el paquete javax.naming para proporcionar la funcionalidad de acceder al servicio de directorio, además del servicio de nombres. Este paquete permite a las aplicaciones recuperar atributos asociados con objetos almacenados en el directorio y buscar objetos usando los atributos especificados.
- javax.naming.event: Este paquete contiene las clases e interfaces que soportan notificación de eventos tanto en los servicios de nombres y como en los de directorios.
- javax.naming.ldap: Este paquete contiene las clases e interfaces que permiten usar las características especificas para el LDAP v3 que no están ya cubiertas por el paquete más genérico javax.naming.directory. De hecho, la mayoría de las aplicaciones JNDI encontrarán suficiente el paquete javax.naming.directory y no necesitarán usar el paquete javax.naming.ldap. Este paquete es principalmente para aquellas aplicaciones que necesitan usar las operaciones "extendidas", controles o notificaciones no solicitadas que aporta la v3 de LDAP.
- javax.naming.spi: Este paquete proporciona la clave de la carga dinámica del proveedor de servicios a través de su interfaz. Es un paquete que al igual que paquetes como la API de JDBC, esta diseñado a partir de interfaces, con lo que se consigue independencia total de la implementación específica del servicio de nombres o directorios. Este es el paquete que le da significado a por qué los desarrolladores de diferentes proveedores de servicios de nombres o directorios pueden desarrollar y mantener sus implementaciones para que los servicios correspondientes sean accesibles desde aplicaciones que usan JNDI.

En los siguientes apartados se estudiarán más detalladamente las clases y los interfaces que pertenecen a los paquetes directamente relacionados con los sistemas de nombrado y de directorios.

2.1.3.2.1. Paquete Naming

La interfaz Context

Este paquete define la interfaz Context, que consiste en un conjunto de asociaciones o uniones nombre-objeto. Este interfaz es el encargado de proporcionar las herramientas para localizar, crear asociaciones o destruirlas, renombrar objetos y crear y destruir subcontextos.

A continuación se enumeran las operaciones más utilizadas que proporciona este interfaz:

- public Object lookup(Name name) throws NamingException;

  El método lookup() recibe como parámetro el nombre del objeto que queremos localizar, y devuelve el objeto unido a ese nombre. Por ejemplo, el siguiente fragmento de código localiza una impresora y envía un documento al objeto impresora para imprimirlo.

```
Printer printer = (Printer)ctx.lookup("laser_print");
printer.print(report);
```

  Cada método de este interfaz Context tiene dos sobrecargas: una que acepta un argumento Name y otra que acepta un nombre java.lang.String.

  Name es un interfaz que representa un nombre genérico (como una secuencia ordenada de cero o más componentes). Para los métodos del interfaz Context, un argumento Name, que es un tipo de la clase CompositeName, representa un compuesto, por eso podemos nombrar un objeto usando un nombre que abarque varios espacios de nombres. Las sobrecargas que aceptan Name son útiles para aplicaciones que necesitan manipular nombres, es decir, componerlos, comparar componentes, etc.

Por su parte, los métodos sobrecargados que aceptan como argumento nombres del tipo java.lang.String son más útiles para aplicaciones sencillas como aquellas que simplemente leen un nombre y localizan el objeto correspondiente.

- public NamingEnumeration listBindings(Name name) throws NamingException;

El método listBindings(), al igual que el método anterior esta sobrecargado y puede recibir como argumento un Name u otro nombre de tipo java.lang.String. En cualquiera de los dos casos devuelve en la clase NamingEnumeration la enumeración de todas las uniones nombre-objeto que pertenecen al contexto pasado como parámetro. El contenido de los subcontextos no es incluido en la respuesta del método.

Cada unión esta representada por un objeto de la clase Binding. Esta unión es un paquete que contiene el nombre del objeto unido, el nombre de la clase del objeto, y el propio objeto.

El método listBindings( ) se ha pensado para aplicaciones que necesitan realizar operaciones en masa sobre los objetos de un contexto. Por ejemplo, un programa administrador de impresoras podría querer rearrancar todas las impresoras de un edificio. Para realizar dichas operaciones, estas aplicaciones necesitan obtener todos los objetos unidos en el contexto. Por eso es mucho más expeditivo devolver los objetos como parte de la enumeración.

- public NamingEnumeration list(String name) throws NamingException;

Este método es similar a listBindings(), excepto en que devuelve en la enumeración objetos del tipo NameClassPair y no Binding. NameClassPair contiene un nombre de objeto y el nombre de la clase del objeto. list ( ) es útil para aplicaciones como navegadores que sólo quiere nombrar los nombres de los objetos de un contexto pero no necesitan realmente los objetos. Por ejemplo, un navegador podría listar los nombres de un contexto y esperar a que el usuario seleccione uno o unos pocos de los nombres mostrados para realizar operaciones futuras. Dichas aplicaciones normalmente no necesitan acceder a todos los objetos

de un contexto. Aunque listBindings( ) proporciona la misma información, es potencialmente una operación más costosa.

La clase InitialContext

La clase InitialContext implementa el interfaz Context por lo que sus métodos más importantes ya se han comentado anteriormente.

Antes de realizar cualquier operación de servicio de nombres o de directorio, necesitamos adquirir un contexto inicial que es el punto de arranque dentro del espacio de nombres. En JNDI, todos las operaciones de nombrado y directorio se realizan en relación a un contexto. Por lo tanto, JNDI define un contexto inicial, denominado InitialContext, que proporciona un punto de arranque para la resolución de nombres y en general para todas las operaciones de nombrado y directorio.

Cuando JNDI crea el InitialContext requiere por lo menos de un par de propiedades que deben ser inicializadas previamente. Estas propiedades son el Initial Context Factory, que indica el proveedor de servicio del sistema de nombrado que se va a utilizar y el Provider URL, que indica la localización del árbol JNDI que se va a utilizar.

Estas propiedades se pueden definir, bien como propiedades de entorno definidas para pasárselas al constructor en el código del cliente, como propiedades que se le pasan al cliente mediante la consola de comandos, como variables de entorno definidas en el sistema, como parámetros que se pasan a un applet o incluso con ficheros de recurso de la aplicación, como puede ser el fichero jndi.properties.

Por ejemplo, se puede especificar el proveedor de servicio a utilizar para el contexto inicial creando un conjunto de propiedades de entorno mediante una Hashtable y añadirle el nombre de la clase del proveedor de servicio y la localización de su árbol JNDI. Si en este caso se usa el proveedor de servicios LDAP de Sun Microsystems, que asume que el servidor ha sido instalado en el puerto 389 de la máquina local dentro de la entrada llamada "o=JNDITutorial" y no se requiere autentificación para

actualizar este directorio, el código para configurar el entorno se debería parecerse a este:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=JNDITutorial");
```

Por otro lado, cuando una clase de la API JNDI necesita determinar el valor de una determinada propiedad, esta hace una búsqueda en las siguientes fuentes de datos en el orden que se expone a continuación:

1. La primera ocurrencia de la propiedad en la variable de entorno que se pasa al contexto como argumento.
2. El fichero proveedor de recursos *jndi.properties* que se encuentra donde apunte la variable *classpath*.

Una vez que tenemos el contexto inicial, podemos usarlo para localizar otros contextos y objetos.

La clase Reference

Los objetos se almacenan de diferentes formas en los servicios de nombres y directorios. Un servicio que soporta almacenamiento de objetos Java podría soportar el almacenamiento de un objeto de forma serializada. Sin embargo, algunos servicios de nombres y directorios no soportan este almacenamiento de objetos Java, por lo que en este caso, un objeto Java serializable podría no ser la representación más apropiada.

Por lo tanto, para estos casos podría ser una buena solución el guardar una referencia al objeto. Esa referencia contiene la información sobre cómo construir una copia del mismo y la JNDI define la clase Reference para representar esa referencia.

La API JNDI intentará convertir las referencias localizadas en un directorio en los objetos Java que esas referencias realmente representan, para que así los clientes

JNDI tengan la sensación de que lo que hay almacenado en el directorio son realmente objetos Java.

Sin embargo hay que mencionar que una referencia podría ser una representación muy compacta de un objeto, mientras que su forma serializada podría contener mucha más información sobre el estado del objeto.

Jerarquía de Excepciones

JNDI define un árbol de clases para excepciones que se pueden lanzar en el curso de las operaciones de nombrado y directorio. La raíz de este árbol de clases es NamingException. Aquello programas interesados en tratar con una excepción particular pueden capturar la subclase correspondiente de la excepción, de otra forma, deberían capturar NamingException.

2.1.3.2.2. Paquete Directory

El interfaz DirContext

El interfaz DirContext, que extiende el interfaz Context, representa un contexto de directorio y define los métodos para examinar y actualizar los atributos asociados con un objeto directorio.

Al extender DirContext del interfaz Context y por lo tanto comportarse como él, esto significa que cualquier objeto directorio también puede proporcionar un contexto de nombrado. Por ejemplo, un objeto directorio para una persona podría contener atributos sobre la persona y también proporcionar un contexto para nombrado de objetos, como la impresora de la persona y el sistema de ficheros relativo al objeto directorio de esa persona.

A continuación se enumeran las operaciones más utilizadas que proporciona este interfaz:

- public Attributes getAttributes(Name name) throws NamingException;

  El método getAttributes (), al igual que los método vistos anteriormente en el paquete javax.naming esta sobrecargado y puede recibir como argumento un Name u otro nombre de tipo java.lang.String. En cualquiera de los dos casos devuelve en la clase Attributes todos los atributos asociados con el objeto directorio pasado como parámetro.

- public void modifyAttributes(Name name, int mod_op, Attributes attrs) throws NamingException;

  Los atributos se modifican usando modifyAttributes ( ). Podemos añadir, reemplazar o eliminar atributos y/o valores de atributos usando esta operación. En el primer argumento pasamos el nombre del objeto cuyos atributos queremos modificar. El tipo de operación dependerá del parámetro que pasemos al método en el segundo argumento: ADD_ATTRIBUTE, REPLACE_ATTRIBUTE, REMOVE_ATTRIBUTE. Y finalmente, en el tercero pasaremos los atributos de la modificación. Este método, al igual que el anterior esta sobrecargado para poder recibir tipos Name o java.lang.String.

- public NamingEnumeration search(Name name, Attributes matchingAttributes) throws NamingException;

  El interfaz DirContext contiene métodos para realizar búsquedas basadas en contenido. Este método también esta sobrecargado y en su forma más sencilla y común de uso, que es la que se comenta ahora mismo, el método busca en el contexto que se pasa como primer parámetro, objetos que contienen el conjunto de atributos especificados en el segundo parámetro, con sus respectivos valores específicos.

  El método devuelve todos los atributos de los objetos encontrados durante la búsqueda en un objeto NamingEnumeration.

- public NamingEnumeration search(Name name, String filter, SearchControl cons) throws NamingException;

  Este método es otra de las formas sobrecargadas de search( ) que soporta filtros de búsqueda más sofisticados.

3. Tecnología de componentes en Java

3.1. Paradigma[1] de componentes

3.1.1. Programación basada en componentes

La programación basada en componentes (PBC) es actualmente uno de los paradigmas de la ingeniería de software que intenta posicionarse como una solución que sea utilizada ampliamente en el mercado de software.

El gran problema que siempre ha sufrido la ingeniería del software radica en que en la actividad del desarrollo el nivel de reutilización sigue siendo muy escaso a pesar del gran número de librerías que existen en el mercado. El uso de estas librerías siempre ha presentado varios inconvenientes:

- Su instalación puede ser compleja.
- Es frecuente que tengan dependencias con otras librerías.
- Suelen tener una aplicación demasiado general: estructuras de datos, interfaces gráficos o comunicación por red, pero rara vez solucionan aspectos de un problema concreto.
- Aunque es evidente que ahorran trabajo, sigue siendo necesario una gran cantidad de programación.

Estos inconvenientes impiden la generalización de la reutilización de código existente en el desarrollo de software. Por lo tanto, la solución a la crisis de la ingeniería del software pasaba por cambiar el modelo de desarrollo actual a un desarrollo orientado a componentes.

Este modelo se basa en la construcción de las aplicaciones a partir de componentes software, bien sean comerciales o gratuitos ya existentes, limitando en lo posible el

[1] Desde el punto de vista de los sistemas de información o de la ingeniería de aplicaciones un paradigma define todo un marco conceptual para una determinada tecnología. Desde un plano más filosófico un paradigma científico define las teorias, terminologias, métodos, presupuestos o instrumental dentro del que se enmarca la investigación científica en un momento dado. El geocentrismo de Ptolomeo y Aristóteles dominó el pensamiento hasta la llegada del heliocentrismo de Copérnico y el sistema de Newton lo hizo hasta que se impuso la relatividad de Einstein.

desarrollo de código nuevo. Además, se demostró que el producir, distribuir, comprar y usar componentes de software, es el camino a seguir, debido a que todas las demás disciplinas de ingeniería han introducido el modelo de componentes desde que han llegado a ser maduras.

La implantación de este modelo trae consigo múltiples beneficios que se enumeran a continuación:

- El tiempo de desarrollo y el coste de de una aplicación es muy inferior.
- El desarrollo será también mucho más sencillo: se trata de seleccionar, instalar e integrar componentes ya implementados.
- El código necesario para esta integración es mínimo.
- La posibilidad de errores es mínima ya que los componentes son sometidos a un riguroso control de calidad antes de ser comercializados.
- Las empresas de desarrollo de software especializadas en un ámbito determinado desarrollan con el tiempo una serie de componentes de calidad para su uso interno que también pueden comercializar

Además, con la aparición de los servidores de aplicaciones, el desarrollo de sistemas software se vio beneficiado por la facilidad de inclusión de todos los servicios que proporcionaban los *middleware*. De esta forma el desarrollador se olvidaba de tener que implementar él mismo esos servicios y podía enfocar toda su labor a solucionar sus problemas de lógica de negocio.

Pero eso no fue todo. Posteriormente surgió la posibilidad de adquirir (bien comprando o desarrollando dentro de la propia organización) soluciones parciales a esos problemas de lógica de negocio a los que antes se hacia mención. Para conseguir esto, surgió la arquitectura de componentes. A partir de ese momento, las aplicaciones que solucionan los problemas se construyen utilizando piezas de puzzle denominadas: componentes software.

En definitiva, la programación basada en componentes (PBC) es aquella que se basa en la implementación de sistemas utilizando componentes previamente programados

y probados. Por su parte, la construcción de esos componentes se realiza mediante la programación orientada a componentes (POC).

La programación orientada a componentes (POC) aparece como una variante natural de la programación orientada a objetos (POO) para los sistemas abiertos, en donde la POO presenta algunas limitaciones; por ejemplo, la POO no define una unidad concreta de composición independiente de las aplicaciones (los objetos no lo son, claramente), y define interfaces de demasiado bajo nivel como para que sirvan de contratos entre las distintas partes que deseen reutilizar objetos.

En cambio numerosas son las características que aporta la programación orientada a componentes (POC) frente a la programación orientada a objetos (POO) tradicional. Entre ellas, el desarrollo de los componentes de forma independiente del contexto en donde serán ejecutados, la reutilización por composición (frente a herencia, al tratarse de entidades binarias o cajas negras), la introspección (facilidad para interrogar al componente sobre sus propiedades y métodos de forma dinámica, normalmente mediante el uso de reflexión), nuevas formas de comunicación (como los eventos y las comunicaciones asíncronas frente a los rudimentarios mecanismos de los objetos), el enlazado dinámico o la composición tardía.

Desde otra perspectiva, la programación orientada a objetos (POO) se centra en las relaciones entre las clases que están combinadas dentro de un programa ejecutable. En el mundo de la POO tradicional, cuando las clases son compiladas, el resultado es una aplicación monolítica binaria. Todas esas clases comparten la misma unidad física de implementación, típicamente un EXE, que también comparten los procesos, los espacios de direcciones y hasta los privilegios de seguridad. Por su parte, la programación orientada a componentes (POC) se centra en los módulos de código intercambiables que trabajan independientemente y que no requieren que quien los utiliza esté familiarizado con su forma de trabajar interna.

La POC nace con el objetivo de construir un mercado global de componentes software, cuyos usuarios son los propios desarrolladores de aplicaciones que necesitan reutilizar componentes ya hechos y probados para construir sus aplicaciones de forma más rápida y robusta. Las entidades básicas de la POC son los componentes, en el

mismo sentido que se han definido anteriormente, es decir cajas negras que encapsulan cierta funcionalidad y que son diseñadas para formar parte de ese mercado global de componentes, sin saber ni quién los utilizará, ni cómo, ni cuándo. Los usuarios conocen acerca de los servicios que ofrecen los componentes a través de sus interfaces y requisitos, pero normalmente ni quieren ni pueden modificar su implementación.

3.1.2. Componentes software

Un componente software en una unidad de software independiente con una interfaz explícita que puede utilizarse para componer aplicaciones. Los componentes no son aplicaciones enteras que pueden ejecutar por ellas mismas, por el contrario, los componentes son fragmentos de lógica de negocio gestionables y bien definidos. Más bien, un componente software puede considerarse como una colección bien definida de objetos.

Se considera que un componente software debe tener las siguientes características:

1. Es una unidad software compilada reutilizable, con una interfaz bien definida que puede colaborar con otros componentes para resolver un problema. Para que un componente pueda funcionar con otros componentes de terceras partes, éste necesita ser suficientemente independiente, y poseer unas especificaciones lo suficientemente detalladas como para saber que es lo que requiere y que es lo que provee. En otras palabras, un componente necesita encapsular su implementación e interactuar con su ambiente a través de interfaces bien definidas.

2. Se distribuye en un único paquete instalable que contiene dentro sí todo lo necesario para su funcionamiento, con ninguna o muy pocas dependencias con otros componentes o librerías. Un componente deberá encapsular las características que lo constituyen y nunca deberá ser desarrollado parcialmente. En este contexto una tercera parte no debe esperar tener acceso a los detalles de construcción de todo lo que involucra a los componentes.

3. Puede estar implementado en cualquier lenguaje de programación y ser utilizado también para el desarrollo en cualquier lenguaje de programación. Sin embargo, hay que comentar que los lenguajes orientados a objetos son especialmente adecuados para este fin.

4. Normalmente es un producto comercial de calidad, vendible y realizado por un fabricante especializado, aunque por supuesto pueden existir componentes gratuitos.

Un componente software resulta una herramienta muy poderosa porque, por ejemplo, permite a una organización comprar módulos software bien definidos, que solucionan determinados problemas y que combinados con otros componentes software permite solucionar problemas globales.

En la actualidad existen tres tecnologías de componentes predominantes y que se caracterizan por su alto nivel de incompatibilidad entre sí:

1. VBX/OCX/ActiveX de Microsoft: los VBX surgieron como una forma de poder distribuir controles entre los desarrolladores de VisualBasic. Los OCX y ActiveX son evoluciones posteriores de los VBX.
2. VCL de Borland. Es la tecnología de componentes utilizados por los entornos de desarrollo Delphi y C++ Builder. Curiosamente estos dos entornos también permiten la utilización de componentes OCX y ActiveX.
3. JavaBeans de Sun: es la tecnología de componentes basada en Java, que al contrario que otras tecnologías, funciona en cualquier plataforma.

Nota de seguimiento: todo por la pasta

No hay que perder de vista el hecho de que los componentes, entendidos como activos reutilizables, son finalmente creados para ser vendidos como productos separados. Por este motivo, al ser comparados con soluciones específicas para problemas específicos, se aprecia como los componentes al necesitar ser cuidadosamente generalizados para permitir su uso en una variedad de contextos, requieren de un mayor trabajo que irremediablemente influirá en la inversión realizada.

Resulta evidente que resolver un problema general en lugar de uno específico implica más trabajo. Además, debido a la variedad de contextos de implementación, la creación de documentación, las pruebas, los tutoriales, las ayudas en línea, entre otros aspectos, es más costosa para los componentes que para las soluciones específicas.

Así, los componentes solo serán viables si la inversión realizada durante su creación es menor que los beneficios resultado de su implementación. Por supuesto, los beneficios deberían buscarse en la comercialización de dichos componentes.

Por lo tanto, los componentes de software, como el resto de los componentes en cualquier área, necesitan ser entendidos en términos del mercado en el se encuentran dentro. Hay que tener claro que una tecnología imperfecta en un mercado de trabajo es sostenible mientras que una tecnología perfecta sin ningún mercado será vana.

Un nuevo producto puede crear un mercado solo si su llegada ya se estaba esperando de otra manera, el costo de crear demanda puede ser prohibitivo. Los pronósticos para el mercado de componentes son tan confiables como los pronósticos del tiempo. Sin embargo, lo que si resulta evidente es la importancia que tiene para el mercado de desarrollo de componentes, los estándares de componentes. Los estándares son útiles para establecer acuerdos en modelos comunes e incluso llegar a habilitar la interoperabilidad.

3.1.3. JavaBeans

Los JavaBeans son la implementación de la tecnología de componentes particularizada al lenguaje de programación Java. En resumidas cuentas un JavaBean es una clase Java que se ajusta a los siguientes criterios:

1. Es pública que implementa el interfaz java.io.Serializable.
2. Expone una serie de propiedades que pueden ser leidas y modificadas desde el entorno de desarrollo.
3. Posee un constructor público sin argumentos.
4. Posee métodos *get* y *set* públicos para acceder a las propiedades. Los métodos *get* no tendrán argumentos, salvo que se refieran a propiedades indexadas.
5. Expone una serie de eventos que pueden ser capturados y asociados a una serie de acciones.

Un JavaBean es un componente Java reutilizable con propiedades, eventos y métodos que pueden ser fácilmente conectado para crear aplicaciones enteras. La apariencia y comportamiento de un JavaBean viene determinada por sus propiedades.

Una propiedad simple queda identificada en un JavaBean por un par de operaciones *get* y *set* de la siguiente forma:

```
public <TipoProp> get<NombreProp>() { ... }

public void set<NombreProp> (<TipoProp> p) { ... }
```

En un JavaBean es posible definir también propiedades indexadas. Estas propiedades representan colecciones de elementos y se identifican mediante los siguientes patrones de operaciones:

```
// Acceso al array completo de de valores:

public <TipoProp>[] get<NombreProp>()
public void set<NombreProp> (<TipoProp>[] p)

//Acceso valores individuales:

public <TipoProp> get<NombreProp>(int index)
public void set<NombreProp> (int index, <TipoProp> p)
```

Un entorno de desarrollo (IDE) orientado a componentes JavaBeans debe facilitar la instalación de dichos componentes y su configuración e integración en una aplicación. Una vez instalado un JavaBean, el entorno de desarrollo debe ser capaz de identificar sus propiedades simplemente detectando las parejas de operaciones *get*/*set*, mediante la capacidad que Java denominada introspección. Este mecanismo automático de introspección, conseguido a través de la reflexión, es suficiente para que el entorno obtenga toda la información de propiedades y eventos del bean.

Finalmente hay que comentar que la forma habitual de distribución de un JavaBean es a través de un fichero *.jar* que debe contener además un fichero *manifest*.

Nota de seguimiento: no hay que mezclar churras con merinas

Cuidado, no hay que confundir “atributo” con “propiedad”. Ambos describen el estado interno y el comportamiento, pero en dos contextos totalmente diferentes: por un lado, los atributos pertenecen al mundo de las clases dentro de la programación orientada a objetos (POO) y por otro, las propiedades pertenecen al mundo de los componentes dentro de la programación orientada a componentes (POC).

Dice una de las máximas del paradigma orientado a objetos que los atributos nunca deben ser públicos y que en su lugar, deben existir métodos públicos *accesor* y *mutator* (*get* y *set*) que permitan acceder a dichos atributos. En cambio las propiedades siempre son características públicas de los componentes. Por otro lado, los atributos son accesibles únicamente a nivel de programación. En cambio las propiedades son accesibles tanto a nivel de programación como interactivamente desde el entorno de desarrollo.

Desde un punto de vista estricto de diseño, cuando se implementa un componente mediante una clase del paradigma de objetos, si que puede darse el caso de que cada propiedad pueda estar asociada internamente a un atributo. Pero, cuidado, esto no siempre ocurre.

Como se puede comprobar, hay suficientes argumentos "de peso" como para no confundir atributos con propiedades. Casi los mismos argumentos "de peso" que debían tener los esquiladores para no mezclar las churras, ovejas de lana gruesa, con las merinas, también ovejas, pero de lana mucho más fina.

3.1.4. JavaBeans vs Enterprise JavaBeans

Antes de entrar de lleno en el mundo de los componentes distribuidos Java es conveniente hacer un pequeño alto en el camino y consolidar un par de términos que pueden causar cierta confusión y que como se verá a continuación son completamente distintos: JavaBeans y Enterprise JavaBeans.

Los JavaBeans se diseñaron para permitir la posibilidad de ampliar las herramientas de desarrollo Java con nuevos componentes y facilitar así la labor del desarrollador. Esos nuevos componentes se podían integrar en otros sistemas y permitían al desarrollador el empleo de funcionalidades que de ese modo no requerían una re-implementación ya que el JavaBean tenía encapsulada esa funcionalidad y era capaz de exportar sus propiedades y métodos.

Ahora bien, por su parte los Enterprise JavaBeans son algo mucho más grande, poderoso e interesante que los JavaBeans, como se verá en el transcurso de este libro. De hecho, se podrían considerar a los JavaBeans como los hermanos pequeños de los Enterprise JavaBeans.

Sin embargo, la diferencia fundamental entre ambas tecnologías no es ni mucho menos su tamaño. La mayor diferencia radica en que los JavaBeans son componentes de desarrollo y no componentes desplegables como lo son los Enterprise JavaBeans. Es decir, un JavaBean no es un componente que sea desplegable dentro de un entorno

de ejecución, sino que un JavaBean colabora en el desarrollo de otras entidades software mas grandes que son realmente las que si se despliegan en un entorno de ejecución como puede ser un servidor de aplicaciones.

De otra forma, un JavaBean es un componente, que al ser como una clase Java sirve para construir una aplicación, pero que no necesita para vivir un entorno de ejecución como un servidor de aplicaciones que la instancie o la destruya, como se verá que pasa con los Enterprise JavaBeans.

3.2. Arquitectura de componentes distribuidos Java: EJB

Un sistema de componentes distribuidos es un sistema de componentes software que pueden estar ejecutándose en diferentes máquinas. Para este tipo de escenarios la plataforma Java 2 Enterprise Edition (J2EE) aporta la tecnología de EJB que surge como una especificación que pretende proporcionar un estándar para el desarrollo de aplicaciones software distribuidas de empresa.

Por lo tanto, la especificación de Enterprise JavaBeans (EJB) surge ante la necesidad de dar respuesta a un problema como el desarrollo de aplicaciones distribuidas, a través de un nuevo modelo de programación, como es el de los componentes software. Esta situación hace necesaria la existencia de una serie de acuerdos o protocolos que dicten las normas que permitan llevar a cabo la implementación de software distribuido de acuerdo a las premisas que define dicho paradigma de componentes.

La especificación 2.1 de Enterprise JavaBeans (EJB) define una arquitectura para el desarrollo y despliegue de aplicaciones basadas en objetos distribuidos transaccionales o componentes software del lado del servidor. Las organizaciones pueden construir sus propios componentes o comprarlos a vendedores de terceras partes. Estos componentes del lado del servidor, llamados Enterprise JavaBeans, son

objetos distribuidos que están localizados en contenedores de Enterprise JavaBeans y que proporcionan servicios remotos para clientes distribuidos a lo largo de la red.

La especificación de Enterprise JavaBeans define una arquitectura para un sistema transaccional de objetos distribuidos basado en componentes. La especificación define un modelo de programación; es decir, convenciones o protocolos y un conjunto de clases e interfaces que crean el API EJB. Este modelo de programación proporciona a los desarrolladores de JavaBeans y a los vendedores de servidores EJB un conjunto de contratos que definen una plataforma de desarrollo común. El objetivo de estos contratos es asegurar la portabilidad a través de los vendedores y el soporte de un rico conjunto de funcionalidades.

Finalmente, hay que decir que la especificación de Enterprise JavaBeans determina una serie de interfaces entre los servidores de aplicaciones y los propios componentes. Esta especificación permitirá así que cualquier componente pueda ejecutar dentro de cualquier servidor de aplicaciones y que cualquier componente pueda ser intercambiado entre servidores de aplicaciones sin necesidad de cambiar su código o potencialmente tener que recompilarlo.

3.2.1. Aplicaciones con EJB

La arquitectura EJB es una tecnología para el desarrollo y despliegue de componentes para aplicaciones del lado servidor que encapsulan la lógica de negocio y el acceso a datos de una aplicación de empresa.

Dentro de esta arquitectura, se entenderá como Enterprise JavaBean (en adelante simplemente bean), un componente del lado del servidor que representa un concepto de negocio de empresa como puede ser un cliente o una cuenta bancaria. Además, dichos beans serán componentes escalables, transaccionales y seguros a nivel multiusuario.

El desarrollo de aplicaciones o sistemas software, con la tecnología de EJB requiere una estrecha colaboración entre distintas partes del sistema como se irá viendo a lo

largo de los siguientes capítulos. Esta tecnología requiere, por un lado a los beans, por otro al contenedor de beans o servidor de aplicaciones, e incluso a los proveedores de herramientas para el desarrollo o el despliegue de las aplicaciones. Además, ésta tecnología requiere de la existencia de una serie de roles durante la fase de desarrollo que también se verán a continuación.

La especificación EJB define, además de esos roles, una serie de interfaces y directrices que estos roles deben seguir para asegurar que todo el proceso de desarrollo se lleva a cabo correctamente. Un ejemplo de estas directrices puede ser el interfaz o contrato bean-contenedor que está diseñado para hacer portables los beans entre distintos contenedores EJB y para que los beans puedan ser desarrollados una sola vez y puedan ejecutarse en cualquier contenedor EJB. Idealmente, cualquier bean que cumpla la especificación debería poder ejecutarse en cualquier contenedor EJB compatible.

Así pues, una aplicación EJB desde el lado servidor consiste en beans que viven en un contenedor EJB y que son accedidos por aplicaciones clientes (bien aplicaciones *stand-alone* o aplicaciones en entorno *web*) a través de la red usando sus interfaces remotos. Estos interfaces exponen las capacidades del bean y proporcionan todos los métodos necesarios para crear, actualizar, borrar e interactuar con el bean. Obviamente, todos los proveedores que venden servidores de aplicaciones, como BEA, Borland, IBM, o iPlanet incluyen también los contenedores EJB que albergarán a los beans.

Por otro lado, la especificación EJB requiere que usemos una versión especializada del API Java RMI cuando trabajamos con un bean de forma remota. Java RMI es un API para acceder a objetos distribuidos y de alguna forma es un protocolo genérico, de la misma forma que JDBC accede a cualquier base de datos de forma genérica. Sin embargo, los servidores EJB soportan el protocolo IIOP, por lo que la especificación de EJB usa la tecnología denominada Java RMI-IIOP como protocolo estándar, que incluye tanto IIOP como protocolo de comunicación y el API de Java RMI.

Estos protocolos son bastantes complicados, pero afortunadamente no tenemos que preocuparnos de ellos, ya que son gestionados automáticamente por el servidor de

aplicaciones. El programador de beans y aplicaciones sólo ve la clase del bean y su interfaz remoto, quedándose los detalles de la comunicación de red ocultos para el programador.

En definitiva, la portabilidad y la reutilización de componentes para el desarrollo de aplicaciones distribuidas en múltiples infraestructuras de servicio, son uno de los principales valores que la tecnología EJB pone encima de la mesa. La portabilidad asegura que un bean desarrollado para un determinado contenedor o servidor de aplicaciones, puede migrarse a otro, si ese otro, ofrece mejor rendimiento, características o ahorros.

Además de la portabilidad y la reutilización, la simplicidad del modelo de programación EJB hace que éste sea muy valioso par el desarrollo de aplicaciones en entorno distribuido. Debido a que el contenedor tiene cuidado de gestionar las complejas tareas como la seguridad, las transacciones, la persistencia, la concurrencia y el control de recursos, el desarrollador de beans se encuentra libre para enfocar su atención en las reglas del negocio y en un modelo de programación muy sencillo. Esto quiere decir que los beans pueden desarrollarse rápidamente sin necesidad de profundos conocimientos en objetos distribuidos, transacciones y otros sistemas de empresa.

3.2.2. Roles en la tecnología EJB

La especificación de EJB define una serie de distintos roles de forma clara, habilitando así que determinadas tareas del desarrollo sean ejecutadas por expertos en sus áreas sin que se pierda interoperabilidad. El desarrollo de aplicaciones con la tecnología de EJB introduce una serie de actores y roles que se pueden apreciar en la siguiente figura:

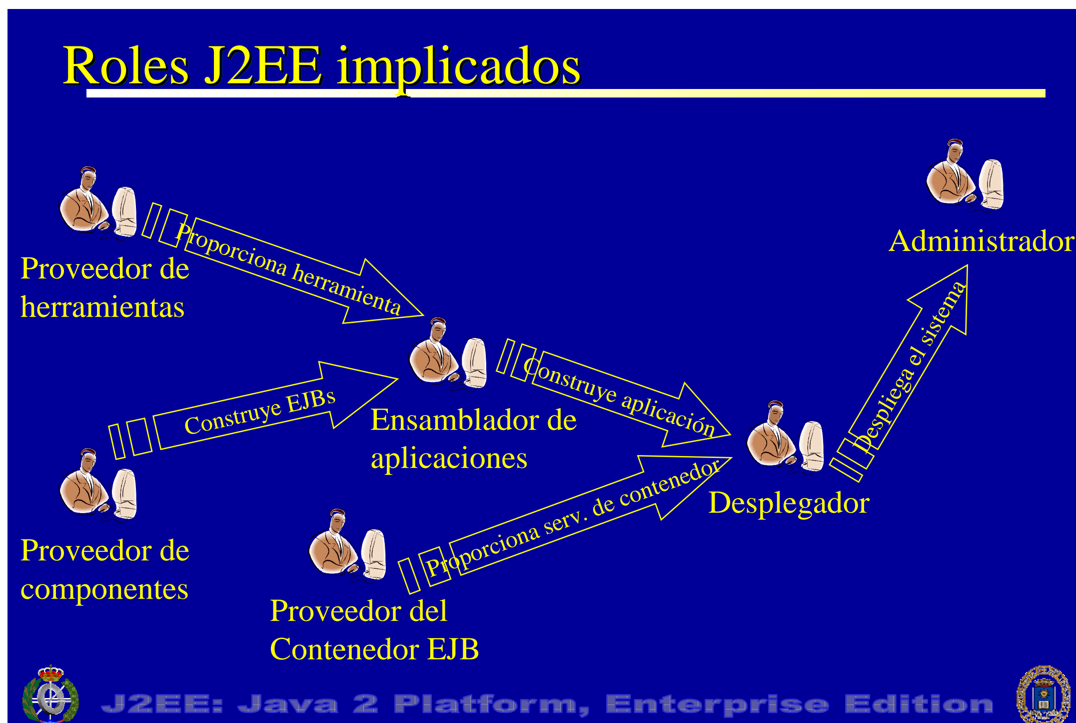


Figura 2: Roles implicados en la especificación EJB

Por lo tanto, para conseguir que una aplicación EJB se despliegue y funcione correctamente no solo es necesario disponer de un servidor de aplicaciones y de los componentes apropiados. Por el contrario, se requiere que todos los roles vistos en la figura anterior, realicen el trabajo que se les exige, del que se suponen son expertos, y que colaboren para alcanzar el objetivo que se persigue. A continuación se profundiza más en cada uno de ellos.

3.2.2.1. Proveedor de beans

El proveedor de beans proporciona los componentes de la lógica de negocio o Enterprise JavaBeans. Los beans no son aplicaciones completas pero si que son componentes desplegables que pueden ser ensamblados dentro de soluciones mas completas. Los Enterprise JavaBeans pueden ser comprados a proveedores externos, de los que en la actualidad hay una lista enorme en el mercado, o ser componentes reutilizables desarrollados y adquiridos a través de departamentos internos de la propia empresa.

3.2.2.2. Ensamblador de aplicaciones

Es el arquitecto de la aplicación. Es el encargado de entender como varios componentes pueden trabajar juntos y de escribir las aplicaciones que permiten combinarlos de forma adecuada. El ensamblador de aplicaciones se puede entender como un consumidor de los beans suministrados por los proveedores de beans, aunque, en realidad, él mismo puede haber sido el autor de alguno de esos componentes.

Las tareas típicas de un ensamblador de aplicaciones pueden ser algunas de las siguientes:

1. Determinar el plan de ensamblaje de la aplicación. Es decir, desde el conocimiento profundo de la lógica de negocio de la aplicación que se pretende desarrollar, decidir cual es la mejor combinación para alcanzar la mejor solución. Esta combinación de componentes puede incluir tanto a beans de los que ya dispone su organización, como a los nuevos a comprar o desarrollar.
2. Proporcionar un interfaz de usuario a la aplicación que permita el uso de los componentes.
3. Escribir nuevos beans que solucionen problemas específicos de la lógica de negocio.
4. Escribir el código que llama a los componentes suministrados por los proveedores de beans.
5. Escribir el código que permite la integración y el mapeo de datos entre los componentes de distintos proveedores. En definitiva, si en una aplicación se integran distintos componentes de distintos proveedores, es posible que todos ellos no trabajen de forma correcta para solucionar un problema y que haya que ajustarlos.

3.2.2.3. Responsable del despliegue de EJB

Después de que el ensamblador de aplicaciones construya la aplicación, ésta debe ser desplegada en el entorno de ejecución apropiado para que pueda ser accedida por los

clientes. Algunas de las características que se pretenden incluir en este momento del desarrollo son:

1. Introducir mecanismos de seguridad y protección durante el despliegue, utilizando por ejemplo un firewall.
2. Integrar con un servidor LDAP para tenar listas de acceso.
3. Elegir el hardware que proporcione el suficiente rendimiento para el sistema que se va implantar.
4. Proporcionar el hardware redundante y el resto de recursos necesarios para asegurar fiabilidad y tolerancia a fallos.
5. Hacer los ajustes de rendimiento necesarios en el sistema.

Normalmente el ensamblador de aplicaciones, que suele ser un analista de sistemas o desarrollador, no dispone de los conocimientos suficientes para realizar este tipo de tareas. Por lo tanto, es mucho más conveniente delegar el trabajo de despliegue de una aplicación EJB en un experto en estas tareas de rendimiento y requisitos operacionales.

Los responsables del despliegue saben de cómo integrar beans en un servidor de aplicaciones y de cómo adaptar esos beans al entorno específico. Son libres de adaptar tanto los beans como los servidores a las características del entorno, modificando para ello los parámetros adecuados.

3.2.2.4. Administrador del sistema

Una vez que el despliegue esta hecho y funciona correctamente, el administrador del sistema se encarga de hacer el seguimiento de la estabilidad del sistema instalado. El administrador del sistema es el encargado del mantenimiento y monitorización del sistema desplegado y debe hacer uso de las herramientas que los servidores de aplicaciones proporcionan para la ello. Por ejemplo, algunos servidores de aplicaciones alertan a los administradores de sistemas cuando un error grave ha ocurrido y requiere atención inmediata.

3.2.2.5. Proveedor del contenedor y servidor de aplicaciones

El proveedor del contenedor proporciona el contenedor EJB que es el entorno de ejecución de los beans y que habitualmente es parte de propio servidor de aplicaciones.

Un servidor de aplicaciones proporciona servicios middleware y en el caso del contenedor EJB, esos servicios se los suministra a los beans que de hecho el propio contenedor gestiona. Como en cualquier servidor de aplicaciones, el contenedor EJB proporciona servicios comunes como la gestión de recursos, las conexiones de red, la persistencia y demás, evitando que el desarrollador se preocupe de algo más que de sus problemas de negocio.

Ejemplos de contenedores EJB son: BEA´s WebLogic, Borland´s AppServer, IBM´s WebSphere, iPlanet´s iPlanet Application Server, Macromedia´s JRun, Oracle´s Oracle 9i, Persistence´s PowerTier y el servidor de aplicaciones de código abierto JBoss Application Server que será el que se utilice para la ejecución de los ejemplos de este libro.

3.2.2.6. Proveedor de herramientas

Para facilitar todo el proceso de desarrollo de componentes se impone la existencia una forma estándar de construirlos, gestionarlos y mantenerlos. Con este fin surgen una serie de entornos integrados de desarrollo o Integrated Development Environment (IDE) que asisten en el desarrollo y depuración rápido de componentes. Ejemplos de esos entornos pueden ser IBM´s WebSphere Application Developer Studio o el Borland´s JBuilder.

Existen otro tipo de herramientas que permiten realizar el modelo de componentes en Unified Modeling Language (UML), realizar testing o simplemente herramientas de construcción de beans como Ant.

3.2.3. Contenedor de EJB

Como se ha comentado, los beans son componentes de software que se ejecutan en un entorno especial llamado un contenedor EJB. El contenedor contiene y maneja un bean de igual forma que el servidor web Java contiene un servlet o un servidor web contiene un applet Java. Un bean no puede funcionar fuera de un contenedor EJB.

La especificación EJB define un contrato bean-contenedor, que incluye sus mecanismos de interacción así como un estricto conjunto de reglas que describe cómo se comportarán los beans y sus contenedores en tiempo de ejecución. El contenedor EJB controla cada aspecto del bean en tiempo de ejecución, incluyendo accesos remotos al bean, seguridad, persistencia, transacciones, concurrencia, y accesos a un almacén de recursos. Por lo tanto, un contenedor de EJB además de proporcionar los servicios estándar de todo contenedor, proporciona servicios para la gestión de las transacciones, la aplicación de la persistencia, el acceso a las API de servicio y la comunicación según el estándar J2EE.

El contenedor aísla al bean de accesos directos por parte de aplicaciones cliente. Cuando una aplicación cliente invoca un método remoto de un bean, el contenedor primero intercepta la llamada para asegurar que la persistencia, las transacciones, y la seguridad son aplicadas apropiadamente a cada operación que el cliente realiza en el bean. El contenedor gestiona estos aspectos de forma automática, por eso el desarrollador no tiene que escribir este tipo de lógica dentro del propio código del bean. El desarrollador de beans puede enfocarse en encapsular las reglas del negocio, mientras el contenedor se ocupa de todo lo demás.

Los contenedores manejan muchos beans simultáneamente de igual forma que un Java WebServer maneja muchos servlets. Para reducir el consumo de memoria y de proceso, los contenedores almacenan los recursos y gestionan los ciclos de vida de todos los beans de forma muy cuidadosa. Cuando un bean no está siendo utilizado, un contenedor lo situará en un almacén para ser reutilizado por otros clientes, o posiblemente lo sacará de la memoria y sólo lo traerá de vuelta cuando sea necesario.

Como las aplicaciones cliente no tienen acceso directo a los beans (el contenedor trata con el cliente y el bean) la aplicación cliente se despreocupa completamente de las actividades de control de recursos del contenedor. Por ejemplo, un bean que no está en uso, podría ser eliminado de la memoria del servidor, mientras que su referencia

remota en el cliente permanece intacta. Cuando un cliente invoca a un método de la referencia remota, el contenedor simplemente re-activa el bean para servir la petición. La aplicación cliente se despreocupa de todo el proceso.

Un bean depende del contenedor para todo lo que necesite. Si un bean necesita acceder a una conexión JDBC o a otro bean, lo hace a través del contenedor; si un bean necesita acceder a la identidad de su llamador, obtiene una referencia a sí mismo, o accede a las propiedades a través e su contenedor.

3.2.3.1. El contenedor EJB como middleware implícito

Esta tarea de interceptador de peticiones que realiza el contenedor de EJB´s hace que éste pueda actuar de forma automática como un middleware implícito. Este rol del contenedor resulta muy útil para los desarrolladores ya que les simplifica enormemente la vida.

De esta forma un desarrollador puede crear componentes de forma rápida sin preocuparse de escribir, depurar o mantener código que llame a otras API´s intermedias. Es decir, que los desarrolladores se olvidan de tener que desarrollar su propio middleware (o sistema intermedio) con las ventajas que eso conlleva o se olvidan de escribir una API ad-hoc que les permita acceder a un middleware ya creado que se encarga de realizar los servicios (lo que comúnmente se conoce como middleware explícito).

Por lo tanto, el hecho de disponer de un contenedor que actúe de middleware implícito permite disponer de servicios como:

1. Gestión distribuida de transacciones. Las transacciones permiten que se realicen de forma robusta y determinista operaciones en un entorno distribuido haciendo que los beans estén configurados con los atributos apropiados. El servicio de transacciones se consigue a través de la Java Transaction API (JTA). El contenedor EJB debe encargarse de gestionar que pasa si dos

clientes acceden simultáneamente a una tupla de una base de datos o si la base de datos se cae.

2. Seguridad. La seguridad es una de las cuestiones más importantes en el despliegue de aplicaciones multi-capa. La J2SE aporta un robusto servicio de seguridad que permite despliegues seguros ante visitantes no deseados. El contenedor EJB debe gestionar el que solo usuarios con privilegios puedan realizar ciertas operaciones.

3. Gestión de recursos. El contenedor EJB debe gestionar el pool de recursos de los beans, como los threads o las conexiones a las bases de datos, para que sean utilizados por todos los clientes.

4. Ciclo de vida del componente. El contenedor EJB debe gestionar los objetos que viven dentro del servidor ya que necesitan ser creados y destruidos según sean accedidos por los clientes. Además, podría permitir la reutilización de instancias si fuera necesario.

5. Persistencia. La persistencia es un requisito evidente en cualquier aplicación que requiera almacenamiento permanente de información. El contenedor EJB debe ofrecer asistencia para el almacenamiento y recuperación de esa información desde la capa de almacenamiento subyacente.

6. Acceso remoto. Debido a que los beans no son accedidos directamente a través de la red por lo clientes, el contenedor EJB debe gestionar la lógica de conexión entre los clientes y los servidores a través de la red. El desarrollador no debe preocuparse de las peculiaridades de las conexiones por la red ya que el contenedor también ofrece este servicio.

7. Concurrencia. El contenedor EJB debe gestionar el que haya distintos clientes conectándose simultáneamente a un servidor. Para ello el contenedor instancia varias copias de un bean y le asigna un thread a cada una. Si hay varias llamadas a un método de un bean, el contenedor solo permite a un cliente

llamar a un bean a la vez, por lo que los otros clientes son redirigidos a otras instancias o forzados a esperar.

8. Transparencia de localización. Los clientes de los beans están totalmente desacoplados de la localización real de los componentes que están usando.

9. Auditoria. El contenedor EJB debe gestionar mecanismos de aviso que permitan saber si algo va mal en el sistema, proveyendo distintos mecanismos de monitoreo o log.

3.2.4. J2EE y JNDI: descubriendo componentes en su entorno

JNDI es una pieza clave de la especificación de la J2EE y en el caso particular de la tecnología EJB resulta fundamental. La clave de su éxito está en el desacoplamiento que brinda entre el código de los componentes y el entorno en el que esos mismos componentes se van a localizar. Por lo tanto, una de las mayores ventajas que ofrece la tecnología EJB es que las aplicaciones deben ser escritas una vez y se ejecutan en cualquier entorno. Si se despliega un bean en una máquina y luego se decide llevar ese bean a otra distinta, el código de ese bean no debe sufrir ningún cambio porque es transparente de localización.

La tecnología EJB consigue la transparencia de localización, y por lo tanto, la portabilidad de código mediante la definición de un entorno para los beans de la aplicación. Este entorno o Enterprise Naming Context (ENC) es utilizado por los distintos participantes o componentes de la aplicación distribuida para solucionar su independencia de localización.

Es decir que el desacople entre el código de un bean y el entorno en el que se despliega se consigue con la existencia de los sistemas de nombrado y directorio, que crearán un ENC para los componentes y al que se accederá mediante la utilización de la API JNDI. Es por tanto responsabilidad del contenedor el hacer disponible un ENC a los beans de la aplicación mediante un contexto JNDI que les sea accesible. Hay que decir que muchos servidores de nombres y directorios suelen estar incluidos dentro de

los servidores de aplicaciones, y por lo tanto, suelen estar controlados por el propio contenedor de EJB.

En definitiva los distintos componentes de la aplicación usan el ENC para las siguientes operaciones:

1. El proveedor de beans usa los descriptores de despliegue estándar de los beans para especificar toda la información y recursos que necesita el bean para ejecutar correctamente. Esta información se organiza, como se mostrará posteriormente cuando se vean dichos descriptores de despliegue mas en profundidad, mediante elementos ENC como pueden ser <ejb-name> o <ejb-ref>.

2. El contenedor de beans proporciona las herramientas que le permiten al encargado de hacer el despliegue de la aplicación, hacer el mapeo entre las referencias hechas por el proveedor de beans, mediante las referencias ENC, con los recursos que los satisfagan durante el despliegue.

3. El contenedor de beans utiliza la información de despliegue para construir el ENC completo de los componentes en tiempo de ejecución.

Los sistemas de nombrado son productos que permiten a las aplicaciones almacenar y posteriormente encontrar recursos a través de una red utilizando para ello código que en ningún momento depende de nombres de máquinas o localizaciones específicas.

Tradicionalmente las empresas han utilizado servicios de nombres y directorios para almacenar usuarios, claves, localización de máquinas o impresoras. EJB se apoya en esta idea y la explota para permitir el almacenamiento y la localización de los recursos que las aplicaciones a nivel empresarial usan. Si se decide posteriormente que los recursos deben ser localizados en otro sitios el código de las aplicaciones que usan estos recursos no debe ser modificado y recompilado porque los servicios de nombres pueden ser fácilmente actualizados para reflejar las nuevas localizaciones de los recursos. Estos recursos pueden ser desde objetos Home, pasando por conexiones

JDBC, hasta otros componentes distribuidos o propiedades específicas de esos componentes.

Esta transparencia de localización facilita enormemente el mantenimiento de sistemas que evolucionan a lo largo del tiempo, y además, se convierte en absolutamente necesario cuando se trabaja con componentes adquiridos a terceras partes, como pueden ser los Enterprise JavaBeans, ya que en estos casos es bastante probable que no se tenga acceso al código fuente de los mismos, por lo que resultaría imposible cambiar su código.

3.2.5. Fases de desarrollo[2] de un EJB

Cuando se desarrolla un componente EJB es conveniente seguir una serie de operaciones estándar que facilitan la tarea y que se irán explicando más detalladamente a lo largo de las siguientes secciones del libro. Dicho pasos se enumeran a continuación:

1. Escribir los ficheros .java que componen el bean: los interfaces del componente (locales o remotos), el interfaz home, la propia clase del bean y cualquier clase helper que sea necesaria.
2. Escribir los descriptores de despliegue.
3. Compilar los ficheros .java del primer punto en ficheros .class.
4. Usar la utilidades jar para crear un fichero Ejb-jar que contenga el descriptor de despliegue y los ficheros .class.
5. Realizar el despliegue del fichero Ejb-jar en un contenedor de la forma que el vendedor aconseje. Por ejemplo, utilizando las herramientas específicas que ese vendedor proporciona o simplemente copiando el fichero Ejb-jar en la carpeta donde el contenedor mira para cargar esos ficheros.
6. Configurar de forma apropiada el servidor EJB que se este utilizando para que el despliegue del fichero Ejb-jar sea el apropiado. Se deberían configurar apropiadamente cosas como las conexiones a las bases de datos o los pool de threads. Este paso es dependiente del vendedor y debería llevarse a cabo mediante consolas Web o editando directamente los ficheros de configuración.

[2] En la terminología de Enterprise JavaBeans el proceso de instalación es llamado "desarrollo".

7. Arrancar el contenedor de EJB y confirmar que el fichero Ejb-jar que se ha desplegado ha sido correctamente cargado.
8. Opcionalmente se puede escribir una aplicación de test que actúe como cliente EJB en un fichero .java. Esta clase se compilará en un fichero .class y se podrá ejecutar desde la línea de comandos.

3.2.6. Estructura de un EJB: la parte servidora

Una vez vistas las fases de desarrollo y el contenedor de EJB como interceptador de peticiones de los clientes resulta imprescindible adentrarnos en lo que realmente constituye a un Enterprise JavaBean. Como hemos visto anteriormente un Enterprise JavaBean no es un simple fichero monolítico. Por el contrario, un bean es un conjunto de ficheros, tanto del lado servidor como del lado cliente, que se estudiarán a continuación y que trabajarán de forma conjunta para cumplir su cometido.

En esta sección se tratará la estructura de los beans desde la perspectiva del lado servidor. Tal y como se aprecia en la figura xx, los ficheros que forman un EJB del lado servidor son los siguientes:

- Interfaz Remoto o Remote Interface
- Interfaz Home o Home Interface
- Clase del Bean o Enterprise Bean Class
- Local Inteface y Local Home Interface: solo en el caso de beans en el mismo proceso.
- Descriptor de Despliegue o Deployment Descriptor
- Ficheros de Despliegue Especificos o Vendor-specific Files

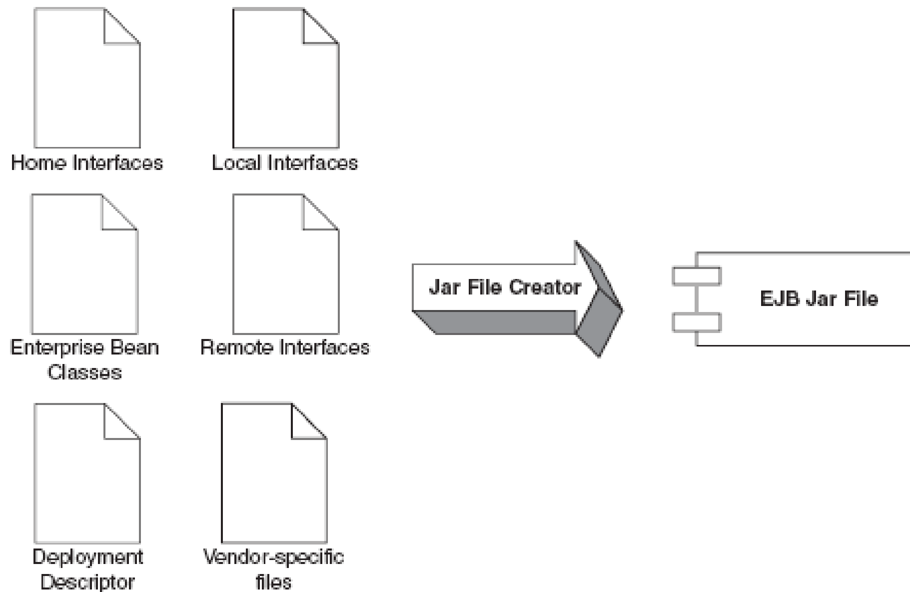


Figura 3: Composición del componente EJB

Para crear un componente EJB del lado del servidor, un desarrollador de beans debe proporcionar dos interfaces que definen los métodos de negocio del bean, además de la implementación real de la clase bean que será llamada clase del bean y que será instanciada en tiempo de ejecución, convirtiéndose en un objeto distribuido.

Los interfaces nombrados anteriormente son los interfaces Home y Remote que representan al bean. Por un lado, el cliente usará el interfaz Home del bean, que será público, para crear, manipular, y eliminar beans del servidor EJB. Sin embrago, el contenedor aísla a los beans de accesos directos desde aplicaciones cliente, de forma que cada vez que un bean es solicitado, creado, o borrado, el contenedor es quien gestiona todo el proceso. Y por otro lado, el cliente usará el interfaz Remote del bean para definir la propia lógica de negocio del bean.

Las siguientes secciones entrarán más en detalle en cada una de las partes que componen un componente EJB. Además, para facilitar su estudio, se ilustrarán cada una de esas partes con código de ejemplo. Finalmente hay que mencionar, que dicho código de ejemplo se utilizará más adelante para ver ejecutando el primer ejemplo práctico de componentes con la arquitectura EJB, basado en el ya mítico ejemplo del Hola Mundo.

3.2.6.1. Clase del bean

La primera parte de cualquier bean es la propia implementación del mismo que contiene toda la lógica y la funcionalidad que realmente aporta el componente. Es lo que se denomina clase del bean y simplemente será una clase Java que cumple unas ciertas reglas de acuerdo a un interfaz bien definido. Estas reglas son necesarias y son las que garantizan que cualquier bean desarrollado puede ejecutar en cualquier contenedor de EJB.

La clase del bean contiene los detalles de implementación del componente y a pesar de que no hay reglas escritas y estrictas al respecto, esa implementación puede variar dependiendo del tipo de bean que estemos desarrollando. Por ejemplo, un bean de sesión contendrá en su implementación una lógica relacionada con el proceso de negocio, como puede ser la realización de transferencias entre cuentas. Por su parte, un bean de entidad contendrá en su implementación una lógica más relacionada con los datos, como puede ser el incremento del saldo en una cuenta.

La especificación 2.1 de la API EJB define una serie de interfaces que la clase del bean puede implementar. Todos estos interfaces obligan a la clase del bean a implementar ciertos métodos que todos los beans deben poseer para seguir la especificación de componentes EJB. De esta forma se garantiza que el bean posee todos los métodos que el contenedor de EJB, por ejemplo, necesita para gestionar el ciclo de vida del bean.

El interfaz más básico que todos los beans deben implementar es el javax.ejb.EnterpriseBean que convierte la clase del bean en una clase enterprise bean y que como aspecto más importante señala a la clase del bean como serializable, tal y como se indica a continuación:

```
public interface javax.ejb.EnterpriseBean extends java.io.Serializable
{

}
```

El hecho de que un bean sea serializable quiere decir que cualquier bean puede ser convertido a una ristra de bits, característica fundamental en la programación distribuida a través de una red, tal y como se comento en el apartado donde se trató la serialización.

Finalmente, hay que mencionar que la especificación de EJB define una serie de interfaces más específicos que extienden el interfaz javax.ejb.EnterpriseBean y que se deberán emplear dependiendo del tipo de bean que se quiera desarrollar. De hecho en la práctica la clase del bean nunca implementa el interfaz javax.ejb.EnterpriseBean sino que implementa alguno de esos interfaces más específicos. Así pues, los beans de sesión deberán implementar el interfaz javax.ejb.SessionBean, los beans de entidad el interfaz javax.ejb.EntityBean o los beans manejados por mensajes el interfaz javax.ejb.MessageDrivenBean.

A continuación se muestran los métodos definidos en el interfaz javax.ejb.SessionBean y una descripción de los mismos.

```
public interface javax.ejb.SessionBean extends java.io.EnterpriseBean
{
        public void ejbActivate() throws EJBException, RemoteException;

        public void ejbPassivate () throws EJBException, RemoteException;

        public void ejbRemove () throws EJBException, RemoteException;

        public void setSessionContext (SessionContext ctx)throws
                                          EJBException, RemoteException;

}
```

- public void ejbActivate( ) throws EJBException, RemoteException;

  Este método es llamado cuando la instancia del bean pasa de un estado pasivo a un estado activo. En este momento la instancia del bean podrá volver a disponer de aquellos recursos que fueron liberados cuando el bean estaba en estado pasivo. Este método solo es llamado por el contenedor y para contextos que no tienen transacciones.

- public void ejbPassivate ( ) throws EJBException, RemoteException;

  Este método es llamado cuando la instancia del bean pasa de un estado activo a un estado pasivo. En este momento la instancia deberá liberar todos los recursos de los que dispone y que no recuperará hasta que no vuelva a un estado activo. Después de terminar la ejecución de este método, la instancia del bean debe estar en un estado que le permita al contenedor usar la serialización para mantener de esta forma el estado de la instancia del bean. Este método solo es llamado en contextos sin transacciones.

- public void ejbRemove ( ) throws EJBException, RemoteException;

  El contenedor llama a este método antes de que termine el ciclo de vida del bean de sesión. La llamada a este método puede darse, bien por que el cliente invoca explícitamente el método de eliminación del bean, o bien, por que el propio contenedor decide que es momento de eliminar el bean después de que se cumpla su time-out.

- public void setSessionContext (SessionContext ctx) throws EJBException, RemoteException;

  Este método asigna al bean el contexto de sesión que se le pasa como argumento. El contenedor llama a este método después de crear la instancia del bean ya que esta instancia guarda una referencia a ese contexto en una variable de instancia. Este método, al igual que los dos primeros, solo es llamado en contextos sin transacciones.

3.2.6.2. EL objeto EJB

Los beans de la tecnología EJB no pueden considerarse como objetos remotos puros. Es decir, cuando un cliente quiere usar una instancia de un bean, este cliente nunca invoca el método remoto directamente sobre una instancia del bean. En su lugar esa

invocación es interceptada, tal y como ya se ha comentado anteriormente, por el contenedor de EJB y delegada entonces por el contenedor en la instancia del bean.

En realidad, las capacidades del contenedor EJB como nivel de in-dirección entre el cliente y el bean se concretan en un objeto llamado Objeto Ejb. Es realmente este objeto quien intercepta la petición y quien conoce de conexiones de red, de transacciones, de seguridad y mucho más. Es un objeto inteligente que sabe como realizar las operaciones lógicas que el contenedor de EJB requiere antes de llamar a los métodos que ofrece la clase del bean. Y es este mismo objeto quien finalmente delegar las peticiones de los clientes en los propios beans.

Un objeto EJB debe considerarse físicamente como una parte integrante del contenedor de EJB´s. Todo objeto EJB publica todos los métodos de la lógica de negocio que ofrece el propio bean y además, tiene código específico del contenedor en el que va a ser desplegado.

Por lo tanto, el proveedor de los beans no debe preocuparse de cómo trabaja cada contenedor. La generación del código Java de la clase del Objeto Ejb será automático y responsabilidad de las herramientas que proporciona para tal efecto el proveedor del contenedor. Estas herramientas están diseñadas para integrar beans dentro de un contenedor y por lo tanto generarán el código Java necesario (los helpers, los stubs, los skeletons, las clases de accesos a datos y otras necesarias y específicas del contenedor) que convertirán al bean en un componente distribuido y totalmente ajustado a las necesidades de seguridad, transacciones, y el resto de servicios que de él se esperan.

3.2.6.2.1. Interfaz Remoto

Como se ha dicho antes el Objeto Ejb clona y publica todos los métodos de la lógica de negocio que tiene la clase del bean. Por lo tanto, ahora cabe preguntarse como conocen las herramientas que generan el código del Objecto Ejb los métodos que se deben clonar. Pues bien, la respuesta está en un interfaz especial que el proveedor de beans debe proporcionar: el interfaz remoto.

El interfaz remoto duplica los métodos de la lógica de negocio, que son aquellos métodos específicos del negocio que representa el bean. Estos métodos pueden ser modificadores como, por ejemplo, para cambiar el nombre de un cliente. O también pueden realizar tareas de la lógica de negocio como decrementar el saldo de una cuenta corriente diseñada como un bean.

Los interfaces remotos deben seguir ciertas normas impuestas por la especificación de EJB. El interfaz remoto del bean debe extender del interfaz javax.ejb.EJBObject que a su vez es una subclase del interfaz java.rmi.Remote. Por lo tanto, la labor del desarrollador de beans es la de proporcionar un interfaz remoto que extienda el interfaz javax.ejb.EJBObject, que se muestra en la figura XXXX, y que incluya los métodos de negocio clonados del bean.

A continuación se muestran los métodos definidos en el interfaz javax.ejb.EJBObject y una descripción de los mismos.

```
public interface javax.ejb.EJBObject extends java.rmi.Remote
{

  public javax.ejb.EJBHome getEJBHome() throws java.rmi.RemoteException;

  public java.lang.Object getPrimaryKey() throws java.rmi.RemoteException;

  public void remove() throws java.rmi.RemoteException,
                                         javax.ejb.RemoveException;

  public javax.ejb.Handle getHandle() throws java.rmi.RemoteException;

  public boolean isIdentical(javax.ejb.EJBObject obj) throws
                                          java.rmi.RemoteException;

}
```

- public javax.ejb.EJBHome getEJBHome() throws java.rmi.RemoteException;

  Este método devuelve una referencia al interfaz home del bean. Es decir, devuelve una referencia al interfaz que define los métodos de creación, destrucción, búsqueda del bean.

- public java.lang.Object getPrimaryKey () throws java.rmi.RemoteException;

Este método devuelve la clave primaria del bean. Este método solo puede ser llamado para beans de entidad. Una llamada a este método para beans de sesión producirá una RemoteException.

- public void remove() throws java.rmi.RemoteException, java.rmi.RemoveException;

  Este método elimina un Objeto Ejb y lanza una java.rmi.RemoteException si se produce un fallo a nivel de sistema como un error de conexión o una java.rmi.RemoveException si el bean o el propio contenedor no permiten el borrado del objeto.

- public javax.ejb.Handle getHandle() throws java.rmi.RemoteException;

  Este método devuelve un manejador para le objeto Ejb. Este manejador se puede usar posteriormente para volver a conseguir una referencia al objeto Ejb incluso en una máquina virtual distinta.

- public boolean isIdentical (javax.ejb.EJBObject obj) throws java.rmi.RemoteException;

  Este método verifica si el objeto EJB pasado como argumento es idéntico al objeto desde el que se invoca el método.

Una vez vistos los métodos del interfaz del que tiene que extender el interfaz remoto, hay que mencionar que la implementación de los métodos definidos en ese interfaz remoto reside en dos sitios distintos que es importante tener muy claro. Por un lado, la implementación de los métodos del interfaz javax.ejb.EJBObject, vistos anteriormente, estará en el Objeto Ejb, que hay que recordar, que es una clase que no implementa el desarrollador de beans sino que es trabajo de las herramientas propietarias del contenedor. Y por otro lado, la implementación de los métodos duplicados del bean residirá en la propia implementación del bean, es decir en la clase del bean. De esta forma, cuando un cliente llama a uno de los métodos de la lógica de

negocio, el Objeto Ejb (o el contenedor) que no tiene esa implementación, delega la petición en la implementación del bean.

Finalmente, hay que destacar que al extender el interfaz remoto de javax.ejb.EJBObject, que a su vez extiende del interfaz java.rmi.Remote, nos encontramos en el mundo de RMI-IIOP. Podemos considerar al Objeto Ejb, que implementa indirectamente el interfaz java.rmi.Remote, como un objeto totalmente compatible con RMI-IIOP, ya que java.rmi.Remote es el interfaz que deben implementar los objetos remotos. Por esta razón, cualquier método del interfaz remoto lanza RemoteException, o una subclase suya, y cualquier paso de parámetros debe ser compatible con RMI-IIOP. No cualquier cosa puede ser pasada como parámetro en una llamada remota a través de una red. Los parámetros que se pasan en los métodos definidos en el interfaz remoto deben ser tipos primitivos, objetos serializables y objetos RMI-IIOP.

3.2.6.3. El objeto Home

Como hemos visto los clientes interactúan con los Objetos Ejb por medio del contenedor y nunca tratan directamente con los beans. Por lo tanto, vuelve a ser evidente plantearse como conocen los clientes las referencias o localización de los Objetos Ejb.

Un cliente no puede instanciar un Objeto Ejb directamente porque ese objeto puede residir en un espacio de direcciones distinto y además, la especificación de EJB promueve la transparencia de localización, por lo que los clientes nunca deberían preocuparse por donde residen los Objetos Ejb.

Para conseguir la referencia a un objeto Ejb el código de los clientes pregunta a una fábrica de Objetos Ejb. Esta fábrica será la responsable de la creación, destrucción y búsqueda de los Objetos Ejb. Dicha fábrica es conocida en la especificación de EJB como Objeto Home y sus principales cometidos consisten en la creación y destrucción de los Objetos Ejb y de encontrar Objetos Ejb que estén ya instanciados en el contenedor para aprovecharlos.

Al igual que los Objetos Ejb, los Objetos Home son propietarios del contenedor EJB y contienen lógica específica del contenedor, por ejemplo, para el balanceo de carga o para consolas gráficas de control administrativo. Por lo tanto, estos objetos también pueden considerarse físicamente como partes integrantes del contenedor EJB al igual que los Objetos Ejb.

3.2.6.3.1. Interfaz Home

Hemos visto que los Objetos Home son fábricas de Objetos Ejb, pero para poder crear un Objeto Ejb puede resultar necesario saber como se quiere crear ese objeto. Es decir, el contenedor, o el Objeto Home más exactamente, necesitan saber, por ejemplo, los parámetros de inicialización que necesita el Objeto Ejb.

Por lo tanto esa información de creación de Objetos Ejb se suministra al contenedor mediante el interfaz home. El interfaz home simplemente define los métodos de ciclo de vida para crear, destruir o localizar beans. Estos comportamientos de ciclo de vida están separados del interfaz remoto porque los comportamientos que representa no son específicos para un sólo objeto bean. Obviamente será el Objeto Home del contenedor quien implementa los métodos definidos en el interfaz home.

Como es habitual la especificación de EJB define una serie de requisitos que los interfaces home deben cumplir. El interfaz home extiende el interfaz javax.ejb.EJBHome que a su vez extiende el interfaz java.rmi.Remote. Por lo tanto, la labor del desarrollador de beans es la de proporcionar un interfaz home que extienda el interfaz javax.ejb.EJBHome y que incluya los métodos de inicialización del Objeto Ejb.

A continuación se muestran los métodos definidos en el interfaz javax.ejb.EJBHome y una descripción de los mismos.

```
public interface javax.ejb.EJBHome extends java.rmi.Remote
{

   public EJBMetaData getEJBMetaData()throws java.rmi.RemoteException;

   public javax.ejb.HomeHandle getHomeHandle()throws
                                         java.rmi.RemoteException;
```

```
    public void remove(javax.ejb.Handle handle) throws
                java.rmi.RemoteException, javax.ejb.RemoveException;

    public void remove(Object primaryKey) throws java.rmi.RemoteException,
                                           javax.ejb.RemoveException;

}
```

- public EJBMetaData getEJBMetaData()throws java.rmi.RemoteException;

  Este método devuelve el interfaz EJBMetaData relacionado con el bean y que permite a los clientes conseguir información sobre el propio bean. La información que se obtiene vía EJBMetaData está pensada para ser usada por las herramientas de desarrollo usadas en la construcción de aplicaciones que utilizan Enterprise JavaBeans.

- public javax.ejb.HomeHandle getHomeHandle ( ) throws java.rmi.RemoteException;

  Este método devuelve un manejador para le objeto Home. Este manejador se puede usar posteriormente para volver a conseguir una referencia al objeto Home incluso en una máquina virtual distinta.

- public remove (javax.ejb.Handle handle) throws java.rmi.RemoteException, java.rmi.RemoveException;

  Este método elimina un Objeto Ejb identificado mediante su manejador que es pasado como argumento al método. Además lanza una java.rmi.RemoteException si se produce un fallo a nivel de sistema como un error de conexión o una java.rmi.RemoveException si el bean o el propio contenedor no permiten el borrado del objeto.

- public remove (Object primaryKey) throws java.rmi.RemoteException, java.rmi.RemoveException;

Este método, al igual que el anterior elimina un Objeto Ejb pero identificándolo ahora mediante su clave primaria que también es pasada como argumento al método. Este método solo puede ser llamado para beans de entidad. Una llamada a este método para beans de sesión producirá una excepción del tipo RemoveException.

Finalmente, hay que destacar que al extender el interfaz remoto de javax.ejb.EJBHome, que a su vez extiende del interfaz java.rmi.Remote, nos encontramos de nuevo en el mundo de RMI-IIOP. Por esta razón, cualquier método del interfaz home lanza RemoteException, o una subclase suya, y cualquier paso de parámetros debe ser compatible con RMI-IIOP.

3.2.6.3.2. Interfaz Local

Uno de los problemas fundamentales de la utilización de los interfaces remotos es que la creación y la llamada de los beans creados a través de ellos son bastante lentas. Para tener una idea mas clara del proceso, se enseñan a continuación los pasos que suceden en una llamada a un Objeto Ejb desde un cliente:

1. El cliente llama al stub local.
2. EL stub empaqueta los parámetros de forma apropiada para mandarlos pro la red.
3. El stub establece una conexión a través de la red con el skeleton.
4. El skeleton desempaqueta los parámetros de forma apropiada para el lenguaje Java.
5. El skeleton llama al Objeto Ejb
6. El Objeto Ejb realiza las operaciones típicas del middleware implícito.
7. El Objeto Ejb llama a la instancia de la clase del bean para que haga su trabajo.
8. Una vez el bean ha terminado su trabajo se deshacen los pasos en el camino de vuelta.

Como se puede apreciar es un trabajo largo y pesado que tiene sentido y es necesario si las llamadas son a métodos remotos. Pero cabe preguntarse ahora si todo ese trabajo

no podría reducirse cuando estemos seguros que no se van a producir llamadas remotas.

Pues bien, desde la especificación 2.0 de EJB se ha introducido la figura del interfaz local. Con los interfaces locales se puede llamar a Enterprise JavaBeans de una forma eficiente y rápida a través de los Objetos Locales en vez de los Objetos Ejb. Los Objetos Locales implementarán un interfaz local en vez de un interfaz remoto y los Objetos Local Home implementarán un interfaz local home en vez de un interfaz home.

Con esta solución se evitan todos los pasos relacionados con los stubs, los skeletons, las conexiones a través de la red y el empaquetamiento y des-empaquetamiento. El proceso ahora sería:

1. El cliente llama a Objeto Local.
2. El Objeto Local lleva a cabo las tareas del middleware.
3. El Objeto Local llama a la instancia de la clase del bean para que haga su trabajo.
4. Se retorna el control a Objeto Local para que este finalmente devuelva el control al cliente.

Como es habitual la especificación de EJB define una serie de requisitos que los interfaces locales deben cumplir. El interfaz local extiende el interfaz javax.ejb.LocalObject, mientras que el interfaz local home extiende el interfaz javax.ejb.EJBLocalHome. Hay que destacar que en ambos casos, tanto javax.ejb.LocalObject, como javax.ejb.EJBLocalHome no extienden el interfaz java.rmi.Remote y por eso no se permiten llamadas remotas. Ambos interfaces definen una serie de métodos que se muestran a continuación.

```
public interface javax.ejb.EJBLocalObject {

        public javax.ejb.EJBLocalHome getEJBLocalHome() throws
                                  javax.ejb.EJBException;

        public Object getPrimaryKey() throws javax.ejb.EJBException;

        public boolean isIdentical(javax.ejb.EJBLocalObject) throws
```

```
                                          javax.ejb.EJBException;

        public void remove() throws javax.ejb.RemoveException,
                                          javax.ejb.EJBException;
}


public interface javax.ejb.EJBLocalHome {

        public void remove(java.lang.Object) throws
                         javax.ejb.RemoveException, javax.ejb.EJBException;
}
```

De esta forma se pueden desarrollar beans más pequeños y con un grano más fino en sus tareas. Sin embargo, hay que destacar que estos interfaces son completamente opcionales y se pueden utilizar como reemplazo o complemento a los interfaces remotos siempre y cuando se tenga claro que este tipo de interfaces presentan las siguientes limitaciones:

1. Solo funcionan cuando se realizan llamadas entre beans dentro del mismo proceso y no se pueden hacer llamadas a métodos remotos. Si se quiere modificar el comportamiento de un bean que a priori se basaba en interfaces locales para ser usado de forma remota, es necesario cambiar el código del bean para convertirlo en remoto.

2. El paso de parámetros con interfaces locales se realiza por referencia en lugar de por valor ya que los beans residen en el mismo espacio de direcciones. Este detalle que obviamente acelera el proceso de llamada, ya que los parámetros no requieren ser copiados, si que cambia la filosofía de codificación de los desarrolladores.

3.2.6.4. Descriptores de despliegue

Un desarrollador de beans debe utilizar los ficheros descriptores de despliegue para declarar e informar al contenedor de EJB (y más específicamente al Objeto Ejb que es quien realiza las tareas de middleware implícito) de los servicios que debe proporcionar el contenedor. Por ejemplo, se puede se puede utilizar un fichero

descriptor de despliegue para declarar como el contenedor debería gestionar el ciclo de vida, la persistencia, el control de transacciones o la seguridad de un bean. El contenedor inspeccionará el fichero descriptor de despliegue para encontrar los requisitos que el desarrollador requiera para su bean.

A continuación se presentan distintos ejemplos de requisitos que puede pedir un desarrollador para un bean y que por lo tanto, que debe declarar en un fichero descriptor de despliegue.

- Requisitos de gestión y ciclo de vida del bean. Este parámetro del descriptor de despliegue indicará como el contenedor debe gestionar los beans. Por ejemplo, especifica el nombre de la clase del bean, el tipo del bean o los interfaces que forman parte de el.

- Requisitos de persistencia (solo para los beans de entidad). Se usará el fichero descriptor de despliegue para informarle al contenedor si el bean gestionará su propia persistencia o si delegará esta tarea en el propio contenedor donde el bean será desplegado.

- Requisitos de transacciones. Hay parámetros en el descriptor de despliegue que permiten especificar como se ejecuta una transacción. Estos parámetros pueden determinar si una transacción empieza siempre que un cliente llama un bean o si debe finalizar justo después de que termine la llamada a un método.

- Requisitos de seguridad. Los ficheros descriptores de despliegue pueden contener entradas de control de acceso que usarán, tanto los beans como el contenedor, para restringir el acceso a ciertas operaciones. Por ejemplo, se puede especificar quien puede utilizar que bean e incluso quien puede utilizar que métodos de un determinado bean. Por otro lado, se pueden especificar roles de seguridad que serán asociados al propio bean y que de esta forma faciliten llevar a cabo las operaciones de seguridad. Por ejemplo, un bean como una cuenta corriente solo puede ser creada por un bean con el rol financiero.

Al igual que en las versiones anteriores, la especificación de EJB 2.1 define el fichero descriptor de despliegue como un fichero XML. Estos ficheros se pueden escribir, bien a mano, con la ayuda de un entorno de desarrollo integrado (Integrated Development Environment o IDE) o mediante herramientas que proporcione el propio proveedor del contenedor EJB.

Finalmente, hay que decir que la creación del descriptor de despliegue es tarea del desarrollador de beans. Sin embargo, otros participantes pueden modificar el contenido de ese fichero agregando los parámetros que considere necesarios para llevar a cabo su tarea. Por ejemplo, el responsable del despliegue de la aplicación puede modificar el fichero de despliegue en el momento de instalar los beans en el contenedor. Esto es posible porque estos ficheros únicamente declaran como los beans deben usar los servicios del middleware en lugar de tener que escribir el código que llame a ese middleware.

En la tecnología de EJB siempre se van a utilizar componentes desarrollados por terceras partes y normalmente no tendremos acceso a su código fuente por lo que resulta imprescindible tener mecanismos de despliegue adaptables sin necesidad de modificar código. Por lo tanto, los descriptores de despliegue son uno de los principales aportes de la arquitectura EJB y dejan claro las ventajas de declarar frente a programar.

Ejemplos de ficheros descriptores de despliegue son los ficheros ejb-jar.xml y web.xml, dependiendo del tipo de cliente al que den servicio los componentes EJB y que se verán a continuación.

3.2.6.4.1. Fichero ejb-jar-xml

La especificación de EJB define el fichero descriptor de despliegue como una colección de componentes EJB y los atributos que necesita cada uno de ellos para ejecutar. Es decir, la descripción del ejb.jar.xml es una vista lógica del entorno que necesitan los beans para funcionar correctamente. De esta forma un desarrollador de componentes, que normalmente no conoce el entorno dentro del que su EJB puede ser desplegado, se limita a describir sus componentes software de una forma

independiente del entorno usando, por ejemplo, nombres lógicos. Será por tanto responsabilidad de quien despliega la aplicación, el enlazar esos nombres lógicos puestos por el desarrollador con los correspondientes recursos durante la fase de despliegue.

La siguiente figura muestra una visión grafica del DTD del fichero descriptor de despliegue ejb-jar.xml. El DTD entero del fichero ejb-jar.xml está disponible en la página web de Sun en http://java.sun.com/dtd/ejb-jar_2_0.dtd.

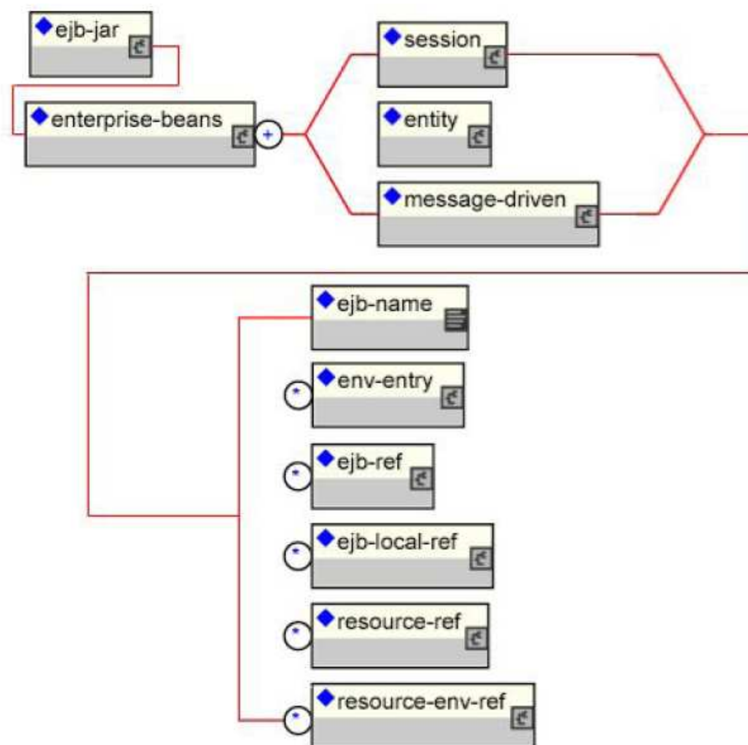


Figura 4: Estructura del fichero descriptor de despliegue ejb-jar.xml

3.2.6.4.2. Fichero web.xml

Como se ha comentado anteriormente las aplicaciones cliente de los componentes EJB pueden ser tanto aplicaciones *stand-alone* como aplicaciones en entorno *web*. Por lo tanto, para aplicaciones en entorno *web* resulta imprescindible disponer de un mecanismo que permita definir, de forma no programática y si declarativa (es decir, desacoplada del código fuente), cuales son los recursos de los que va a hacer uso dicha aplicación. Es decir, un fichero que declare qué componentes o beans, para el

caso particular de la tecnología EJB, colaborarán para llevar a cabo las funcionalidades que se esperan del sistema.

Para conseguir este objetivo, la especificación EJB aprovecha la existencia de un fichero XML que debe estar presente en aplicaciones en entorno *web* cuando se utilizan tecnologías como Servlets. Este fichero es el descriptor de despliegue de Servlets 2.3, llamado web.xml, que además de definir toda la información que esperan las tecnologías *web* de J2EE, se aprovecha ahora para describir una colección de componentes y el entorno de ejecución que necesitarán las aplicaciones *web* para funcionar.

La siguiente figura muestra una visión grafica del DTD del fichero descriptor de despliegue web.xml. El DTD entero del fichero web.xml está disponible en la página web de Sun en http://java.sun.com/dtd/web-app_2_3.dtd.

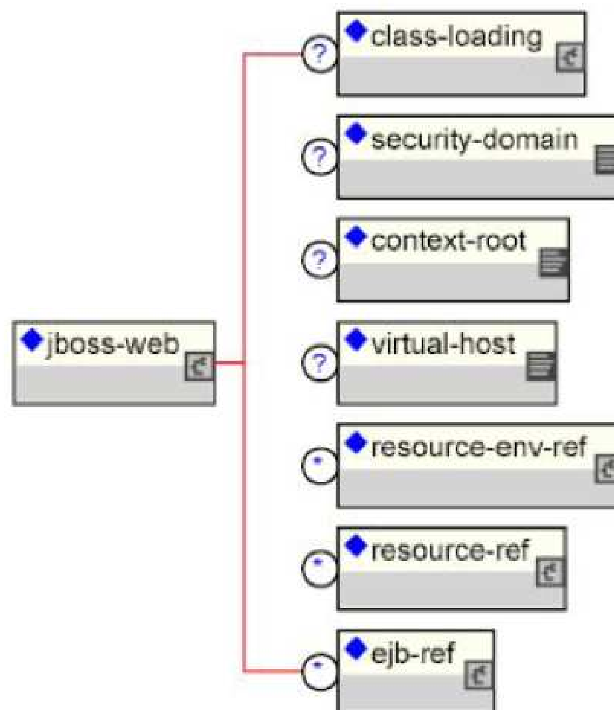


Figura 5: Estructura del fichero descriptor de despliegue web.xml

3.2.6.5. Fichero específicos del proveedor

Como se ha visto al inicio de este capítulo cuando se estudiaron los roles involucrados en el desarrollo de un componente EJB, por un lado existe la figura del proveedor de beans y por otro, la del proveedor del contenedor o servidor de aplicaciones. Los proveedores del servidor de aplicaciones o proveedores del servidor EJB, son todos distintos y cada uno de ellos, obviamente, tiene sus propias características y aportará su valor añadido.

Como es lógico, al brindar cada proveedor unos servicios independientes del resto, la especificación de la tecnología EJB no entra en las funcionalidades propias que aporta cada uno. Ejemplos de esas funcionalidades pueden ser la configuración del balanceo de carga o la monitorización de la actividad del servidor de aplicaciones. Por lo tanto, resulta lógico pensar que cada servidor de aplicaciones requiere incluir una serie de ficheros específicos, en formato XML, de texto o binarios, en los se de soporte a su funcionalidad propia.

Al ser estos ficheros específicos del servidor de aplicaciones que se emplee, no se estudiarán en esta sección. Sin embargo, cuando en el siguiente capítulo se vea un caso real de componentes EJB, con un servidor de aplicaciones real, si que se entrará en el detalle de estos ficheros.

3.2.6.6. Fichero de empaquetamiento Ejb-jar

Una vez que se han generado la clase del bean, los interfaces home y remote y el fichero descriptor de despliegue es el momento de empaquetar todos estos ficheros en un fichero Ejb-jar. Este fichero es un fichero comprimido de acuerdo al formato .zip como lo es cualquier fichero .jar, que lo que hace es compactar software Java para facilitar su portabilidad. En un fichero Ejb-jar puede haber uno o más beans empaquetados, permitiéndose así el transporte de un conjunto de componentes o de un bean en particular.

La generación de este fichero se puede hacer de forma manual llamando a las herramientas de empaquetado de las distribuciones de Java o utilizando las herramientas automáticas disponibles para tal efecto. La especificación de EJB 2.1 define que el soporte a los ficheros Ejb-jar es un requisito obligado y estándar para cualquier herramienta EJB.

Una vez creado el fichero Ejb-jar, el bean está completo y se ha convertido en una unidad desplegable dentro de un servidor de aplicaciones. Cuando estos ficheros son desplegados, las herramientas de los proveedores del contenedor son las responsables de la descompresión, la lectura y la extracción de la información contenida en el fichero. A partir de ese momento el responsable del despliegue de la aplicación tendrá que realizar las tareas relacionadas con el propio contenedor. Estas tareas habitualmente son realizadas por el contenedor o el servidor de aplicaciones de forma totalmente automática en el propio momento del despliegue, e incluyen desde la creación de los objetos Ejb y Home a partir de sus respectivos interfaces, hasta importar el bean dentro del contenedor.

3.2.6.7. Ficheros de empaquetamiento del cliente

La especificación de EJB define un fichero de empaquetamiento distinto para cada no de los posibles tipos de clientes que pueda tener una aplicaron distribuida que utilice la tecnología EJB. Es decir, que existirá un fichero web-client.war para aplicaciones en entorno web y un fichero, opcional según la especificación, app-client.jar para aplicaciones stand-alone.

Para el caso de aplicaciones web la especificación EJB define el fichero web-client.war, con extensión WAR (Web Archive), que permite agrupar los ficheros y las clases que deben ser desplegadas para cualquier cliente web de un determinado fichero Ejb-jar. Cuando se crea un fichero web-client.jar se deben incluir en él todos los ficheros que necesita el cliente web. Estos ficheros serán tanto los ficheros que conforman el front-end de la aplicación y que les permite a los usuarios interactuar con los beans que tiene la aplicación, como aquellos recursos propios de los componentes EJB y que requiere el cliente web para realizar las llamadas, como los interfaces EJB.

Por otro lado la especificación EJB define el fichero app-client.jar como una parte opcional dentro del proceso de desarrollo de EJB y que permite agrupar los ficheros y las clases que deben ser desplegadas para cualquier cliente stand-alone de un determinado fichero Ejb-jar. Cuando se crea un fichero app-client.jar se deben incluir en él solo los ficheros que necesita el cliente stand-alone. Estos ficheros normalmente son los interfaces, las clases helper, los stubs de los objetos remotos y los ficheros descriptores de despliegue correspondientes, como el application-client.xml o el fichero especifico del proveedor del servidor de aplicaciones JBoss llamado jboss-client.xml. Estos dos ficheros, por su condición de opcionales y específicos del servidor, no se estudiarán en esta sección sino que se verán en el siguiente capítulo cuando se vea un caso real de componentes EJB.

Sin embargo, sería conveniente recordar para terminar, que la utilización del fichero app-client.jar resulta útil para ahorrar espacio en el disco duro ya que evita tener que copiar entero en la parte cliente el fichero Ejb-jar.jar que contiene, por ejemplo, los interfaces EJB del bean.

3.2.6.8. Ficheros de empaquetamiento de la aplicación

Cualquier aplicación J2EE debe ser finalmente presentada en un fichero EAR (Enterprise Archive), que no es más que un fichero JAR (Java Archive) con extensión .ear. Este fichero .ear agrupa a distintos módulos J2EE y además contiene un descriptor de despliegue de la aplicación.

Hay que aclarar que un módulo J2EE estará formado por uno o más componentes J2EE que pertenecerán al mismo tipo de contenedor y que serán descritos por el mismo tipo de descriptor de despliegue. Por ejemplo, un módulo de Enterprise JavaBeans declarará atributos de transacciones y de seguridad para un determinado bean. Además, un modulo J2EE sin descriptor de despliegue de aplicación se considerará como un modulo independiente.

Por lo tanto, un fichero de empaquetamiento EAR puede contener cualquiera de los siguientes tipos de módulos que se enumeran a continuación y que se pueden ver en la siguiente figura:

1. Módulo EJB: Se empaqueta en un fichero con extensión .jar y contiene las clases y el descriptor de despliegue de los Enterprise JavaBeans.
2. Módulo Web: Se empaqueta en un fichero con extensión .war y contiene los servlets, paginas JSP, ficheros GIF y HTML y el descriptor de despliegue de la aplicación web.
3. Módulo Application Client: Se empaqueta en un fichero con extensión .jar y contiene las clases del cliente stand-alone y el descriptor de despliegue.
4. Módulo Resource Adapter: Se empaqueta en un fichero con extensión .rar y contiene los interfaces Java, las clases y librerías nativas, los adaptadores de recursos y el descriptor de despliegue, necesarios todos ellos, para resolver el accesos a sistemas legados. Todos estos elementos forman parte de de la J2EE Connector Architecture (JCA) que se implementará para un EIS (Enterprise Information System) en particular. Por lo tanto, la conexión entre el servidor de aplicaciones y el EIS se realizará a través de los llamados adaptadores de recursos. Usando JCA, los EJBs que se ejecutan en el servidor de aplicaciones J2EE pueden acceder a los sistemas externos mediante mecanismos homogéneos y bien definidos, logrando transparencia y portabilidad con respecto al EIS.

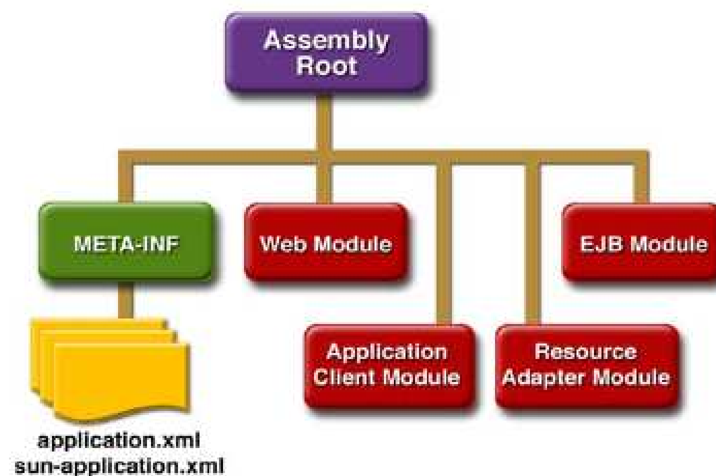


Figura 6: Estructura del fichero EAR

En cuanto al descriptor de despliegue de la aplicación hay que decir que es un fichero en formato XML y que se denomina application.xml. Este fichero describe los parámetros de despliegue de la aplicación, así como de los módulos que la forman. Finalmente, hay que recordar que debido a que la información que almacena el descriptor de despliegue es declarativa, esta información puede ser modificada sin necesidad de cambiar el código. En tiempo de ejecución el servidor J2EE leerá el descriptor de despliegue y actuará de acuerdo a lo parámetros declarados para la aplicación.

3.2.7. Llamando a los EJB´s: la parte cliente

Ahora es el momento de ver la otra mitad del mundo, es decir, dejar a un lado la parte servidora de EJB y estudiar mas en profundidad la parte cliente que es quien hace uso de toda la lógica de negocio que ayudará a solucionar los problemas reales usando uno o mas beans que trabajan juntos.

Hay que mencionar en este punto que existen dos tipos posibles de clientes EJB:

1. Clientes Java RMI-IIOP: Que serán aquellos clientes que usen Java Naming and Directory Interface (JNDI) para buscar los objetos a través de la red y que usen Java Transaction API (JTA) para controlar las transacciones. Estos son el tipo de clientes que se van a estudiar en este libro ya que son los que se encuentran dentro del mundo Java.

2. Clientes CORBA: Los clientes también podrían estar escritos de acuerdo al estándar CORBA. Esto podría resultar muy útil si se quisiese que nuestros componentes EJB fueran llamados desde clientes escritos en otros lenguajes de programación como C++. Los clientes CORBA usan el CORBA Naming Service (COS Naming) para buscar los objetos a través de la red y el servicio CORBA´s Object Transaction Service (OTS) para el control de las transacciones.

Pues bien, sea cual sea el tipo de cliente que llame a nuestros componentes EJB, CORBA o RMI-IIOP, dicho cliente siempre va a tener que hacer las siguientes operaciones:

1. Encontrar el objeto Home.
2. Usar el objeto Home para crear el EJB Object.
3. Llamar a los métodos de negocio que brinda ese EJB Object
4. Borrar el objeto EJB Object.

A continuación se describirá más en detalle la operación de búsqueda del objeto Home viendo las distintas tareas que incluyen esta operación, para profundizar así en el estudio de la búsqueda de referencias de métodos remotos. El resto de los pasos involucrados en una llamada a un componente EJB, debido a que resultan más triviales y menos didácticos se verán en el siguiente capítulo ya sobre el ejemplo práctico.

3.2.7.1. Buscando una referencia

Es importante destacar en este punto que existen distintos niveles en el ámbito de nombrado para cuando se busca una referencia dentro de un sistema de nombrado usando JNDI. Puede haber un sistema de nombrado local en el que solo hay visibilidad a los contextos particulares dentro de la misma maquina virtual u otro sistema accesible a clientes remotos en el que se requiere nombres con una visibilidad global y un soporte a serialización.

El local es un sistema que define un entorno de componentes de aplicación que solo permite acceder al contenedor para buscar los beans que maneja y no permitiría que un bean accediese a los elementos del ENC de otro bean o que una aplicación web accediese a los elementos del ENC de otro bean. En realidad, en este caso se puede afirmar que cada ENC de un determinado bean está desacoplado de cada ENC del resto de los componentes. Cada uno de los componentes lo que hace es definir su propio ENC, con lo que incluso dos componentes pueden tener el mismo nombre y referenciar a objetos distintos.

En este caso los contextos y los objetos enlazados cuelgan bajo el contexto *java:* y son únicamente visibles dentro de la máquina virtual en la que reside el contenedor o el servidor de aplicaciones y nunca por los clientes remotos. Un ejemplo en el que las restricciones de visibilidad del contexto *java:* podría resultar útil sería el caso de una fábrica de conexiones javax.sql.DataSource que solo pueda ser utilizada dentro de un servidor de aplicaciones que reside en la misma máquina que el pool de conexiones a una base de datos.

El siguiente fragmento de código ilustra como se accede a un ENC local creando un objeto javax.naming.InitialContext a partir de un constructor sin argumentos. Luego se busca en el entorno de nombrado *java:comp/env*, que es particular y restringido solo para los componentes asociados a ese contexto.

```
// Obteniendo el ENC
Context iniCtx = new InitialContext();

Context compEnv = (Context) iniCtx.lookup("java:comp/env");
```

Por su parte, cualquier otro nombre que no cuelgue de *java:* se considera un espacio de nombres global. Es decir, cualquier otro nombre de contexto o nombre enlazado estará disponible para clientes remotos, proporcionándoseles un contexto y soporte a la serialización de objetos. Es este nivel de nombrado remoto el que se estudiará en profundidad en el ejemplo práctico que se verá mas adelante, ya que es el que aporta una mayor nivel de dificultad.

En los siguientes apartados se describen los pasos que habría que llevar a cabo para la búsqueda del objeto Home.

3.2.7.1.1. Creando el contexto inicial

Para obtener un contexto inicial tenemos que seguir los siguientes pasos:

1. Seleccionar el proveedor de servicios correspondiente al servicio al que queremos acceder.
2. Especificar la configuración que necesite el contexto inicial.
3. Llamar al constructor InitialContext.

Para realizar los dos primeros pasos podemos hacerlo mediante código en la aplicación cliente, o por el contrario, utilizando el fichero jndi.propierties. Ambas posibilidades se muestran a continuación:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.NamingContextFac
tory");
env.put(Context.PROVIDER_URL, "jnp://localhost:1099");
```

```
### JBossNS properties
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
```

Ahora ya se puede crear el contexto inicial. Para hacer esto, pasamos al constructor de InitialContext las propiedades de entorno que hemos creado anteriormente de la siguiente forma:

```
Context ctx = new InitialContext(env);
```

Ahora ya se tiene una referencia a un objeto Context, con el que se puede empezar a acceder al servicio de nombres. Si se quieren realizar operaciones de directorio, se necesitaría usar la clase InitialDirContext en vez de la de InicialContext como se aprecia a continuación:

```
DirContext ctx = new InitialDirContext(env);
```

3.2.7.1.2. Buscando el objeto Home

Para conseguir la transparencia de localización, el contenedor de EJB oculta la localización de los objetos Home al código de los clientes de los beans. Los clientes por lo tanto, no tienen en su código ni siquiera el nombre de las máquinas en las que residen los objetos Home. En vez de eso, los clientes usan los métodos de la API JNDI para descubrir dichos objetos.

Los objetos Home estarna localizados físicamente en algún sitio de la red (quizás en el propio espacio de direcciones del contenedor de EJB o en otro distinto) pero esa es una cuestión de la que el desarrollador del código cliente de los EJB no debe preocuparle.

Para que el cliente pueda localizar el objeto Home, se debe proporcionar el nombre simbólico que le representa dentro del sistema de nombrado de JNDI. Este nombre se enlaza automáticamente al objeto Home cuando se realiza el despliegue de un bean dentro de un contenedor EJB. El desarrollador especifica ese nombre simbólico usando los ficheros específicos que los proveedores suministran y que están ligados con los beans que se desarrollan mediante ficheros como ejb-jar.jar.

Para localizar un objeto desde el servicio de nombres, usamos Context.lookup( ) y le pasamos el nombre del objeto que queremos recuperar. Por ejemplo, el nombre HelloHome será el nombre que los clientes usarán para identificar el objeto Home y así cualquier cliente en cualquier maquina a través de la red podrá encontrar dicho objeto.

```
Object obj = ctx.lookup("HelloHome");
```

Los clientes usan la API JNDI para realizar esta tarea y será JNDI quien explore a través de la red para descubrir el servicio de nombres, su árbol o quizás quien conecte con uno o más servicios de nombres. Finalmente el objeto Home será encontrado y se le devolverá al cliente una referencia a suya para que la utilice.

La siguiente grafica ilustra claramente cual es el orden de operaciones necesaria en este procedimiento:

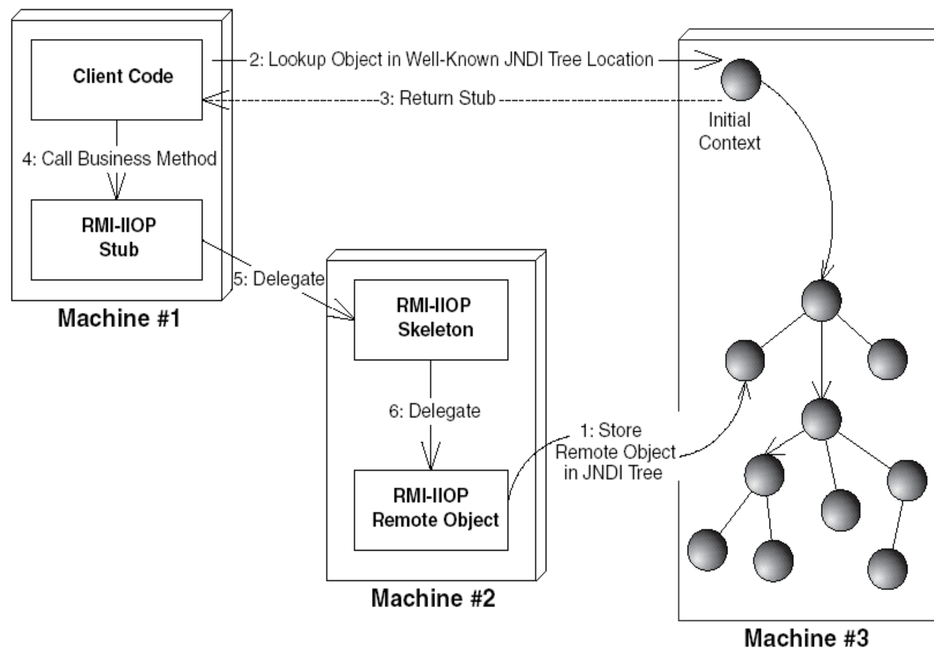


Figura 7: Diagrama de secuencia para JNDI

Finalmente, si en lugar de obtener un sólo objeto, como se consigue con Context.lookup( ), se quiere conocer el contenido entero de un contexto, se puede listar un contexto completo usando una sola operación. Hay dos métodos para listar un contexto: uno que devuelve las uniones y otro que sólo devuelve las parejas de nombres de clases nombre-a-objeto como ya se ha comentado anteriormente.

Por un lado el método Context.list( ) devuelve una enumeración de NameClassPair, en el que cada uno de ellos consiste en el nombre del objeto y el nombre de la clase. Mientras que el método Context.listBindings( ) devuelve una enumeración de tipos Bindings.

El siguiente fragmento de código lista los contenidos del contexto "ejb" (es decir, los subcontextos encontrados en el contexto "ejb"):

```
NamingEnumeration list = ctx.list("ejb");
```

```
while (list.hasMore()) {
    NameClassPair nc = (NameClassPair)list.next();
    System.out.println(nc);
}
```

4. JBoss aplication server: ejecutando la aplicación

Una vez visto todo el entorno tecnológico que necesitan los Enterprise JavaBeans para funcionar, con sus actores, sus partes, sus roles y sus características, es hora de ver como funciona verdaderamente la tecnología de EJB´s con el primer ejemplo que veremos a continuación.

Para ejecutar este ejemplo, al igual que para el resto de ejemplos del libro, se utilizarán una serie de herramientas dentro de las que destaca el servidor de aplicaciones de código abierto y libre distribución JBoss Application Server 4.0 y la herramienta de desarrollo Ant 1.6.2 de la Apache Software Foundation. De hecho, durante los siguientes apartados, tanto el servidor de aplicaciones JBoss como la herramienta Ant, serán bastante utilizadas ya que son partes imprescindibles dentro del desarrollo de este tipo aplicaciones.

Por lo tanto, antes de continuar hay que aclarar que el lector no se encontrará en este libro con toda una presentación general y explicación en profundidad de toda la información referente a dichas herramientas. Esto es debido a que esa información se sale del ambito estricto del libro y a que podría ser, en si misma, otro libro entero. Sin embargo, lo que si se expondrá, con el máximo detalle, serán cada una de las operaciones que se vayan realizando con cada una de esas herramientas. De esta forma, cada operación realizada durante el proceso de desarrollo será debidamente detallada y comentada para asi conseguir un claro entendimiento de lo que se está haciendo.

También es conveniente aclarar que no se le exige al lector ningún estudio pormenorizado por su cuenta de las generalidades de las herramientas o estar familiarizado con los entornos de ejecución para conseguir los resultados esperados con los ejemplos. Por el contrario, se asume de entrada que los conocimentos por parte del lector en lo que a estas dos herramientas respecta son escasos, y de hecho, lo que se pretende es que al terminar el recorrido por el contenido del libro, el lector pueda haber conseguido bastante soltura a la hora de trabajar con estas herramientas.

Como es lógico, cualquier conocimiento previo de estas herramientas lo único que hará es sumar y en último termino acelerar la consecución de resultados. Sin embargo, a pesar de que no son necesarios demasidos conocimientos previos, si que se pide, como también es lógico, un mínimo de interes que asegure la consecución de unos resultados ciertamente favorables.

En cualquier caso, para encontrar mayor cantidad de información y solucionar las posibles dudas que surgan con respecto a estas herramientas, y que el presente libro no sea capaz de resolver, se recomienda consultar la documentación que se puede encontrar en las páginas *web* correspondientes de cada herramienta y que se comentarán cuando se vean los requisitos mínimos para poder ejecutar cualquier ejemplo del libro.

4.1. Ejemplo Hola Mundo!!

4.1.1. Introducción

Por fin ha llegado la hora de ver como se lleva a la práctica toda la teoría que se ha visto hasta ahora. En el ejemplo que se presenta a continuación se verá paso a paso como escribir cada uno de los ficheros que componen un EJB, como se empaquetan esos ficheros en distintos formatos para facilitar su portabilidad, como se despliegan dentro de un servidor de aplicaciones, y finalmente, como se accede a ese EJB tanto desde una aplicación cliente como desde una aplicación en entorno *web* a través de una pagina JSP.

El bean del ejemplo tendrá la no desdeñable misión de devolver el *string* "Hola Mundo!!" al cliente que lo llamó, ya sea un cliente Java stand-alone o un cliente web a través de un pagina JSP. Este ejemplo, que a priori puede resultar escaso resulta bastante útil para descubrir como se desarrolla y se ejecuta de forma muy rápida y sencilla con la tecnología EJB. Si bien es cierto que este primer ejemplo no hará una demostración impresionante del poder de los EJB, si que se pretende que este ejemplo ilustre claramente los fundamentos de esta arquitectura y sobre todo que siente las

bases de una plantilla que será útil para la construcción de componentes mucho mas complejos.

Como se comentó al inicio de este capitulo para la realización de los ejemplos propuestos en este libro se utilizará el servidor de aplicaciones JBoss Application Server 4.0 y la herramienta de desarrollo Ant. El hecho de utilizar una herramienta como Ant facilita enormemente el desarrollo y despliegue de aplicaciones que utilicen la tecnología EJB.

Por lo tanto, las fases de desarrollo de los beans utilizados en los ejemplos de este libro, se basarán en las fases de la especificación de EJB, explicadas ya anteriormente, y se ejecutarán utilizando los mecanismos que proporciona la herramienta Ant para su consecución. Es decir, cada uno de los pasos de desarrollo se ejecutará de acuerdo al plan de ejecución descrito en el fichero de configuración jboss-build.xml. Este fichero contendrá todas las tareas que se deben realizar en fase de desarrollo. Por este motivo resulta muy recomendable que el lector siga paso a paso la realización del ejemplo, tal y como se expone, ya que todas esas tareas se irán explicando a medida que van siendo necesarias. Este seguimiento permitirá irse familiarizando poco a poco con las particularidades de la herramienta Ant, y por supuesto, con el servidor de aplicaciones JBoss.

Finalmente, hay que mencionar que para este ejemplo se utilizará un bean de sesión sin estado. Este bean tiene quizás el tipo más sencillo que se puede implementar, pero por ahora cubre completamente las expectativas que de él se esperan en este primer ejemplo. A medida que se vayan introduciendo más conceptos y profundizando en los conocimientos de esta arquitectura se irán utilizando en los ejemplos otros tipos de beans más complejos.

4.1.2. Que se hace en el ejemplo, que no se hace y ¿por qué?

El objetivo final de este libro es el de guiar al lector por los caminos de la programación de aplicaciones distribuidas de la mano de la tecnología EJB. Sin lugar a dudas, con un poco de actitud e interés por su parte, al final del mismo dicho objetivo estará más que conseguido. Sin embargo, muchas veces no resulta sencillo

poner en práctica todo lo que se estudia de forma teórica, debido a inconvenientes, como por ejemplo, no disponer de una verdadera LAN para ejecutar en un entorno distribuido.

Y es aquí donde se pretende hacer hincapié. No sería lógico, después de lo recorrido hasta ahora y debido a este tipo de inconvenientes, recurrir a un ejemplo "poco" distribuido o local. Por el contrario, se pretende que el ejemplo propuesto sea lo mas completo posible, siendo en ejemplo real de sistema distribuido y que de alguna manera pase por encima de las limitaciones de ejecutar clientes y servidores en la misma máquina.

Por lo tanto, es a estas alturas, y antes de ver el propio ejemplo, cuando es conveniente preguntarse y saber en cada momento, qué se va a hacer, cómo se va a hacer, cual es el alcance de lo que se va a hacer y por qué no se puede hacer algo más. Es decir, cabe preguntarse: ¿Es el ejemplo un sistema distribuido?, ¿Está el ejemplo realmente distribuido?, ¿Tendría que hacer algo para distribuirlo?.

Pues bien, en primer lugar hay que decir que el ejemplo de Hola Mundo!! que se propone en el libro es un sistema distribuido por los cuatro costados. Es decir, esta diseñado para que clientes y servidores ejecuten en espacios de direcciones distintos, es decir, en maquinas distintas y para que funcione correctamente. De esta forma, una aplicación cliente instalada en una determinada máquina puede llamar al bean que se encuentra distribuido en otra máquina y que devuelve el Hola Mundo!!. O que un cliente *web* mediante el protocolo HTTP descargue una pagina JSP que también accede a ese mismo bean. En definitiva, el sistema esta diseñado y desarrollado para que tanto los clientes, bien sean aplicaciones Java stand-alone o clientes web, y el servidor de aplicaciones donde viven los beans, en este caso JBoss, residan en puntos distantes dentro de una LAN.

Desafortunadamente, la mayor parte de los lectores del libro no dispondrán más que de una máquina donde ejecutar los ejemplos de este libro. Por lo tanto, el ejemplo está enfocado a un despliegue en una sola máquina, o sea, que en la misma máquina donde residen los clientes residen los servidores. En ese sentido el ejemplo a pesar de ser distribuido, no lo está realmente ya que el despliegue del cliente Java, por ejemplo, se

hace en el mismo espacio de direcciones donde el lector tendrá instalado el servidor de aplicaciones JBoss.

Sin embargo, debe quedar claro que esta situación (ser un ejemplo distribuido pero no estarlo realmente), no es más que una cuestión circunstancial por la que se ha optado por cuestiones didácticas, de simplicidad y rapidez a la hora de poder ejecutar el ejemplo. Este mismo ejemplo puede ser distribuido en cualquier momento dentro de una LAN, cuando se disponga de ella, sin necesidad de recompilar ninguna parte del sistema. Esto es posible ya que no se ha recurrido a llamadas locales dentro de la misma máquina virtual (esa es la razón de que no se utilicen los interfaces locales en el ejemplo) sino a llamadas a métodos remotos, tal y como se haría si efectivamente el sistema fuese a ser distribuido en distintas máquinas. Quizás la única tarea adicional que debería llevar a cabo el desarrollador en caso de intentar ejecutar este ejemplo en un entorno de una LAN sería la de comprobar que, efectivamente, los *stubs* de los objetos remotos residen en la parte cliente cuando se realice el despliegue en dicha máquina.

4.1.3. Requisitos para ejecutar el ejemplo

Para poder ejecutar el ejemplo Hola Mundo!! el lector debe tener instalado y funcionando correctamente en su máquina el servidor de aplicaciones JBoss Application Server 4.0. Se puede descargar de forma completamente gratuita, ya que es código abierto y de libre distribución, la última versión de JBoss del siguiente sitio web: http://www.jboss.org.

Una vez descargado el fichero comprimido (.zip, .tar.gz, etc) de este sitio web, dependiendo de la plataforma en la que se quiera ejecutar, éste simplemente se debe descomprimir en algún directorio de la máquina en la que se quiera instalar, teniendo siempre cuidado con los nombres de directorios con espacios que puede dar algún problema.

El único requisito adicional que necesita el JBoss para funcionar es el de tener una versión actualizada de Java instalada en la maquina donde se instalará el servidor de aplicaciones. JBoss 4.0 necesita para funcionar por lo menos la versión 1.4 del JDK.

Hay que asegurarse, además, que lo que se tiene instalado es la versión 1.4 del JDK y no la de la JRE, ya que aunque en este último caso el servidor de aplicaciones arrancaría correctamente, cuando se requiera realizar tareas de compilación de páginas JSP nos encontraremos con ciertos problemas debido a que no se encuentran ciertas clases. Por lo tanto, para que el servidor de aplicaciones funcione correctamente necesita conocer donde está el JDK y eso se hace mediante la variable de entorno denominada JAVA_HOME que debe apuntar justo al directorio raíz de la instalación del JDK.

Por otro lado, se requiere tener instalada la herramienta Ant de la Apache Software Foundation. Para los ejemplos de este libro se ha utilizado la última versión disponible en el sitio web http://ant.apache.org/, que es la versión 1.6.2. Esta versión requiere tener instalado como mínimo la versión 1.2 o posterior del JDK. Al igual que en el caso del servidor JBoss, si el JDK no está instalado y lo está el JRE, no todas las tareas funcionarán correctamente.

Para instalar Ant hay que descomprimir la distribución binaria de ficheros que se descargó de la página web en cualquier directorio de la máquina. Por lo tanto, este directorio será el directorio home de la aplicación y se debe declarar la variable de entorno ANT_HOME que punte justo a ese directorio. Además, resulta muy útil añadir el directorio bin, del directorio donde apunta ANT_HOME, a la variable de entorno path para poder tener acceso a la herramienta Ant en cualquier directorio de la máquina.

4.1.4. Preparando el entorno de ejecución

4.1.4.1. Arrancando el JBoss

Para comprobar que el servidor de aplicaciones está bien instalado hay que arrancarlo. Para llevar a cabo esta tarea hay que buscar el directorio *bin* dentro del directorio principal del JBoss. Este directorio contiene varios scripts y para arrancar el servidor hay que ejecutar el de run.bat si se está en el sistema operativo Windows, o el de run.sh si se esté en Linux. Obviamente lo que hace este script es arrancar el servidor y

por lo tanto, lo que se debe ver es una consola de *log* del JBoss con los mensajes de todos los componentes que hay desplegados hasta ese momento y que se van arrancando. El ultimo mensaje que debe aparecer en este log, evidentemente con valores distintos en cuento a fecha, hora y tiempo de arranque, debe ser muy parecido al que se muestra a continuación:

Para verificar que el servidor está funcionando correctamente, una vez arrancado, se puede ejecutar el servidor web del servidor de aplicaciones que escucha por el puerto 8080. Por lo tanto, asegurándonos de que ninguna otra aplicación usa este puerto en nuestra máquina, se puede acceder a la página por defecto del servidor JBoss poniendo la siguiente url: http://localhost:8080/. Esta página mostrará enlaces a algunos recursos útiles de JBoss.

El servidor de aplicaciones se detiene usando el comando Ctrl-C sobre la consola de *log* del JBoss o utilizando, bien el script shutdown.bat para Windows o el de shutdown.sh para Linux, que se encuentran dentro del directorio *bin* del directorio principal del JBoss.

Es importante destacar en este momento que el servidor de aplicaciones JBoss viene con la versión 5.0 del Tomcat como contenedor web. Este contenedor web se encuentra embebido dentro del servidor de aplicaciones como un servicio web. De hecho se puede ver como hay un directorio dentro del directorio de despliegue llamado jbossweb-tomcat50.sar, que es el fichero Service Archive de JBoss (SAR) que tiene el servicio del Tomcat.

Pues bien, el hecho de que Tomcat sea un servicio web y no un servidor *stand-alone* que corre de forma independiente, como lo era antes, hace que la forma de utilizarlo sea ligeramente distinta a como se hacia anteriormente. Por ejemplo, en el caso del despliegue de una aplicación web, ésta se hace ahora descargándola directamente dentro del directorio *deploy* del servidor JBoss y no en un directorio del contenedor web Tomcat. Será luego el propio servidor de aplicaciones quien se encargue de dialogar y trabajar con el contenedor Tomcat. Entonces habría que dejar claro que ahora el responsable del despliegue de la aplicación en ningún momento debe acceder directamente a los directorios del Tomcat como se hacia antes. Por otra parte, tanto el

log interno del Tomcat como los accesos al mismo, se encuentran ahora en el directorio *log* de JBoss.

Por lo tanto, resulta imprescindible ser concientes del nuevo entorno de ejecución en el que se apoya la tecnología EJB y conocer las particularidades y matices que introduce el trabajar con un servidor de aplicaciones como JBoss. Por ejemplo, no es necesario tener que re-arrancar el contenedor web o el servidor de aplicaciones cuando se despliega una nueva aplicación web dentro del servidor JBoss ya que ese despliegue se hace completamente en caliente. Los servidores de aplicaciones soportan el despliegue iterativo, es decir, que si en algún momento se requiere hacer algún cambio en la aplicación que se esta desarrollando esta aplicación se puede volver a desplegar sin problemas en el servidor. Un ejemplo de esto último podría ser el caso en el que se necesite cambiar el código fuente de un fichero de un bean. En este caso se recompila el fichero, se ensambla de nuevo y se vuelve a desplegar la aplicación.

4.1.4.2. Preparando la herramienta Ant

Finalmente hay que hablar del directorio de trabajo del ejemplo, de donde se van crear los ficheros, de cómo se van ejecutar las tareas de desarrollo de los EJB o de cómo se hará el despliegue dentro del servidor de aplicaciones JBoss. En definitiva queda hablar de Ant.

Para la realización de cada uno de los ejemplo de este libro se utilizará un directorio de trabajo que el lector puede localizar donde le resulte mas conveniente. En ese directorio se guardarán tanto el fichero de configuración jboss-build.xml que tendrá el plan de desarrollo del proyecto EJB, como los ficheros fuente de la aplicación. Por lo tanto, será en el directorio de trabajo elegido por el lector donde se ejecute la herramienta Ant. Ant leerá el fichero de configuración y realizará las tareas allí descritas utilizando los recursos o ficheros fuente que se hayan distribuido en su estructura de directorios.

Así pues, se recomienda por coherencia (no hay reglas de obligado cumplimiento) que los ficheros fuente se localicen en distintos subdirectorios de acuerdo al papel que desempeñen dentro de la aplicación distribuida que se va a desarrollar. Es decir, los ficheros descriptores de despliegue estarán dentro del directorio dd (deployment descriptor), los ficheros fuentes del bean a desarrollar estarán en el directorio src (source) y finalmente, por ejemplo para aplicaciones en entorno web, los ficheros JSP estarán en el directorio web. Esta estructura de directorios que se propone se aprecia en la siguiente figura:

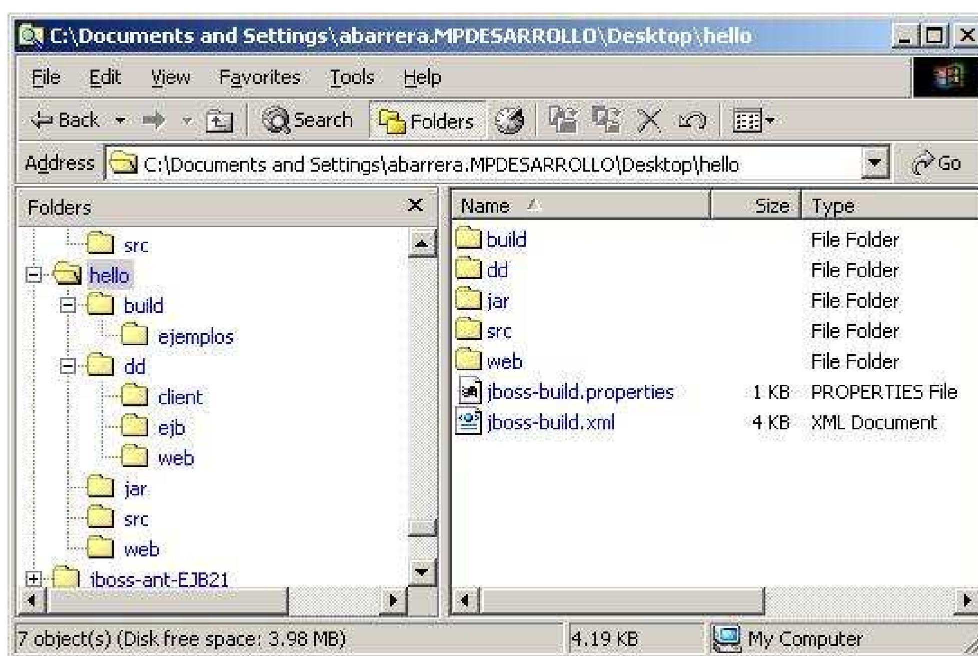


Figura 8: Estructura de directorios de trabajo

Esta estructura de subdirectorios del directorio de trabajo resulta muy importante ya que está íntimamente ligada con el fichero de configuración jboss-build.xml y facilita enormemente la tarea de desarrollo de la tecnología EJB. Es en este fichero en el que se le indica a la herramienta las tareas que tiene que realizar y donde están los recursos que debe utilizar para esas tareas de las fases de desarrollo. Por lo tanto, resulta evidente solicitar al desarrollador consistencia entre lo que se le dice a la herramienta y la ubicación real de esos recursos, para que todo funcione de forma correcta.

A continuación se muestra el fichero de configuración jboss-build.xml que se utilizará en el desarrollo del ejemplo del Hola Mundo!!, y que además, debería servir como plantilla para el desarrollo del resto de ejemplos del libro con solo unos pequeños cambios para adaptarlo a cada caso.

```
<project name="HelloWorld" default="all" basedir=".">

    <property file="jboss-build.properties"/>

    <property name="lib.dir"   value="../../libs"/>
    <property name="src.dir"   value="${basedir}/src"/>
    <property name="build.dir" value="${basedir}/build"/>

    <!-- El classpath para ejecutar el cliente -->
    <path id="client.classpath">
        <fileset dir="${jboss.home}/client">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <!-- Classpath para la compilación-->
    <path id="build.classpath">
        <path refid="client.classpath"/>
         <fileset dir="${jboss.server}/lib/">
            <include name="javax.servlet*.jar"/>
        </fileset>
    </path>

    <!-- ============================================================== -->
    <!—Directorio de creación                                          -->
    <!-- ==============================================================-->
    <target name="prepare">
        <mkdir dir="${build.dir}"/>
    </target>

    <!-- ============================================================== -->
    <!— Compilación del código fuente                                  -->
    <!-- ============================================================== -->
    <target name="compile" depends="prepare">
        <javac destdir="${build.dir}" classpathref="build.classpath"
debug="on">
            <src path="${src.dir}"/>
        </javac>
    </target>

    <target name="package-ejb" depends="compile">
        <mkdir dir="jar" />
        <delete file="jar/hello-ejb.jar"/>
        <jar jarfile="jar/hello-ejb.jar">
            <metainf dir="dd/ejb" includes="**/*.xml" />
            <fileset dir="${build.dir}">
                <include name="**/*.class"/>
               <exclude name="*.class"/>
            </fileset>
        </jar>
    </target>

    <target name="package-client" depends="compile">
        <mkdir dir="jar" />
         <delete file="jar/app-client.jar"/>
        <jar jarfile="jar/app-client.jar">
            <metainf dir="dd/client" includes="*.xml"/>
            <fileset dir="${build.dir}">
```

```
                <include name="**/*.class"/>
            </fileset>
            <fileset dir="${src.dir}">
                <include name="jndi.properties"/>
            </fileset>
        </jar>
    </target>

    <target name="package-web" depends="compile">
        <mkdir dir="jar" />
        <delete file="jar/web-client.war"/>
        <war warfile="jar/web-client.war" webxml="dd/web/web.xml">
            <fileset dir="web">
                <include name="*.jsp"/>
                <include name="template/*"/>
                <include name="images/*.gif"/>
            </fileset>
            <webinf dir="dd/web">
                <include name="jboss-web.xml"/>
            </webinf>
            <classes dir="${build.dir}">
                <include name="**/*.class"/>
            </classes>
        </war>
    </target>

    <!—Crea un fichero ear que contiene los ejbjars y el webclient war. -->
    <target name="assemble-app">
        <delete file="jar/Hello.ear"/>
        <ear destfile="jar/Hello.ear" appxml="dd/application.xml">
            <fileset dir="jar" includes="*-ejb.jar,app-client.jar,*.war"/>
        </ear>
    </target>

    <!—Despliega el fichero EAR copiándolo en el directorio deploy JBoss -->
    <target name="deploy" depends="assemble-app">
        <copy file="jar/Hello.ear" todir="${jboss.server}/deploy"/>
    </target>

    <!—Ejecuta el cliente standalone -->
    <target name="run-client">
        <echo>${java.class.path}</echo>
        <java classname="HelloClient" fork="yes">
            <classpath>
                <pathelement path="jar/app-client.jar"/>
                <path refid="client.classpath"/>
            </classpath>
        </java>
    </target>

    <target name="clean">
       <delete dir="${build.dir}" />
       <mkdir dir="${build.dir}"  />
    </target>

    <target name="all" depends="compile,package-ejb,package-web,package-
client,assemble-app,deploy" />

</project>
```

Fichero jboss-build.xml

Como se puede apreciar en este código XML que lee la herramienta Ant, jboss-build.xml define una serie de tareas que posteriormente se irán llamando de acuerdo a la fase de desarrollo en la que se encuentre la aplicación. La explicación detallada de lo que hace cada una de estas tareas se abordará cuando se tenga que llamar a dicha tarea especifica durante las fases de desarrollo que se verán en los siguientes apartados del ejemplo.

Finalmente, hay que destacar que en la segunda línea del fichero jboss-build.xml se define un fichero de propiedades llamado jboss-build.properties. Este fichero específico de JBoss declarará propiedades que se utilizarán posteriormente en el fichero jboss-build.xml. Un ejemplo de estas propiedades podría ser jboss.server en el que se definirá el directorio en el que reside el servidor de aplicaciones JBoss. A continuación se muestra ese fichero de propiedades en el que ya se ha dado valor a la propiedad jboss.home.

```
# Set the path to JBoss directory containing the JBoss application server
# (This is the one containing directories like "bin", "client" etc.)

jboss.home=H:\Program Files/JBOSS/4.0/jboss-4.0.1sp1
jboss.server=${jboss.home}/server/default
jboss.deploy.dir=${jboss.server}/deploy
```

Nota de seguimiento: partiendo de una base suficientemente sólida

Puede ser que a estas alturas alguien piense que el camino ya esta siendo bastante largo cuando apenas se empieza a andar. Para llegar hasta aquí se ha visto bastante teoría: fundamentos históricos, introducción a nuevas tecnologías o la necesidad de usar herramientas externas, y en general se ha visto poca práctica. Es verdad, puede haber sido largo. Pero para adentrarse en una tecnología como EJB es necesario conocer bien el terreno sobre el que se fundamenta y ser concientes de qué se esta haciendo en cada momento y por qué. Este conocimiento previo ayudará a que los ejemplos propuestos sean realmente didácticos en el sentido de que se vea en funcionamiento lo aprendido hasta ese instante.

Por lo tanto, llegados a este punto es importante que el lector sepa claramente cual es el objetivo de la aplicación propuesta en el ejemplo, entendiéndola claro, dentro del marco de las aplicaciones distribuidas. Una vez conseguido este objetivo en el plano estrictamente conceptual es momento de preocuparse del plano estrictamente práctico. Y en ese sentido, por el momento lo único importante es, en primer lugar, tener el servidor de aplicaciones JBoss funcionando correctamente, en segundo lugar, tener en el directorio de trabajo el fichero jboss-build.xml y su correspondiente fichero de propiedades jboss-build.properties, tal y como se mostró anteriormente, y finalmente, tener la estructura de directorios vacía similar a la propuesta. Nada más, es hora de empezar a crear los ficheros que necesita la aplicación EJB y que irán poblando esa estructura.

4.1.5. Preparando los ficheros

En este apartado se mostrará el código de todos los ficheros implicados en el desarrollo de la aplicación distribuida del ejemplo del Hola Mundo!!. Para cada uno de estos ficheros se hará una breve explicación del papel que desempeñan dentro del sistema, se explicarán las principales características del código relativas a la tecnología de EJB, así como su ubicación dentro de la estructura del directorio de trabajo.

4.1.5.1. Interfaz remoto

El interfaz remoto duplica cada uno de los métodos de la lógica de negocio que el bean finalmente va a publicar. Este interfaz es con el que operan los clientes cuando interactúan con los objetos EJB. Además, el contenedor se encargará de implementar este interfaz creando el objeto EJB que delega realmente a su vez en la instancia de la clase del bean.

El interfaz remoto se guardará en un fichero Hola.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz remoto para el Enterprise JavaBean: Hola
*/

package ejemplos;

public interface Hola extends javax.ejb.EJBObject {

        /*
         * Método que devuelve un string de saludo al cliente
        */
        public String hola() throws java.rmi.RemoteException;

}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

1. El interfaz extiende de javax.ejb.EJBObject lo que indica que el objeto EJB que implementará el contenedor contendrá todos los métodos (como remove o isIdentical) definidos en este interfaz.

2. El interfaz define el método clave de la lógica de negocio denominado hola( ) que devolverá el String “Hola Mundo!!” al cliente que lo llame. Este método se implementará en la clase del bean. Al ser un interfaz remoto RMI-IIOP debe lanzar una excepción remota java.rmi.RemoteException que indicaría, por ejemplo, algún problema de conexión a través de la red. Esta es la única diferencia entre las signaturas del método hola( ) del interfaz remoto y la signatura del método hola( ) del bean que se verá a continuación.

4.1.5.2. Interfaz home

El interfaz remoto tiene los métodos de creación y destrucción del objeto EJB. La implementación de este interfaz es responsabilidad del propio contenedor y se materializa en el objeto Home.

El interfaz home se guardará en un fichero HolaHome.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz Home para el Enterprise JavaBean: Hola
 */

package ejemplos;

public interface HolaHome extends javax.ejb.EJBHome {

        /*
         * Método que crea la instancia del bean de sesión Hola
         */
        public ejemplos.Hola create()
                throws javax.ejb.CreateException, java.rmi.RemoteException;

}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

1. El interfaz home tiene que definir un método create( ) que devuelva el tipo del bean que se quiere crear. Este método será implementado como una fábrica de objetos que los clientes usan para conseguir una referencia a los objetos EJB.

2. Hay que recordar que el método create( ) no crea beans sino que crea objetos EJB. El ciclo de vida de los beans es gestionado por el contenedor y create( ) solo crea objetos EJB que posteriormente delegarán en los beans. Por lo tanto, este método se corresponderá con un método ejbCreate( ) de la clase del bean, que como resulta lógico, se utilizará también para inicializarlo.

3. El interfaz home al extender de javax.ejb.EJBHome define por herencia un método de destrucción del bean por lo que no es necesario sobrescribirlo.

4. El método create( ) lanza excepciones java.rmi.RemoteException como todos los objetos RMI-IIOP y además excepciones javax.ejb.CreateException como todos los métodos create( ). Este último tipo de excepciones denominadas excepciones de nivel de aplicación, se corresponden, por ejemplo, a situaciones como un paso de parámetros incorrecto en la inicialización de un bean. Este tipo de excepciones, como norma, debe ser devuelta al cliente ya que son situaciones que a él le interesan, mientras que excepciones de nivel se sistema, como fallos en las conexiones de red, deben ser transparentes y solucionadas por el propio contenedor cuando el objeto EJB intercepta las peticiones.

4.1.5.3. La clase del bean

La clase del bean contiene la implementación tanto de los métodos de la lógica de negocio definidos en el interfaz remoto, como la implementación de los métodos del interfaz javax.ejb.SessionBean ya que en este ejemplo se implementa el bean como un bean se sesión sin estado. Es decir, que la clase del bean reúne e implementa los métodos de la lógica de negocio y los métodos relacionados con el ciclo de vida del bean.

La clase del bean se guardará en un fichero HolaBean.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Implementación de la clase del Enterprise JavaBean: Hola
*/

package ejemplos;

public class HolaBean implements javax.ejb.SessionBean {

        /*
         * Atributo que representa el contexto del bean
         */
        private javax.ejb.SessionContext mySessionCtx;

        /*
         * Método hola que devuelve el string
         */
        public String hola() {
                System.out.println("ejecutando hola()");
                return "Hola Mundo!!";
        }

        /*
         * Método de creación en el ciclo de vida del bean
         */
        public void ejbCreate() throws javax.ejb.CreateException {
                System.out.println("ejecutando ejbCreate()");
        }

        /*
         * Método activación dentro del ciclo de vida del bean
         */
        public void ejbActivate() {
                System.out.println("ejecutando ejbActivate()");
        }

        /*
         * Método del ciclo de vida del bean
         */
        public void ejbPassivate() {
                System.out.println("ejecutando ejbPassivate()");
        }

        /*
         * Método finalización en el ciclo de vida del bean
         */
        public void ejbRemove() {
                System.out.println("ejecutando ejbRemove()");
        }

        /*
         * Método getSessionContext
         */
        public javax.ejb.SessionContext getSessionContext() {
                return mySessionCtx;
        }

        /*
         * Método setSessionContext
         */
        public void setSessionContext(javax.ejb.SessionContext ctx) {
                mySessionCtx = ctx;
```

```
	}
}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

1. La clase del bean deberá tener un método ejbCreate( ) sin parámetros y que es el método que llama el objeto Home desde su método create( ). Al ser ejbCreate( ) llamado desde el objeto Home, que es parte del contenedor, éste método no lanza ninguna excepción remota.

2. La clase del bean implementa el interfaz javax.ejb.SessionBean y por lo tanto, todos los métodos relacionados con el ciclo de vida. Todos estos métodos son llamados exclusivamente por el contenedor y nunca por el cliente. Por lo tanto, para facilitar el entendimiento de la gestión hecha por el contendor del ciclo de vida del bean, se introducirán trazas de información de actividad por la salida estándar. Este será el único código asociado a cada uno de esos métodos ya que el bean no requiere ninguna inicialización ni gestión especial cuando se destruya. Además, los métodos ejbActivate( ) y ejbPasivate( ) no aplican para beans de sesión sin estado.

3. La clase del bean define una propiedad de tipo javax.ejb.SessionContext. Como se ha comentado en repetidas ocasiones los beans viven en el contenedor y éste es el responsable de llamar a sus métodos de acuerdo a sus políticas de gestión particulares. Sin embargo, resulta útil disponer de un mecanismo que permita acceder desde los beans a información propia del contenedor como, por ejemplo, su estado. Este mecanismo se define en la especificación de EJB y es el objeto contexto que implementa el contenedor. Por lo tanto, el interfaz javax.ejb.EJBContext, interfaz padre de javax.ejb.SessionContext, define una serie de métodos que permiten al objeto contexto del contenedor ir evolucionando en el tiempo según van evolucionando los beans que contiene. Así por ejemplo, un bean que participa en una nueva transacción modifica el estado del contexto del contenedor. Todos estos métodos del interfaz EJBContext se verán de forma detallada en

el apartado correspondiente dentro del capitulo dedicado a los beans de entidad.

Nota de seguimiento: analizando el meollo del asunto

Segundo hito alcanzado en el desarrollo de la aplicación Hola Mundo!!. Hasta aquí se han declarado todas las partes de la aplicación que pertenecen al lado del servidor y que ofrecen una determinada funcionalidad mediante Enterprise JavaBeans. Hasta aquí llega el auténtico trabajo que le corresponde al proveedor de beans porque estamos ante la parte de la aplicación que aporta la lógica de negocio y donde realmente hay que hacer un esfuerzo grande de análisis y diseño de componentes.

Bien es cierto que en el caso particular que nos ocupa, ese trabajo no ha sido demasiado costoso debido a la simplicidad del ejemplo propuesto (no esta de más recordar que con el ejemplo se pretende ilustrar las particularidades de la tecnología EJB y en ningún caso las dificultades de desarrollo de una aplicación de gran entidad), pero las cosas no son siempre así de fáciles en aplicaciones de mayor complejidad. En estos casos, un buen análisis y diseño de componentes, con interfaces robustos y funcionalidades bien definidas, facilitará enormemente todo el proceso de desarrollo e integración de las aplicaciones a nivel empresarial que los utilizarán.

4.1.5.4. Descriptores de despliegue

JNDI se utiliza como una API para externalizar una gran cantidad de información de los componentes EJB. Por ejemplo, los nombres JNDI de los componentes de las aplicaciones estarán declarados en los ficheros descriptores de despliegue estándar como nombres lógicos, totalmente independientes de servidor de aplicaciones en el que finalmente se vayan a desplegar. Estos nombre lógicos podrán estar, bien en el fichero ejb-jar.xml para componentes EJB o en el fichero web.xml para componentes en entorno *web*. Incluso es posible declarar información de los componentes en un fichero opcional de la tecnología EJB llamado application-client.xml, que usarían los clientes *stand-alone*.

Como ya se comento en su momento, los ficheros descriptores de despliegue estándar son unas de las principales aportaciones de la tecnología EJB ya que permiten especificar de forma declarativa atributos de los beans en lugar de hacerlo de forma programática. Por lo tanto, en los ficheros descriptores de despliegue que se utilizarán en el ejemplo se especificarán las necesidades que requiere el bean, que se está desarrollando, del contenedor EJB. Para el ejemplo planteado se utilizarán en la

aplicación los tres descriptores de despliegue enumerados anteriormente y que se estudiarán en detalle a continuación.

En primer lugar, el fichero descriptor de despliegue del componente EJB se guardará en el fichero ejb-jar.xml en el directorio /dd/ejb del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE ejb-jar PUBLIC
          "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
          "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
         <session>

                 <ejb-name>Hola</ejb-name>
                 <home>ejemplos.HolaHome</home>
                 <remote>ejemplos.Hola</remote>
                 <ejb-class>ejemplos.HolaBean</ejb-class>
                 <session-type>Stateless</session-type>
                 <transaction-type>Container</transaction-type>

         </session>
    </enterprise-beans>
</ejb-jar>
```

Hay que destacar que existen muchos elementos y parámetros de configuración que componen un descriptor de despliegue y que irán surgiendo de acuerdo a las necesidades de los ejemplos propuestos. Por ejemplo, el elemento <session> indica que el bean que se empleará para este caso es un bean de tipo sesión, de la misma forma que un elemento <entity> indicaría que el bean incluido en el fichero es de tipo entidad.

Sin embargo, los parámetros más importantes a destacar del ejemplo que se propone se enumeran a continuación:

1. Parámetro <ejb-name>: Es el alias que se define para ese bean en particular. Este nombre puede ser usado posteriormente en el momento del despliegue, por ejemplo, para hacer referencia al bean y asignarle nuevos parámetros de configuración. Además, en ausencia del fichero específico descriptor de

despliegue que aporte un nombre JNDI, el interfaz Home se enlazará con el valor del parámetro <ejb-name>.

2. Parámetro <home>: Es el nombre completamente cualificado del interfaz home, es decir, el nombre completo del home con los paquetes a los que pertenece.
3. Parámetro <remote>: Es el nombre completamente cualificado del interfaz remote.
4. Parámetro <ejb-class>: Es el nombre completamente cualificado de la clase del bean.
5. Parámetro <session-type>: Determina si el bean de sesión es con estado (stateful) o sin estado (stateless).
6. Parámetro <transaction-type>: Determina el tipo de transacción del bean y es el típico parámetro que por ahora se puede ignorar ya que para el ejemplo no tiene mayor relevancia.

En segundo lugar, el fichero descriptor de despliegue de los componentes en entorno web se guardará en el fichero web.xml en el directorio /dd/web del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <display-name>HolaWeb</display-name>

    <ejb-ref>

        <description>Referencia web</description>
        <ejb-ref-name>refWeb</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>ejemplos.HolaHome</home>
        <remote>ejemplos.Hola</remote>

    </ejb-ref>

</web-app>
```

Antes de explicar cada uno de los parámetros que presenta este fichero es conveniente hablar del concepto de referencia. Es habitual que los beans o los componentes web

interactúen a su vez con otros beans. Sin embargo debido a que el nombre JNDI bajo el que se enlazará el interfaz home de ese bean no se conoce hasta el momento del despliegue de la aplicación (hay que recordar que ese mapeo es tarea del responsable del despliegue) se hace necesario la existencia de un mecanismo que le permita a los desarrolladores de beans el declarar que un bean colabora con otro. Y todo esto en fase de desarrollo, es decir, utilizando los ficheros de despliegue estándar y antes de realizar el despliegue.

Por lo tanto, el mecanismo que satisface ésta funcionalidad requerida son las referencias, que no son más que una unión entre un nombre lógico puesto por el desarrollador y un interfaz home. Es decir, es un mecanismo bastante similar al que se realiza en el caso del fichero ejb-jar.xml, en cuanto a que el desarrollador recurre a nombres lógicos, que posteriormente, el encargado del despliegue, se encargará de ir asignando a recursos desplegados mediante la utilización de los ficheros específicos del servidor.

Una referencia EJB se declara usando el elemento <ejb-ref> dentro de un fichero descriptor de despliegue. De esta forma se consigue indicarle a las aplicaciones (clientes o en entorno web) o a los propios beans, que pueden colaborar con otro bean para lo que deberán utilizar la referencia definida dentro de ese elemento. Cada elemento <ejb-ref> describe los interfaces necesarios para que cualquier componente pueda acceder a la referencia de un bean.

Finalmente, hay que mencionar que una referencia EJB tiene su ámbito restringido únicamente a la aplicación que la haya definido, es decir a la aplicación que contenga la declaración en su fichero descriptor de despliegue dentro del elemento <ejb-ref> correspondiente. Por lo tanto, una referencia EJB no es accesible desde cualquier otra aplicación en tiempo de ejecución e incluso otra aplicación podría definir un elemento <ejb-ref> con el mismo nombre <ejb-ref-name> sin que se produzcan conflictos en el sistema de nombrado.

Los elementos hijos o parámetros más importantes del elemento <ejb-ref>, relativos al ejemplo, se enumeran a continuación.

1. Parámetro <display-name>: Elemento que permite especificar un nombre representativo para el item.
2. Parámetro <description>: Elemento opcional que define el propósito de la referencia.
3. Parámetro <ejb-ref-name>: Especifica el nombre de la referencia relativa al contexto. Se recomienda usar como nombre de la referencia ejb/nombre_del_enlace.
4. Parámetro <ejb-ref-type>: Especifica el tipo del bean, por lo que sus posibles valores son Session o Entity.
5. Parámetro <home>: Es el nombre completamente cualificado del interfaz home.
6. Parámetro <remote>: Es el nombre completamente cualificado del interfaz remote.
7. Parámetro <ejb-link>: Es un elemento opcional que crea una referencia a otro bean que se encuentra dentro del mismo fichero ejb-jar.jar. El valor de este elemento será el valor del parámetro <ejb-name> del bean referenciado. El elemento <ejb-link> no puede ser utilizado para referenciar beans de otras aplicaciones (que lógicamente tendrán su propio fichero ejb-jar.jar). Para referenciar beans externos a una aplicación hay que recurrir al nombre JNDI puesto al objeto EJB durante el despliegue de la aplicación usando el elemento <jndi-name> del fichero jboss.xml

Finalmente, la especificación EJB define un fichero descriptor de despliegue opcional como es el de las aplicaciones stand-alone, que también se incluirá en este ejemplo. Este descriptor se guardará en el fichero application-client.xml en el directorio /dd/client del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE application-client PUBLIC
          '-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//EN'
          'http://java.sun.com/dtd/application-client_1_3.dtd'>

<application-client>

        <display-name>HolaClient</display-name>
```

```
        <ejb-ref>

                <description>Referencia para el cliente</description>
                <ejb-ref-name>refClient</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
                <home>examples.HelloHome</home>
                <remote>examples.Hello</remote>

        </ejb-ref>

</application-client>
```

Como se puede apreciar en el código, este descriptor utiliza los mismos elementos <ejb-ref> que el descriptor de despliegue web.xml.

4.1.5.5. Cliente stand-alone

El cliente que se va a desarrollar está diseñado para ejecutar completamente al margen del servidor de aplicaciones, aunque en realidad por cuestiones de infraestructura, como ya se indicó al comienzo de este ejemplo, tanto cliente como servidor residirán en el mismo espacio de direcciones. Sin embargo, es conveniente aclarar que éste cliente podría ejecutar en una MVJ (maquina virtual Java) diferente e incluso en un *host* físicamente distinto. Obviamente esto es posible debido a que EJB es una arquitectura distribuida.

La clase del cliente debe contener el código que permita realizar las llamadas a los métodos remotos definidos y exportados por lo beans. Los clientes stand-alone de los beans habitualmente se encuentran en otro espacio de direcciones distinto al de los propios beans, pero la tecnología EJB permite que las llamadas a los métodos de esos beans se hagan como si estuviesen en local y de forma transparente. Por lo tanto, lo primero que debe hacer un cliente es encontrar un servidor de nombres JNDI y un determinado protocolo que le permita conocer la localización de los objetos home de los beans que necesita.

Para encontrar el servidor de nombre hay que tener en cuenta los siguientes detalles. Bien es cierto que en la práctica, en la mayoría de las ocasiones, el servidor de nombres y el servidor de aplicaciones residen en la misma máquina. Pues bien, esta

circunstancia hace que la búsqueda del servicio de nombre sea más sencilla cuando hablamos de aplicaciones cliente en entorno web. En este tipo de aplicaciones, que se podrían definir como auto-contenidas porque todos sus recursos son desplegados y residen en el servidor (servlets, paginas JSP e incluso beans) y el cliente simplemente accede vía HTTP, el servidor de aplicaciones asume como servidor de nombres el suyo propio, que lógicamente conoce y gestiona.

Sin embargo, en el caso de aplicaciones stand-alone que pueden residir en maquinas distintas se requiere darles una ayuda para encontrar ese servicio. La distintas posibilidades de acceder a ese servicio de nombres ya fueron comentadas anteriormente y para el caso particular de ejemplo se recurrirá a un fichero jndi.properties. Este fichero permite desacoplar al máximo el código del cliente de la localización del servicio de nombres JNDI, por lo que resulta interesante. El contenido de ese fichero se presenta a continuación y se guardará en el directorio /src del directorio de trabajo

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
```

Este fichero debe contener al menos estas dos propiedades. Por un lado el nombre del interfaz que va a permitir que la aplicación cliente se comunique con el servidor de nombres y por otro la URL de dicho servidor. Estas dos propiedades son específicas del proveedor del servidor de aplicaciones y en el ejemplo anterior son las correspondientes al servidor JBoss.

Nota de seguimiento: un poco de imaginación, algo de tranquilidad y mucha perspectiva

Seguro que alguien estará pensando: “Bien, muy bonito eso de que realmente se pueda distribuir la aplicación distribuida (que por algo será distribuida) si se tuviese la famosa infraestructura. Pero parece que no es el caso. De todas formas, si por alguna casualidad se pudiese ejecutar en una LAN ¿Que habría que hacer?”

Pues bien, en primer lugar habría que modificar en el fichero jndi.properties, visto anteriormente, la propiedad java.naming.provider.url = jnp://localhost:1099 y poner en lugar de *localhost* el nombre del *host* en el que reside el servidor JBoss. Con *localhost* se le está indicado al cliente stand-alone que busque el servidor de nombres en la misma máquina donde reside el cliente y ya se sabe de sobra donde residen ambos. Pero si realmente se va

a distribuir la aplicación a través de una red el cliente estará en otra máquina y habrá que decirle donde debe buscar su servidor de nombre.

Además, si se sigue imaginando un entorno realmente distribuido aparece una de las principales inquietudes con las que se encuentra todo desarrollador de aplicaciones con tecnología EJB. ¿Cómo llegan los stubs al espacio de direcciones del cliente, si resulta que éstos los crea el contenedor en tiempo de despliegue, una vez hechos ya los empaquetamientos?

Pues bien, la respuesta a esta gran pregunta por ahora solamente es: tranquilidad. Aun no es el momento de comentarlo, ya se verá. Lo que si es conveniente comentar a estas alturas es que el mecanismo de llevar los stubs al cliente está íntimamente ligado al atributo asignado a la primera de las propiedades del fichero jndi.properties. Es decir, el interfaz org.jnp.interfaces.NamingContextFactory. Este interfaz es un interfaz especifico de JBoss y pertenece al paquete jnp (JBoss Naming Protocol). Este protocolo JNP de JBoss, al igual que el resto de los que hay en el mercado de otros proveedores deben ser compatibles con el protocolo CORBA IIOP y es quien tiene el secreto de cómo llegan los stubs al lado de los clientes.

Realmente, el como llegan esos stubs al cliente se comentará en profundidad más adelante cuando se vean las tareas de la fase de despliegue. Lo único importante por ahora a ese respecto es saber que los stubs efectivamente llegan al espacio de direcciones del cliente, y mas vale que lleguen, que si no habría que preocuparse de demasiados aspectos de la comunicación remota.

Y ahora si, no se puede olvidar la clase principal del cliente que es quien realmente llama a la lógica de negocio que aportan los beans. Esta clase se guardará en un fichero HolaClient.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Aplicación cliente
 */

import javax.ejb.CreateException;
import javax.naming.*;
import java.rmi.RemoteException;
import java.util.*;

import ejemplos.Hola;
import ejemplos.HolaHome;

public class HolaClient {

        public static void main(String[] args) {

                 try {
                    Context initialContext = new InitialContext();
                    Object obj = initialContext.lookup("HolaClient/refClient
                                                                            ");
                    HolaHome home = (HelloHome)
                                  javax.rmi.PortableRemoteObject.narrow(obj,
                                                            HolaHome.class);
                    Hola hello = home.create();
                    System.out.println(hello.hello());
```

```
                hello.remove();
            } catch (Exception e) {
                    System.out.println(e);
            }

        }
}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

1. El método lookup( ) obtiene una referencia al objeto Home que es la fábrica de objetos EJB Hola. Para conseguir esta referencia se le pasa al método como argumento el nombre JNDI de la referencia con la que el recurso Home ha sido desplegado en el sistema de nombrado del servidor de aplicaciones. Esa referencia se mapea con el nombre JNDI o ubicación real del recurso dentro del servidor de nombres en tiempo de despliegue y se realiza en el fichero jboss-client.xml.

2. Al ser el objeto Home un objeto RMI-IIOP se requiere hacer un tipo especial de cast utilizando el método narrow( ) que convierte el objeto obj de tipo Object en un objeto HolaHome.

3. Se llama al método create( ) de la fábrica de objetos Home para crear una instancia del objeto EJB y posteriormente sobre esa instancia llamar al método de la lógica de negocio hello( ). La llamada al método hello( ) del objeto EJB (que no tiene implementada la funcionalidad requerida) lo que hace es delegar en el objeto de la clase del bean que devuelve el resultado y se imprime por pantalla.

4. Un detalle importante a destacar es que tanto el método create( ) como el método hello( ) son dos métodos que deben ser llamados, al igual que cualquier método en Java, sobre objetos y en ambos casos se llaman sobre interfaces. Entonces cabe preguntarse entonces quien implementa estos interfaces, porque ya se ha dejado claro anteriormente que la clase del bean no es. La respuesta es que quien implementa estos interfaces son los *stubs* que crea el servidor JBoss a partir de los interfaces en tiempo de despliegue. El

stub por tanto contiene los mismos métodos que tiene y expone el bean. Así cuando un cliente llama a un método del bean en realidad llama al método del *stub*, que a su vez llama al método de lado servidor y que, finalmente, delega en el propio método del bean.

5. Finalmente, una vez utilizado el método del bean se llama al destructor del objeto EJB que hace que el contenedor lo elimine.

4.1.5.6. Cliente web

En la aplicación "Hola Mundo!!" que se ha propuesto como ejemplo práctico existe un segundo tipo de cliente que es el cliente web y que se trata en este apartado. En esta aplicación, el cliente web es usado por los usuarios para acceder de forma remota, vía protocolo HTTP, a una pagina JSP cuyo único cometido es devolver al cliente, al igual que en el caso de la aplicación stand-alone, el string "Hola Mundo!!". Por lo tanto el cliente web estará contenido en una pagina JSP que es un documento basado en texto (bien en formato HTML o XML) con elementos JSP y que se construye de forma dinámica.

Para el ejemplo propuesto, el trabajo principal de la pagina JSP es el de presentación al cliente de los datos, que en este caso son únicamente el string "Hola Mundo!!" indicado. Además, esta cadena siempre será la misma, por lo que incluso la generación dinámica de páginas HTML queda como un objetivo fuera del alcance del ejemplo. Hay que recordar de nuevo, que el ejemplo intenta abarcar toda la tecnología EJB y no pretende ser en ningún caso exhaustivo en cuanto al resto de tecnologías como JSP, de la que además, se suponen por parte del lector unos conocimientos mínimos de le permitan avanzar sin mayores esfuerzos.

Sin embargo, si sería conveniente aclarar que en la mayoría de casos reales los clientes web accediendo a través de paginas JSP, por ejemplo, necesitan para la creación de sus contenidos dinámicos delegar muchas tareas de proceso dinámicos en distintos Enterprise JavaBeans. Ejemplos de tareas realizadas por estos beans serían la gestión de las interacciones con la base de datos o el interactuar con otros

componentes de la aplicación para adquirir de ellos otros datos o resultados. En definitiva, con la utilización de estos beans se facilitará la adquisición de unos datos mucho más complejos que se usarán en la creación dinámica de los documentos que visualizará el cliente.

A continuación se enseña el código de la pagina JSP correspondiente al cliente web, que se guardará en el fichero index.jsp en el directorio /web del directorio de trabajo

```
<%--
        Pagina JSP cliente con las declaraciones y el código HTML
        que muestra el string
--%>

<%@ page import="ejemplos.Hola, ejemplos.HolaHome, javax.ejb.*, java.math.*,
javax.naming.*, javax.rmi.PortableRemoteObject, java.rmi.RemoteException" %>

<%!

   private Hola hello = null;

   public void jspInit() {

      try {
         InitialContext ic = new InitialContext();
         Object objRef = ic.lookup("java:comp/env/refWeb");
         HolaHome home = (HolaHome)PortableRemoteObject.narrow(objRef,
                                                   HolaHome.class);
         hello = home.create();

      } catch (RemoteException ex) {
            System.out.println("No se crea hello bean."+ ex.getMessage());
      } catch (CreateException ex) {
            System.out.println("No se crea hello bean."+ ex.getMessage());
      } catch (NamingException ex) {
            System.out.println("NO se ha hecho lookup. "+ ex.getMessage());
      }
   }

   public void jspDestroy() {
         hello = null;
   }
%>

<html>
<head>
    <title>Ejemplo Hola Mundo</title>
</head>
<body bgcolor="white">

<h1><b><center>Saludo</center></b></h1>
<hr>
<p>El saludo: <%= hello.hello() %>, es el string devuelto por el EJB.<p>

</body>
</html>
```

Las cuestiones mas importantes a destacar del código de esta pagina JSP se enumeran a continuación:

1. En primer lugar hay que destacar que todas las sentencias incluidas en el método jspInit y que permiten localizar el objeto Home, crear su instancia e invocar al método de creación son prácticamente idénticas al código que se utilizó en el cliente stand-alone.

2. Sin embargo, una primera diferencia apreciable puede ser el hecho de que en el caso de las aplicaciones en entorno web al constructor del InitialContext no se le pasan argumentos. Hay que recordar que en el caso de la aplicación stand-alone esos parámetros se pasaron vía fichero jndi.properties. Pues bien, si tenemos en cuenta que la mayoría de las aplicaciones web harán uso de componentes que residen ya en el propio servidor de aplicaciones donde reside la aplicación web, es decir, en el mismo espacio de direcciones, no es de extrañar que un InitialContext sin argumentos se asuma como un servicio de nombres del propio servidor de aplicaciones. En cualquier caso, si se necesita que el cliente web utilice un servicio de nombres distinto al del servidor de aplicaciones, al InitialContext se le podía pasar la información de ese sistema de nombrado en código (como argumento en su constructor) dentro del método jspInit.

3. Las aplicaciones en entorno *web* son un claro ejemplo de un cliente (una página JSP en el caso de la aplicación del ejemplo), que se ejecuta en el servidor, y que llama a un bean, también residente en el servidor y más específicamente en el contenedor. Por lo tanto, es evidente que dichas llamadas deben hacerse de forma local ya que en ellas se accede a beans ubicados dentro de la propia máquina virtual del servidor de aplicaciones. Por esta razón en el código de la pagina JSP se utiliza el contexto local *java:comp/env* del sistema de nombrado.

4. Profundizando en la cuestión de que los clientes *web* acceden a los beans de forma local incluso resultaría coherente plantearse a esta alturas si no sería más sencillo utilizar durante las fases de desarrollo de la aplicación los

interfaces locales en vez de los interfaces remotos. La respuesta podría ser que si, efectivamente, pero hay que recordar que en el caso de utilizar interfaces locales, dichos beans no podrían ser accedidos por aplicaciones o clientes remotos. Por lo tanto, en este caso no sería una solución válida ya que la aplicación del ejemplo aprovecha el mismo bean para ser accedido por ambos tipos de clientes.

5. Otro detalle importante es que el ENC o el contexto de nombrado de todos los componentes o beans de una aplicación *web* es declarado de forma global para todos los servlets y las páginas JSP contenidas dentro de dicha aplicación *web*. Esto es así para poder compartir recursos. Es decir, que cualquier recurso de la aplicación *web* (servlet o página JSP) tiene acceso a cualquier componente definido para dicha aplicación.

6. Por otro lado, a pesar de que la aplicación stand-alone y el cliente web acceden al mismo bean, hay que mencionar que se ha considerado conveniente por razones didácticas, que ambas aplicaciones referencien al objeto Home en la llamada al método lookup( ) mediante nombres de referencia diferentes. La aplicación stand-alone referencia al bean con el nombre HolaClient/refClient mientras que la página JSP lo hace ahora con el nombre java:comp/env/refWeb. Esto es posible ya que el nombre JNDI con el que se mapeará el bean en los ficheros descriptores de despliegue específicos de JBoss es el mismo, es decir, es el mismo recurso, aunque el nombre utilizado en cada aplicación para referenciarlo sea distinto. Hay que recordar que los ficheros de despliegue, que incluyen un *nivel de in-dirección*[3], son los que permiten desacoplar el código de los clientes de la localización real de los beans, haciendo mucho más fácil el ensamblaje de los componentes que forman las aplicaciones J2EE.

7. También es importante destacar que las clases que el cliente necesita, como los interfaces remotos y el resto de paquetes que permiten hacer las llamadas

[3] Dice una máxima en sistemas de información: un nivel de in-dirección soluciona cualquier problema en informática.

remotas, se tiene que declarar entre los caracteres <%@ y %>, indicando que es una directiva JSP de importación.

8. Debido a que la localización del interfaz remoto se hace una única vez, el código de esa localización se encuentra dentro de una declaración JSP entre los caracteres <%! Y %> y que contiene los método de inicialización y finalización de la pagina JSP.

9. Después de la declaración se encuentra el código HTML que realmente se mostrará en la pagina al cliente y que contiene una expresión JSP encerrada entre los caracteres <%= y %>. Dentro de esta expresión se hace la llamada al método del bean que devuelve el string indicado para que pueda ser evaluada y mostrada en la página resultado.

10. Finalmente hay que recordar que si estamos utilizando una pagina JSP para el cliente web, esta página no requiere ser compilada antes de hacer el despliegue de la aplicación ya que esta tarea la llevará a cabo el servidor de aplicaciones automáticamente, a través de su contenedor de servlets y JSP´s, cuando se realice la primera petición a dicha pagina JSP. Por el contrario, si el cliente estuviese basado en servlets y no en JSP, si que sería necesario realizar la compilación de las clases correspondientes a los servlets antes de hacer el despliegue.

Nota de seguimiento: el cliente siempre tiene la razón, sus derechos y obligaciones

Seguimos en desarrollo. Hasta la nota de seguimiento anterior estábamos en la fase de desarrollo que le correspondía al proveedor de beans y llegados a este punto, seguimos en fase de desarrollo, pero con el trabajo que le corresponde ahora al ensamblador de la aplicación. Como se comentó anteriormente cuando se vieron las funciones del rol de ensamblador de aplicaciones, éste es el encargado, entre otras cosas, de determinar el plan de ensamblaje de la aplicación, de escribir el código de integración entre los distintos componente de la aplicación e incluso de aportar el código de los clientes que llaman a los componentes. Eso es precisamente lo que se ha hecho hasta este punto, ayudados lógicamente, del profundo conocimiento que se supone tiene el ensamblador de la aplicación que está construyendo. De nuevo es conveniente destacar que la aplicación propuesta no aporta una excesiva dificultad pero en aplicaciones reales es evidente que para llegar hasta aquí, se habrá tenido que pasar por distintas y laboriosas tareas de análisis y diseño de la aplicación distribuida.

Una vez vista la parte servidora de la aplicación es hora de ver la otra parte del sistema. Basándose en una arquitectura cliente/servidor, una aplicación software distribuida no sólo se compone de la parte que da servicio sino que, por el contrario, se compone de la parte que presta el servicio y de la parte que lo consume. Esta última parte, el consumidor de funcionalidad, es el cliente que se acaba de ver en sus dos vertientes: cliente stand-alone y cliente web.

Para ser un buen cliente EJB de una aplicación distribuida y funcionar de forma adecuada se deben cumplir con una serie se derechos y obligaciones. Un cliente tiene el derecho a que se le facilite al máximo su trabajo y poder llamar a las funcionalidades que le brindan los beans con total transparencia en cuanto a su localización. El cliente no debe preocuparse de donde están las cosas, por el contrario debe, y ahí tiene una obligación, saber donde buscar lo que necesita, es decir tiene que conocer su servidor de nombres. Por otro lado, el cliente tiene el derecho a conocer toda la funcionalidad que le brindan los beans para llevar a cabo su labor. Para saber de lo que dispone, el cliente tiene la obligación de conseguir los interfaces remotos que le definen lo que los beans exportan.

Hasta ahora la labor de los desarrolladores de beans (proveedor de beans y ensamblador de aplicaciones) definiendo tanto la parte cliente como la servidora de la aplicación, ha sido bastante manual, sin recurrir aun a ningún tipo de herramientas externas. Pero cuidado, aun no se ha hecho nada palpable. A partir de ahora, se dejan conceptos y definiciones a un lado y se pasa a un mundo más de ejecución y eminentemente práctico. Es ahora cuando se empezará a utilizar la herramienta Ant que facilita enormemente todas las tareas de las fases de desarrollo de una aplicación EJB.

4.1.6. Compilando los fuentes

Una vez terminada la parte conceptual es hora de empezar a realizar las tareas que requiere cualquier aplicación de la tecnología EJB. A continuación se van a llevar a cabo una serie de pasos con la ayuda de la herramienta Ant. Esta herramienta tomará como datos de entrada todos los ficheros fuente descritos hasta ahora para ir ensamblándolos de acuerdo a las necesidades, permitirá hacer el despliegue de la aplicación dentro del servidor de aplicaciones y finalmente permitirá ejecutar los clientes para comprobar que todo funciona correctamente. Cada una de las tareas que se va a ejecutar se explicará en detalle describiendo las cosas que hace y los efectos que produce.

En primer lugar, para ejecutar cualquier tarea, y en particular la tarea de compilación que es la primera y a la que se refiere este apartado, hay que dirigirse en una consola de línea de comandos al directorio de trabajo donde están todos los ficheros fuentes de la aplicación. Será desde este directorio desde donde se van a ejecutar todos los comandos Ant para la creación de la aplicación EJB. Una vez allí, para ejecutar la

tarea de compilación de los ficheros con código Java hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml compile
```

Este comando ejecuta todas las subtareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada compile incluida en el fichero jboss-build.xml que se presentó en el apartado 4.1.4.2. Es decir, busca los ficheros con extensión .java en el directorio /src, los compila y deja los ficheros bytecode con extensión .class en el directorio /build del directorio de trabajo si todo ha ido correctamente. De lo contrario enseñará un mensaje: Build failed. Además, esta tarea depende de la tarea denominada prepare, por lo que automáticamente se ejecuta antes de que se ejecute compile y lo que hace es crear el directorio /build donde se van a dejar los ficheros.

Nota de seguimiento: no va más, hasta aquí llega el trabajo del desarrollador

Con esta tarea de compilación se podría afirmar que se acaban las tareas asignadas al desarrollador de beans. Se acaba con las tareas de la fase de desarrollo realizado por el proveedor de beans y las tareas del ensamblador de aplicaciones y se empieza con las tareas asignadas al encargado del despliegue de la aplicación. Sin embargo, hay que aclarar que no existe una línea clara que divida las labores de los desarrolladores y de los encargados del despliegue. Esto es debido a que el ensamblador de aplicaciones, por ejemplo, se encarga de aportar los ficheros descriptores de despliegue que luego pueden ser modificados por quien despliega de la aplicación para afinar, por ejemplo, ciertas funcionalidades específicas del contenedor que deban estar de forma explicita en esos ficheros de despliegue. Por lo tanto, es posible que haya cierto solapamiento en las tareas de ambos dependiendo de cada caso particular. Finalmente, como es lógico cualquier modificación de alguno de los ficheros aportados en alguna fase anterior, implicaría la creación de nuevo de todos los ficheros de empaquetamiento que debe crear el encargado del despliegue.

4.1.7. Ficheros específicos del servidor JBoss

La tecnología J2EE esta diseñada para que el código sea totalmente independiente del servidor de aplicaciones en el cual sea desplegado. Los descriptores de despliegue para los EJB´s (ejb-jar.xml), para las aplicaciones web (web.xml) o para las aplicaciones stand-alone (application-client.xml) son estándar y no cambian de un

contenedor J2EE a otro. Sin embargo, hay ciertas cosas específicas de los servidores de aplicaciones, y en particular del JBoss, que requieren ser especificadas en ficheros descriptores de despliegue específicos y en tiempo de despliegue, que es cuando se conoce al proveedor del contenedor.

Estos ficheros específicos del servidor de aplicaciones existen porque la especificación de la tecnología EJB no puede cubrir de forma exhaustiva todas áreas relacionadas con el rendimiento de las aplicaciones, como pueden ser los algoritmos de pooling o los de clustering. Los ficheros específicos del servidor de aplicaciones se utilizarán para declarar esos servicios o funcionalidades específicas que requieren los beans y que son proporcionados por el middleware del servidor de aplicaciones que conoce sus propias políticas y sabe como resolver esos problemas de forma particular. Por lo tanto, estos ficheros específicos, a diferencia de los ficheros estándar vistos en el apartado anterior, no son portables y pueden utilizar cualquier formato XML, binario e incluso soportes como una base de datos.

Para el caso particular del ejemplo propuesto, debido a su simplicidad, los ficheros específicos descriptores de despliegue aun no definirán servicios de middleware porque no son necesarios para cubrir las funcionalidades que se requieren. Estas definiciones, sin embargo, se verán en ejemplos posteriores cuando se vean tipos de beans más complejos y se entre en el detalle de esos servicios. Por el momento lo más importante es saber que el servidor de aplicaciones JBoss utiliza el formato XML para definir sus ficheros específicos y de esta forma se necesitará para el ejemplo propuesto un fichero XML por cada uno de los descriptores de despliegue estándar que se hayan utilizado. A continuación se presenta el código y las características de esos ficheros específicos.

En primer lugar, el fichero descriptor de despliegue EJB de JBoss, o jboss.xml, proporciona el mapeo entre el nombre lógico del bean (es decir, el nombre JNDI del componente que el desarrollador de beans declaró de forma independiente en el fichero ejb-jar.xml) y el nombre JNDI actual del recurso físico que realmente se desplegará en el servidor de aplicaciones. Es decir, que el nombre JNDI bajo el cual se enlaza el interfaz home de un EJB, es una decisión tomada en tiempo de

despliegue, hecha por el responsable de ese despliegue, y que queda reflejada de forma explícita en el fichero jboss.xml.

A continuación se enseña el código correspondiente al fichero descriptor de despliegue EJB para JBoss, que se guardará en el fichero jboss.xml en el directorio /dd/ejb del directorio de trabajo.

```
<!DOCTYPE jboss PUBLIC
          "-//JBoss//DTD JBOSS 4.0//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">

<jboss>
    <enterprise-beans>
        <session>

            <ejb-name>Hola</ejb-name>
            <jndi-name>ejb/examples/HelloHome</jndi-name>

        </session>
    </enterprise-beans>
</jboss>
```

Los elementos más importantes a destacar del código de este descriptor de despliegue se enumeran a continuación:

1. Elemento <ejb-name>: En JBoss el mecanismo de mapeo se implementa haciendo coincidir los parámetros <ejb-name> tanto del descriptor de despliegue ejb-jar.xml como del descriptor de despliegue jboss.xml. Por lo tanto, el valor de este parámetro debe ser el que mismo en ambos casos.

2. Elemento <jndi-name>: Declara el nombre JNDI completamente cualificado del recurso desplegado. Ante la ausencia de este elemento en la especificación del fichero descriptor de despliegue jboss.xml, el interfaz Home del bean se enlazará con el nombre definido en el parámetro <ejb-name> del fichero ejb-jar.xml.

En segundo lugar, la aplicación del ejemplo requiere un fichero descriptor de despliegue web de JBoss, o jboss-web.xml, que al igual que el descriptor anterior proporciona un mapeo entre un nombre lógico y el destino final del componente. En

este caso el mapeo es entre el nombre lógico de la referencia que va a utilizar la aplicación web (declarado en el fichero web.xml) y el recurso que realmente se desplegará en el servidor de aplicaciones.

A continuación se enseña el código correspondiente al fichero descriptor de despliegue web para JBoss, que se guardará en el fichero jboss-web.xml en el directorio /dd/web del directorio de trabajo.

```
<!DOCTYPE jboss-web PUBLIC
          "-//JBoss//DTD Web Application 2.4//EN"
          "http://www.jboss.org/j2ee/dtd/jboss-web_4_0.dtd">

<jboss-web>
    <ejb-ref>

        <ejb-ref-name>refWeb</ejb-ref-name>
        <jndi-name>ejb/examples/HelloHome</jndi-name>

    </ejb-ref>
</jboss-web>
```

Los elementos más importantes a destacar del código de este descriptor de despliegue se enumeran a continuación:

1. Elemento <ejb-ref-name>: Este elemento se corresponderá con el elemento <ejb-ref-name>, bien del fichero descriptor de despliegue estándar ejb-jar.xml o del web.xml.

2. Elemento <jndi-name>: Especifica el nombre JNDI completamente cualificado de la ubicación del interfaz home durante el despliegue de la aplicación.

Finalmente, como es lógico, la aplicación requiere un tercer fichero descriptor de despliegue de las aplicaciones stand-alone para JBoss, que como ya se comentó tenia carácter opcional. El objetivo de este fichero es exactamente el mismo que el del fichero jboss-web.xml, pero ahora para el entorno de aplicaciones stand-alone y no de aplicaciones web.

A continuación se enseña el código correspondiente al fichero descriptor de despliegue stand-alone para JBoss, que se guardará en el fichero jboss-client.xml en el directorio /dd/client del directorio de trabajo.

```
<!DOCTYPE jboss-client PUBLIC
       "-//JBoss//DTD Application Client 4.0//EN"
       "http://www.jboss.org/j2ee/dtd/jboss-client_4_0.dtd">

<jboss-client>
   <ejb-ref>

      <ejb-ref-name>refClient</ejb-ref-name>
      <jndi-name>ejb/examples/HelloHome</jndi-name>

   </ejb-ref>
</jboss-client>
```

Como se puede apreciar en el código, este descriptor utiliza exactamente los mismos elementos que el descriptor de despliegue jboss-web.xml y que ya han sido explicados anteriormente.

4.1.8. Empaquetando el EJB

Una vez escritos todos los ficheros necesarios para la aplicación EJB que se está desarrollando es necesario ir empaquetando todos esos recursos de forma ordenada y conjunta en los diferentes ficheros de empaquetamiento (.jar, .war o .ear) que se explicarán a continuación y que son, realmente, los que se van a desplegar en los servidores de aplicaciones.

El objetivo fundamental de la creación de este paquete EJB es el de disponer de un recurso fácilmente portable que incluya toda la información que se necesita para realizar el despliegue de los EJB en un servidor de aplicaciones y poder así, dar el servicio que prestan los beans incluidos en ese paquete. Por lo tanto, hay que considerar este paquete como el núcleo de la parte servidora de la aplicación EJB, ya que es el paquete que cuando se despliega en el servidor, hace que el contenedor, a través de sus herramientas, cree realmente todos los recursos EJB que necesita la aplicación para funcionar correctamente. Estos recursos se crearán a partir de los interfaces incluidos en el fichero de empaquetamiento e incluyen desde los objetos

asociados a dichos interfaces, así como los stubs, los skeletons y demás recursos. Por su parte, los clientes de los beans, como es lógico, se limitarán a tener sólo los interfaces remoto y home que hacen referencia a los beans, pero en ninguna caso generan nada en la parte servidora.

Este empaquetamiento se puede realizar de forma manual utilizando la herramienta jar, o como es el caso, con la utilización de la herramienta de desarrollo Ant. Para ejecutar esta tarea de empaquetamiento del fichero Ejb-jar.jar de la aplicación hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ ant -f jboss-build.xml package-ejb
```

Este comando ejecuta todas las subtareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada package-ejb incluida en el fichero jboss-build.xml que se presentó en el apartado 4.1.4.2. Es decir, se creará un fichero .jar en el directorio creado para tal efecto dentro del directorio de trabajo. Este fichero, por un lado, contendrá el código de los beans (con los distintos interfaces y la clase del bean) que siempre deben estar guardados en los directorios apropiados teniendo en cuenta los paquetes a los que pertenecen estas clases, y por otro lado, contendrá los ficheros descriptores de despliegue ejb-jar.xml y jboss.xml dentro del directorio META-INF.

Además de estos ficheros, el fichero ejb-jar.jar, al igual que el resto de ficheros de empaquetamiento, incluye de forma automática (por el mero hecho de ser un fichero jar) un fichero denominado MANIFEST. Este fichero estará ubicado también dentro del directorio META-INF y su misión es simplemente listar los ficheros incluidos en el fichero de empaquetamiento.

4.1.9. Empaquetando el cliente

El siguiente paso en la fase de despliegue dentro del desarrollo de la aplicación EJB es la creación del fichero app-client.jar correspondiente a la aplicación stan-alone. Por

lo tanto, en este fichero de empaquetamiento se deben guardar únicamente las clases particulares que requiere el cliente para ejecutar, así como los interfaces EJB de la aplicación, las clases helper, los stubs de los objetos remotos y los ficheros descriptores de despliegue correspondientes, como el application-client.xml o el fichero especifico del proveedor del servidor de aplicaciones JBoss llamado jboss-client.xml.

Los interfaces EJB deben estar en el fichero .war ya que las clases clientes que hacen las llamadas a los beans hacen referencia a dichos interfaces. Además, hay que aclarar que, si bien es cierto que los interfaces EJB son necesarios en este fichero de empaquetamiento, la clase del bean nunca lo será, ya que hay que recordar de nuevo, que los clientes nunca acceden a los beans directamente sino que lo hacen a traves del contenedor y los objetos home. Finalmente, destacar que en los ficheros descriptores de despliegue de la aplicación stand-alone estarán declarados solo los beans que van a colaborar en dicha aplicación.

Para ejecutar la tarea de creación del fichero app-client.jar de la aplicación atand-alone hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml package-client
```

Este comando ejecuta todas las subtareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada package-client incluida en el fichero jboss-build.xml que se presentó en el apartado 4.1.4.2. Es decir, crea el fichero app-client.jar con el descriptor de despliegue application-client.xml y el descriptor jboss-client.xml extraídos ambos del directorio dd/client y los guarda dentro del directorio META-INF. Incluye las clases correspondientes al cliente y a los interfaces EJB de la aplicación web ya compilados en el directorio /build y finalmente, incluye el fichero de propiedades JNDI, que necesitará la aplicación cliente para conocer donde reside el servicio de nombres.

4.1.10. Empaquetando la aplicación WEB

El siguiente paso en el desarrollo de la aplicación EJB es la creación del fichero web-client.war correspondiente a la aplicación web. En este fichero se guardará tanto el front-end de la aplicación, como aquellos recursos propios de los componentes EJB y que requiere el cliente web para realizar las llamadas, como los interfaces EJB y los descriptores de despliegue correspondientes.

Es importante destacar que en el caso de aplicaciones en entorno web que utilizan páginas JSP también son necesarios los interfaces EJB en el paquete web del cliente. Esto es debido a que al realizarse la creación de la clase del servlet asociada a la página JSP en la primera llamada, se realiza el proceso de compilación de dicha página que requiere los interfaces EJB.

De esta forma el fichero .war contendrá todos los componentes web (JSP´s, servlets, imágenes, etc) que estén contenidos en el directorio /web del directorio de trabajo y que serán añadidos al fichero .war sin ninguna modificación. Por otro lado se añadirá un directorio WEB-INF que contendrá todos los ficheros que no son accedidos por el browser del cliente, que son específicos de la parte del servidora y, lógicamente, parte de la aplicación web. Ejemplos de estos últimos son los descriptores de despliegue web.xml y jboss-web.xml, ficheros de clases, como servlets e interfaces EJB e incluso otros ficheros .jar o descriptores de etiquetas JSP que también pueda necesitar la aplicación.

Para ejecutar la tarea de creación del fichero web-client.war de la aplicación web hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml package-web
```

Este comando ejecuta todas las subtareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada package-web incluida en el fichero jboss-build.xml que se presentó en el apartado 4.1.4.2. Es decir, crea el fichero web-client.war con el descriptor de despliegue web.xml extraído del directorio dd/web, incluye la pagina JSP sacada del directorio /web del directorio de trabajo y lo incluye en el fichero .war. Incluye el descriptor de despliegue jboss-web.xml del directorio

/dd/web, y finalmente, incluye los interfaces EJB de la aplicación web ya compilados en el directorio /build.

4.1.11. Ensamblando el EAR

Finalmente, como ultimo paso de empaquetamiento dentro de las tareas típicas de la fase de despliegue de la aplicación, esta el ensamblar la aplicación entera. El fichero EAR (Enterprise Archive) resultado de la ejecución de este paso recoge la aplicación entera, conteniendo el módulo EJB ejb-jar.jar, el módulo cliente app-client.jar y el módulo web-client.war. Además este fichero de empaquetamiento incluye un fichero descriptor adicional llamado application.xml que declara los distintos módulos que componen la aplicación.

Hay que aclarar que es posible realizar el despliegue por separado de los distintos módulos que componen el fichero .ear, pero este fichero proporciona una entidad única que hace más fácil su portabilidad. Por lo tanto, éste será en definitiva, el fichero que se despliegue en el servidor de aplicaciones con el nombre representativo dado en fase de despliegue.

Para ejecutar la tarea de creación del fichero Hello.ear de la aplicación EJB hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml assemble-app
```

Este comando ejecuta todas las subtareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada assemble-app incluida en el fichero jboss-build.xml que se presentó en el apartado 4.1.4.2. Es decir, crea el fichero Hello.ear con el descriptor de despliegue application.xml extraído del directorio /dd, e incluye todos los ficheros de empaquetamiento que se encuentren en el directorio /jar del directorio de trabajo.

El contenido del descriptor de despliegue application.xml, que declara todos los módulos J2EE incluidos dentro de la aplicación, se presenta a continuación y se guardará en el directorio /dd del directorio de trabajo.

```
<!DOCTYPE application PUBLIC
         "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
         "http://java.sun.com/dtd/application_1_3.dtd">

<application>
    <description>Application description</description>
    <display-name>Hello</display-name>

    <module>
        <ejb>hello-ejb.jar</ejb>
    </module>

    <module>
        <java>app-client.jar</java>
    </module>

    <module>
        <web>
            <web-uri>web-client.war</web-uri>
            <context-root>hello</context-root>
        </web>
    </module>

</application>
```

Como elemento más importante a destacar del código de este descriptor de despliegue de la aplicación hay que mencionar que, a parte de declarar cada uno de los módulos J2EE de los que se compone la aplicación, en el caso del módulo de la aplicación *web* mediante el elemento <context-root> se especifica la raíz del contexto asociada dicha aplicación *web*.

Nota de seguimiento: JAR, WAR, EAR, empaquetando que es gerundio

Este, sin duda, es el mundo del encargado de realizar el despliegue de la aplicación. Este es el momento, siguiendo las instrucciones de proveedores de beans y ensambladores de aplicaciones, de resolver dependencias externas, de especificar parámetros de seguridad o de asignar atributos de transacciones.

Y una vez hecho todo este trabajo, será el momento de que se cree el Enterprise Archive (EAR) de la aplicación para desplegarlo luego en el servidor de aplicaciones. Por lo tanto, el uso de ficheros .ear permite ensamblar distintas aplicaciones usando los mismos módulos J2EE sin necesidad de código extra y preocupándose simplemente de agruparlos convenientemente de acuerdo a las necesidades de la aplicación.

En el caso de la aplicación de ejemplo se han llegado a crear hasta 4 paquetes. Un JAR para los EJB, otro JAR para un cliente, un WAR para un cliente web y finalmente, un EAR para la

aplicación entera. Pero realmente, ¿Es necesario hacer tanto paquete?. La respuesta es que si. Como ya se comentó en capítulos anteriores cuando se habló de los ficheros de empaquetamiento de los módulos J2EE éste es un mecanismo que favorece la portabildad y la reutilización de módulos. Además, son paquetes especializados en su tarea y cada uno de ellos cumple su misión perfectamente.

Bueno, vale. Pero ahora surge la siguiente cuestión. Se admite como coherente y válida la idea de crear un paquete EAR que se desplegará en el servidor y que ese EAR agrupe al módulo EJB y al módulo WAR. Al fin y al cabo, por un lado el módulo EJB contiene "cosas" relacionadas con los EJB´s y éstos va a residir en el servidor, y por otro lado, el módulo WAR contiene otras "cosas" relacionadas con el mundo web y también debe estar en el servidor. Pero que pinta el cliente stand-alone en el EAR?

Si se asume de nuevo el escenario perfectamente distribuido: ¿El módulo del cliente stand-alone no debería ir en la máquina cliente y punto, sin incluírsele en el EAR? Pues no. El cliente debe estar también en el EAR. ¿Por qué?. Pues porque el paquete app-client.jar tiene en su interior "cosas" que tiene sentido para el cliente y "cosas" que tienen sentido para el servidor. Así que, a no ser que se prefieran crear dos ficheros de empaquetamiento distintos para distribuirlos cada uno en su sitio apropiado, no queda más remedio que desplegar el mismo paquete en ambos sitios.

Y aunque se opine lo contrario hay que reconocer que es una solución bastante sencilla y válida. Sino habría que ocuparse de unas cuantas incongruencias que actualmente se cometen. Un ejemplo: la clase principal del cliente sólo tiene sentido en el lado del cliente y es absurdo desplegarla en el servidor, y se hace. Otro ejemplo más: un fichero descriptor de despliegue, tal y como su nombre indica, es para que sea leído por el servidor de aplicaciones en tiempo de despliegue y no tienen sentido en el lado del cliente y allí también está ese fichero. Uno más: ¿Que pinta el fichero jndi.properties en el lado servidor en un aplicación stand-alone?. Bueno vale, ya no más, aunque seguro que las hay. En definitiva, demasiadas cosas de las que ocuparse.

Finalmente, si que resultaría útil como caso práctico y aclaratorio, el probar a no hacer el despliegue del módulo app-client.jar en el EAR y hacerlo sólo en el lado cliente. Así se verá que pasa cuando el cliente realiza la operación de lookup en el servicio de nombres con el nombre lógico declarado en el descriptor de despliegue. ¿Que pasa? Excepcion: javax.naming.NameNotFoundException: HolaCliente not bound. Lógico, no hay descriptores de despliegue del módulo cliente en el lado servidor que permitan hacer el mapeo correspondiente.

4.1.12. Desplegando la aplicación

Por fin ha llegado el momento de desplegar el bean dentro del contenedor EJB del servidor de aplicaciones JBoss. Cuando se despliega un fichero .ear dentro de un contenedor suceden las siguientes acciones:

1. Los ficheros de empaquetamiento eras, jars o wars son verificados. El contenedor verifica que las clases de los beans, los interfaces remotos, los ficheros descriptores de despliegue y demás componentes de la aplicación sean válidos. Cualquier servidor de aplicaciones del mercado cuenta con herramientas que reportan de forma mas o menos inteligente errores en fase de

despliegue como el siguiente: “Necesita definir un método ejbCreate( ) en el bean”.

2. Las herramientas del contenedor generan los objetos Ejb y los objetos Home a partir de los interfaces remotos EJB.

3. Las herramientas del contenedor generan cualquier objeto RMI-IIOP necesario para la comunicación remota como pueden ser los stubs y los skeletons a partir de los interfaces remotos EJB.

Para ejecutar la tarea de despliegue del fichero Hello.ear en el servidor de aplicaciones hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml deploy
```

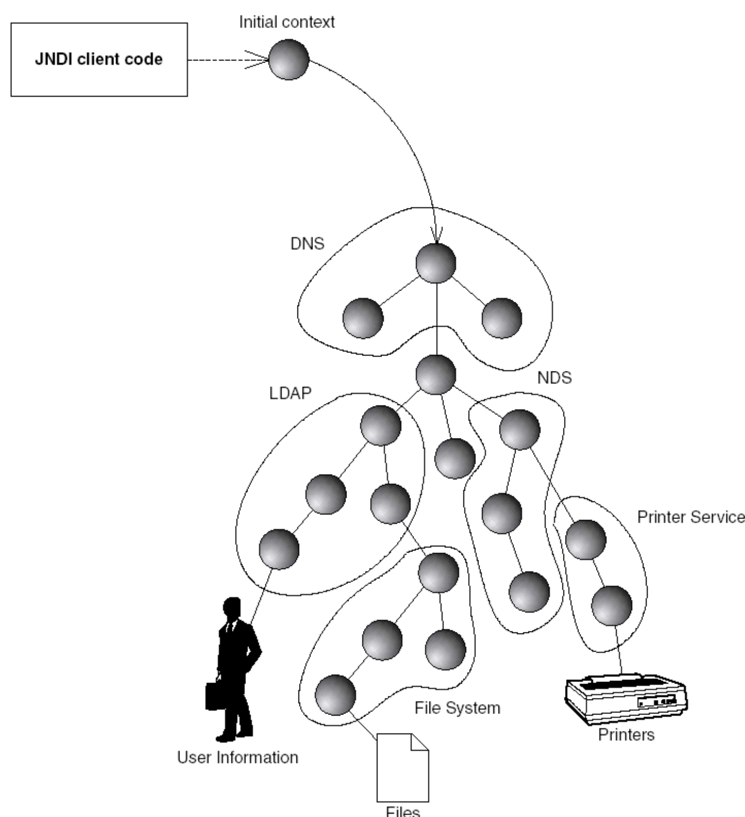
Este comando ejecuta todas las subtareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada deploy incluida en el fichero jboss-build.xml que se presentó en el apartado 4.1.4.2. Es decir, esta tarea lo único que hace es ejecutar en primer término la tarea assemble-app ya que depende de ella, y posteriormente, copia el fichero Hello.ear desde el directorio jar del directorio de trabajo a el directorio de despliegue del servidor de aplicaciones ubicado en el directorio deploy.

Finalmente, hay que destacar que a partir del momento de despliegue del fichero ejb-jar.jar, incluido como se ha visto dentro del fichero .ear, se puede considerar que los EJB incluidos en ese fichero están disponibles para ser llamados desde cualquier cliente. Esto es debido a que durante el despliegue de la aplicación es cuando produce la publicación de los interfaces home en el sistema de nombrado del servidor de aplicaciones. Por lo tanto, en el caso particular del ejemplo, se puede comprobar como justo después de realizar el despliegue el objeto Home del bean del ejemplo está publicado en el servidor de nombre. Para comprobarlo basta con abrir la JMX

Console[4] del servidor JBoss, acceder al servicio JNDIView y dentro ya del servicio invocar al método list ( ) que proporciona una vista con las entradas de los recursos enlazados en el sistema de nombrado.

Para acceder a la consola JMX de servidor JBoss basta con introducir en un navegador web la URL de la página de bienvenida del servidor: http://localhost:8080 y una vez allí pinchar en el link JMX Console. Obviamente, para poder ver la JMX Console debe estar arrancado convenientemente el servidor de aplicaciones, si no lo estaba ya. Y finalmente, recordar que el despliegue de las aplicaciones puede hacerse en caliente, es decir, sin necesidad de volver a re-arrancar el servidor cuando haya que hacer un nuevo despliegue.

En la siguiente figura se muestra el aspecto que podría tener el árbol JNDI del servidor de aplicaciones, después de llevar a cabo el despliegue de la aplicación del ejemplo, incluyendo la nueva entrada de acuerdo a los descriptores de despliegue suministrados.



[4] JMX Console es la consola de gestión del *framework* Java Management Extensions que permite definir componentes conectables o MBeans (Managed Beans) que dan servicio a la arquitectura JBoss.

Figura 9: Namespaces para JNDI

Nota de seguimiento: este ya no es mi problema, pero interesa

Como se acaba de ver en el apartado de despliegue de la aplicación, una vez incluido el fichero .ear en el sitio adecuado del servidor de aplicaciones se desencadenan una serie importante de acciones realizadas por las herramientas del contenedor. Se verifican ficheros, se publican nombres, se crean objetos y lo que más preocupa, porque se había quedado en el tintero: se crean los stubs y los skeletons.

Ahora si es el momento de responder a la pregunta: ¿Cómo llegan los stubs al espacio de direcciones del cliente, si resulta que éstos los crea el contenedor en tiempo de despliegue, una vez hechos ya los empaquetamientos? Obviamente la pregunta tiene todo el sentido del mundo y más teniendo en cuenta que cliente y servidor pueden residir en máquinas diferentes. La respuesta se basa en dos posibles soluciones. En primer lugar se pueden copiar manualmente los stubs a la máquina del cliente una vez generados por el contenedor, o por el contrario, podemos confiar en la descarga dinámica de clases. Como ya se avanzó en una nota de seguimiento anterior el protocolo JNP, propietario de JBoss, es el secreto para solucionar esta circunstancia. Cuando el cliente utiliza este protocolo JNP para el servidor de nombres, los stubs son automáticamente transferidos al cliente como parte de la ejecución del método lookup( ) realizada por el cliente.

Así de simple, los stubs y skeletons se generan en tiempo de despliegue en el servidor y se envían al cliente para que este último no se tenga que preocupar de distribuir esos objetos y se olvide completamente de todos problemas relacionados con las conexiones de red en las llamadas remotas.

Enhorabuena, se han terminado todos los pasos en el desarrollo de una aplicación distribuida utilizando la tecnología EJB. Ahora solo queda ejecutar y se podrá hacer distribuyendo o sin distribuir, con sistema de nombrado local o global, con cliente stand-alone o cliente web, como se quiera. Se han tocado todos estos temas y se esta en disposición de ponerlos ahora en práctica. Lo único que se requiere para que todo funcione y se reciban los resultados esperados es haber seguido las explicaciones del ejemplo, haberlas entendido, tener una perspectiva global y clara de lo que se está haciendo y como no, tener esa pizca de suerte que nunca está de más. Animo que solo es el primer ejemplo.

4.1.13. Ejecutando Hola Mundo!!

Por fin ha llegado el momento de la verdad. Ahora si se podrán ejecutar las aplicaciones cliente y comprobar que los componentes EJB desplegados funcionan correctamente. La única complicación en este paso surge por la necesidad que tiene la aplicación cliente stand-alone, y solo ella, de una cantidad importante de clases que den soporte a las operaciones de comunicación. Este mismo problema no existe en el caso del módulo cliente web o incluso en el módulo EJB, ya que ambos residen en la misma máquina que el servidor de aplicaciones JBoss y se asume que todas las clases

antes mencionadas se encuentran correctamente configuradas ya en el variable de entorno classpath.

Por lo tanto, a continuación se verán por separado las características particulares en la ejecución de cada uno de los clientes.

4.1.13.1. Ejecutando el cliente stand-alone

Como ya se ha comentado, el cliente stand-alone es diferente, bien porque realmente se encuentre distribuido en una máquina distinta de donde está el servidor de aplicaciones JBoss, o bien porque a pesar de residir en la misma máquina conceptualmente se considera fuera del mismo. Por lo tanto, requiere de un tratamiento especial. Necesita conocer y tener acceso, por ejemplo, a las clases JNDI de JBoss, a las clases que permiten la descarga dinámica de los stubs, a los protocolos de red involucrados en la comunicación remota e incluso a las clases de la API de la tecnología EJB.

Por todos estos motivos se necesita, a la hora de ejecutar el cliente stand-alone, de la ayuda de la herramienta Ant. Para ejecutar la tarea de ejecución del cliente se utiliza el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml run-client
```

Este comando ejecuta todas las subtareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada run-client incluida en el fichero jboss-build.xml que se presentó en el apartado 4.1.4.2. Es decir, esta tarea ejecuta utilizando la máquina virtual Java mediante el comando java la clase HolaClient que pertenece al paquete app-client.jar ubicado en el directorio /jar del directorio de trabajo. Además, para dicha ejecución amplia la variable classpath con todas los ficheros .jar incluidos en al directorio /client del directorio HOME de la instalación de servidor de aplicaciones JBoss. Por lo tanto, la aplicación ahora si que tendrá acceso a clases empaquetadas en ficheros .jar como jnp-client.jar o jboss-j2ee que tienen las clases comentadas anteriormente. Obviamente, este es el caso particular para el ejemplo

propuesto, en el que ya se sabe que el cliente residirá en la misma máquina donde está el servidor de aplicaciones, pero ese elemento path cambiará en cualquier otra circunstancia.

Finalmente, hay que destacar que tanto los interfaces EJB del componente como el fichero de propiedades jndi.properties se encuentran ya dentro del paquete app-client.jar, por lo que al ejecutar con java la clase HolaClient desde el propio paquete esos recursos son accesibles.

En el lado cliente, una vez ejecutado la aplicación, lo que se debe visualizar en la consola de comandos desde la que se lanzó la máquina virtual Java debe ser la siguiente secuencia:

```
[java] Hola Mundo!!
```

En el lado servidor, una vez ejecutado la aplicación, lo que se debe visualizar en la consola de *log* del JBoss debe ser una secuencia similar a la siguiente:

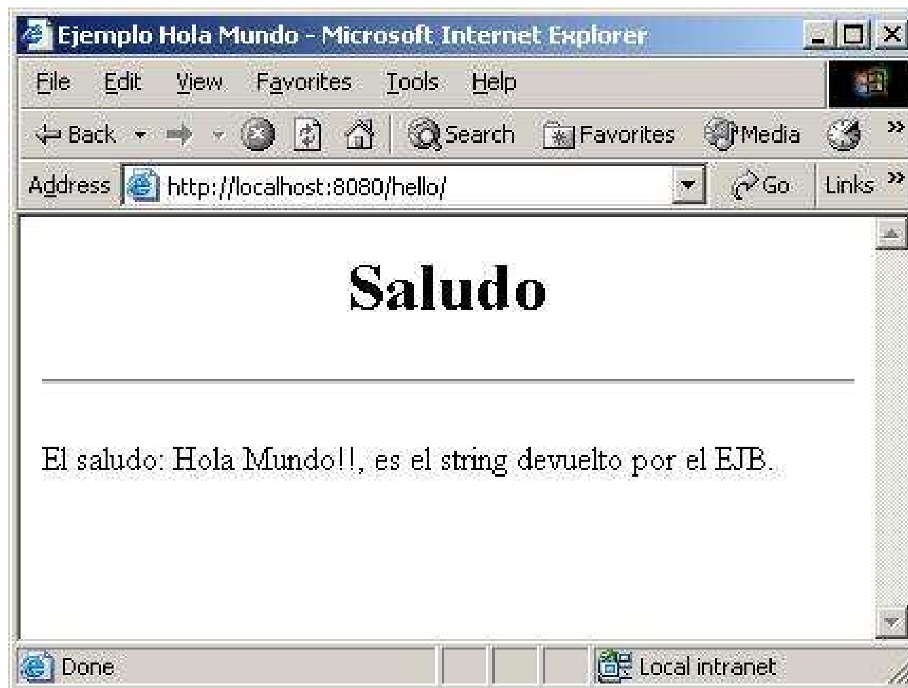
```
00:31:30,218 INFO  [STDOUT] ejecutando ejbCreate()
00:31:30,238 INFO  [STDOUT] ejecutando hola()
```

Como se puede apreciar el contenedor en primer lugar llama al método create( ) del bean para posteriormente llamar al método hola( ).

4.1.13.2. Ejecutando el cliente web

La ejecución de este cliente es mucho más sencilla que la del cliente stand-alone. Para ejecutar en este caso basta con utilizar un navegador web, tener obviamente el servidor de aplicaciones JBoss arrancado correctamente, poner la siguiente URL: http://localhost:8080/hello/index.jsp y voila!!

El lado cliente en este caso se conecta vía HTTP al servidor, solicita la descarga de una página dinámica HTML que genera la pagina JSP y visualiza en su navegador el siguiente resultado:



Navegador con resultado

En el lado servidor del cliente stand-alone lo que se debe visualizar en el *log* del servidor de aplicaciones JBoss debe ser una secuencia bastante similar a la anterior. Esto se debe, obviamente, a que ambas aplicaciones hacen uso del mismo bean. Sin embargo, hay que mencionar que puede darse el caso de que no se ejecute el método ejbCreate( ). Esta situación se puede dar si cuando el cliente hace la llamada al método hello( ) del bean, éste ya estuviese creado. Por ejemplo si se ha ejecutado previamente el cliente stand-alone y aún el contenedor no ha ejecutado el método ejbRemove( ).

```
00:38:10,536 INFO  [STDOUT] ejecutando hola()
```

5. Profundizando en la arquitectura EJB

Bienvenidos al mundo real del desarrollo de aplicaciones distribuidas utilizando la tecnología EJB. Hasta este momento se han sentado unas bases suficientemente sólidas que permitirán al lector seguir profundizando en todo el conocimiento que aún no se ha tratado de esta tecnología de sistemas distribuidos. Hasta ahora se ha planteado un ejemplo práctico bastante simple, que como ya se dijo, pretendía ser más didáctico que representativo. Es hora de empezar a estudiar problemas más cotidianos, reales y, por lo tanto, irremediablemente más complicados.

Para ello será necesario saber, no sólo como se ejecuta una aplicación distribuida sencilla, sino que será necesario saber, por ejemplo, que tipos de EJB´s define la especificación EJB para elegir el que mejor se adapte a las necesidades de la aplicación, saber de qué servicios se puede aprovechar un desarrollador de aplicaciones distribuidas y quien los debe gestionar, saber qué tecnologías de las existentes en la distribución J2EE es necesario conocer en profundidad para utilizar esos servicios, o finalmente, saber como se implementa todo esto en un servidor de aplicaciones como JBoss.

5.1. Tipos de componentes EJB

La especificación de la tecnología EJB define tres tipos básicos de beans: beans de sesión, beans de entidad y los beans manejados por mensaje. Sin embargo, en un primer momento nos vamos a centrar en los dos primeros tipos incluidos desde la especificación EJB 1.1. La especificación EJB 2.0 posterior, incluyó el soporte a los beans manejados por mensaje pero su estudio se pospone al apéndice que se encuentra al final del libro.

Por lo tanto, este capítulo y los siguientes se centrarán en el estudio de estos beans. Se hablará de los beans de sesión que representan procesos o tareas que actúan como agentes que realizan tareas en beneficio de la aplicación cliente y de los beans de entidad que contiene los datos que necesita esa aplicación y que proporcionan los

métodos de acceso a los mismos. Efectivamente, es esto lo que se va a estudiar a partir de hora. A partir de ahora se pretende profundizar en la tecnología de EJB, ya no está de mas aclarar, que el estudio que se ha hecho de los EJB hasta este instante ha sido un estudio, extenso si se quiere, pero bastante superficial de lo que es la tecnología. La aplicación del ejemplo era una aplicación supremamente sencilla que utilizaba un único bean y del tipo más simple posible (de sesión y sin estado). A partir de ahora se verá como, lógicamente, las aplicaciones requieren de la existencia de más de un bean, que estos colaboran entre si y que, además, se requieren de distintos tipos de beans.

De esta forma cuando construyamos aplicaciones distribuidas con tecnología EJB crearemos muchos beans, cada uno representando un concepto diferente de la lógica de negocio. Cada concepto del negocio quedará entonces reflejado como un bean de entidad o de sesión y se elegirá ese tipo de bean basándose en lo que se espera que haga. Por ejemplo, un bean denominado "Cliente" que represente el cliente de un servicio particular contendrá información que debe ser persistente (como su nombre y su número de cuenta bancaria) y seria un buen candidato a ser un bean de entidad. Por su parte, un bean denominado "BuscadorClientes", que busca determinados clientes que cumplen ciertas características, se acerca bastante más a un bean de sesión, ya que en este caso, el bean no requiere información que deba ser mantenida entre distintas interacciones de clientes.

A continuación se estudiarán más en profundidad cada uno de estos tipos de beans para ver sus características principales y se propondrán ejemplos concretos que muestren sus aspectos más importantes. Pero en primer lugar se estudiarán los servicios que el propio contenedor EJB proporciona a los beans y que son aprovechados por los distintos tipos de beans para solucionar los problemas que se encuentran durante su desarrollo.

5.2. Servicios de los beans gestionados por el contenedor

Como ya se ha comentado anteriormente, el contenedor EJB en su papel de middleware, es quien manejará la gestión de recursos, la persistencia, las

transacciones, la concurrencia y el control de acceso de forma automática para los Enterprise JavaBeans. A continuación se realizará un estudio más en detalle de cada uno de estos servicios que proporciona el contenedor.

5.2.1. Gestión de recursos

Uno de los primeros problemas con los que se encuentra cualquier desarrollador a la hora de implementar un Enterprise JavaBean es el problema de la escalabilidad. Es decir, el mantener el mayor grado de eficiencia de servicio cuando el número de clientes aumenta.

Para solucionar este problema surgen una serie de acciones que se pueden poner en práctica como las siguientes:

1. Pooling: La utilización del concepto de *pool* tiene dos vertientes diferentes. Por un lado el *pooling* de recursos permite reutilizar varios recursos, como pueden ser conexiones a bases de datos, para ser aprovechados por distintos clientes, Y por otro lado, el *pooling* de instancias, que permite que los mismos objetos sean utilizados por diferentes peticiones, evitando así la creación de nuevos objetos para cada petición. Los *pools* de instancias han sido utilizados desde los inicios de Java. Se trata de definir un espacio en memoria para varios objetos, de tal forma que en vez de tener que instanciar un objeto cada vez, podemos reutilizar las instancias existentes en el *pool*. Esto conlleva un mayor consumo de memoria pero ofrece una mayor velocidad. Además, hace trabajar menos al recolector de basura, con lo que la mejora de resultados es bastante notable.

2. Activación y pasivación: la gestión de la activación y la pasivación permite almacenar valores de un objeto en memoria secundaria para luego poder recuperarlos. La pasivación es el proceso de serialización a disco, u otro soporte como una base de datos, de aquellas instancias menos recientemente usadas (Least Recently Used o LRU, aunque es posible especificar otra política de liberación dependiendo del servidor de aplicaciones) cuando el número de instancias del bean requeridas por los clientes es mayor de las que

puede soportar el *pool*. Este proceso se puede dar, bien porque se acaba la memoria o porque se ha llegado al límite de instancias máximas configurado en el descriptor de despliegue. Al proceso contrario se le llama activación, y se produce cuando un cliente llama a un método de su bean de sesión con estado asociado y éste se encuentra en estado de pasivación. Cuando se vayan viendo los distintos tipos de beans, y mas específicamente los beans de sesión con y sin estado, se comprobará como estos mecanismos solo tienen sentido para los beans de sesión con estado.

3. Balance de carga: la utilización de este tipo de acciones permite distribuir las distintas peticiones a elementos con menor carga de trabajo para mejorar la escalabilidad de la aplicación.

4. Clustering: el empleo de clustering permite el uso de varios servidores de aplicaciones para repartir la carga.

5.2.2. Persistencia

La persistencia es un mecanismo que consiste en el almacenamiento en memoria secundaria del estado de los objetos que participan en una aplicación. Como memoria secundaria puede entenderse cualquier dispositivo de almacenamiento fuera de memoria, como ficheros en disco y en la mayor parte de los casos sistemas gestores de bases de datos.

Una de las principales aportaciones de la tecnología EJB es que el mecanismo de persistencia puede ser delegado en el propio contenedor EJB para que él mismo se encargue de gestionar el almacenamiento. De esta forma resulta mucho más sencilla para los desarrolladores la gestión de los datos que maneja la aplicación. El desarrollador se despreocupa en gran medida de toda la lógica de consulta, inserción, modificación, recuperación y borrado de los datos, que antes soportaba el código de sus aplicaciones. Sin embargo, esta facilidad que brinda el contenedor trae consigo, como es lógico, ciertas implicaciones como puede ser la necesidad de que el modelo de objetos de la aplicación coincida con el modelo relacional de los datos de la base

de datos. Es evidente, un servicio de persistencia así requiere de la existencia de una relación directa o mapeo lógico entre atributos de objetos y campos en la base de datos para funcionar.

En cuanto a la propia especificación EJB, hay que decir que uno de los tipos que se definen en ella, los beans de entidad, son objetos que controlan su persistencia. Además, dentro de este tipo de beans, como se verá más adelante, habrá dos posibilidades: los beans que gestionan su propia persistencia o Bean Managed Persistence (BMP) y los beans que delegan esa acción en el contenedor, es decir, Container Managed Persistence (CMP).

5.2.3. Transacciones

Las transacciones dividen una aplicación en un conjunto de unidades de trabajo atómicas o indivisibles y evitan interferencias entre éstas y otros procesos. Este mecanismo aplica a multitud de campos dentro de la ingeniería de aplicaciones, desde procesos de negocio con una alta interdependencia hasta operaciones interpretadas como atómicas en bases de datos. Por lo tanto, durante el desarrollo de cualquier proceso software es importante saber si la lógica de negocio que éste gestiona requiere o no de transacciones.

Por ejemplo, en cualquier aplicación de reserva de productos, es necesario que la operación de selección de la reserva siempre vaya acompañada de la operación de pago por dicha reserva, conformando ambas operaciones una única transacción. Esta situación es absolutamente lógica, ya que si se produjese un fallo en la operación de pago después de realizar la operación de selección y en la selección del producto no se ejecutase el roll-back pertinente, entonces se entraría en un estado inconsistente de la aplicación entre la selección y el pago.

Por lo tanto, cualquier aplicación que vaya a requerir de transacciones, necesitaría cumplir con una serie de propiedades conocidas como ACID:

1. (A) Atomicidad: El trabajo de los componentes se realiza en su totalidad o no se realiza.

2. (C) Consistencia: Ante fallos en el sistema se mantiene la coherencia de los datos.
3. (I) Isolation: Cada transacción es autónoma y se ejecuta de forma independiente al resto.
4. (D) Durabilidad: Los resultados de las operaciones permanecen a pesar de producirse fallos.

Por todo esto, resulta fundamental saber si un proceso en su lógica de negocio requiere que un conjunto de tareas se ejecuten como una única unidad. Como respuesta a esta necesidad aparece la gestión de transacciones de la J2EE o Java Transaction API (JTA).

Con la API JTA se permite a las aplicaciones acceder a transacciones de forma independiente de la implementación. Para conseguir esa independencia de implementación JTA, al igual que muchas otras API´s de la distribución J2EE que se basan en la definición de interfaces, especifica un conjunto estándar de interfaces entre un gestor de transacciones y las diferentes partes implicadas en un sistema transaccional distribuido. Estas partes implicadas son la propia aplicación transaccional, el servidor J2EE y el gestor que controla el acceso a los recursos compartidos o transaccionales.

El servidor de aplicaciones J2EE será el encargado de implementar el gestor de transacciones mediante el Java Transaction Service (JTS). Sin embrago, una aplicación transaccional nunca llama directamente a métodos JTS, en su lugar se utilizan los métodos proporcionados por los interfaces JTA que posteriormente delegarán en rutinas implementadas en JTS. JTS especifica la implementación de un gestor de transacciones que soporta JTA e implementa la versión Java de la especificación OMG Transaction Service 1.1. Además, JTS proporciona los servicios y funciones de gestión requeridos para soportar la demarcación transaccional, gestión de recursos transaccionales, sincronización y propagación de información específica de una instancia transaccional.

Las transacciones JTA son transacciones planas, es decir no se permite la anidación de transacciones o transacciones hijas. En otras palabras, una instancia no puede

empezar una nueva transacción hasta que la transacción anterior no haya terminado. Además, estas transacciones simplifican el desarrollo de las aplicaciones al liberar al proveedor de componentes de la aplicación de las tareas de recuperación ante errores y programación multiusuario. La plataforma J2EE manejará implícitamente muchos de estos detalles transaccionales, y también se encargará de la coordinación entre múltiples gestores transaccionales o de la propagación de información especifica de una instancia determinada.

En definitiva, con la API JTA se tiene acceso a transacciones que pueden implicar a múltiples componentes y gestores de recursos. Para demarcar una transacción JTA, una aplicación debería invocar los métodos begin( ), commit( ) y rollback( ) del interfaz javax.transaction.UserTransaction cuya implementación es quien permitirá crear y gestionar esas transacciones.

Para implementar las transacciones JTA los distintos módulos software tienen diferentes formas de acceder al objeto javax.transaction.UserTransaction. De esta forma, los componentes web, al no estar diseñados para ser transaccionales, suelen delegar el trabajo transaccional en los EJB´s, al igual que los clientes stand-alone, que dependen de su contenedor para tener acceso o no a las transacciones, pero que también suelen delegar en los EJB´s. En definitiva, son los propios EJB los encargados de gestionar las transacciones bien por si mismos (a través del método EJBContext.getUserTransaction( )) o delegando esa tarea en el propio contenedor EJB.

Por lo tanto, para el caso específico de los Enterprise JavaBeans la especificación EJB define dos tipos de transacciones:

1. Gestionadas por el propio componente: este tipo requiere que la aplicación que quiere manejar transacciones cree el objeto transacción, realice el arranque explícito de la misma (begin) y que, finalmente, la termine (commit / roll-back).

2. Gestionadas por el contenedor: en este caso el contenedor sigue las instrucciones del descriptor de despliegue indicadas por el ensamblador de

aplicaciones, liberando así de trabajo al proveedor de componentes de la aplicación.

Finalmente, hay que aclarar que un componente no puede usar los dos tipos de transacción a la vez. Los componentes entidad solo pueden usar transacciones gestionadas por el contenedor y el proveedor de componentes de la aplicación decidirá qué tipo de transacción utilizará un componente de sesión a través del descriptor de despliegue EJB

5.2.4. Concurrencia

La programación concurrente permite la ejecución de múltiples hilos de ejecución o threads en único programa. Esta característica hace que este tipo de programación resulte bastante útil en la implementación de los servicios que residirán en la parte servidora de cualquier aplicación en un entorno distribuido. Con este tipo de programación se favorece el desarrollo de código re-entrante, es decir, código que puede ser compartido por varios procesos. Además, la programación concurrente ayuda a la escalabilidad del sistema, es decir, a la cantidad de clientes que puede acceder al sistema ya que permite, por ejemplo, que una misma instancia de un objeto sirva a multitud de clientes.

Sin embargo, la programación concurrente se complica cuando los diferentes hilos de ejecución deben interactuar entre si por las posibles interacciones entre ellos. Por lo tanto, se debe recurrir a mecanismos de sincronización que permitan ejecutar de forma segura y correcta a dichos hilos. Esa es la razón por la que la programación concurrente requiere técnicas de programación avanzadas como bloqueos, recursos compartidos o sincronización.

Finalmente hay que decir, que la especificación EJB determina que sea el propio contenedor EJB quien realice la gestión de la concurrencia, liberando al desarrollador de esa difícil tarea. Para ello la especificación EJB, siguiendo la misma filosofía que Java emplea en otras tecnologías, propone limitar al desarrollador las tareas que le resultan más complicadas, como sería la de la creación explícita de hilos que luego se

debería preocupar de sincronizar. En su lugar se delegan todas esas tareas de sincronización en el contenedor EJB que implementará los mecanismos necesarios para conseguir la concurrencia.

Por lo tanto, la especificación EJB determina que todos los tipos de beans sigan un modelo de ejecución *single-threaded* o de hilo único, es decir, que distintos clientes no pueden compartir una misma instancia de un bean, a pesar de que en realidad puedan admitir código re-entrante. Esta limitación es debida a dos razones fundamentales. En primer lugar, si se quiere aprovechar una misma instancia para distintos clientes, hay que implementar beans *thread-safe* (o beans que implementen mecanismos como regiones criticas para gestionar la concurrencia) que son bastante complicados. Este hecho no favorece una de las principales mejoras que pretende la especificación EJB como es la de un rápido desarrollo de las aplicaciones. Además, este tipo de código se convierte, finalmente y debido a su dificultad, en una fuente constante de errores.

Y en segundo lugar, este tipo de implementación con múltiples *threads* de ejecución accediendo a la misma instancia de un bean, hace que la gestión de transacciones realizada por el sistema de transacciones asociado al contenedor EJB se convierta en una tarea casi imposible.

Sin embargo, el hecho de que la especificación EJB determine que modelo *single-threaded* para cualquier tipo de bean, puede llevar a cualquier aplicación de medianas dimensiones a ofrecer un rendimiento caótico. Por este motivo, lo que promueve la especificación es que el contenedor EJB utilice distintas instancias de un mismo bean para dar servicio a distintos clientes. De esta forma los clientes no deben ser procesados de forma secuencial sino que puede prestárseles un servicio de forma concurrente.

5.2.5. Seguridad

Uno de los mayores retos que afronta cualquier aplicación distribuida es el de la seguridad. Existen distintas razones para restringir el acceso a los clientes o usuarios a las aplicaciones. Por ejemplo, se puede querer proteger una vista, o partes de la

misma, de la posibilidad de acceso directo por parte de un determinado cliente. También se puede querer gestionar el registro de entrada a la aplicación, o querer realizar una gestión de usuarios dividido por roles, grupos y dominios. Incluso se puede necesitar gestionar el control del flujo de un usuario a través de la aplicación. En todos estos casos es necesario realizar un exhaustivo control de la seguridad en la aplicación.

Por lo tanto, cualquier aplicación debe cumplir los siguientes requisitos en cuanto a seguridad:

1. Integridad: debe garantizar que los recursos (documentos, mensajes o sus componentes) no han sido alterados.
2. Autenticación: debe garantizar que una entidad (persona o sistema) es quien dice que es.
3. Autorización: determina los privilegios asociados a una determinada entidad.
4. Confidencialidad: debe garantizar que elementos no autorizados no pueden acceder a determinados recursos.
5. No repudio: debe garantizar que una entidad pueda realizar su trabajo si cumple las condiciones para hacerlo.

Existen distintas metodologías relativas a la seguridad. Por un lado, está la metodología declarativa. La especificación J2EE define un modelo de seguridad simple para EJB´s y componentes web basado en roles. Esta metodología se basa en la API Java Authentication and Authorization Service (JAAS) y define en el descriptor de despliegue del componente la información de seguridad que especifique el proveedor de componentes de la aplicación. De esta forma, en la especificación de EJB, de nuevo es el propio contenedor quien facilita la gestión de autenticación y autorización utilizando la forma declarativa. Se considera declarativa ya que se describen roles y permisos usando el descriptor de despliegue XML, en vez de incluir toda la información de seguridad dentro del propio componente como hace la siguiente metodología.

Y por otro lado, está la metodología programática que incluye toda la lógica de seguridad dentro del propio componente. En este caso la seguridad no puede ser

añadida al componente de forma independiente de su lógica de negocio y, por lo tanto, no puede ser descrita utilizando el modelo declarativo. Esta metodología hace uso de las APIs de seguridad de la plataforma J2EE especificadas por el proveedor de componentes de la aplicación.

Sin embargo, la especificación EJB aboga por definir el servicio de seguridad declarativo basado en roles que permite controlar el acceso de los clientes a recursos en dos pasos: autenticación y autorización. A continuación se explicarán más en detalle cada uno de estos pasos:

1. Autenticación: la autenticación es un proceso que verifica la identidad de un usuario, dispositivo o cualquier otra entidad dentro de un sistema. Habitualmente es un prerrequisito para permitir el acceso a los recursos de un sistema. La especificación define el concepto de *principal* como la identidad que adopta una entidad como resultado de la autenticación. Para ello define el interfaz java.security.Principal.

2. Autorización: la autorización es el proceso que decide el acceso de una entidad de seguridad o *principal* a un método o recurso. La autorización se basa en la identificación, que permite el reconocimiento de una entidad en un sistema y la autenticación. Por lo tanto, la autorización decide si el principal asociado a la petición pertenece al rol de seguridad asociado. Un rol no es más que una agrupación de usuarios o clientes definidos por el desarrollador (es decir, por el proveedor de componentes de la aplicación o por el ensamblador de la aplicación). En fase de despliegue, el encargado del mismo, asociará o mapeará cada rol a un grupo de identidades o *principals* en el propio entorno de despliegue.

Sin embargo, la fase de autenticación tiene ciertas restricciones. No puede autenticarse un *principal* ante un contenedor EJB. El servicio de autenticación del servidor J2EE trabaja con usuarios J2EE (*users*) y no tiene ningún conocimiento acerca de los usuarios del sistema operativo. Por lo que hay dos tipos de usuarios que provienen de dos mecanismos de seguridad que están separados y considerados como

dos *realms* (esferas o mundos) diferentes. Además, un bean puede autenticarse ante un recurso directamente o a través de su contenedor según establezca el proveedor de componentes de la aplicación. En este último caso es el encargado de realizar el despliegue quien especifica la información de autenticación para cada referencia a un recurso.

Finalmente, hay que decir que la API JAAS define el término *subject* para representar la fuente de una petición de un recurso por parte de una entidad. *Subject* es la clase central del JAAS y representa la toda la información de una entidad como un usuario o un dispositivo. Durante el proceso de autenticación el objeto *subject* es rellenado con los *principals* o identidades asociados. Es importante tener en cuenta que un *subject* puede tener distintos *principals*. Por ejemplo, un usuario puede tener como *principals* el nombre (alejo), su DNI (01182811) o su *username* (abarrera), todos ellos permitiéndole distinguirse de otros *subjects*.

5.3. Servicios del contenedor en JBoss

Hasta este momento no se han tenido que utilizar los servicios del contenedor como la persistencia, la concurrencia, las transacciones o el control de acceso, sin embargo, la especificación EJB utiliza el mismo mecanismo declarativo para indicar cómo se manejan todos estos servicios que el que utilizaba para definir el tipo de los beans o los interfaces EJB´s de los mismos.

Este mecanismo descriptivo se basa en el uso de los descriptores de despliegue XML. Estos descriptores de despliegue, como se ha visto hasta ahora, describirán el tipo de bean (si es de sesión o de entidad) y las clases usadas para el interface remoto, home y la clase del bean. Sin embargo, para los servicios que proporciona el contenedor, debido a que el contenedor es específico del proveedor y que aporta soluciones ad-hoc, se necesitará de un descriptor también especifico del proveedor. Este descriptor, por tanto, se utilizará para especificar los atributos transaccionales de cada método del bean, qué roles de seguridad pueden acceder a cada método (control de acceso), y finalmente, si la persistencia del bean de entidad será manejada automáticamente o será manejada por el bean. Como es lógico y bien sabido a estas alturas, tanto los

ficheros descriptores de despliegue estándar (ejb-jar.xml) como los específicos del proveedor (en el caso de JBoss, el fichero jboss.xml) se empaquetarán en un fichero denominado ejb-jar.jar.

Este fichero ejb-jar.jar, puede ser desplegado en cualquier contenedor EJB que soporte la especificación EJB. De esta forma, cuando un bean es desplegado en cualquier contenedor EJB, su descriptor de despliegue específico es leído desde el fichero para determinar cómo manejar el bean en tiempo de ejecución. Por lo tanto, cuando un bean es desplegado dentro de un contenedor, el contenedor conocerá cómo serán manejadas las transacciones, la persistencia en el caso de los beans de entidad, y el control de acceso. La persona que despliega el bean usa está información y especificará información adicional para proporcionarle al bean estas facilidades en tiempo de ejecución.

Finalmente, hay que recordar que la persona que despliega el bean mapeará los atributos del descriptor de despliegue al entorno del contenedor. Esto incluirá el mapeo de la seguridad de accesos al sistema de seguridad del entorno, añadir el bean al sistema de nombrado del contenedor EJB, demás. Una vez que el desarrollador del bean ha finalizado su despliegue el bean estará disponible para que lo utilicen las aplicaciones cliente y otros beans.

El estudio detallado de cada uno de los elemento requeridos en los descriptores de despliegue por el servidor de aplicaciones JBoss se realizará según se vayan necesitando durante la realización de los ejemplos que se propondrán en los siguientes capítulos. Son en esos ejemplos en los que verán cada uno de los elementos XML que se deben incluir en los descriptores de despliegue ejb-jar.xml, jboss.xml o jbosscmp-jdbc.xml, este último para el caso de los beans de entidad CMP, como se verá más adelante. Cada uno de esos elementos ayudará a definir el comportamiento de los diferentes beans de acuerdo a los servicios que proporciona el contenedor.

6. Beans de sesión

Los bean de sesión representan un proceso o tarea, que se realizan en beneficio de la aplicación cliente. Los beans de sesión se encargan de resolver la lógica de negocio de la aplicación. Se podría decir que cada método contenido en un bean de sesión resolverá un caso de uso de la aplicación. Los beans de sesión podrían usar otros beans para realizar tareas o acceder directamente a bases de datos. Por ejemplo, puede haber un bean de sesión que haga ambas cosas, disponiendo de un método que usa otros beans para realizar una tarea y un método que usa JDBC para acceder directamente a una base de datos. Ejemplos típicos de beans de sesión podrían ser un Control de Usuarios, un Gestor de Precios o un Control de Sanciones de Tráfico para el carné por puntos.

Los beans de sesión son usados para manejar las interacciones entre entidades y otros beans de sesión, para acceder a determinados recursos, y generalmente para realizar tareas en beneficio del cliente. Los beans de sesión no son objetos del negocio persistentes como lo son los beans de entidad. No representan datos en la base de datos. Los beans de sesión se podrían corresponder con la parte controlador en una arquitectura modelo-vista-controlador porque están encapsulando la lógica de negocio de una arquitectura de tres capas.

Dentro de la especificación EJB se pueden encontrar dos tipos de beans de sesión: State Less Session Bean (SLSB) o beans de sesión sin estado y State Full Session Bean (SFSB) o beans de sesión con estado. Los beans de sesión sin estado configuran métodos de negocio que se comportan como procedimientos y que operan sólo sobre los argumentos que se les pasan. Además, los bean sin estado son temporales y como su nombre indica no mantienen el estado del negocio entre distintas llamadas. Por su parte, los beans de sesión con estado encapsulan la lógica del negocio y el estado específico de un cliente. Los beans con estado si que mantienen el estado de negocio entre distintas llamadas al método, mantienen la información en memoria y no la hacen nunca persistente.

Una vez visto que existen dos tipos de beans de sesión, seria conveniente hacer un estudio más profundo de las situaciones en las que resulta necesario utilizar uno u otro tipo en las aplicaciones EJB.

6.1. Bean de sesión con o sin estado?

A la hora de elegir el tipo de bean de sesión mas adecuado a las necesidades de la aplicación que se está desarrollando es conveniente plantearse una serie de peguntas genéricas que ayudarán a decidir cual es la mejor opción. Algunas de dichas preguntas pueden ser las siguientes:

1. ¿Cuántas invocaciones requiere un proceso de negocio del bean para completar el servicio que de él se espera? Es decir, hay que saber si el bean requiere o no el mantenimiento de un estado conversacional. O lo que es lo mismo, con la ejecución de una única vez del proceso de negocio se consigue toda la funcionalidad esperada o es necesario hacer sucesivas llamadas al proceso de negocio.

2. ¿Un mismo cliente puede ser atendido por distintas instancias de un mismo bean? Es decir, hay que saber si un cliente es independiente de la ejecución de proceso. O lo que es lo mismo saber si el procesamiento del bean es transparente para el cliente.

3. ¿Una misma instancia de un bean puede ser utilizada por múltiples clientes? Es decir, hay que saber si la ejecución de proceso de negocio es independiente del cliente que la requiera o no. O lo que es lo mismo hay que saber si el cliente es transparente para el bean.

4. ¿El proceso de negocio del bean será accedido sólo por aplicaciones en entorno web, sólo por aplicaciones stand-alone o por ambos tipos de aplicaciones? Es decir, hay que saber si el bean puede o no aprovechar, por ejemplo, las facilidades de objetos como el de la clase HttpSession para gestionar las sesiones en el caso de aplicaciones en entorno web.

5. ¿El bean hace uso de recursos que también hay que gestionar? Es decir, hay que saber si el bean va a requerir o no de una lógica de proceso de negocio extra para gestionar recursos. O lo que es lo mismo, si se va a necesitar, o no, la ayuda del contenedor EJB con la gestión, que por ejemplo, realiza de recursos mediante pools.

Una vez planteadas las preguntas, es hora de ver las posibles conclusiones y sus implicaciones en cuanto al tipo de beans a utilizar.

1. Un bean de sesión sin estado es más fácil de reutilizar y apenas produce sobrecarga. Sin embargo, el paso de información específica de un cliente a un bean sin estado requiere que el paso de dicha información sea a través de parámetros. Este hecho complica claramente el desarrollo de los beans sin estado ya que dichos parámetros estarán sometidos en accesos remotos a operaciones de marshalling y unmarshalling por lo que deben ser, en primer lugar, parámetros serializables. Otro ejemplo de problemas en el paso de parámetros podría ser el paso de identificadores clave que podrían ser un cuello de botella en la BBDD.

2. Un bean de sesión sin estado no puede mantener un estado conversacional y sus instancias pueden reutilizarse por múltiples clientes. Además en el caso de los bean de sesión sin estado invocaciones sucesivas de un mismo cliente pueden ser atendidas por diferentes instancias.

3. Por su parte, un bean de sesión con estado si que puede mantener un estado conversacional y al contrario que los beans de sesión sin estado, los clientes no comparten los beans de sesión con estado. Un bean de sesión con estado tiene su ciclo de vida gestionado por el contenedor a través de un pool con el propósito de optimizar el coste de invocación y recursos. Sin embargo, las fases de ciclo de vida del bean de activación y pasivación (o desactivación, es decir, llevar al bean al estado pasivo desde el activo) de beans con estado puede suponer un cuello de botella de entrada/salida. Por lo tanto, si se prevé que con asiduidad van a ocurrir estos procesos es conveniente intentar buscar

otra alternativa, como por ejemplo manteniendo la sesión en el servidor de Servlets-JSP.

4. Como se acaba de mencionar los beans pueden solucionar su problema de mantenimiento de la sesión gestionándola directamente en el nivel de presentación mediante el uso de la clase HttpSession gestionado por el propio servidor de Servlets-JSP. Este mantenimiento se puede producir excepto en los siguientes casos:

    - Se necesita un objeto con estado y transaccional ya que el objeto HttpSession no tiene soporte a transacciones.

    - Hay aplicaciones stand-alone que acceden a los beans de la aplicación y que no pueden acceder a la clase HttpSession y que requieren mantener el estado.

    - Se necesita un objeto con estado para almacenar temporalmente el estado de un proceso de negocio que ocurre en el contexto de una única petición HTTP e implica múltiples beans.

Para finalizar este apartado se intentará ilustrar el dilema planteado, de la elección de uno u otro tipo de bean, con un ejemplo práctico. Este ejemplo no se desarrollará como el del anterior capítulo, por el contrario, lo único que se pretende es definir un hipotético caso que permita hacer un estudio teórico de las posibles soluciones que podrían aplicarse.

También es importante aclarar, que en este apartado se han intentado poner sobre la mesa unas posibles normas básicas y genéricas (en ningún caso reglas absolutas), que permitan al desarrollador recoger información suficiente acerca de las características de su aplicación y que le ayuden a discriminar el tipo de componentes que necesita. No hay que olvidar que siempre se considerará como buena práctica, en general en muchos casos, y más aun en un campo como la ingeniería de aplicaciones, el llevar a cabo una buena fase de adquisición de información, el tomar las decisiones

apropiadas en fase de diseño de acuerdo a esa información recogida, y finalmente, desarrollar, mirando poco hacia atrás y confiando en el diseño planteado. Adquisición de información, reflexión, toma de decisiones y ejecución pueden ser las claves para conseguir los objetivos que se persiguen.

El ejemplo planteado es el siguiente: un sistema de reservas on-line de entradas para el cine. Para este caso podríamos pensar en un bean que simula el cine y otro que simula las reservas de entradas. Este bean de reservas en un primer momento puede pensarse como un bean de sesión sin estado en el que cada reserva realizada por un cliente es independiente de otra hecha por el mismo. Sin embargo, este bean puede ser modificado para que sea un bean con estado que pueda mantener un estado conversacional entre llamadas a métodos de reserva de entradas. Esto sería útil, por ejemplo, si queremos que el bean que representa al cine pueda realizar muchas reservas y que pueda procesarlas todas juntas bajo una misma tarjeta de crédito de un cliente.

6.2. Beans de sesión sin estado

6.2.1. Fundamentos de los beans de sesión sin estado

Los beans de sesión sin estado son el tipo más simple de beans que aporta la especificación EJB. Los beans de sesión sin estado, como todos los beans de sesión, no son objetos de negocio persistentes y no representan datos en la base de datos. Debido a que no son persistentes se pueden considerar como servicios temporales, que además, no sobreviven a las caídas o fallos en el contenedor. En su lugar, este tipo de beans de sesión si que representan procesos de negocio o tareas que son realizadas en beneficio del cliente que las usa e incluso pueden llegar a manejar transacciones en la ejecución de esas tareas.

Los beans de sesión sin estado son muy sencillos de manejar por parte el contenedor EJB, por eso procesan las peticiones muy rápido y usan menos recursos. El contenedor garantiza que a una instancia de un EJB sólo puede acceder un hilo, con lo cual nunca dentro de un EJB tendremos un problema de concurrencia, el contenedor lo soluciona por nosotros.

Sin embargo, esta ventaja de rendimiento tiene un precio: los beans de sesión sin estado no recuerdan entre una llamada a método y otra. Cada llamada a un método de negocio es independiente de las llamadas anteriores, es decir, no guardan ninguna relación entre diferentes llamadas de un mismo cliente. Es mas, entre dos llamadas a un mismo tipo de bean de sesión sin estado, es posible que el contenedor direccione al cliente a diferentes instancias del componente.

En definitiva, cada cliente que usa el mismo bean de sesión sin estado obtiene el mismo servicio. Para conseguir este objetivo el contenedor de EJB se apoya en los pools de EJB, cuyo objetivo, como ya se comentó previamente, es el de reutilizar las instancias de los beans en memoria. Por lo tanto, resulta evidente que la utilización de este mecanismo de pooling resulta muy sencillo y útil en el caso de este tipo de beans ya que todos ofrecen el mismo comportamiento. Un dato esclarecedor de cómo se comportan este tipo de componentes sin estado es la relación existente entre el numero de beans y el número de clientes. Para N clientes hay M instancias siendo $M < N$ en la mayoría de los casos. Con estos dos valores se define el tamaño del pool de EJB. Además, hay que recordar que el valor máximo y mínimo de M es configurable en la mayoría de los servidores.

Finalmente hay que decir, que todos los interfaces EJB home de los beans de sesión sin estado definirán un sólo método. Este es el método create( ) que no tiene argumentos, porque los beans de sesión sin estado no ofrecen métodos de búsqueda y no pueden ser inicializados con ningún argumento cuando son creados porque son todos equivalentes.

En otras palabras, no hay beans de sesión sin estado que se consideren únicos y diferenciables y que por lo tanto puedan ser localizados de alguna manera en la base de datos. Sin embargo, los beans de sesión sin estado si que pueden usarse para acceder a la base de datos así como para coordinar la interacción de otros beans para realizar una tarea.

6.2.2. Ejemplos de beans sin estado

Un primer ejemplo de un bean de sesión sin estado es, como ya se ha comentado repetidas veces, la aplicación el ejemplo propuesto en el capitulo anterior. Era un ejemplo sencillo en el que el bean únicamente se dedicaba a devolver el string: Hola Mundo!!. No era persistente, no toleraba fallos del contenedor, no manejaba transacciones debido a la simplicidad de su funcionalidad, y finalmente, tampoco necesitaba gestionar su estado.

Otro ejemplo de bean de sesión sin estado seria un bean que represente un servicio de crédito que puede validar y procesar cargos en tarjetas de crédito. Una cadena de hoteles podría desarrollar ese bean para encapsular el proceso de verificación del número de una tarjeta de crédito, hacer un cargo, y grabar el cargo en la base de datos para propósitos de contabilidad.

Finalmente, hay que comentar que no se va a realizar ningún ejemplo práctico de beans de sesión sin estado ya que es el tipo de beans que se utilizó para el primer ejemplo. Para los siguientes tipos de beans que se estudiarán a partir de ahora si que se aportarán ejemplos prácticos.

6.3. Beans de sesión con estado

6.3.1. Fundamentos de los beans de sesión con estado

Por su parte los beans de sesión con estado, al igual que los beans de sesión sin estado, se pueden considerar como servicios temporales, no son objetos de negocio persistentes y, por lo tanto, no sobreviven a las caídas en el contenedor. En su lugar, los beans de sesión con estado representan procesos de negocio que son realizados en beneficio de un único cliente, generalmente existen sólo durante una única sesión con ese cliente y pueden soportar transacciones. Es decir, que en los beans de sesión con estado su ciclo de vida puede expirar, y normalmente termina con la finalización de la sesión del cliente.

Sin embargo, la característica principal, o hecho diferencial, que define un bean de sesión con estado, es que estos beans pueden mantener estado entre llamadas a métodos. De esta forma, los beans de sesión con estado están dedicados a un cliente y pueden mantener un estado conversacional entre llamadas de métodos aprovechándose de ese mantenimiento del estado. Por lo tanto, es evidente que los clientes no pueden compartir los beans de sesión con estado, al contrario de lo que pasaba en el caso de los beans de sesión sin estado, en donde un mismo bean ofrecía siempre el mismo servicio. Ahora, cuando un cliente crea un bean con estado, el objeto del bean tiene un estado asociado que le caracteriza y estará dedicado a servir únicamente a ese cliente.

Como se ha comentado previamente, la forma que tienen este tipo de beans de implementar y mantener un estado conversacional, es decir, la forma en la que mantienen una conversación "uno a uno" con cada cliente que los invoca, es a través del mantenimiento del estado asociado a dicho bean. Este estado de negocio puede ser compartido por métodos del mismo bean.

Es importante destacar que en el caso de los beans con estado es el propio contenedor quien gestiona ese estado. Es decir, el contenedor es el encargado de almacenar el estado si el bean sale de memoria a algún dispositivo de almacenamiento y de recuperarlo cuando vuelva a memoria. Sin embargo, hay que recordar que en el caso de los beans de sesión con estado, a pesar de que sea el contenedor quien gestione el mantenimiento de la sesión, debe ser el propio bean quien debe gestionar su información persistente.

Por otro lado, todos los interfaces EJB home de los beans de sesión con estado podrán definir varios métodos create( ) cada uno de ellos recibiendo distintos argumentos, porque los beans de sesión con estado pueden ser inicializados con distintos argumento cuando son creados.

Finalmente para muchos, este tipo de EJB´s pueden ser tachados como verdaderas "máquinas pesadas" debido a los problemas de rendimiento que puede ocasionar la pasivación de los mismos, por lo que desaconsejan su uso en la mayoría de los casos.

Otros mantienen su utilidad por la posibilidad de mantener una sesión distribuida entre varios servidores.

6.3.2. Pooling y pasivación en beans con estado

La utilización del concepto de *pooling* visto para el caso de los beans de sesión sin estado no resulta tan sencilla en el caso de los beans con estado. Cuando un cliente invoca un bean, el cliente empieza una conversación y su el estado del bean debe estar disponible para el mismo cliente la próxima vez que ejecute una petición a un método del bean. Por lo tanto, en este caso el contenedor EJB no tiene tan fácil la tarea de consultar el *pool* y asignar de forma dinámica a un cliente un menejador de bean ya que cada uno tendrá un estado asociado. Sin embargo, la necesidad de utilizar el mecanismo de *pooling* para beans de sesión con estado, con el fin de gestionar los recursos (memoria, conexiones a bases de datos o sockets) y conservar la escalabilidad del sistema sigue existiendo.

En consecuencia, para limitar el número de beans de sesión con estado en memoria el contenedor EJB lo que hace es liberar la memoria (*swap-out*) guardando su estado conversacional en disco o en otro dispositivo de almacenamiento. Obviamente el mecanismo implementado para solucionar este problema es la pasivación y la activación explicadas previamente.

Por lo tanto, la especificación EJB efectivamente si que incluye el mecanismo de *pooling* para los beans de sesión con estado. Solo unas pocas instancias de los beans pueden estar en memoria cuando hay muchos clientes haciendo uso de ellos. Pero ya se sabe, este mecanismo de activación/pasivación constituirá un cuello de botella en el caso de los beans de sesión con estado. En este caso cada bean tiene asociado un estado que debe ser salvado, mientras que en los beans sin estado, obviamente esto no sucedía.

Por una lado, la pasivación que realiza el contenedor de una instancia puede ocurrir en cualquier momento, aunque el bean que tenga que sufrir la pasivación no este involucrada en la llamada a un método. La única excepción a este paso surge cuando el bean que va a sufrir la pasivación esta inmerso en medio de una transacción ya que

el contenedor deberá asegurar primero que la transacción en curso de dicho bean ha sido previamente completada. Y por otro lado, la activación de las instancias la realizan los contenedores bajo demanda, es decir, cuando un cliente vuelve a hacer una petición sobre el bean.

Finalmente, hay que mencionar un concepto bastante relacionado con la activación y pasivación de beans con estado como es el concepto de Handle. Muchas aplicaciones requieren que los clientes desconecten de los beans para posteriormente volver a reconectar. La especificación EJB define para ello el concepto de EJB object handle. El objeto Handle es una referencia serializable a un objeto EJB. Guardando esta referencia se puede acceder posteriormente al bean con el que el cliente había establecido una relación y no perder así el estado conversacional con el bean. Para los beans de sesión con estado cada cliente tiene asociado un objeto EJB durante una conversación. En caso de que esta referencia se pierda, se puede recuperar gracias al objeto Handle. Por eso es el cliente quien se tiene que encargar de preservar la instancia de estos objetos si quiere volver a llamar a su bean de sesión con estado. Un EJB object Handle es básicamente una referencia persistente a un objeto EJB. Sin embargo, la especificación no soporta la portabilidad de Handles entre distintos contenedores o máquinas.

6.3.3. Pasivación y serialización en beans con estado

Es importante destacar que el estado conversacional de un bean está íntimamente ligado a la serialización. En tiempo de pasivación el contenedor usa la serialización para convertir el estado de un bean (representada como datos en memoria) en una ristra de bits (o bit-blob[5]) y salvarla a disco. De esta forma la instancia del bean, que aun existe, es liberada de su estado y puede ser reasignada a otro cliente para comenzar otra conversación. Todo este mecanismo es posible debido a que el interfaz javax.ejb.EnterpriseBean extiende del interfaz java.io.Serializable y por lo tanto cualquier bean lo será.

Y es que, es de la serialización y de cómo se realiza ésta, de donde surge uno de los secretos que permite que estos mecanismos de activación/pasivación funcionen.

[5] BLOB: Acrónimo de Binary Large Object

Cualquier objeto Java implicado en un estado conversacional de un bean será serializado de forma recursiva para que quede registrado y pueda ser posteriormente reconstruido. Más concretamente, cualquier variable miembro de un bean se considerará como parte del estado conversacional y, por lo tanto, serializada si cumple alguna de las siguientes características:

1. La variable es un tipo simple
2. La variable es un objeto Java persistente (o nontransient) y, por lo tanto, serializable.

Este es el secreto, cuando el contenedor quiere realizar la tarea de pasivación de un bean en primer lugar guarda el estado para posteriormente informar al bean de que va a realizar esa tarea llamando al método ejbPassivate( ). Esta llamada al método ejbPassivate( ) en forma de aviso que realiza el contenedor al bean es fundamental ya que es la forma de indicarle al bean que debe abandonar todos los recursos de los que dispone. Por lo tanto, con la llamada al método ejbPassivate( ) el contenedor da la oportunidad al bean de aportar el código necesario para liberar los recursos que tiene apropiados. Es decir, conexiones a base de datos, sockets abiertos, ficheros y todos aquellos objetos que o bien no tiene sentido salvar en disco o es imposible hacerlo ya que no son serializables. Como es lógico, la tarea opuesta de activación del bean debe realizar las acciones contrarias de adquisición de los recursos que en su momento fueron liberados poniendo el código correspondiente en el método ejbActivate( ).

Sin embargo, se puede asegurar que en la mayoría de los casos el desarrollador de beans no debe preocuparse de la implementación de los métodos ejbPassivate( ) y ejbActivate( ), pudiendo dejar su implementación vacía, a no ser que el bean use dichos recursos. Además, hay que tener en cuenta que en el caso de un bean se sesión con estado, deberá ser el contenedor, y nunca el desarrollador, quien se haga de salvar los valores de referencias de los objetos implementados por el propio contenedor que puede tener asociado dicho bean durante las fases de activación/pasivación. Estas referencias podrían ser referencia a objetos EJB, referencias a objetos Home, referencias a contextos EJB o incluso referencias a contextos de nombrado JNDI.

6.3.4. Ciclo de vida de una bean de sesion con estado

A continuación de muestran una serie de gráficas que seguramente ilustrarán de forma mucho mas clara y precisa cada uno de las fases por las que pasa un bean de sesión con estado durante su ciclo de vida.

En primer lugar, la gráfica que se muestra a continuación muestra el escenario tipico que tiene lugar durante el proceso de pasivación de un bean de sesion con estado. En esta gráfica se aprecia como el cliente invoca un método de un bean. Esta invocación es interceptada por el objeto EJB correspondiente que no tiene una instancia del bean en ese momento en memoria para prestar el servicio. Si ademas, el tamaño del *pool* de instancias que gestiona el contenedor EJB es el máximo en ese momento, al contenedor no le queda otra alternativa que realizar el proceso de pasivación de una de las instancias del *pool* antes de aceptar la petición que le acaba de llegar del cliente.

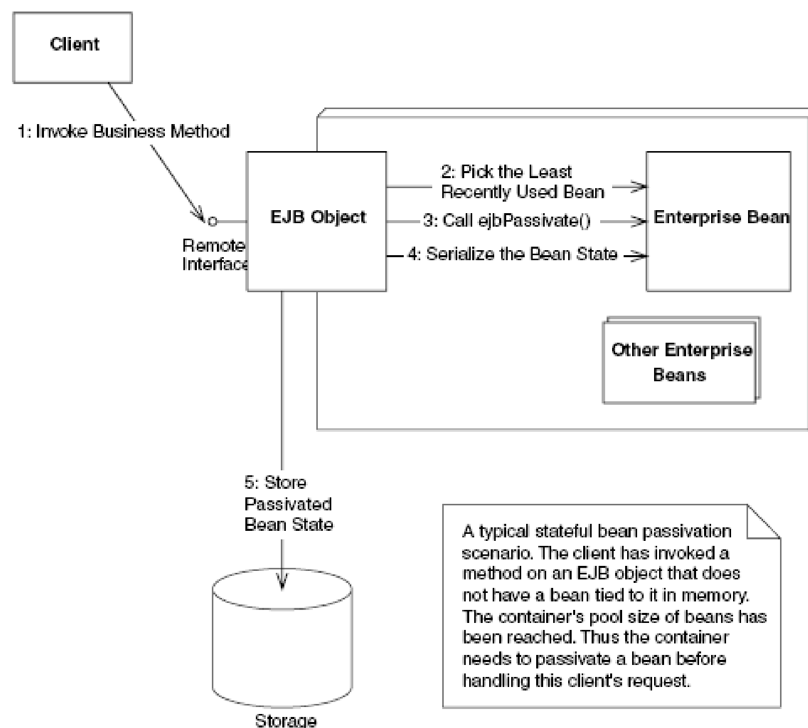


Figura 10: Pasivación de un bean de sesión con estado

El como soluciona el contenedor EJB quien es la instancia candidata a pasar al estado pasivo es cuestión del tipo de política de cacheo de instancias que emplee el propio

contenedor EJB. Como se verá mas adelante esta política es en parámetro de configuración del propio contenedor.

El diagrama de secuencia entero que representa el ciclo de vida de una bean de sesión con estado se presenta a continuación.

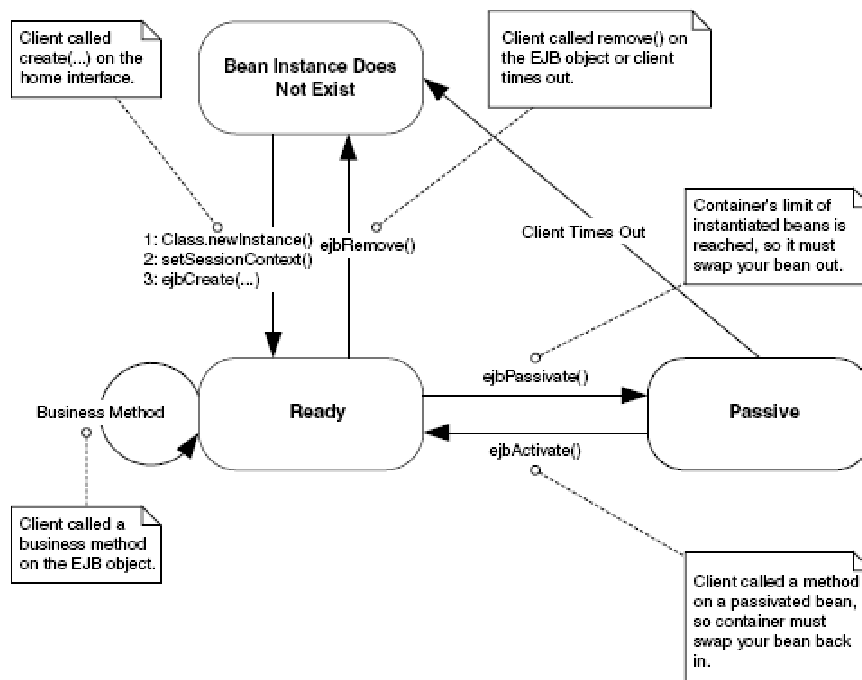


Figura 11: Ciclo de vida de un bean de sesión con estado

Hay que recordar que todas las llamadas que aparecen en este diagrama pertenecen al ciclo de vida de un bean, y por lo tanto, dichas llamadas que se realizan sobre la instancia de un determinado bean son siempre responsabilidad del contenedor EJB y nunca por el propio cliente.

Como se puede apreciar en la figura, la llamada 1 se produce cuando el cliente realiza una llamada al método create( ) sobre el interfaz Home. La llamada 2 se produce cuando el cliente llama a un método de negocio del objeto EJB y la instancia está en memoria en ese momento. La llamada 3 se produce cuando el contenedor alcanza el límite de instancias en memoria, y por lo tanto, se debe hacer el swap out de una instancia. La llamada 4 se produce cuando un cliente realiza una llamada a un método de una instancia de bean que previamente ha pasado a un estado pasivo por lo que el contenedor debe realizar la operación inversa de swap in. La llamada 5 se produce

cuando ha expirado el tiempo permitido para que una instancia pueda estar en estado pasivo y el contenedor la elimina. Y finalmente, la llamada 6 se produce cuando el cliente realiza una llamada remove( ) sobre el objeto EJB o cuando ha expirado el tiempo de un cliente y la instancia se elimina.

Nota de seguimiento: una de examen, el dilema

Hasta este momento se ha visto como se implementa el mantenimiento del estado en los beans de sesión con estado mediante el pooling, la serialización y la activación/pasivación. Pero también, se ha comprobado que dichos mecanismos soportados en la especificación EJB, también son compartidos por los beans de sesión sin estado. Incluso, se habla de métodos ejbPassivate( ) y ejbActivate( ) como los vistos en el ejemplo de beans de sesión sin estado. Métodos de los que incluso se dice que su código puede, dependiendo de los casos, estar vacío.

Por lo tanto, cabe preguntarse a estas alturas, como sabe el contenedor que una determinada instancia de un bean puede ser destruida sin miramientos por el contenedor porque es un bean que no mantiene el estado. O quizás el contenedor debe ejecutar la tarea de pasivación, porque es un bean de sesión con estado y propiedad de un cliente particular. Como diferencia el contenedor uno u otro tipo de bean. Tampoco se ha mencionado en ningún momento, por ejemplo, que un bean de sesión con estado se diferencie de uno sin estado en que uno implemente un interfaz diferente al otro o que extienden de clases distintas.

En definitiva, ¿Donde se deja patente que un bean es de un tipo o de otro?, que al fin y al cabo es el dilema que se trata de solucionar. Eso si que es de lo único de lo que debe preocuparse el desarrollador, el donde poner de forma explicita que un bean es de un tipo u otro, nada mas.

Soluciones al dilema?. No?. Pista. Se busca, como siempre, algo alejado de la idea programática, como podría ser el extender el bean de una u otra clase. Por el contrario, lo que se busca es algo más flexible y declarativo.

Si no se sabe, no hay problema, por lo menos es positivo haberle dado alguna vuelta a la cuestión y plantearse las cosas desde otra perspectiva. La respuesta?… inmediatamente, cuando se vea el ejemplo de este tipo de beans.

6.3.5. Ejemplo de bean de sesión con estado: una BD literaria

6.3.5.1. Introducción

Ha llegado el momento de poner en práctica todos los conocimientos adquiridos hasta este momento. Es hora de ver un ejemplo que ilustre de forma práctica las características propias de los beans de sesión con estado que se acaban de estudiar.

Ejemplos que permitan poner de manifiesto esas características, como es lógico, hay miles. Sin embargo, el ejemplo propuesto pretende, no solo ser una aplicación que cubra el objetivo didáctico, en cuanto a la consolidación de conocimientos teóricos se refiere (meta que debe cubrir cualquier ejemplo práctico y que obviamente también se cumple en este caso), sino que además, pretende ser un ejemplo con cierto contenido de fondo. Un ejemplo que aporte algo más. Algo tan simple, como un poquito de cultura literaria.

Por lo tanto, lo que se va implementar como aplicación de ejemplo para ver el funcionamiento de los beans de sesión con estado, será una aplicación *stand-alone* que consulte una pequeña base de datos que contenga información sobre grandes obras de la literatura universal. La funcionalidad principal que va a prestar esta aplicación es la de consulta de toda la información almacenada de cada obra maestra para ir presentándola en pantalla al cliente que ha solicitado el servicio. De esta forma, el cliente irá pidiendo la información de las obras a un bean que reside en servidor y que devuelve al cliente dicha información agrupada por obra, dando su titulo, su autor y la nacionalidad.

En primer lugar, hay que destacar que lo mas interesante del ejemplo que se propone es que en este caso los beans de sesión son capaces de gestionar su estado. Por lo tanto, resulta fundamental estudiar en que consiste la gestión del estado que hacen los beans de la aplicación.

Para asegurar que se tiene que llevar a cabo una gestión del estado en el ejemplo propuesto se va utilizar un cliente que creará varios beans para que le den servicio. Sin embargo, no todas las instancias del bean van a poder estar en memoria a la vez, ya que impondrán limitaciones al contenedor EJB para ello. Esto provocará que las instancias deban ser gestionadas de forma controlada por el contenedor y que su estado sea tenido en cuenta.

Hay que recordar que en el caso de los beans de sesión con estado, no cualquier bean puede prestar el servicio, como pasaba en los beans de sesión sin estado en el que el contendor gestionaba indiscriminadamente las instancias de los beans ya que todas podían prestar el mismo servicio. En este caso de los beans de sesión con estado al

existir un estado conversacional entre cliente y servidor es necesario que las instancias de los beans estén bien diferenciadas. Para conseguirlo, hay que recordar que los beans de sesión con estado pueden recibir argumentos durante su creación, característica que va a permitir esa diferenciación de las instancias antes mencionada. Finalmente, decir que para este tipo de beans resulta fundamental que se puedan llevar a cabo los procesos de activación y pasivación con los que se mantiene el estado del bean fuera de memoria.

Estos procesos van a ser necesarios en el ejemplo porque solo se va a permitir dos instancias del bean que gestiona la base de datos literaria en memoria, y como se ha dicho anteriormente, la aplicación cliente va a crear tres instancias. Cada una de estas instancias se va a encargar de gestionar un tipo de información de las obras maestras. La primera gestionará los títulos, otra instancia los autores y la última gestionará las nacionalidades de los autores.

La gestión del estado de cada uno de estos beans debe ser gestionado ya que es necesario saber de alguna manera la obra que ha sido consultada en última posición para enseñar la información de la siguiente obra en la siguiente llamada. Ese dato, por ejemplo, es el que determina el estado del bean y el que se debe mantener vivo en cada instancia del bean, esté o no en memoria dicha instancia, como va a producirse cuando el contenedor EJB lo requiera.

Por otro lado, hay que decir que la base de datos de la obras de la que de habla, no será un sistema gestor de base de datos relacional (SGBDR) real. La base de datos de información literaria se simulará mediante la definición en el propio bean de distintos conjuntos de títulos, autores y nacionalidades. Por ejemplo, una primera posible base de datos literaria sería la siguiente:

{La Iliada,	Don Quijote de la Mancha,	Cien años de soledad,	Hamlet	},
{Homero,	Miguel de Cervantes,	Gabriel García Márquez,	W. Shakespiare	},
{Grecia,	España,	Colombia,	Inglaterra	}

De una estructura como esta es de la que los distintos beans de sesión consultarán la información literaria que enseñará la aplicación del ejemplo. Como se puede apreciar hay una relación posicional entre los diferentes conjuntos de obras, autores y nacionalidades. De esta forma la primera posición de cada uno de estos conjuntos se corresponde con distinta información acerca de la misma obra. Lo mismo pasa con la segunda posición, la tercera, la cuarta y así consecutivamente si se decidiesen introducir mas obras.

Nota de seguimiento: el talón de Aquiles

Puede que alguien se haya cuestionado la decisión de diseño relativa a que varias instancias de un mismo bean gestionen cada una de ellas un tipo distinto de información literaria. ¿No podría ser toda esa información gestionada por una misma instancia?. Efectivamente, e incluso no se debería permitir que la capa de datos esté incluida dentro de la capa de lógica de negocio, definiendo la mini base de datos dentro del propio bean. ¿No seria lo correcto utilizar una arquitectura basada en capas con un SGDBR al que accediéramos mediante JDBC?. Pues la verdad es que si, una arquitectura que siguiese el modelo MVC (*model-view-controller*) sería lo políticamente correcto.

Sin embargo, implementar para este caso una solución así, se saldría ligeramente del ámbito estrictamente didáctico que persigue el ejemplo y no permitiría ver los fundamentos y las particularidades que distinguen a los beans de sesión con estado. Enseñar esos fundamentos es aquí lo importante y ese objetivo trata de conseguirse quizás, es verdad, sacrificando en parte algunos de los cánones básicos de cualquier diseño software. Esto es aceptable aquí, en el ambito de este libro, pero hay que tener claro que sería una práctica muy poco recomendable en una aplicación coorporativa de ambito empresarial como las que permite desarrollar la J2EE para el mundo real.

En definitiva, puede afirmarse (siendo puristas, si se quiere), que el diseño de la aplicación podría ser su verdadero talón de Aquiles. Pero algunas veces, como bien saben los protagonistas de la Iliada, hay que saber perder batallas en aras de conseguir la victoria final en la guerra. Que se lo digan a los que iban en el caballo de Troya.

6.3.5.2. Requisitos para ejecutar el ejemplo

Para poder ejecutar el ejemplo BD Literaria se requieren básicamente los mismos requisitos que para el ejemplo anterior del Hola Mundo!!. Por lo tanto, el lector debe tener instalado y funcionando correctamente en su máquina el servidor de aplicaciones JBoss Application Server 4.0 por un lado y requiere tener instalada la herramienta Ant.

Con respecto al servidor de aplicaciones JBoss, como se ha comentado en la introducción del ejemplo, para asegurar que la aplicación debe gestionar explícitamente el estado de los beans, se impone como restricción al sistema que el número de instancias posibles cargadas en memoria sean 2 como máximo. Esta consideración obliga a que se deba configurar el servidor de aplicaciones y en particular el contenedor EJB. Por lo tanto, la configuración previa del contendor EJB se convierte en un requisito previo de ejecución del ejemplo BD Literaria. Para configurar el número de instancias permitidas por el contenedor hay que acceder al fichero de configuración del contendor como se explicará más detalladamente en el siguiente apartado.

Con respecto a la herramienta Ant, para este nuevo ejemplo se utilizará un nuevo fichero jboss-build.xml, tal y como se verá en el siguiente apartado, bastante parecido al ya utilizado anteriormente en el ejemplo Hola Mundo!!, pero con las modificaciones ad-hoc para el ejemplo actual.

El resto de información, código y demás consideraciones para ejecutar la aplicación BD Literaria propuesta como ejemplo, se irán enumerando según se vayan necesitando a partir de ahora.

6.3.5.3. Configurando el servidor JBoss

La información de configuración del contenedor se encuentra en un fichero llamado standardjboss.xml que se localiza en el directorio *conf* de cualquiera de las configuraciones definidas para el servidor de aplicaciones JBoss. Este fichero almacena la configuración estándar para los cuatro tipos de beans que define la especificación EJB. Debido a esta cuestión suele ser un fichero bastante extenso y que almacena una información que requiere un profundo conocimiento del diseño arquitectónico del contenedor JBoss.

Este fichero se ordena de acuerdo a distintos elementos todos ellos colgando del elemento de más alto nivel <jboss>. Dentro de este elemento puede haber otros como el denominado <container-configurations> que permite definir distintas

configuraciones para el contenedor EJB. Cada una de estas configuraciones particulares estará definida dentro de un sub-elemento del elemento <container-configurations> denominado <container-configuration>. También es importante destacar, que cada una de las configuraciones definidas en standardjboss.xml dentro de un elemento <container-configuration> se corresponde con un tipo particular de bean. Es decir, que el fichero standardjboss.xml define una serie de configuraciones estándar para cada tipo de bean. El nombre de esas configuraciones, de hecho, lleva implícito el tipo asociado y serán las se utilice el contenedor EJB para cada bean dependiendo precisamente de su tipo.

Cada configuración definida dentro de una <container-configuration> especifica información como, por ejemplo, la identidad del gestor de persistencia, la identidad del gestor de seguridad o los atributos de la *cache* o del *pool* de instancias. La *cache* de instancias del contenedor EJB es una estructura gestionada por el propio contenedor que se encarga de albergar todas las instancias de los beans que están en estado activo y que deben ser pasivadas debido a que no hay más capacidad en el *pool*. Únicamente los beans de entidad y los beans de sesión con estado serán cacheados ya que son los únicos tipos que mantienen el estado entre distintas invocaciones de métodos. Por lo tanto, solo cuando la aplicación EJB utilice este tipo de beans será necesario, por ejemplo, configurar el tamaño de esa *cache*. Por otro lado, dentro de los atributos particulares del *pool* de instancias destacan el tamaño de dicho *pool*, que es el parámetro que interesa modificar en la configuración del contenedor para la aplicación del ejemplo.

También, es importante destacar que la configuración del contenedor puede ser especificada a dos niveles distintos. El primero de ellos es con la utilización del fichero standardjboss.xml en la ubicación antes mencionada. El segundo de los niveles es directamente a nivel EJB JAR. Es decir, se puede incluir información de configuración del contenedor directamente incluyendo un fichero jboss.xml en el directorio META-INF del fichero EJB JAR. Por lo tanto, en este ultimo fichero, bien se podría sobrescribir la configuración del contenedor definida en el fichero standardjboss.xml, o bien se podrían incluir nuevos parámetros de configuración no existentes en el primero. Este mecanismo provee de una gran flexibilidad a la configuración del contenedor.

Por lo tanto, para cumplir el requisito del ejemplo, se utilizará la primera de las opciones. Hay que acceder al fichero standardjboss.xml de la configuración por defecto del servidor (directorio *default*) y dentro de esa configuración al directorio *conf*. Una vez allí, se buscará la configuración correspondiente a un bean de sesión con estado cuyo elemento <container-name> dentro de <container-configuration> debe ser algo parecido a: Standard Stateful SessionBean. Aquí se modificará el número que se encuentra entre los elementos <MaximumSize> y </MaximumSize> que a su vez está dentro del elemento <container-pool-conf>. Este valor debe ponerse a 2 para indicar que solo se permiten dos instancias en memoria para el bean del ejemplo. Con este atributo se representa el tamaño máximo del pool de instancias que gestiona el contenedor EJB.

A continuación se muestra un fragmento de un fichero standardjboss.xml en que se pueden apreciar algunas de los elementos mas importantes que pertenecen a ese fichero y en el que, además, está reflejado el cambio en el tamaño del pool de instancias que necesita el ejemplo.

```
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 3.2//EN"
                      "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">

<jboss>

<!-- ... -->

<container-configurations>

        <container-configuration>
                <contaner-name>Standar Stateful SessionBean</contaner-name>
                <!-- ... -->

                <container-cache-conf>
                        <cache-policy>
                                LRUEnterpriseContextCachePolicy
                        </cache-policy>
                        <cache-policy-conf>
                                <min-capacity>50</min-capacity>
                                <max-capacity>1000000</max-capacity>
                                <overager-period>300</overager-period>
                                <max-bean-age>600</max-bean-age>
                                <resizer-period>400</resizer-period>
                                <cache-load-factor>0.7</cache-load-factor>
                        </cache-policy-conf>
                </container-cache-conf>
                <container-pool-conf>
                        <MaximumSize>2</MaximumSize>
                </container-pool-conf>
        </container-configuration>
```

```
        <!-- ... -->
</container-configurations>
</jboss>
```

Finalmente, sería conveniente comentar que la forma que tiene un bean durante la fase de despliegue de elegir la configuración de contenedor EJB que requiere, es a través del elemento <configuration-name> dentro de los elementos <enterprise-bean> y <bean-type>. Es a través de este elemento <configuration-name> con el que se tiene una referencia a una configuración de contenedor. Esta configuración estará definida en alguno de los ficheros de configuración mencionados anteriormente dentro del elemento <container-configurations>.

Nota de seguimiento: a los aficionados al tuning

Esta nota de seguimiento va dirigida a todos aquellos interesados en como funcionan las tripas de un servidor de aplicaciones como JBoss. A los interesados en como se afina (*tune* en inglés) con mayor precisión la configuración de un contenedor EJB. Es decir, a aquellos con una mayor vocación de sistemas.

En primer lugar, hay que destacar que en el código de ejemplo que se ha mostrado anteriormente, el comportamiento del elemento <MaximumSize> en la configuración de contenedor EJB del servidor de aplicaciones no es el esperado. La semántica de este elemento es ligeramente diferente de los que se espera. En JBoss <MaximumSize> representa el número de instancias de un bean que pueden crearse, o dicho de otra forma, el número de instancias activas que soporta el contendor EJB aunque dicho número supere el tamaño del *pool*.

Por lo tanto, el contenedor con la configuración tal y como se ha mostrado puede crear mas de 2 instancias activas de un bean que es precisamente lo que se pretende en el ejemplo. Sin embargo, si realmente se quiere limitar el número de instancias activas al del tamaño del *pool* utilizando el elemento <MaximumSize>, es necesario configurar un elemento adicional llamado <strictMaximumSize> a *true*. Configurando este valor se permiten únicamente un <MaximumSize> de instancias activas. Si se actúa de esta manera hay que plantearse el poco sentido que tendría la tarea de pasivación ya que el número máximo de instancias activas coincide con el del *pool.*

Por otro lado, hay que plantearse que la configuración del *pool* de instancias del contenedor debe ir muy estrechamente ligada a la configuración de la *cache* de instancias. Hay que recordar que cualquier instancia activa que no pueda estar en memoria dentro del *pool* debe estar en la *cache* de almacenamiento secundario que gestione el contenedor EJB. Por lo tanto, para que funcione correctamente la aplicación es necesario afinar un par de elementos de los mostrados anteriormente en el ejemplo. El elemento <max-bean-age> indica el número de segundos que una instancia puede pasar inactiva antes de que sea considerada candidata a ser pasivada por el proceso que verifica la caducidad de las instancias. Por lo tanto, un valor de 10 minutos como el mostrado en el ejemplo es poco recomendable. Para el ejemplo seria conveniente cualquier valor de unos pocos segundos. Y el otro elemento a afinar es <overager-period> que indica el tiempo en segundos que tarda en ejecutar la tarea

que verifica la caducidad de las instancias para proceder a pasivar aquellas que hayan sobrepasado el <max-bean-age>. De nuevo un valor de unos pocos segundos sería lo adecuado para que el ejemplo funcione correctamente.

Finalmente, hay que decir que el mundo del tuning en general, y en particular el de un servidor de aplicaciones como JBoss es un mundo supremamente interesante y extenso. Esto requiere un esfuerzo extra por parte del lector para recabar toda la información relativa a este tema y que puede considerarse que queda fuera del ámbito de este libro. Sin embargo, todo este conocimiento le permitirá conocer todas las particularidades de configuración del servidor que sin duda resultan fundamentales a la hora de desarrollar y desplegar una aplicación EJB. Un buen documento de referencia que toca temas de configuración en profundidad puede ser el documento de administración del servidor JBoss que ellos mismos proporcionan su sitio *web* dentro de la página de documentación: http://www.jboss.org/docs/index.

Espero que no se hayan sentido engañados por el titulo aquellos que esperaban alerones, bajos y llantas. Aquello es tunning, creo.

6.3.5.4. Preparando el fichero jboss-build.xml de Ant

Como ya se ha comentado en el ejemplo anterior, para la realización de cada uno de los ejemplo de este libro se utilizará un directorio de trabajo que se puede localizar donde resulte mas conveniente. En ese directorio es donde se guardará el fichero de configuración jboss-build.xml con el plan de desarrollo de la aplicación EJB del ejemplo, así como el resto de ficheros fuente.

A continuación se muestra el fichero de configuración jboss-build.xml que se utilizará en el desarrollo del ejemplo BD Literaria. Este fichero ha sido construido utilizando como plantilla el del ejemplo anterior del libro e introduciendo solo unos pequeños cambios para adaptarlo a este caso.

```
<project name="BaseDatos Literaria" default="all" basedir=".">

    <property file="jboss-build.properties"/>

    <property name="lib.dir"   value="../../libs"/>
    <property name="src.dir"   value="${basedir}/src"/>
    <property name="build.dir" value="${basedir}/build"/>

    <!-- The classpath for running the client -->
    <path id="client.classpath">
        <fileset dir="${jboss.home}/client">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <!-- The build classpath -->
    <path id="build.classpath">
        <path refid="client.classpath"/>
```

```
        <fileset dir="${jboss.server}/lib/">
        </fileset>
    </path>

    <!-- =============================================================== -->
    <!— Directorio de creación                                           -->
    <!-- =============================================================== -->
    <target name="prepare">
        <mkdir dir="${build.dir}"/>
    </target>

    <!-- =============================================================== -->
    <!— Compilación del código fuente                                   -->
    <!-- =============================================================== -->
    <target name="compile" depends="prepare">
        <javac destdir="${build.dir}" classpathref="build.classpath"
                                                      debug="on">
            <src path="${src.dir}"/>
        </javac>
    </target>

    <target name="package-ejb" depends="compile">
        <mkdir dir="jar" />
        <delete file="jar/BDLiteraria-ejb.jar"/>
        <jar jarfile="jar/BDLiteraria-ejb.jar">
            <metainf dir="dd/ejb" includes="**/*.xml" />
            <fileset dir="${build.dir}">
                <include name="**/*.class"/>
                <exclude name="*.class"/>
            </fileset>
        </jar>
    </target>

    <target name="package-client" depends="compile">
        <mkdir dir="jar" />
        <delete file="jar/app-client.jar"/>
        <jar jarfile="jar/app-client.jar">
            <metainf dir="dd/client" includes="*.xml"/>
            <fileset dir="${build.dir}">
                <include name="**/*.class"/>
            </fileset>
            <fileset dir="dd/client">
                <include name="jndi.properties"/>
            </fileset>
        </jar>
    </target>

    <!—Crea un fichero ear que contiene los ejbjars y el webclient war. -->
    <target name="assemble-app">
        <delete file="jar/BDLiteraria.ear"/>
        <ear destfile="jar/BDLiteraria.ear" appxml="dd/application.xml">
            <fileset dir="jar" includes="*.jar"/>
        </ear>
    </target>

    <!—Despliega el fichero EAR copiándolo en el directorio deploy JBoss -->
    <target name="deploy" depends="assemble-app">
        <copy file="jar/BDLiteraria.ear" todir="${jboss.server}/deploy"/>
    </target>


    <!—Ejecuta el cliente standalone -->
    <target name="run-client">
        <java classname="BDLiterariaClient" fork="yes">
            <classpath>
                <pathelement path="jar/app-client.jar"/>
                <path refid="client.classpath"/>
            </classpath>
```

```
        </java>
    </target>

    <target name="clean">
       <delete dir="${build.dir}" />
       <mkdir dir="${build.dir}"  />
    </target>

    <target name="all" depends="compile,package-ejb,package-client,assemble-
app,deploy" />

</project>
```

Fichero jboss-build.xml

Como se puede apreciar en este código XML que lee la herramienta Ant, jboss-build.xml define básicamente las mismas tareas que las del ejemplo anterior y que posteriormente se irán llamando de acuerdo a la fase de desarrollo en la que se encuentre la aplicación. La explicación detallada de lo que hace cada una de estas tareas ya se abordó previamente en el ejemplo anterior. Por lo tanto, no se entrará en este segundo ejemplo en demasiados detalles a no ser que sea estrictamente necesario para seguir la evolución del ejemplo.

6.3.5.5. Preparando los ficheros

En este apartado se mostrará el código de todos los ficheros implicados en el desarrollo de la aplicación distribuida Base de Datos Literaria elegida como ejemplo. Para cada uno de estos ficheros se hará una breve explicación del papel que desempeñan dentro del sistema, se explicarán las principales características del código relativas a la tecnología de EJB, así como su ubicación dentro de la estructura del directorio de trabajo.

6.3.5.5.1. Interfaz remoto

El interfaz remoto expone cada uno de los métodos de la lógica de negocio que el bean finalmente va a publicar. Este interfaz es con el que operan los clientes cuando interactúan con los objetos EJB.

El interfaz remoto se guardará en un fichero BDLiteraria.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz remoto para el Enterprise JavaBean: BDLiteraria
*/

package ejemplos;

public interface BDLiteraria extends javax.ejb.EJBObject {

        /*
         * Método que devuelve un string con la información de la obra
         * maestra almacenada en la base de datos. Esta información
         * puede ser cualquier dato de la misma: titulo, autor o
         * nacionalidad.
        */
        public String dameDato() throws java.rmi.RemoteException;

        /*
         * Método que devuelve el número de obras de la base de datos
        */
        public int numeroObras() throws java.rmi.RemoteException;
}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

3. El interfaz define el método denominado dameDato( ) que no recibe argumentos y que devolverá el String correspondiente a la información de la obra maestra que corresponda. Este método se implementará de la forma más genérica posible, de forma que en cada momento, la información que devuelve el método será del tipo que corresponda a la instancia creada del bean y siempre de la obra siguiente. Por este motivo resulta tan interesante en el ejemplo el mantener el estado del bean, controlando su tipo y sobre todo el índice de la última obra consultada.

4. El interfaz define un segundo método numeroObras( ) que devuelve un entero que representa el número de obras almacenadas en el bean. Es un método que se utiliza desde el cliente para saber el número de consultas que hay que realizar sin necesidad de cablear en código el número de consultas. Esto independizaría al cliente de cambios en el código si se introduce una nueva obra maestra en la base de datos que maneja el bean.

5. Una posible mejora que no se ha implementado por sencillez y claridad y que también redundaría en una mayor independencia del código de los clientes sería la de incluir además, un método que devolviese el número de tipos de información que maneja la base de datos (en este caso 3, títulos, autores y nacionalidades). De esta forma si en el futuro se plantea la posibilidad de incluir un nuevo tipo de información, como personajes de la obra, no habría que recompilar el cliente.

6.3.5.5.2. Interfaz home

El interfaz remoto tiene los métodos de creación y destrucción del objeto EJB. La implementación de este interfaz es responsabilidad del propio contenedor y se materializa en el objeto Home.

El interfaz Home se guardará en un fichero BDLiterariaHome.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz Home para el Enterprise JavaBean: BDLiteraria
 */

package ejemplos;

public interface BDLiterariaHome extends javax.ejb.EJBHome {

        /*
         * Método que crea la instancia del bean de sesión con estado
         * BDLiteraria. Este método recibe como argumento un entero que
         * representa el tipo de información que gestiona el bean. Con
         * un 0 maneja títulos,
         * un 1 maneja autores,
         * un 2 maneja nacionalidades.
         */
        public ejemplos.BDLiteraria create(int tipoDato)
                throws javax.ejb.CreateException, java.rmi.RemoteException;

}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

5. El interfaz Home tiene que definir los distintos métodos create( ) con los que se pueda crear el bean. En el caso del ejemplo un único método que recibe como parámetro un entero con el que se le indica al bean el tipo de información que va a ser capaz de manejar, tal y como se ve en la descripción del código mostrado. Este método lógicamente devuelve el tipo del bean que se quiere crear.

6. Hay que recordar que este método será implementado como una fábrica de objetos que los clientes usan para conseguir una referencia a los Objetos EJB ya que el método create( ) no crea beans sino que crea objetos EJB que posteriormente delegarán en los beans. Por lo tanto, este método se corresponderá con un método ejbCreate( ) de la clase del bean, que se verá a continuación.

6.3.5.5.3. La clase del bean

La clase del bean debe contener la implementación tanto de los métodos de la lógica de negocio definidos en el interfaz remoto, como la implementación de los métodos del interfaz javax.ejb.SessionBean.

La clase del bean se guardará en un fichero BDLiterariaBean.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Implementación de la clase del Enterprise JavaBean: BDLiteraria
*/

package ejemplos;

public class BDLiterariaBean implements javax.ejb.SessionBean {

        // Atributo que representa el tipo de información que maneja el bean
        private int tipoDato = 0;

        // Atributo que representa el índice de la obra consultada hasta ese
        // momento.
        private int indexObra = 0;

        // Atributo que representa la BD de obra maestras
        String [][] obrasMaestras = {
                {"La Iliada","Don Quijote de la Mancha","Cien años de
                                         soledad",  "Hamlet"},
```

```
        {"Homero","Miguel de Cervantes","Gabriel Garcia
                                Marquez","W. Shakespiare"},
        {"Grecia","España","Colombia","Inglaterra"},
};


// Atributo que representa el contexto del bean
private javax.ejb.SessionContext mySessionCtx;

/*
 * Método genérico dameDato que devuelve la información
 * correspondiente a la obra maestra dependiendo del tipo de
 * información gestionada por la instancia del bean (titulo,
 * autor o nacionalidad) y del índice de la ultima obra consultada.
 */
public String dameDato() {
        System.out.println("ejecutando dameDato()");
        return obrasMaestras [tipoDato][indexObra++];
}

/*
 * Método que devuelve el número de obras de la mini base de datos
 */
public int numeroObras() {
        System.out.println("ejecutando numeroObras()");
        return obrasMaestras[0].length;
}

/*
 * Método del ciclo de vida del bean ejbCreate que inicializa con
 * el argumento recibido como parámetro el tipo de información que
 * va a manejar el bean.
 */
public void ejbCreate(int tipoDato) throws javax.ejb.CreateException
{
        System.out.println("ejecutando ejbCreate()");
        this.tipoDato = tipoDato;
}

/*
 * Método del ciclo de vida del bean ejbActivate
 */
public void ejbActivate() {
        System.out.println("ejecutando ejbActivate()");

}

/*
 * Método del ciclo de vida del bean ejbPassivate
 */
public void ejbPassivate() {
        System.out.println("ejecutando ejbPassivate()");

}

/*
 * Método del ciclo de vida del bean ejbRemove
 */
public void ejbRemove() {
        System.out.println("ejecutando ejbRemove()");

}

/*
 * getSessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
```

```
	}

	/*
	 * setSessionContext
	 */
	public void setSessionContext(javax.ejb.SessionContext ctx) {
		mySessionCtx = ctx;
	}

}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

4. La clase del bean deberá tener un método ejbCreate( ) que recibe un argumento con el que se inicializa el tipo de información que va a manejar el bean. Este método es el que llama el objeto Home desde su método create( ).

5. La clase del bean implementa el interfaz javax.ejb.SessionBean y por lo tanto, implementa todos los métodos relacionados con el ciclo de vida. Todos estos métodos son llamados exclusivamente por el contenedor y nunca por el cliente. Al igual que el ejemplo anterior se han incluido trazas para verificar cuando llama el contendor a esos métodos. Por lo tanto, es importante destacar que las trazas serán el único código asociado a cada uno de los métodos del ciclo de vida, ya que el bean no requiere ninguna inicialización ni gestión especial cuando se destruya sus instancias.

6. Sin embargo, al estar ante un bean de sesión, ahora, con estado, hay que recordar que los métodos ejbActivate( ) y ejbPasivate( ) si que tiene sentido. Ahora será necesario mantener el estado conversacional del bean, es decir, guardar y recuperar su estado ejecutando esos métodos. Por lo tanto, cabe preguntarse por que no se incluye código en dichos métodos y solo contienen las trazas. La respuesta ya se vió cuando se vieron los fundamentos teóricos de este tipo de beans y tiene que ver con los atributos definidos en la clase que se verán a continuación.

7. La clase del bean además de los métodos vistos anteriormente, define una serie de atributos que serán los que permitan mantener el estado del bean

cuando se tenga que realizar las tareas de pasivación y activación. No hay que olvidar que lo que se denomina atributo en el entorno de la clase del bean, son lo que se denominan propiedades en el entorno del componente. Cada uno de estos atributos, bien son tipos simples como los dos enteros que definen el tipo de información o el índice de la última obra consultada, o bien, son serializables como el vector de Strings que define la base de datos o el propio contexto del bean. Cuando se tenga que pasar el bean a un estado pasivo en algún dispositivo de almacenamiento secundario, se serializarán estos atributos para mantener sus valores hasta que deban ser recuperados en una activación posterior.

8. Los métodos ejbActivate( ) y ejbPasivate( ) solo tendrán código asociado si se requiere realizar alguna tarea adicional como puede ser la de adquirir o liberar (dependiendo obviamente del método) algún recurso por parte del bean como conexión a una base de datos. Si al bean no posee recursos externos el propio estado del bean será gestionado por el contendor EJB a través de la persistencia de sus atributos.

6.3.5.5.4. Descriptores de despliegue estándar

En los ficheros descriptores de despliegue estándar que se utilizarán en el ejemplo se definirán y especificarán las necesidades que requiere el bean del contenedor EJB. Para el ejemplo planteado se utilizarán en la aplicación los dos descriptores de despliegue correspondientes, en primer lugar, a los componentes EJB en el fichero ejb-jar.xml, y en segundo lugar, al fichero opcional de la tecnología EJB llamado application-client.xml, que se usará para el cliente *stand-alone*. Ambos se estudiarán en detalle a continuación.

En primer lugar, el fichero descriptor de despliegue del componente EJB se guardará en el fichero ejb-jar.xml en el directorio /dd/ejb del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE ejb-jar PUBLIC
```

```
          "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
          "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>

    <enterprise-beans>
        <session>
                <ejb-name>ObrasMaestras</ejb-name>
                <home>ejemplos.BDLiterariaHome</home>
                <remote>ejemplos.BDLiteraria</remote>
                <ejb-class>ejemplos.BDLiterariaBean</ejb-class>
                <session-type>Stateful</session-type>
                <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>

</ejb-jar>
```

Como se puede apreciar en el código del descriptor de despliegue que se acaba de mostrar todos los elementos utilizados ya fueron explicados en el ejemplo anterior por lo que no se entrará en más detalles.

En segundo lugar, está el fichero descriptor de despliegue opcional para las aplicaciones *stand-alone*, que también se incluirá en este ejemplo. Este descriptor se guardará en el fichero application-client.xml en el directorio /dd/client del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE application-client PUBLIC
          "-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//EN"
          "http://java.sun.com/dtd/application-client_1_3.dtd">

<application-client>
        <display-name>BiblioCliente</display-name>

        <ejb-ref>
                <description></description>
                <ejb-ref-name>refBiblio</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
                <home>ejemplos.BDLiterariaHome</home>
                <remote>ejemplos.BDLiteraria</remote>
        </ejb-ref>

</application-client>
```

Como se puede apreciar en el código presentado, este descriptor también utiliza básicamente los mismos elementos vistos en descriptores de despliegue del ejemplo

anterior. Las únicas diferencias radican únicamente en que se utilizan, como es lógico, valores distintos para esos elementos. Por este motivo tampoco se va a entrar en demasiadas consideraciones.

Nota de seguimiento: la solución al dilema

Como se habrá podido percatar mas de uno, la solución al dilema planteado de cómo se hacia patente que un bean era de un tipo o de otro, se soluciona por vía declarativa. Es decir, que es en el fichero descriptor de despliegue donde se configura que el bean de sesión, por estar dentro del elemento <session>, es del tipo con estado poniendo el valor Stateful en el elemento <session-type>.

6.3.5.5.5. Ficheros específicos del servidor JBoss

Al igual que con cualquier otra aplicación que se quiera desplegar en el servidor de aplicaciones JBoss, hay ciertas caracteristicas específicas de dicho servidor, que requieren ser especificadas para la aplicación Base de Datos Literaria en ficheros descriptores de despliegue específicos.

Como es lógico se necesitará para el ejemplo propuesto un fichero XML por cada uno de los descriptores de despliegue estándar que se hayan utilizado. A continuación se presenta el código y las características de cada uno de esos ficheros descriptores de despliegue específicos.

El código correspondiente al fichero descriptor de despliegue EJB para JBoss, que se guardará en el fichero jboss.xml en el directorio /dd/ejb del directorio de trabajo es el siguiente:

```
<!DOCTYPE jboss PUBLIC
          "-//JBoss//DTD JBOSS 4.0//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">
<jboss>

    <enterprise-beans>
        <session>
            <ejb-name>ObrasMaestras</ejb-name>
            <jndi-name>ejb/examples/BDLiterariaHome</jndi-name>
        </session>
    </enterprise-beans>
```

```
</jboss>
```

Y en segundo lugar, la aplicación requiere del fichero descriptor de despliegue de las aplicaciones *stand-alone* para el servidor JBoss. A continuación se enseña el código correspondiente, que se guardará en el fichero jboss-client.xml en el directorio /dd/client del directorio de trabajo.

```
<!--<!DOCTYPE jboss-client PUBLIC
       "-//JBoss//DTD Application Client 4.0//EN"
       "http://www.jboss.org/j2ee/dtd/jboss-client_4_0.dtd">-->

<jboss-client>

   <ejb-ref>
      <ejb-ref-name>refBiblio</ejb-ref-name>
      <jndi-name>ejb/examples/BDLiterariaHome</jndi-name>
   </ejb-ref>

</jboss-client>
```

Como se puede apreciar en los dos fragmentos de código mostrados, ambos descriptores utilizan exactamente los mismos elementos que los descriptores de despliegue del ejemplo anterior que ya han sido explicados.

6.3.5.5.6. Cliente stand-alone

Tal y como sucedía en el ejemplo anterior, el cliente de la aplicación Base de Datos Literaria que se va a desarrollar, está diseñado para ejecutar completamente al margen del servidor de aplicaciones, aunque en realidad, tanto cliente como servidor residirán en el mismo espacio de direcciones.

De nuevo el cliente accederá al servicio de nombres recurriendo al fichero jndi.properties tal y como se hizo para el ejemplo anterior. Hay que recordar que este fichero permite desacoplar al máximo el código del cliente de la localización del

servicio de nombres JNDI. El contenido de ese fichero se presenta a continuación y se guardará en el directorio /src del directorio de trabajo

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
```

La clase principal del cliente creará en este caso los distintos beans de sesion que aportan la lógica de negocio. Esta clase se guardará en un fichero BDLiterariaClient.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Aplicación cliente
 */

import javax.ejb.CreateException;
import javax.naming.*;
import java.rmi.RemoteException;
import java.util.*;

import ejemplos.BDLiteraria;
import ejemplos.BDLiterariaHome;

public class BDLiterariaClient {
        public static void main(String[] args) {

                try {
                   Context initialContext = new InitialContext();
                   Object obj =
                         initialContext.lookup("BiblioCliente/refBiblio");
                   BDLiterariaHome home = (BDLiterariaHome)
                         javax.rmi.PortableRemoteObject.narrow(obj,
                                          BDLiterariaHome.class);

                   // Array de beans
                   BDLiteraria dbLiteraria[] = new BDLiteraria[3];

                   // Se crean los distintos beans, cada uno manejando un
                   // tipo de información
                   for (int i=0; i<3; i++){
                         dbLiteraria[i] = home.create(i);
                   }

                   // Consulto el numero de obras de la base de datos
                   int numObras = dbLiteraria[0].numeroObras();

                   // Se muestra dicha información llamando siempre al
                   // mismo método
                   for (int n=0; n<numObras; n++){
                        System.out.println(
                         "***Información de la siguiente obra maestra***");
                         for (int i=0; i<3; i++){
                            System.out.println(dbLiteraria[i].dameDato());
                         }
```

```
                        Thread.sleep(2000);
                  }

                  // Una vez utilizados se eliminan los objetos home
                  for (int i=0; i<3; i++){
                        dbLiteraria[i].remove();
                  }

            } catch (Exception e) {
                        System.out.println(e);
            }
      }
}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

6. El método lookup( ) obtiene la referencia al objeto Home pasándole como argumento el nombre del recurso Home con el que ha sido desplegado en el sistema de nombrado del servidor de aplicaciones. Esa referencia se mapeará con el nombre JNDI o ubicación real del recurso dentro del servidor de nombres en tiempo de despliegue y se realiza en el fichero jboss-client.xml.

7. Una vez obtenida la referencia al objeto Home se crean tres instancias diferentes utilizando para ello un vector de instancias definido previamente. Cada una se encargará de gestionar un tipo de información diferente ya que al método create( ) se la pasa en cada caso un argumento distinto. Por lo tanto, la aplicación ha creado tres instancias perfectamente diferenciables y a las que es necesario gestionar su estado.

8. Una vez creadas las intancias se hace la primera llamada a un metodo de logica de negocio. La llamada a numeroObras( ) devuelve el numero de obras maestras almacenadas en el bean y puede llamarse sobre cualquier instancia ya que todas comparten esa información.

9. Posteriormente se empiezan a hacer llamadas consecutivas al mismo método dameDato( ) sobre las distintas instancias y tantas veces como obras maestras existan. Este método controla y conoce el estado del bean sabiendo el tipo de información que gestiona cada instancia, y sobre todo, cual fue la ultima obra consultada. Esto es fundamental ya que cuando se ejecute la aplicación,

debido al número de instancias creadas y a que la configuración del contenedor EJB no permite tener en el *pool* de memoria todas esas instancias, es inevitable que se produzcan las tareas de pasivación y activación. La ejecución de estas tareas son las que obligan a que la gestión del estado del bean sea tratado convenientemente para que la información mostrada al cliente sea siempre consistente.

10. Finalmente, la aplicación cliente una vez utilizadas las instancias del bean, llama al destructor del objeto EJB que hace que el contenedor las elimine.

6.3.5.5.7. Fichero application.xml

Como se podrá ver en el siguiente apartado se va a tratar de aprovechar al máximo la potencia de una herramienta de desarrollo como Ant intentando ejecutar varias tareas de forma consecutiva. Sin embargo, para poder realizar todas esas tareas de forma rápida y automática hay que asegurarse primero de que todos los ficheros necesarios están presentes y bien construidos.

Pues bien, el único fichero que falta para que todo funcione como es debido es el fichero descriptor de despliegue application.xml. Este fichero se utiliza para crear el fichero BDLiteraria.ear. Este fichero .ear lee la información almacenada en el descriptor de despliegue application.xml para conocer los módulos que componen la aplicación J2EE e incluye todos los ficheros de empaquetamiento que se encuentren en el directorio /jar del directorio de trabajo.

El contenido del descriptor de despliegue application.xml, que declara todos los módulos J2EE incluidos dentro de la aplicación, se presenta a continuación y se guardará en el directorio /dd del directorio de trabajo.

```
<!DOCTYPE application PUBLIC
        "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
        "http://java.sun.com/dtd/application_1_3.dtd">

<application>

    <description>Una descripción cualquiera</description>
```

```
    <display-name>BaseDatosLiteratura</display-name>

    <module>
        <ejb>BDLiteraria-ejb.jar</ejb>
    </module>

    <module>
        <java>app-client.jar</java>
    </module>

</application>
```

6.3.5.6. Aprovechando al máximo Ant

De nuevo se ha terminado la parte conceptual con la definición y desarrollo de todos los ficheros incluidos en el ejemplo y es hora de empezar a realizar todas las tareas que requiere cualquier aplicación de la tecnología EJB. Por lo tanto, es el momento de llevar a cabo todas las tareas, con la ayuda de la herramienta Ant, que se realizaron para el ejemplo anterior.

Sin embargo, esta vez se propone aprovechar al máximo la potencia de Ant. Por lo tanto, lo que se propone es ejecutar de forma automática y consecutiva toda esa serie de tareas que se llevaron a cabo para el anterior ejemplo de forma individual y ahora agruparlas en una única tarea. Esta tarea se denominará *all* y ya se encuentra definida en el fichero jboss-build.xml.

Como es lógico la ejecución de esta única tarea facilitará enormemente todo el desarrollo y despliegue de la aplicación del ejemplo. Esta tarea *all* simplemente define una serie de tareas de las que depende y que se irán ejecutando de forma ordenada, automática y consecutiva según se hayan definido en el fichero jboss-build.xml. Para este caso particular la tarea *all* implica la ejecución en primer lugar de la tarea de compilación, luego el empaquetamiento de los componentes EJB creando el fichero BDLiteraria-ejb.jar, mas adelante el empaquetamiento del módulo del cliente originando app-client.jar, luego la tarea de ensamblaje de la aplicación que produce el fichero BDLiteraria.ear y, finalmente, el despliegue de dicho fichero dentro del servidor de aplicaciones.

Para ejecutar esta tarea *all* que engloba a todas las demás hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml all
```

Al igual que el resto de los comandos cuando se ejecutan de forma individual este comando *all* ejecuta en primer lugar todas las tareas de las que depende. Cada una de esas tareas a su vez tendrá otras sub-tareas definidas mediante etiquetas XML que también se irán ejecutando de forma secuencial. Finalmente, si todo ha ido correctamente se enseña un mensaje Build Successful y si se ha producido algún error se enseñará el mensaje Build failed.

6.3.5.7. Ejecutando Base de Datos Literaria.

A la hora de ejecutar el cliente stand-alone de la aplicación Base de Datos Literaria, se recurrirá a la ayuda de la herramienta Ant. Para realizar esta tarea de ejecución del cliente se utiliza el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml run-client
```

Este comando ejecuta todas las sub-tareas que se encuentren definidas mediante etiquetas XML dentro de la tarea denominada run-client incluida en el fichero jboss-build.xml. Es decir, esta tarea se encarga de ejecutar mediante el comando java la clase BDLiterariaClient que pertenece al paquete app-client.jar ubicado en el directorio /jar del directorio de trabajo.

En el lado cliente, una vez ejecutado la aplicación, lo que se debe visualizar en la consola de comandos desde la que se lanzó la máquina virtual Java debe ser una secuencia similar a la siguiente:

```
     [java] ****Información de la siguiente obra maestra****

     [java] La Iliada
     [java] Homero
     [java] Grecia

     [java] ****Información de la siguiente obra maestra****

     [java] Don Quijote de la Mancha
     [java] Miguel de Cervantes
     [java] España

     [java] ****Información de la siguiente obra maestra****

     [java] Cien años de soledad
     [java] Gabriel García Márquez
     [java] Colombia

     [java] ****Información de la siguiente obra maestra****

     [java] Hamlet
     [java] W. Shakespiare
     [java] Inglaterra

BUILD SUCCESSFUL
Total time: 11 seconds
```

Para comprobar que efectivamente la aplicación funciona como se esperaba y que el estado se ha mantenido de forma correcta, se tiene que cumplir que los datos de las obras maestras que se muestran en pantalla sean coherentes y ajustados a la realidad. De forma que debe aparecer toda la información de todas las obras que se hayan almacenado en la estructura que manejan los beans. Y obviamente, los títulos de las obras deben ir seguidos por sus verdaderos autores, que a su vez deben ir seguidos de la nacionalidad correcta. Un Hamlet, escrito por el colombiano Miguel de Cervantes Saavedra es una prueba clara de que la aplicación no esta guardando ese estado correctamente.

Por su parte en el lado servidor, una vez ejecutado la aplicación, lo que se debe visualizar en la consola de *log* del JBoss debe ser una secuencia similar a la siguiente:

```
12:52:30,153 INFO  [STDOUT] ejecutando ejbCreate()
12:52:30,183 INFO  [STDOUT] ejecutando ejbCreate()
12:52:30,193 INFO  [STDOUT] ejecutando ejbCreate()
12:52:30,203 INFO  [STDOUT] ejecutando numeroObras()
12:52:30,213 INFO  [STDOUT] ejecutando dameDato()
12:52:30,243 INFO  [STDOUT] ejecutando dameDato()
12:52:30,243 INFO  [STDOUT] ejecutando dameDato()
12:52:31,334 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:31,354 INFO  [STDOUT] ejecutando ejbPassivate()
```

```
12:52:32,266 INFO  [STDOUT] ejecutando ejbActivate()
12:52:32,266 INFO  [STDOUT] ejecutando dameDato()
12:52:32,276 INFO  [STDOUT] ejecutando ejbActivate()
12:52:32,276 INFO  [STDOUT] ejecutando dameDato()
12:52:32,286 INFO  [STDOUT] ejecutando ejbActivate()
12:52:32,286 INFO  [STDOUT] ejecutando dameDato()
12:52:33,337 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:33,347 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:33,347 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:34,289 INFO  [STDOUT] ejecutando ejbActivate()
12:52:34,289 INFO  [STDOUT] ejecutando dameDato()
12:52:34,299 INFO  [STDOUT] ejecutando ejbActivate()
12:52:34,299 INFO  [STDOUT] ejecutando dameDato()
12:52:34,309 INFO  [STDOUT] ejecutando ejbActivate()
12:52:34,309 INFO  [STDOUT] ejecutando dameDato()
12:52:35,340 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:35,340 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:35,340 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:36,321 INFO  [STDOUT] ejecutando ejbActivate()
12:52:36,321 INFO  [STDOUT] ejecutando dameDato()
12:52:36,331 INFO  [STDOUT] ejecutando ejbActivate()
12:52:36,331 INFO  [STDOUT] ejecutando dameDato()
12:52:36,341 INFO  [STDOUT] ejecutando ejbActivate()
12:52:36,341 INFO  [STDOUT] ejecutando dameDato()
12:52:37,343 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:37,343 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:37,353 INFO  [STDOUT] ejecutando ejbPassivate()
12:52:38,354 INFO  [STDOUT] ejecutando ejbActivate()
12:52:38,354 INFO  [STDOUT] ejecutando ejbRemove()
12:52:38,364 INFO  [STDOUT] ejecutando ejbActivate()
12:52:38,364 INFO  [STDOUT] ejecutando ejbRemove()
12:52:38,364 INFO  [STDOUT] ejecutando ejbActivate()
12:52:38,374 INFO  [STDOUT] ejecutando ejbRemove()
```

Como se puede apreciar el contenedor EJB se encarga de ir llamando a los distintos métodos de negocio como dameDato( ) o numeroObras( ) para pedir las información literaria y, también se encarga de ir llamando a los métodos de activación y pasivación. Estos métodos son necesarios ya que en el *pool* de instancias solo se ha permitido la existencia de dos de ellas y la aplicación cliente creo una por cada tipo de información gestionada.

7. Beans de entidad

Los beans de entidad son objetos persistentes que representan la información que maneja la aplicación. Es decir, son objetos que mantienen en memoria los datos relacionados con la aplicación, para que luego dicha información sea hecha persistente manteniéndola en un almacén de datos en cualquier dispositivo de almacenamiento secundario. Además de gestionar los datos de la aplicación, los beans de entidad se caracterizan por no realizar tareas complejas de lógica asociada al *workflow* de la aplicación.

Los beans de entidad pueden vivir tanto como la información que representan. Los beans de entidad contienen información que persiste aun cuando no haya clientes utilizando ninguno de los servicios que proporcionan los beans. El estado de un bean de entidad existe más allá del ciclo de vida de la aplicación en la que se ha creado, o incluso más allá del ciclo de vida del contenedor EJB. Esto implica que el contenedor EJB puede restaurar el bean de entidad a su estado original.

De hecho, la información que gestionan los beans de entidad persiste incluso si el servidor de aplicaciones en el que viven esos beans se reinicia. Por esta razón se dice que los beans de entidad son beans que sobreviven a caídas del sistema, ya que en caso de un fallo del sistema, los datos que hay en memoria estarán guardados en el dispositivo persistente, con lo cual, cuando se reinicie el servidor se recuperaran sin ningún problema. Si en algún momento el contenedor EJB fallase, el componente EJB, la clave primaria en el caso de los beans CMP y cualquier referencia remota sobreviviría al fallo.

Lógicamente, para conseguir ese grado de persistencia de información los beans de entidad deben recurrir a un mecanismo que permita soportarlo, y que normalmente, aunque no la única forma como se verá más adelante, resulta ser una base de datos relacional. Los beans de entidad proporcionan un interfaz orientado a objeto de los datos que normalmente serán accedidos mediante la API JDBC u otra API. Es decir, un bean de entidad proporciona un objeto *view*, o una vista en forma de objeto, de los datos almacenados en ese soporte de almacenamiento. En definitiva, un bean de

entidad proporcionaría una especie de caché o vista sincronizada en memoria de los datos de la base de datos.

En cuanto a los dispositivo de almacenamiento utilizados, los beans de entidad normalmente usan las tablas de una base de datos relacional, aunque también es posible que mantengan la persistencia de los datos mediante serialización en ficheros, por ejemplo un fichero XML, utilizando LDAP o en bases de datos orientadas a objetos o Object Database Management System (ODBMS). En cualquiera de los casos el objetivo de un bean de entidad es el de *cachear* los datos en memoria desde una fuente persistente y mantener una sincronización total entre el estado de los datos en memoria y la fuente de datos.

Otra de las características destacables que introducen los beans de entidad dentro de la tecnología EJB es que este tipo de beans abstraen totalmente la capa de persistencia del sistema. Es decir, independiza la capa de datos del resto de la capas de la aplicación y funciona como una herramienta de traducción de base de datos relacional a objeto. Por ejemplo, se podría cambiar el nombre de una columna de una tabla y el resto de capas no se daría cuenta, ya que a esa columna se accedería a través de un método get/set del bean de entidad. Así, en lugar de escribir toda la lógica de acceso a la base de datos dentro de una aplicación, la aplicación simplemente usaría los interfaces de ese bean para acceder a los datos almacenados en la base de datos y que requiere el cliente.

Los beans de entidad proporcionan un modelo de componentes que permite a los desarrolladores de beans enfocar su atención en la lógica de negocio del bean, mientras el contenedor EJB, a través de los servicios que proporciona en su papel de *middleware*, se encargará de manejar la persistencia, las transacciones, y el control de accesos. Los beans de entidad se consideran componentes de "grano grueso" distribuidos, transaccionales, persistentes, compartidos, multiusuario, de larga duración y transparentes a caídas del servidor.

Finalmente, sería conveniente hacer una última aclaración relacionada con el término bean de entidad. En lo sucesivo se utilizarán dos conceptos que usan dicho término pero que hacen referencia a mundos completamente distintos aunque estrechamente

relacionados. Por un lado, existe la instancia del bean de entidad, que será el objeto o la representación en memoria de los datos persistentes en algún dispositivo y que además, es un objeto que sabe como manipular los datos que representa. Y por otro lado, existen los datos del bean de entidad que son el conjunto propiamente dicho de los datos físicos almacenados en esa base de datos.

7.1. Persistencia de objetos

La persistencia es el proceso de escribir información en una fuente de datos externa. Y como se ha comentado previamente, la persistencia de objetos puede realizarse de distintos modos. Bien utilizando la serialización de los objetos, utilizando bases de datos relacionales o incluso utilizando bases de datos directamente orientadas a objetos. Sin embargo, no todos estos tipos de persistencia presentan las mismas ventajas a la hora de ser utilizados por una tecnología como EJB. La especificación EJB requiere de una serie de mecanismos robustos de persistencia que incluya el almacenamiento, la recuperación y la consulta de toda la información que requieren las aplicaciones distribuidas. A continuación se van a estudiar cada una de estas posibilidades de persistencia para ver sus ventajas e inconvenientes a la hora de ser usadas en la tecnología EJB.

En primer lugar se encuentra la serialización. La serialización de objetos permite transformar un objeto en un flujo de bytes. Se suele utilizar cuando se trabaja con objetos Java para almacenar el estado de dicho objeto en un dispositivo permanente. Por lo tanto, la serialización se convierte en una forma sencilla de empaquetar un objeto utilizando un flujo de bytes con el que luego se puede trabajar. Ese flujo de información posteriormente se puede almacenar, e incluso, pasarlo a través de una red como efectivamente hace RMI en el paso de parámetros.

Así pues, se puede afirmar que la serialización es un mecanismo que funciona bien para ciertos dominios como comunicaciones a través de una red. Sin embargo, de cara a la persistencia, la serialización presenta ciertas limitaciones a la hora de realizarla. Por ejemplo, la serialización ofrece pocas facilidades para realizar de forma eficiente

consultas sobre la información que almacena. En general, cualquier consulta realizada sobre un objeto almacenado mediante serialización es costosa y poco eficiente ya que, por ejemplo, requiere que todos los objetos sean des-serializados y llevados de nuevo a memoria.

Otra forma de realizar la persistencia de objetos Java es utilizando una base de datos relacional. En este caso, más que serializar un objeto entero, el objeto se descompone en partes para posteriormente almacenar cada una de estas partes. Cuando se quiere almacenar un objeto Java se debería usar, por ejemplo, la API JDBC para mapear los datos del objeto en la base de datos relacional. Para realizar la tarea contraria de carga de un objeto a partir de la información almacenada en la base de datos, se debería crear una instancia de la clase almacenada, leer los datos almacenados e ir poblando los atributos con los campos leídos de la base de datos.

Esta tecnología se conoce como el mapeo objeto-relacional (O/R). Esta tecnología resulta ser un mecanismo mucho más sofisticado que la simple serialización de objetos, que entre otras ventajas aporta eficientes mecanismos de consulta y visualización. También, es importante destacar que hay dos formas de realizar este mapeo. Por un lado, se pueden utilizar herramientas que facilitan la realización automática del mapeo, y por otro, se puede hacer ese mapeo de forma manual usando API´s de acceso a base de datos como JDBC o SQL/J. Un motor o convertidor objeto-relacional mapearía una clase Java a una definición de tabla SQL. De esta forma una instancia de la clase sería mapeada a un registro de la tabla, mientras que los atributos del objeto serán los distintos campos dentro del registro.

Finalmente, existe un tipo de persistencia que utiliza una base de datos orientadas a objetos u Object Database Management System (ODBMS). Este tipo de base de datos utiliza un mecanismo de persistencia que gestiona objetos enteros y no sus partes como las bases de datos relacionales. Con este tipo de base de datos no es necesaria la capa de mapeo de O/R, ya que se guardan objetos Java enteros. Sin embargo, si que se introduce un nivel de abstracción adicional con la aparición del interfaz Object Query Language (OQL) que permite realizar consultas de alto nivel a las propiedades de los objetos.

7.2. Mapeo Objeto-Relacional

Para la mayoría de las aplicaciones, sean distribuidas o no, el almacenar y recuperar información de la capa de datos implica alguna forma de interacción con una base de datos relacional. Esto siempre ha representado un problema importante para los desarrolladores porque, a pesar de que algunas veces el diseño de datos relacionales de la capa de datos y el diseño orientado a objetos de la capa de negocio comparten ciertas relaciones, realmente se basan en estructuras muy diferentes dentro de sus respectivos entornos.

Las bases de datos relacionales están estructuradas mediante una configuración de tablas mientras que las instancias orientadas a objetos normalmente están relacionadas más en forma de árbol. Esta diferencia ha llevado históricamente a los desarrolladores de tecnologías de persistencia de objetos a intentar construir un puente entre el mundo relacional y el mundo orientado a objetos. Resultaba, pues imprescindible elegir un mecanismo de persistencia de objetos que permitiera reducir la distancia existente entre los mundos relacional y de objetos.

La tecnología Enterprise JavaBeans a través de la especificación EJB ofrece una posible respuesta a este problema. La especificación EJB define el mapeo objeto-relacional como mecanismo para llevar a cabo dicha persistencia. Este mapeo objeto-relacional implícito en los beans de entidad, implica que el bean de entidad sea el responsable de mantener una consistencia entre los datos que maneja la aplicación y la fuente de datos que los almacena.

Por lo tanto, se deduce que existirá un alto grado de correspondencia entre las instancias de un bean de entidad y los registros o filas de una tabla de la base de datos. De hecho en la práctica, todos los servidores EJB implementan las instancias de los beans de entidad como registros de las bases de datos. Además, hay que decir que ésta correspondencia es tan grande que el concepto de "*primary key*" que utilizan las bases de datos es un concepto de fundamental relevancia en el mundo los beans de entidad. De hecho, los beans de entidad se identificarán unívocamente a través de sus *primary keys* o claves primarias, tal y como se estudiará más adelante en este mismo capítulo cuando se vea la arquitectura de los beans de entidad.

Sin embargo, antes de ver dicha arquitectura y siguiendo con el mundo de las bases de datos, cabe plantearse la siguiente consideración. Si los beans de entidad se caracterizan porque deben ser mapeados a algún dispositivo de almacenamiento debe existir alguien que realmente tenga que escribir el código de acceso al dispositivo de almacenamiento. Pues bien, hay dos formas distintas de hacer persistentes esos beans de entidad.

Por una parte, los beans de entidad con persistencia gestionada por el propio bean (Bean Management Persistant o BMP en adelante) son beans de entidad que deben hacerse persistentes por si mismos, y en pocas palabras, a mano. Es decir, el desarrollador de componentes debe escribir el código que traduce los atributos de las instancias en memoria a sus correspondientes campos de datos del dispositivo de almacenamiento, como pueden ser una base de datos relacional o una base datos orientada a objetos. El desarrollador de beans realiza él mismo y dentro del código del propio bean las operaciones de persistencia (incluyendo el almacenamiento, recuperación y búsqueda de datos). Para conseguirlo, el desarrollador debe hacer uso de las API´s que le proporciona la J2EE como JDBC. Por ejemplo, para el caso de una base de datos relacional, el código del bean debería realizar sentencias SQL como INSERT para insertar un dato en la base de datos o realizar una sentencia SQL DELETE para eliminarlo.

Por otra parte, los beans de entidad con persistencia gestionada por el contenedor (Container Management Persistant o CMP en adelante) son beans que se apoyan en el contenedor EJB para realizar las tareas de persistencia en lugar de realizarlas el propio desarrollador. En este caso, el bean normalmente no contiene ninguna lógica de persistencia y en su lugar hay que informarle al contenedor sobre cómo se quiere que realicen las tareas de persistencia las herramientas del contenedor EJB. Por ejemplo, si se usa una base de datos relacional, el contenedor automáticamente generará la sentencia SQL INSERT que llave a cabo la inserción de un dato en la base de datos. Incluso en el caso de no utilizar una base de datos relacional, y en su lugar usar una base de datos orientada a objetos que sea soportada por el contenedor, las herramientas del contenedor EJB deberían generar la lógica apropiada necesaria para llevar a cabo la persistencia.

De hecho un dato importante a destacar es que la especificación EJB en el caso de los beans CMP permite que no sea hasta tiempo de despliegue cuando se realice la configuración del mapeo objeto-relacional. Esta característica resulta fundamental ya que, en primer lugar, el desarrollador se despreocupa de las tareas de persistencia y, en segundo lugar, se permite que los objetos de datos relacionados con el almacenamiento sean independientes y reusables en distintos entornos de implantación.

Finalmente, hay que destacar que la persistencia gestionada por el contenedor o CMP aporta como una de sus mayores ventajas, como resulta evidente, una importante reducción en el tamaño del código de los beans ya que elimina todo el código JDBC asociado a la persistencia.

7.3. Arquitectura de los beans de entidad

7.3.1. Profundizando en el ciclo de vida de los beans de entidad

Es el momento de introducir un nuevo elemento no visto hasta ahora y que resulta imprescindible en la implementación de los beans de entidad. Dicho nuevo concepto es el de *primary key*. Para entender mejor el por qué de la necesidad de introducir un nuevo elemento, resulta conveniente hacer un pequeño recordatorio del ciclo de vida de los bean, tanto de sesión como de entidad. Este recordatorio va a permitir al lector centrar y focalizar la cuestión, ver la situación con perspectiva y desembocar de forma natural en la necesidad anunciada.

Hasta este momento se ha visto como la especificación EJB impone en los beans de sesión que un cliente nunca llama directamente a la clase de un bean. En su lugar, un cliente invoca a un objeto EJB que a su vez delega en el propio bean. Teniendo en cuenta que el objeto EJB delega en el bean resulta evidente que por cada método de creación, ejbCreate( ) definido en el bean, deberá existir un método create( ) definido en el interfaz Home a partir del cual se crea el objeto EJB.

A continuación se recuerdan la signatura de los métodos create( ) y ejbCreate( ) utilizados en el ejemplo de los beans de sesión.

```
// Signatura del metodo de creacion en el interfaz Home para beans de sesion
public Hola create() throws
                 javax.ejb.CreateException, java.rmi.RemoteException;

// Signatura del metodo de creacion en la clase del bean para bean de sesion
public void ejbCreate() throws javax.ejb.CreateException
```

Como se puede apreciar, el método create( ) del objeto Home devuelve el objeto EJB ejemplos.Hola, creado a partir del interfaz Hola. Por su parte, el método ejbCreate( ) de la clase del bean devuelve *void*. Este último método, ejbCreate( ) es llamado por el contenedor EJB, y mas concretamente, quién llama a ese método es el objeto EJB cuando es creado. Debido a que el contenedor no necesita que se le devuelva ningún tipo especial que le permita, por ejemplo, reconocer la instancia recién creada, entonces devuelve *void*.

A continuación, para aclarar aún más el mecanismo de creación de un bean de sesión se muestra una figura que ilustra claramente los pasos a realizar.

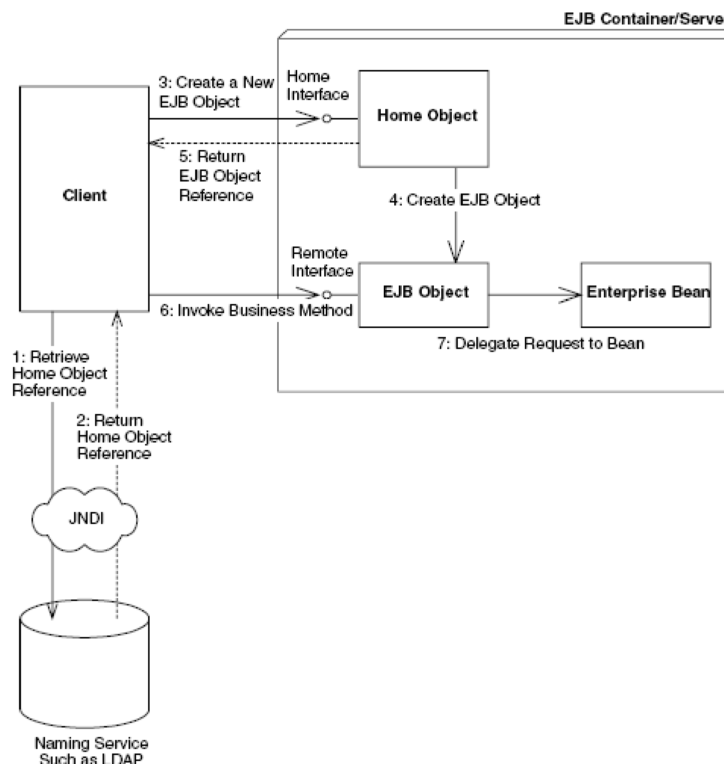


Figura 12: Pasos en la creación de un bean de sesión

Sin embargo, este mecanismo es ligeramente distinto en el caso de los beans de entidad. A continuación se describe como serían los métodos de creación tanto en el interfaz Home como en la clase del bean.

```
// Signatura del método de creación en el interfaz Home para beans de sesión
public Alumno create()
                 throws javax.ejb.CreateException, java.rmi.RemoteException;


// Signatura del método de creación en la clase del bean para bean de sesión
public AlumnoPK ejbCreate() throws javax.ejb.CreateException
```

Como se puede apreciar los dos métodos devuelven ahora tipos distintos. El objeto Home devuelve un objeto EJB, tal y como se hacia antes, mientras que ahora la instancia del bean devuelve un objeto *primary key* llamado AlumnoPK. Es decir, en el caso de los beans de entidad el método ejbCreate( ) de la clase del bean lo que esta devolviendo es un nuevo elemento: AlumnoPK.

Ahora la clase del bean devuelve al contenedor EJB un objeto *primary key* y no *void*. Esto es así porque ahora quien realiza la llamada al método ejbCreate( ) sigue siendo efectivamente el contenedor, pero ahora no desde el objeto EJB como en los beans de sesión, que además aún no está creado, sino que quien realiza la llamada ahora en nombre del contenedor EJB es el objeto *Home*. El contenedor EJB, en el caso de los beans de entidad debe distinguir cual es la instancia que crea, no es válida cualquiera como en los beans de sesión. Por lo tanto, la única forma que tiene el contenedor EJB, y en particular, el objeto *Home* para distinguir que bean tiene que crear es apoyándose en la clase *primary key* que el devuelve la instancia del bean. Una vez que el objeto *Home* tiene el objeto *primary key*, éste puede generar el objeto EJB y devolvérselo al cliente.

A continuación se muestran gráficamente los pasos que se tienen que llevar a cabo para realizar la creación de un bean de entidad y poder así ver claramente las diferencias respecto a los beans de sesión.

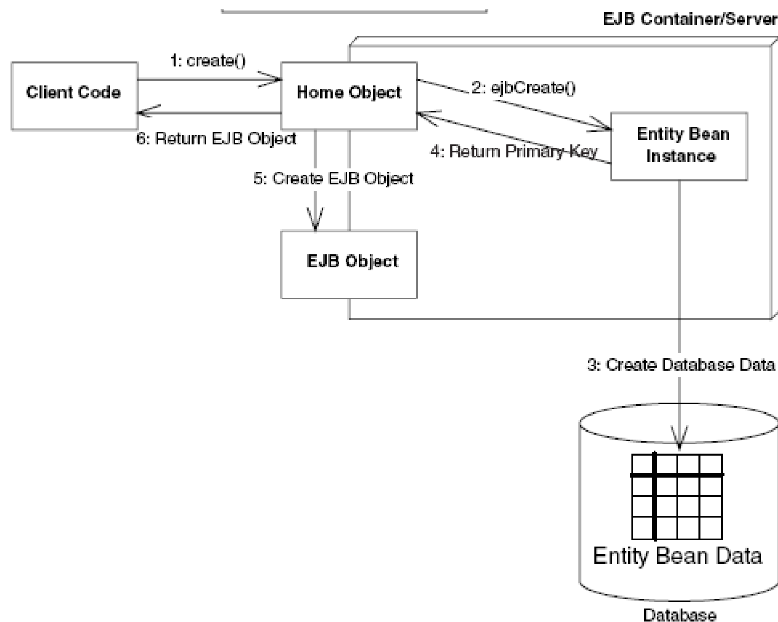


Figura 13: Pasos en la creación de un bean de entidad

Como se aprecia en la figura la creación de los beans de entidad es ligeramente distinta a la de los beans de sesión y entran en juego otros actores con otras responsabilidades. A continuación se van a resumir esos nuevos roles y sus obligaciones.

Una *primary key* es una clase que acompañará a la clase del bean de entidad y que permitirá diferenciar cada instancia del mismo. Una *primary key* será un objeto que puede contener o no varios atributos. Estos atributos definidos como públicos serán los datos necesarios para identificar una instancia particular del bean. La especificación EJB permite que se definan los atributos necesarios para asegurar la unicidad de esa *primary key*. Por lo tanto, una *primary key* puede ser desde un simple atributo hasta un objeto entero cuya única limitación es la ser serializable de acuerdo a las reglas de Java.

Por su parte, la instancia del bean de entidad será la encargada de establecer el mapeo a la definición de entidad que maneja el esquema de la base de datos. Es decir, es la clase que establece el mapeo a una tabla de la base de datos y mas específicamente a un registro de esa tabla. Esta clase puede tener métodos simples que manipulen o accedan a los datos como métodos *get* y *set*. Además, de estos métodos la clase del bean de entidad, al igual que la clase de los bean de sesión, requiere de una serie de

métodos estándar de *call-back* que serán llamados por el contenedor EJB de forma apropiada para gestionar el bean. Dichos métodos ya se vieron con anterioridad cuando se estudiaron los beans de sesión por lo que no se va a profundizar en ellos en este apartado.

7.3.2. Interacción con la base de datos

Los beans de entidad deben considerarse como vistas de una base de datos y debe pensarse que la instancia de un bean de entidad y sus correspondientes datos del dispositivo de almacenamiento son exactamente lo mismo. El encargado de mantener esa sincronización entre ambos mundos es el contenedor EJB. Esto es posible ya que el ciclo de vida de los beans está gestionado por dicho contenedor. El contenedor EJB garantiza la sincronización de la misma forma que el contenedor de servlets garantiza que antes de llamar al método doGet( ) o doPost( ) de un servlet, se ha llamado previamente al método init( ) del mismo.

Efectivamente, existen distintas copias de los mismos datos (en memoria y en base de datos) pero es responsabilidad del contenedor EJB (ya sean beans BMP o CMP) el mantener absolutamente sincronizados ambos mundos (objeto y relacional). Para conseguirlo el contenedor debe implementar mecanismos de transferencia de información. Para ello utiliza rutinas periódicas de sincronización mediante la llamada a dos métodos que deben existir en la implementación de la clase del bean como ejbLoad( ) y ejbStore( ).

Estos dos métodos se definen como métodos *call-back*, es decir, que son métodos implementados en el bean y que son invocados por el contenedor EJB. Dichos métodos nunca deben ser invocados por el cliente de forma explícita, ya que esta es una de las claves o mejoras que aporta la especificación EJB. El desarrollador no debe preocuparse en ningún momento de cuando realizar la sincronización de la información y puede asumir sin ningún riesgo que ambos mundos se encuentran absolutamente sincronizados siempre.

La única diferencia de los métodos ejbLoad( ) y ejbStore( ) entre los beans BMP y CMP es si su implementación tiene o no contenido. En BMP, estos métodos contienen código para leer los datos de la base de datos y para escribir los cambios en la base de datos, respectivamente. En otras palabras, los métodos ejbLoad( ) y ejbStore( ) deben incluir lógica de acceso a base de datos para que el bean pueda cargar y almacenar sus datos cuando el contenedor EJB se lo diga. En cambio en CMP estos métodos estarán implementados a vacío ya que es el contenedor quien se encargará de gestionar la persistencia sobrescribiendo esos dos métodos *call-back* en una subclase que el contendor generará tal y como se verá mas adelante.

Sin embargo, las llamadas que hace el contenedor a esos métodos si que son iguales e independientes del tipo de bean del que se trate. Estos métodos de la instancia del bean se llaman cuando el contenedor EJB decide que es un buen momento para leer o escribir los datos. El contenedor ejecutará automáticamente los métodos ejbLoad( ) y ejbStore( ) cuando lo considere oportuno. Por ejemplo, el método ejbLoad( ) se invoca normalmente al principio de una transacción, justo antes de que el contenedor delegue en un método de negocio del bean.

Por otra parte, hay que aclarar que según la especificación EJB un bean de entidad no existe técnicamente hasta después de que sus datos hayan sido insertados en la base de datos, lo que ocurre durante la ejecución del método ejbCreate( ) de la instancia del bean. Una vez que los datos han sido insertados, el bean de entidad existe y puede acceder a su propia clave primaria y a sus referencias remotas, lo que no es posible hasta que se complete el método ejbCreate( ) y hasta que los datos estén en la base de datos. Por lo tanto, y debido a la sincronización necesaria entre los mundos objeto-relacional sería lógico pensar que cuando se inicializa en memoria una instancia de un bean, durante la ejecución del método ejbCreate( ), se debería desencadenar la creación de los datos correspondientes a ese bean en la base de datos asociada. Y efectivamente eso es lo que sucede.

Cuando el método ejbCreate( ) del bean de entidad con persistencia gestionado por el bean es llamado, este método será el responsable de crear en la base de datos los datos asociados. De la misma forma, cuando se llama al método ejbRemove( ) de la instancia de un bean BMP, el propio bean es el encargado de borrar los datos de la

base de datos. Por el contrario, si se usan beans CMP, al estar la persistencia gestionada por el contenedor EJB, será éste quién realice las tareas correspondientes y en este caso ambos métodos (ejbCreate( ) y ejbRemove( )) estarán implementados a vacío.

Finalmente, nótese que el método remove( ) definido por la especificación EJB para cualquier tipo de bean, no tiene parámetros y por lo tanto, puede considerarse como un método único. Este método puede ser invocado tanto desde el objeto Home, como desde el objeto EJB ya que en ambos está implementado. Cualquiera de estos dos métodos lo que se desencadena es una llamada el método ejbRemove( ) de la instancia del bean. Este método ejbRemove( ) no elimina la instancia del bean ni en el caso de los beans de sesión ni en el de los de entidad. En su lugar, en los beans de entidad el método simplemente elimina los datos relacionados en la base de datos. De esta forma, la instancia del bean puede ser reciclada para manejar otros datos de la base de datos. Además, hay que recordar incluso, que un bean de entidad tiene un ciclo de vida que le permite sobrevivir a fallos en el contenedor y que supera la finalización de la sesión de un cliente.

Esta particularidad del método remove( ), en cambio, no se produce con el método create( ) que solo está implementado en el objeto *Home*, que es el único encargado de la creación del objeto EJB a través del cual el cliente puede acceder a la instancia del bean.

7.3.3. Llamadas entre beans

Coma ya se había comentado anteriormente, un bean de entidad no debe realizar tareas complejas de lógica asociada al *workflow* de la aplicación. Solamente un bean de este tipo puede incluir lógica de negocio si esa lógica no implica relaciones con otras entidades de la aplicación, si el bean no participa en la gestión de flujo de interacción con el cliente o no lleva asociado código de acceso a datos, es decir, que no usa un Data Access Object (DAO) que se usaría en un bean con persistencia manejada por el propio bean.

Cualquier modelo de objetos EJB diseñado para una aplicación de cierta entidad presenta distintos beans que colaboran entre si, y por lo tanto, presenta beans que reciben llamadas de otros beans. En este caso, para que un bean pueda llamar a otro bean se debe seguir el mismo proceso que si se pretende llamar a un bean desde un cliente, es decir:

1. Buscar el objeto Home del bean remoto via JNDI.
2. Invocar el método crear del objeto Home.
3. Invocar los métodos de lógica de negocio sobre el objeto EJB.
4. Invocar el método remove( ) del objeto EJB.

Como es de esperar, la especificación EJB aporta distintas facilidades para la interacción de los beans ofreciendo mecanismos como la búsqueda por defecto en JNDI, la utilización de las referencias EJB y, por supuesto, la utilización de interfaces locales tal y como se expone a continuación.

Con la búsqueda JNDI por defecto, la especificación EJB libera completamente al desarrollador de beans de la preocupación de tener que inicializar los parámetros JNDI que teníamos que realizar cuando se llamaba a los beans desde un cliente. El desarrollador se despreocupa de quien será su servidor JNDI. Cuando se llama a un bean desde otro bean no hace falta esa inicialización de parámetros. Se supone que el bean llamante, que ya corre en un contenedor EJB, tiene un contexto asociado y el bean llamado, simplemente tendrá que utilizar el *initial context* que por defecto ofrece JNDI. Para conseguir ese *initial context* por defecto habría que invocar el constructor sin argumentos InitialContext( ).

Con la utilización de las referencias EJB, como ya se ha comentado anteriormente, se dispone de un *nickname* que determina la localización en la que se debe buscar un determinado recurso o bean en el servicio de nombres. El código que suministra el desarrollador será independiente de la localización real buscando el objeto *home* de acuerdo a ese *nickname* ya que puede no coincidir con la localización real donde se ha desplegado el recurso. En última instancia, será el encargado del despliegue quien solucione el nivel de in-dirección modificando los descriptores de despliegue de la

aplicación y enlazando convenientemente el *nickname* del desarrollador con la ubicación real del recurso en tiempo de despliegue.

Finalmente, otra forma de acceder de un bean a otro bean es a través de los interfaces locales. Para utilizar este tipo de llamadas se debe hacer uso de los interfaces locales en vez de los interfaces remotos. Además, habría que cambiar ligeramente los descriptores de despliegue cambiando el nombre de los elementos <ejb-ref> por el de <ejb-local-ref> y, finalmente, hacer uso del constructor por defecto para crear el contexto inicial por defecto JNDI y buscar en el contexto recomendado por la especificación java:comp/env/ejb.

Sin embargo, no todo son ventajas a la hora de la colaboración entre beans. También es conveniente tener en cuenta que la especificación EJB propone ciertas restricciones a la hora de utilizar dicha colaboración. Por ejemplo, los beans de entidad no deben nunca invocarse directamente desde clientes remotos. En su lugar los beans de entidad deben ser accedidos siempre a través de beans de sesión (implementando así un patrón de diseño de sesión façade[6] o fachada) o a través de otros beans de entidad locales, es decir, que corren en el mismo proceso. En definitiva, un bean de entidad debe implementarse utilizando siempre su interfaz local y nunca ser llamado a través de su interfaz remoto.

Con la utilización de los interfaces locales para los beans de entidad, se eliminan todos los procesos y mecanismos propios de la comunicación remota que ralentizan el tiempo de servicio y que no son necesarios cuando se accede a un bean de entidad desde un bean de sesión o desde otro bean de entidad que reside en el mismo contenedor EJB que el llamado.

[6] El patrón de diseño façade o fachada trata de simplificar la interfaz entre dos sistemas ocultando un sistema complejo detrás de una clase que hace las veces de pantalla o fachada. La idea es ocultar todo lo posible la complejidad de un sistema, el conjunto de clases o componentes que lo forman, de forma que solo se ofrezca un (o unos pocos) punto de entrada al sistema tapado por la fachada.

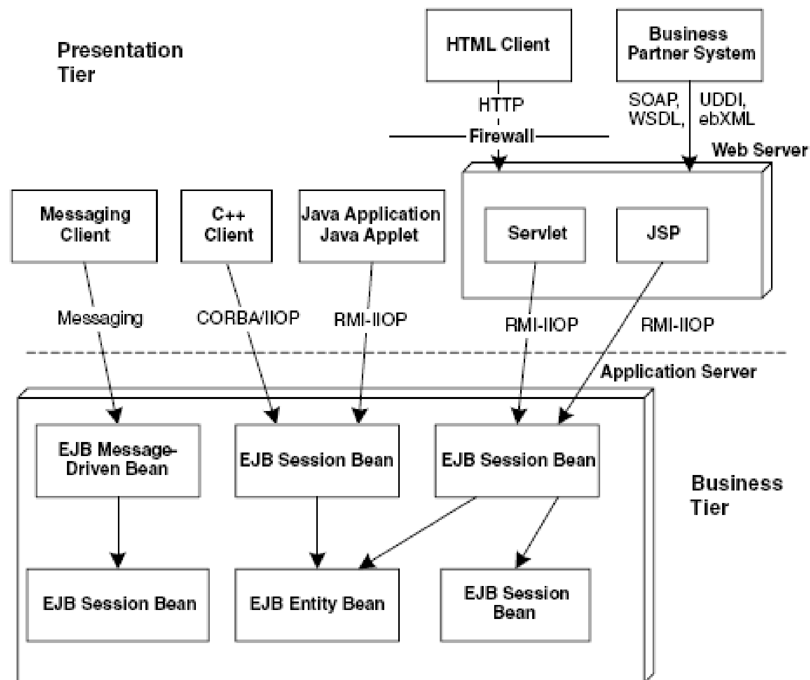


Figura 14: Colaboración entre beans de entidad y sesión

7.3.4. Otras características de los beans de entidad

7.3.4.1. Finders en beans de entidad

Debido a que los beans de entidad pueden identificarse de forma univoca, ya que se corresponden con datos almacenados en base de datos, se dice que los beans de entidad pueden ser más buscados que creados. Por lo tanto, encontrar un bean de entidad resulta una operación muy similar a ejecutar un sentencia SELECT en SQL, en la que se busca una información particular persistente en la base datos. Obviamente la operación de búsqueda no aplica a los beans de sesión, como se vió en su momento, ya que no son objetos permanentes y su ciclo de vida dura únicamente el tiempo que dura la sesión del cliente.

La especificación EJB permite que se definan distintas formas de búsqueda para un bean de entidad. Para conseguirlo, basta con añadir a los métodos estándar de ciclo de vida presentes en el interfaz Home del bean, los métodos correspondientes a las búsquedas. Estos métodos serán conocidos como *finders*. Un interface *Home* de un bean de entidad puede definir uno o más métodos *finder*, para encontrar un objeto

entidad o incluso una colección de ellos. El nombre de cada método *finder* empieza con el prefijo *find*.

La principal diferencia entre los beans BMP y CMP en cuanto a los métodos *finders* es que no existen dichos métodos en la clase del bean CMP como si los habrá en los beans BMP. Esto es debido a que la búsqueda de los beans, en el caso de los CMP, la controla el propio contenedor EJB. Además, en el caso de los beans CMP, tampoco se necesita generar las clases *primary key*, no se necesita implementar clases DAO o de acceso a datos, ni se tiene que escribir ningún descriptor para la fuente de datos como se debe hacer para los beans de entidad BMP, ya que el contenedor EJB es el responsable de todos esto.

Tal y como se verá mas adelante cuando se estudien los beans CMP, todo método *finder*, excepto el método findByPrimaryKey(key), debe ir asociado con un elemento <query> que debe estar presente en el descriptor de despliegue. El proveedor del bean de entidad CMP declarará una consulta de tipo EJB/QL y la asociará con el método *finder* en ese descriptor de despliegue. Un método *finder* en CMP normalmente se caracterizará por un *String* de consulta EJB/QL especificado mediante el elemento <query>.

Quizás las mayores críticas de los beans de entidad vienen por la lentitud en las funciones de búsqueda de los *finders*. En realidad lo que hacen estos métodos es devolver una colección de interfaces remotos que cumplen una serie de condiciones, de forma similar a como lo hacen las instrucciones SQL. Bien es cierto que los métodos *finders* pueden ser más lentos que una simple llamada SQL, pero no lo suelen ser mucho más, si se hace un correcto uso en el método *finder* de las transacciones y se minimiza el número de llamadas a través de la red. En cualquiera de los casos, no suele ser recomendable hacer este tipo de llamas si sólo se va ha hacer operaciones de lectura en la base de datos.

7.3.4.2. Transacciones en beans de entidad

Como ya se ha comentado previamente, la especificación EJB determina que la forma de prestar un servicio a los clientes de forma concurrente es implementando código *single-threaded*, pero a su vez, permitiendo que múltiples instancias de un mismo bean ofrezcan el servicio a los clientes. En el caso particular de los bean de entidad, múltiples instancias de un bean pueden representar simultáneamente los mismos datos almacenados en base de datos. Y es aquí, debido a las distintas representaciones de los mismos datos, en donde aparece el gran problema de los beans de entidad: la consistencia de los datos.

Si se tienen varias instancias que representan los mismos datos, algunas de esas instancias pueden no tener sus datos debidamente actualizados. Por este motivo, para conseguir la consistencia de las instancias de los beans de entidad, estos deben ser sincronizados con la base de datos mediante la utilización periódica de rutinas. Estas rutinas, tal y como se citó anteriormente consisten en las llamadas a los métodos *call-back*, ejbLoad( ) y ejbStore( ).

La frecuencia con la que el contenedor EJB realiza las llamadas a estos métodos de sincronización con la base de datos esta determinada por el mecanismo de transacciones. Las transacciones permiten que cada petición de un cliente sea ejecutada de forma independiente al resto de llamadas de otros clientes. De esta forma, las transacciones evitan las posibles inconsistencias entre dos instancias que se corresponden con los mismos datos almacenados en el dispositivo de almacenamiento. Con las transacciones, los clientes tienen la sensación de trabajar con una única instancia del bean cuando en realidad el contenedor EJB se encarga de manejar las distintas instancias. Así los clientes tienen la sensación de acceder a los datos de forma exclusiva, cuando en realidad hay muchos clientes manejando el mismo conjunto de datos.

En definitiva, los beans de entidad permiten compartir entre varios clientes el acceso a los datos de un mismo bean compartiendo varias instancias de ese bean. Con ello se consigue dar un servicio concurrente que maneja el contenedor EJB. Para conseguir dicha concurrencia los beans de entidad deben participar en transacciones,

especificando unas determinadas propiedades transaccionales. Dichas propiedades transaccionales es especifican declarativamente en los descriptores de despliegue de forma que los límites de las transacciones los maneja el contenedor.

7.3.4.3. Pooling en beans de entidad

La especificación EJB determina que efectivamente se puede crear distintas instancias de un mismo bean para producir concurrencia. Sin embargo, la creación y destrucción de instancias de un bean no es una operación trivial. Por el contrario, se trata de operaciones costosas, y que especialmente, afectan al rendimiento de la aplicación si las peticiones de los clientes llegan con frecuencia.

Para solucionar este inconveniente de rendimiento, los beans se mantienen dentro de un *pool* de instancias de beans que es gestionado y mantenido en memoria por el contenedor EJB. Así las instancias de un bean de entidad pueden ser reutilizadas por parte del contenedor. El contenedor elige una instancia de un bean de entidad de entre las que hay en el *pool* y le asigna dinámicamente un objeto EJB y su clave primaria correspondiente. De esta forma, cada instancia de un bean queda asociada a un cliente concreto.

Por lo tanto, debe estar claro en este momento que el contenedor EJB implementa la concurrencia asignando dinámicamente instancias de un mismo bean de entidad que representan los mismos datos en base de datos. Sin embargo, no hay que olvidar que la reutilización de instancias puede ir más allá. El contenedor EJB incluso, puede reutilizar y asignar instancias de distintos beans de entidad elegidos del *pool*. Lógicamente, dichas instancias representarán diferentes datos en base de datos.

En este último caso, el *pooling* con asignación de instancias a diferentes objetos EJB, cada instancia del bean de entidad perteneciente al *pool* representa un registro distinto de la base de datos en memoria. La utilización del *pooling* con distintas instancias consigue accesos muy rápidos a dichos registros, y por lo tanto, resulta interesante para mantener en memoria aquellos registros de la base de datos que se considera que los clientes van a volver a utilizar con bstante frecuencia. Sin embargo, este caso

introduce ciertas complicaciones debidas a la adquisición y liberación de recursos por parte de esos objetos.

Para solucionar los problemas relacionados con la gestión de los recursos la clase del bean de entidad debe implementar los dos métodos *call-back* utilizados a tal efecto y que ya fueron vistos anteriormente cuando se estudiaron los beans de sesión con estado.

El método ejbActivate( ) es el método que llama el contenedor EJB cuando se reserva un bean del *pool* de instancias y se lleva a memoria. El proceso se denomina activación de beans entidad e implica una carga en memoria del estado y la adquisición de los recursos que pueda necesitar el bean como sockets o conexiones a base de datos.

Por el contrario, el método ejbPassivate( ) realiza la pasivación de los beans de entidad. Este método *call-back* se invoca cuando la instancia del bean sale de memoria y se libera la instancia del *pool*. La pasivación, como es habitual, implica una salvaguarda del estado del bean llevándolo al dispositivo utilizado para su almacenamiento.

Por este motivo, cuando una instancia de un bean es pasivada, no solo debe liberar sus recursos, sino que también se debe salvar su estado actualizado con las últimas modificaciones de los datos hechas hasta ese momento. Esta salvaguarda de estado debe ser almacenado en la base de datos. Para ello el contenedor llama al método *call-back* ejbStore( ) previamente a llamar al método de la pasivación. De la misma forma, durante la activación no solo se reservan los recursos sino que se recupera el estado con los datos mas recientes de la base de datos. Para cargar esos datos en la instancia del bean, el contenedor llama al método ejbLoad( ) después de llevar a cabo la activación.

7.3.5. El objeto EJB Context

Cualquier tipo de bean, sea de sesión o de entidad, tiene un objeto contexto o *context object* asociado, que determina el entorno particular de ejecución de dicho bean. Estos

objetos contienen la información del entorno que el contenedor EJB les configura. De esta forma, los bean pueden acceder a estos objetos con el fin de conocer una serie de información que les resulta útil, como puede ser información de seguridad o de la gestión de transacciones que realiza el contenedor.

La especificación EJB determina que el interfaz que define un contexto EJB genérico es el interfaz javax.ejb.EJBContext. Este interfaz define los métodos que se muestran a continuación:

```
public interface javax.ejb.EJBContext {

        public javax.ejb.EJBHome getEJBHome();
        public javax.ejb.EJBLocalHome getEJBLocalHome();
        public java.security.Principal getCallerPrincipal();
        public boolean isCallerInRole(java.lang.String);
        public javax.transaction.UserTransaction getUserTransaction();
        public void setRollbackOnly();
        public boolean getRollbackOnly();

}
```

- public javax.ejb.EJBHome getEJBHome( );

  Este método lo puede llamar el propio bean para conocer una referencia a su propio objeto Home. No se suele utilizar mucho, pero con esa referencia se podrían crear, destruir o encontrar beans.

- public javax.ejb.EJBLocalHome getEJBLocalHome( );

  El mismo método que el anterior con la excepción de que en este caso se devuelve el interfaz local en lugar del remoto.

- public java.security.Principal getCallerPrincipal();

  Este método devuelve el objeto *Principal* de seguridad asociado al usuario que actualmente esta autenticado en el sistema. Se puede utilizar ese objeto para hacer consultas en la base de datos o para realizar otras operaciones.

- public boolean isCallerInRole(java.lang.String);

  Pregunta al contenedor EJB si el usuario actual está dentro del rol apropiado para realizar una determinada operación. Este método resulta muy útil de cara a la seguridad programática.

- public javax.transaction.UserTransaction getUserTransaction();

  Este método devuelve el interfaz JTA UserTransaction que permite realizar las operaciones programáticas de transacción.

- public void setRollbackOnly();

  Este método se puede llamar para forzar la ejecución de un *roll-back* de la transacción que este en ese momento en curso. Esta llamada se puede producir si dentro de la ejecución del bean se produce algún error.

- public boolean getRollbackOnly();

  Pregunta al contenedor si la transacción está condenada a realizar un *call-back*. Conocer esta información resulta útil ya que se pueden evitar una serie de operaciones que gastan recursos.

Se acaba de especificar el interfaz genérico del contexto EJB. Sin embargo, cada uno de los tipos de beans tiene su propio interfaz que le permite definir sus propios métodos respecto al objeto contexto. Así pues, la especificación EJB define los interfaces javax.ejb.SessionContext y javax.ejb.EntityContext para los beans de sesión y de entidad respectivamente. Estos dos interfaces extienden el interfaz genérico javax.ejb.EJBContext y aportan cada uno los métodos particulares de cada tipo, tal y como se puede ver a continuación:

```
public interface javax.ejb.SessionContext extends javax.ejb.EJBContext {

        public javax.ejb.EJBLocalObject getEJBLocalObject();
        public javax.ejb.EJBObject getEJBObject();

}
```

Estos dos métodos son supremamente importantes si, por ejemplo, un bean quiere llamar a otro bean y para ello requiere pasarle la referencia de si mismo. Este mecanismo es muy útil y en Java esto se suele solucionar con la utilización de la palabra reservada *this*.

Sin embargo, en la tecnología EJB hay que recordar un detalle fundamental y es el hecho de que un bean nunca puede ser accedido por un cliente de forma directa sino que siempre debe realizarse el acceso a través de un objeto EJB. Por lo tanto, si un bean utiliza la referencia *this* para pasarla como parámetro, estará permitiendo que el otro bean, o bean llamante, referencie directamente a la clase del bean llamado y no lo haga a través del objeto EJB asociado. Por este motivo, un bean tiene que pasar su referencia a si mismo usando una referencia a su objeto EJB con alguno de los dos métodos definidos en el interfaz javax.ejb.SessionContext y no a través de la referencia *this*. Como es lógico, la única diferencia entre ambos métodos es que uno devuelve el interfaz local y el otro el remoto.

Por otro lado, el interfaz javax.ejb.EntityContext también aporta los dos métodos que devuelven el objeto EJB asociado y que ya han sido explicados anteriormente para los beans de sesión. Finalmente, este interfaz define un método más, tal y como se muestra a continuación:

```
public interface javax.ejb.EntityContext extends javax.ejb.EJBContext {

        public javax.ejb.EJBLocalObject getEJBLocalObject();
        public javax.ejb.EJBObject getEJBObject();
        public java.lang.Object getPrimaryKey();

}
```

- public java.lang.Object getPrimaryKey();
  Este método lo utilizan solo los beans de entidad. Devuelve la clave primaria que caracteriza de forma univoca al bean de entidad y que está en ese momento asociada con la instancia del bean. Este método se llamará siempre que se necesite saber la información de la base de datos con la que está asociada la instancia del bean. Debido a que las instancias de los beans pueden ser reutilizadas y a que son gestionadas mediante un *pool*, es posible que le contenedor EJB necesite en algún momento utilizar una instancia de un bean para soportar unos datos u otros. Para conseguir esto el contendor EJB debe realizar las tareas de activación y pasivación ya explicadas. Cuando se producen estas tareas, cada instancia del bean extraída del *pool* recibe una nueva instancia de datos de la base de datos, y por lo tanto, una nueva *primary key*. El bean no conoce de forma explícita quien es su nueva

primary key, cuando es activado. Para conocerlo, el bean debe realizar la llamada al método *call-back* getPrimaryKey( ) del contenedor para conocer los datos de la base de datos con los que ahora esta trabajando.

Por lo tanto, este método se usa bastante cuando se tienen beans que realizan sus propias tareas de persistencia, como son los beans BMP. Estos beans deben llamar a este método siempre que necesiten conocer los datos asociados al bean en la base de datos. De hecho, este método getPrimaryKey( ) debe usarse en los siguientes casos específicos:

1. Al ejecutar el método ejbLoad( ): Las llamadas sucesivas que realiza el contenedor a los métodos *call-back* ejbLoad( ) y ejbStore( ) permiten la sincronización del bean con su dispositivo de almacenamiento. El método ejbStore( ) salva los datos y no presenta demasiados problemas porque lo que se tiene que salvar esta en memoria. Sin embargo, ejbLoad( ) trae los datos desde la base de datos con los que se va a poblar la instancia del bean seleccionada del *pool*. Pero el problema es saber qué datos hay que traer. Para saberlo se recurre a la llamada a getPrimaryKey( ) que devuelve la clave que hay que buscar en la base de datos para descubrir los datos relacionados con la instancia.

2. Al ejecutar el método ejbRemove( ): Las llamadas sucesivas que realiza el contenedor a los métodos *call-back* ejbCreate( ) y ejbRemove( ) permiten la creación y destrucción de datos en el dispositivo de almacenamiento. Cuando el contenedor EJB llama al método ejbCreate( ) se sabe qué datos insertar en la base de datos ya que el bean tiene esa información en memoria pasada como parámetros en la creación. Sin embargo, cuando se quiere borrar una instancia, debido a que las instancias se asignan de forma dinámica y a que están gestionadas con un *pool*, resulta imprescindible saber que datos son realmente los que hay que borrar de la base de datos. De nuevo una llamada al método getPrimaryKey( ) del contenedor EJB devolverá, en este caso, la clave del registro que hay que borrar en la base de datos.

7.4. Beans BMP (Bean Managed Persistence)

Hasta este punto se han visto las características principales que distinguen a los beans de entidad en general, tanto BMP como CMP. A partir de este momento se verán las características que diferencian de forma particular a los beans BMP y que ya se han adelantado en los apartados anteriores. Como ya es habitual en los temas que ya se han tratado en el libro, se terminará el estudio de este tipo de beans con el ejemplo correspondiente, acompañado de los comentarios y aclaraciones que se han considerado oportunos.

En BMP, el bean de entidad contiene el código de acceso a la base de datos (normalmente utilizando JDBC) y es el responsable de leer y escribir en la base de datos su propio estado. Sin embargo, los beans BMP tienen mucha ayuda, ya que el contenedor EJB alertará al bean siempre que sea necesario hacer una actualización o leer su estado desde la base de datos. El contenedor EJB también manejará cualquier transacción necesaria para que la base de datos mantenga la integridad de la información que almacena.

La persistencia manejada por el bean le da al desarrollador de beans la flexibilidad de realizar operaciones de persistencia que son demasiado complicadas para el contenedor EJB o para usar una fuente de datos que no está soportada por el contenedor. Este tipo de persistencia le da más flexibilidad al programador porque controla todos los accesos a la fuente de datos.

Con el paso del tiempo y debido al rápido desarrollo de los servidores de aplicaciones se puede decir que cada vez se usan menos este tipo de beans BMP. Esta situación es debida a las mejoras introducidas por los servidores en los contendores EJB que cada vez realizan de forma más acertada las tareas de persistencia para los beans CMP. Sin embargo, todavía hay algunas operaciones que sólo se pueden realizar gracias a las habilidades de los programadores y que requieren el uso de BMP. En cualquiera de los casos siguen siendo imprescindibles cuando la persistencia no se consigue a través de una base de datos relacional.

7.4.1. Arquitectura BMP

A continuación se muestra un diagrama que representa el modelo de objetos típico de un bean de entidad con la persistencia gestionada por el propio bean (BMP). Este diagrama reune todas las clases que se requieren a la hora de desarrollar cualquier bean BMP y que ya se han ido explicando, una a una, a lo largo de los distintos capitulos del libro consumidos hasta ahora.

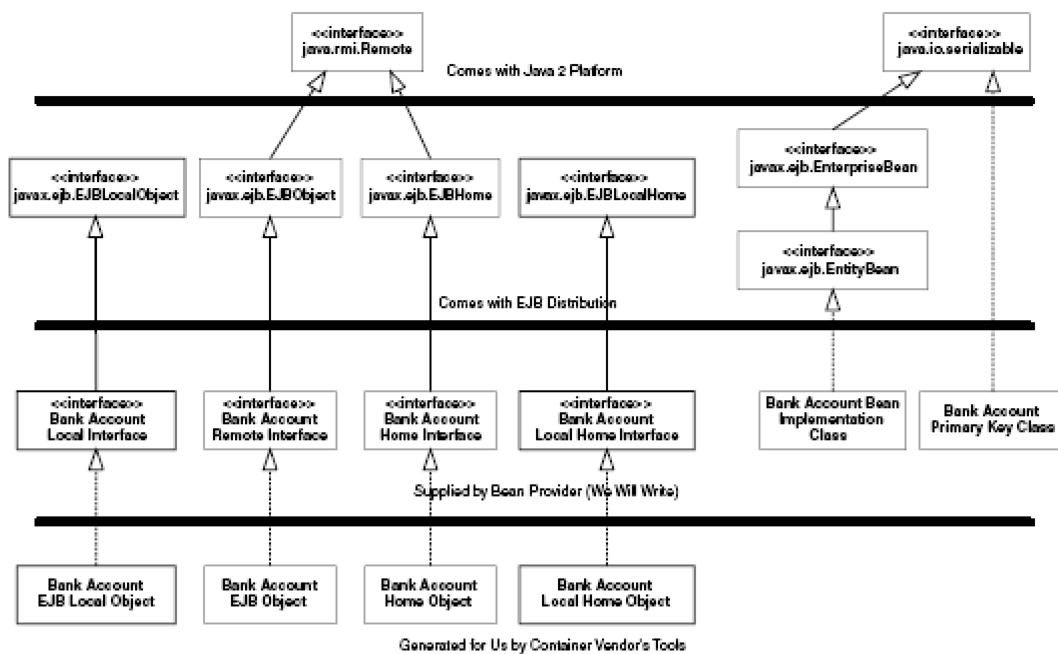


Figura 15: Diagrama de objetos para BMP

Como se puede apreciar en la figura, las distintas clases se encuentran organizadas en dos grupos distintos. Por un lado estan las clases relacionadas con los interfaces EJB y por otro las clases relacionadas con el propio bean. Además, las clases se encuentran divididas de forma horizontal indicando su procedencia y el responsable de aportarlas al desarrollo de la aplicación.

La parte izquierda del diagrama resume las clases relacionadas con las llamadas remotas, y por lo tanto, incluye desde el interfaz java.rmi.Remote que viene con la distribución J2EE, pasando por los interfaces javax.ejb.EJBObject, javax.ejb.EJBHome y los respectivos interfaces locales que pertenecen al paquete EJB, hasta llegar a los interfaces que extienden de los anteriores y que debe aportar el desarrollador para que luego los implemente el propio contenedor EJB. Resulta lógico

pensar que esta parte del diagrama es con la que el lector se encuentra más familiarizado ya que resume todas clases que se han utilizado hasta el momento en el desarrollo de los ejemplos propuestos en el libro.

La parte derecha del diagrama, por su parte, muestra las clases directamente relacionadas con la implementacion del bean. Por lo tanto, en esta parte de incluyen desde el interfaz java.io.Serializable de la J2EE, pasando por los interfaces javax.ejb.EnterpriseBean y javax.ejb.EntityBean del paquete EJB, hasta la clase del bean que debe implementar el desarrollador extendiendo de los interfaces anteriores. Finalmente, para el caso de los beans BMP el diagrama de objetos tambien incluye una clase que debe implementar el desarrollador llamada *primary key* y cuyo cometido ya ha sido expuesto anteriormente.

Es importante destacar que el modelo de objetos para los beans de sesión sería, como cabe esperar debido a la experiencia acumulada hasta este momento, bastante parecido al diagrama anterior. Las escasas diferencias radican en que los beans de sesión extienden del interfaz javax.ejb.SessionBean en lugar del interfaz javax.ejb.EntityBean y que, debido a que los beans de sesión no poseen mecanismos de persistencia, no requieren de la existencia de una clase como la *primary key*, que como ya se ha comentado es específica de los beans de entidad. Por lo demás el diagrama es básicamente el mismo para ambos tipos de beans.

Una vez vistos los fundamentos teóricos de este tipo de beans de entidad BMP y de conocer el entorno arquitectónico y tecnológico en el que se mueven dichos beans, es el momento de ver de forma mas clara su funcionamiento con la realización de la aplicación de ejemplo correspondiente.

7.4.2. Ejemplo BMP: gestión de cuentas corrientes en un banco

7.4.2.1. Introducción

Hasta este momento se han implementado aplicaciones para los ejemplos del libro con un afán más didáctico, si se quiere, que casos realmente prácticos y de la vida real.

Ejemplos como la aplicación HolaMundo!! o la Base de Datos Literaria seguramente no representan el mejor ejemplo de los requisitos software típicos que se esperan de una verdadera aplicación distribuida que utilice la tecnología EJB. Sin embargo, han cumplido a la perfección su papel de aplicaciones con una lógica sencilla, que por encima de todo, permiten descubrir al lector todas las particularidades de una tecnología como es la de EJB.

Por el contrario, con el ejemplo que se propone a continuación se intenta acercar al lector a una aplicación más parecida a lo que se realmente se implementa en la vida real. Un caso menos conceptual y más práctico. Esta aplicación por tanto, aporta, sino todas, si que las principales funcionalidades que necesita un supuesto banco para gestionar las cuentas corrientes de sus clientes. Hay que aclarar que no se ha considerado oportuno implementar toda la posible funcionalidad que debería prestar dicho banco, ya que la cantidad de código que llegaría a generarse podría dificultar un entendimiento claro y preciso de las características de este tipo de beans BMP, que al fin y al cabo, sigue siendo el objetivo prioritario. Aun así, el lector podrá comprobar como la cantidad de código de este ejemplo ha aumentado considerablemente con respecto a los ejemplos anteriores, en gran parte como se podrá ver a continuación, debido a la inclusión de toda la lógica de persistencia JDBC necesitaría para este tipo de beans.

En definitiva, la aplicación de ejemplo propuesta se denominará Gestor de Cuentas y será una aplicación *stand-alone* que permitirá al cliente de la aplicación realizar una serie de operaciones básicas sobre las distintas cuentas corrientes que gestiona el banco. Dichas operaciones pueden ser, por ejemplo, el ingreso o reintegro de dinero de una cuenta o la modificación de los datos personales del titular de esa cuenta. Toda esta información de las cuentas corrientes del banco que gestiona el sistema, como es lógico y habitual en la vida real, se almacenará en una base de datos de la que se hablará a continuación.

El nombre de Gestor de Cuentas para la aplicación surge debido a que es una aplicación que permite al propio banco, entendido como el cliente de la aplicación, interactuar con las cuentas de sus clientes y realizar ciertas comprobaciones de funcionamiento del sistema. En definitiva se podría decir, que la aplicación Gestor de

Cuentas está dirigida más al banco que a los propios clientes del mismo. De esta forma, el banco podrá conocer, por ejemplo, el total de dinero acumulado en el saldo de todas sus cuentas o crear nuevos cuentas corrientes para realizar las operaciones típicas mencionadas anteriormente. Por ejemplo, para que el banco como cliente de la aplicación pueda crear una nueva cuenta corriente se deberán proporcionar al sistema el nombre de un titular y un identificador único de cuenta.

El concepto de cuenta corriente será implementado en la aplicación, como es lógico, como un bean de entidad BMP que encapsule toda la lógica necesaria para cumplir los requisitos y suministrar al banco la funcionalidad requerida. Este bean gestionará toda la información realtiva a una cuenta corriente almacenando para cada una de ellas su código de cuenta, el nombre del titular y el saldo. Además, este bean BMP denominado CuentaCorriente, debe permitirle al banco operar con ella, permitiéndole hacer ingresos, reintegros y conocer su saldo. Este bean lógicamente se encargará de gestionar su propia persistencia como todos los beans BMP, utilizando para ello una base de datos.

Un detalle muy a tener en cuenta en este momento, es que un bean de entidad BMP como el que se propone para la implementacion del ejemplo, introduce ciertas restricciones de diseño que no deben pasar por alto. Tal y como se ha comentado anteriormente cuando se vieron los conceptos teóricos acerca de la colaboración entre beans, no se recomienda que un bean de entidad sea llamado directamente desde un cliente, en este caso el banco.

De hecho, en este tipo de situaciones, se recomienda que sea un bean de sesión (GestorCuentas) quien sea el llamado desde el código del cliente (Banco), que realmente será remoto. Posteriormente ese bean de sesión (GestorCuentas) llamará a un bean de entidad (CuentaCorriente) que se encuentra en su mismo espacio de direcciones y que, por lo tanto, no requiere utilizar los interfaces remotos y puede ser implementado con los interfaces locales.

Por lo tanto, resulta indispensable en la aplicación propuesta la implementación de un bean de sesion sin estado que se encargue de realizar todas las tareas de lógica de negocio que necesita el banco para gestionar sus cuentas. Este bean de sesión se

denominará GestorCuentas y utilizará al bean de entidad BMP CuentaCorriente, del que ya se ha hablado, para gestionar una cuenta. Pues bien, el hecho de disponer de un bean de sesión sin estado que accede al bean de entidad BMP, abre una puerta enorme que permite que se cumpla la recomendación de la especificación EJB en cuanto a que los beans de entidad sean implementados utilizando los interfaces locales en lugar de los remotos. Además, es la mejor excusa para utilizar estos interfaces que hasta ahora no se han usado.

En definitiva, la aplicación del ejemplo tendrá varios beans de distintos tipos colaborando. El bean de sesión sin estado se implementará con interfaces remoto y se encargará de recibir las peticiones de los clientes y llamará al beans de entidad BMP, que correrá en la misma máquina que el de sesión, y que por lo tanto, se podrá implementar utilizando interfaces locales.

Finalmente, ahora si que se puede afirmar que se está frente a una aplicación más real y compleja que basa su funcionamiento en la colaboración de distintos componentes EJB tal y como sucede en la mayoria de la aplicaciones distribuidas implementadas en clientes reales. Todo esto, a pesar de que, es verdad, no se está siendo exhaustivo en cuanto a la funcionalidad que se proporciona en esta aplicación. Como ya se ha dicho, esto se debe a que lo que se pretende es fundamentar los conocimientos sobre los beans BMP de la tecnología EJB y no inundar el libro con el código de una aplicación de proporciones empresariales.

7.4.2.2. Requisitos para ejecutar el ejemplo: la base de datos

Al estar inmersos en el estudio de los beans de entidad BMP, es inevitable hablar de un sistema gestor de base de datos que proporcione la persistencia asociada a estos componentes. Por lo tanto, para conseguir un sopore de almacenamiento robusto para toda la información que gestionan los beans que se desarrollaran en este ejemplo de beans BMP (y que lógicamente también valdrá para el siguiente ejemplo relativo a bean con persistencia gestionada por el contenedor o CMP), se utilizará una base de datos MySQL. MySQL es una de las más populares bases de datos de código abierto

que existen en el mercado y que es usada incluso en proyectos pertenecientes a la NASA.

Para configurar y poblar la base de datos MySQL mencionada se utilizarán una serie de scripts que se verán a continuación. Sin embargo, antes de poder interactuar con dicha base de datos, es necesario tenerla instalada, configurarla convenientemente y sobre todo, configurar el servidor de aplicaciones JBoss para que interctúe con esa base de datos cuando tenga que realizar la tareas de persistencia de los beans creados en la aplicación del ejemplo.

Por lo tanto, en primer lugar para poder ejecutar el ejemplo Gestor de Cuentas, el lector debe tener instalado y funcionando correctamente en su máquina el sistema gestor de base de datos MySQL. Se puede descargar de forma completamente gratuita, ya que es código abierto y de libre distribución, la última versión de este servidor de base de datos del siguiente sitio web: http://www.mysql.com.

Una vez instalada la base de datos lo primero que hay que crear es la base de datos contra la que se realizarán todas las operaciones que requieren los beans de entidad de la aplicación. Para ello, hay que ejecutar el programa cliente de MySQL que permite, en modo consola, ir introduciendo y ejecutando los comandos de administración necesarios para la creación de la base de datos. Es recomendable asegurarse de que se está conectado a la base de datos con suficientes privilegios como para crear la base de datos. Para ello habría que entrar en la aplicación como *root* (que será el usuario con privilegios suficientes y que debió ser configurado durante la instalacion de la herramienta) o especificar una vez dentro de la aplicación la opción "-u root" para cambiar al usuario correspondiente.

Para acceder a esta herramienta hay que arrancar la aplicación MySQL Command Line Client desde el administrador de programas del sistema operativo. Esta aplicación proporciona un interfaz de consola bastante sencillo para realizar distintas operaciones. Una vez dentro de la aplicacion se debe ver el prompt *mysql>* y en el que hay que insertar el siguiente comando para crear la base de datos Banco que se utilizará para el ejemplo:

```
mysql> CREATE DATABASE Banco;

Query OK, 1 row affected (0.05 sec)
```

Para comprobar que la base de datos ha sido creada correctamente se puede introducir el comando que se muestra a continuación y que debe producir un resultado similar al siguiente:

```
mysql> SHOW DATABASES;

+----------+
| Database |
+----------+
| banco |
| mysql |
| test |
+----------+

3 rows in set (0.00 sec)
```

Finalmente, se podría crear un nuevo usuario al que le dariamos todos los privilegios para interactuar con la base de datos recién creada. Tambien, se podría utilizar el propio usuario *root* que ya debe estar creado. No obstante, si se decide crear un nuevo usuario, su nombre podría ser "lector" y su *password* será "password". La creación del usuario se haría de la siguiente forma:

```
mysql> GRANT ALL PRIVILEGES ON banco.* TO lector@localhost IDENTIFIED BY
'password';

Query OK, 0 rows affected (0.06 sec)
```

De nuevo se podría comprobar que el nuevo usuario ha sido creado con exito realizando la siguiente consulta sobre la base de datos:

```
mysql> select User,Host,Password from mysql.User;

+-------+---------+----------------+
| User | Host | Password |
+-------+---------+----------------+
| root | localhost | |
| root | % | |
| lector | localhost | *7A038F15BF53D81C9D2FB5360F51819234EAB82A |
+-------+---------+----------------+
```

```
5 rows in set (0.02 sec)
```

También es importante mencionar que es necesario instalar el driver JDBC que permitirá acceder desde código Java a la base de datos MySQL para realizar las sentencias SQL que requiera la aplicación. El driver que proporciona MySQL se denomina Connector/J y tambien es descargable del sitio web mencionado anteriormente. La configuración del servidor de aplicaciones JBoss para que acceda a la base de datos se verá a continuacion pero antes hay que permitirle al JBoss tener accesible ese driver al que se hace referencia. Para ello es necesario copiar el fichero mysql-connector-java-3.1.10-bin.jar de la distribución Connector/J al directorio /server/default/lib. Es decir, al directorio /lib de la configuración por defecto, suponiendo que esa sea la configuración que se está utilizando al ejecutar el servidor de aplicaciones JBoss.

Por otro lado, para llevar a cabo la configuración del servidor de aplicaciones JBoss y permitir que éste pueda interactuar con la base de datos MySQL recién creada y configurada, se recurrirá a la creación de un fichero XML. Este fichero especifico de configuración de fuente de datos, al igual que todos los demás ficheros de fuente de datos que gestiona JBoss, tendrá un nombre que terminará con el sufijo *-ds* (relativo a *datasource*). Para el caso particular de la aplicación del ejemplo, como se utilizará como sistema gestor de base de datos MySQL, el nombre del fichero de configuración será mysql-ds.xml.

Este fichero deberá sustituir al fichero hsqldb-ds.xml que se encuentra en el directorio /server/default/deploy y que permite la configuración de la fuente de datos por defecto usando la base de datos Hypersonic que incorpora directamente el servidor de aplicaciones JBoss y que corre como un servicio embebido dentro del propio servidor de aplicaciones. Por lo tanto, se deberá borrar el fichero hsqldb-ds.xml y en su lugar poner el fichero mysql-ds.xml cuyo código se verá a continuación. El hecho de sustituir un fichero por otro evita que se produzcan inconsistencia y conflictos entre los nombres JNDI de las dos fuentes de datos ya que, como se verá mas adelante, las conexiones a la base de datos que necesitan los bean de entidad se gestionarán como recursos que se referenciarán a traves de nombre lógicos utilizando para ello la API JNDI.

Por lo tanto, el fichero mysql-ds.xml que se debe guardar en el directorio /server/default/deploy del servidor de aplicaciones JBoss, para que este pueda descubrir la información relativa a la fuente de datos, debe ser algo similar al código que se presenta a continuación:

```
<datasources>

<local-tx-datasource>

        <jndi-name>DefaultDS</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/banco</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>lector</user-name>
        <password>password</password>

</local-tx-datasource>

</datasources>
```

Hay que aclarar que el contenido del fichero que se acaba de mostrar sería util si se ha decidido crear un usuario lector con *password* password. Si no se ha realizado esta tarea el sistema funcionaría correctamente utilizando en este fichero como usuario *root* y con la clave correspondiente que tuvo que establecerse durante la instalación de la base de datos MySQL.

Nota de seguimiento: el macabro sino de cualquier instalación software

Si alguien a estas alturas, después de instalar la base de datos MySQL correctamente, después de crear el fichero fuente de datos mysql-ds.xml y después de cambiarlo por el de la base de datos Hypersonic embebida en JBoss, cree que ha terminado, se equivoca. Si en este momento se reinicia el servidor de aplicaciones JBoss, se podrá apreciar como producen una serie de errores al arrancar el servidor.

Todos esos errores estan relacionados con el arranque del servicio JMS del servidor JBoss y con la utilización que éste hace de ciertas sentencias SQL (o subqueries) a través de su gestor de persistencia y que no son soportadas por la base de datos MySQL. Por lo tanto, es normal que se produzcan todas esas excepciones al arrancar el servidor de aplicaciones ya que hay un último detalle a tener en cuenta.

Este último detalle consiste en que para poder usar MySQL como fuente de datos hay que utilizar un fichero de servicio alternativo. Para ello hay que reemplazar el fichero de servicio de la base de datos Hypersonic llamado hsql-jdbc2-service.xml localizado en el directorio de despliegue /server/default/deploy/jms, por el fichero de servicio de MySQL mysql-jdbc2-service.xml que se puede encontrar en el directorio docs/examples/jms que cuelga del directorio home de JBoss.

Finalmente, una vez reemplazado el fichero en el directorio (es muy conveniente para evitar conflictos e inconsistencias de las fuentes de datos, que los fichero sean efectivamente

reemplazados y que no simplemente convivan en los directoros correspondientes) habría que reemplazar dentro del fichero mysql-jdbc2-service.xml cualquier ocurrecia de la palabra “MySQLDS” por la de “DefaultDS”.

7.4.2.3. Preparando los scripts de base de datos

Ya se ha comentado como configurar la base de datos MySQL que se va a utilizar en el ejemplo propuesto. Sin embargo, a parte de configurar la base de datos hay que ejecutar una serie de scripts que requiere el sistema. La aplicación necesitará crear en la base de datos Banco las tablas que soportarán la persistencia de los beans de entidad CuentaCorriente. Además, dichas tablas también se van poblar con unos datos iniciales que más adelante se verán afectados por actualizaciones realizadas por los clientes. Esas tablas, lógicamente, pueden verse ampliadas debido a la creación de otros registros o cuentas corrientes que realicen los clientes.

El modelo de datos que maneja la aplicación Gestor de Cuentas del ejemplo, es bastante sencillo y básicamente se compone de una única tabla que contendrá todas las cuentas corrientes que gestiona el sistema. Dicha tabla tendrá tres campos diferentes. El primero será el identificador de la cuenta corriente, que debe ser un String y considerado como la clave primaria ya que debe ser único. El segundo campo es el nombre del titular de la cuenta, tambien de tipo String, y en tercer lugar, el saldo de la misma de tipo numérico.

A continuación se muestra el contenido del script SQL de creación de la tabla explicada anteriormente. Este script se guardará en un fichero create-table.sql, dentro del directorio /sql del directorio de trabajo para luego ejecutarlo desde la herramienta Ant.

```
DROP TABLE cuenta;

CREATE TABLE cuenta(      id VARCHAR(12),
                          titular VARCHAR(24),
                          saldo FLOAT(10),
                          CONSTRAINT pk_cuenta PRIMARY KEY (id));
```

En segundo lugar se encuentra el script de inserción de datos en la base de datos, que tambien se guardará en un fichero insert.sql, dentro del directorio /sql del directorio de trabajo para luego ejecutarlo desde la herramienta Ant. A continuación se muestra dicho código:

```
DELETE FROM cuenta;

INSERT INTO cuenta VALUES
        ('123r78jni9', 'Hamlet', 135.61);

INSERT INTO cuenta VALUES
        ('12calm34', 'Alejo', 8500);

INSERT INTO cuenta VALUES
        ('1thief007', 'Gerente', 500);
```

7.4.2.4. Preparando el fichero jboss-build.xml de Ant

A continuación se muestra el fichero de configuración jboss-build.xml que se utilizará en el desarrollo de la aplicación de ejemplo Gestor de Cuentas. Este fichero ha sido construido utilizando como plantilla los ficheros jboss-build.xml de los ejemplos anteriores. Por lo tanto, además de las tareas tipicas vistas hasta el momento, se han introducido una serie de tareas relacionadas con los scripts de creación de tablas e inserción de datos en la base de datos.

Como siempre el fichero jboss-build.xml se guardará en el directorio raiz del directorio de trabajo y su código se muestra a continuación:

```
<project name="GestorCuentas" default="all" basedir=".">

    <property file="jboss-build.properties"/>

    <property name="lib.dir"   value="../../libs"/>
    <property name="src.dir"   value="${basedir}/src"/>
    <property name="build.dir" value="${basedir}/build"/>

    <!-- The classpath for running the client -->
    <path id="client.classpath">
        <fileset dir="${jboss.home}/client">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <!-- The build classpath -->
    <path id="build.classpath">
```

```
        <path refid="client.classpath"/>
         <fileset dir="${jboss.server}/lib/">
        </fileset>
    </path>

    <!-- Hypersonic SQL classpath -->
    <path id="hsql.classpath">
        <pathelement location="${jboss.server}/lib/hsqldb.jar"/>
    </path>

    <!-- =============================================================== -->
    <!—- Directorio de creacion                                          -->
    <!-- =============================================================== -->
    <target name="prepare">
        <mkdir dir="${build.dir}"/>
    </target>

    <!-- =============================================================== -->
    <!—- Compilacion del codigo fuente                                   -->
    <!-- =============================================================== -->
    <target name="compile" depends="prepare">
        <javac destdir="${build.dir}" classpathref="build.classpath"
                                                         debug="on">
            <src path="${src.dir}"/>
        </javac>
    </target>

    <target name="package-ejb" depends="compile">
        <mkdir dir="jar" />
        <delete file="jar/cuentas-ejb.jar"/>
        <jar jarfile="jar/cuentas-ejb.jar">
            <metainf dir="dd/ejb" includes="**/*.xml" />
            <fileset dir="${build.dir}">
                <include name="**/*.class"/>
                <exclude name="*.class"/>
            </fileset>
        </jar>
    </target>

    <target name="package-client" depends="compile">
        <mkdir dir="jar" />
         <delete file="jar/app-client.jar"/>
        <jar jarfile="jar/app-client.jar">
            <metainf dir="dd/client" includes="*.xml"/>
            <fileset dir="${build.dir}">
                <include name="**/*.class"/>
            </fileset>
             <fileset dir="dd/client">
                <include name="jndi.properties"/>
            </fileset>
         </jar>
    </target>

    <!--Crea un fichero ear que contiene los ejbjars y el webclient war. -->
    <target name="assemble-app">
        <delete file="jar/GestorCuentas.ear"/>
        <ear destfile="jar/GestorCuentas.ear" appxml="dd/application.xml">
            <fileset dir="jar" includes="*.jar"/>
        </ear>
    </target>

    <!--Despliega el fichero EAR copiandolo en el directorio deploy JBoss-->
    <target name="deploy" depends="assemble-app">
        <copy file="jar/GestorCuentas.ear" todir="${jboss.server}/deploy"/>
    </target>


    <!--Ejecuta el cliente standalone -->
```

```
    <target name="run-client">
        <java classname="ClienteCuentaCorriente" fork="yes">
            <classpath>
                <pathelement path="jar/app-client.jar"/>
                 <path refid="client.classpath"/>
            </classpath>
        </java>
    </target>

    <target name="clean">
       <delete dir="${build.dir}" />
       <mkdir dir="${build.dir}"  />
    </target>

    <!—Ejecución en serie de todos los pasos -->
    <target name="all" depends="compile,package-ejb,package-client,assemble-
                                                                  app,deploy" />

    <!-- Llamada al script de MySQL de creacion de tabla -->
    <target name="create-table">
     <sql classpathref="build.classpath" driver="com.mysql.jdbc.Driver"
          url="jdbc:mysql://localhost:3306/banco"
          userid="lector"
          password="password">
      <fileset dir="sql">
           <include name="create-table.sql"/>
      </fileset>
    </sql>
   </target>

   <!-- Llamada al script de MySQL de creacion de tabla -->
   <target name="insert-data">
     <sql classpathref="build.classpath" driver="com.mysql.jdbc.Driver"
          url="jdbc:mysql://localhost:3306/banco"
          userid="lector"
          password="password">
      <fileset dir="sql">
           <include name="insert.sql"/>
      </fileset>
     </sql>
   </target>

</project>
```

Como se puede apreciar en el código se han incluido dos tareas nuevas al final del fichero denominadas create-table e insert-data. Estas dos tareas se ejecutarán utilizando la tarea Ant SQL a través de la etiqueta <sql>. Esta tarea permite ejecutar sentencias SQL contra una base de datos via JDBC.

Para llevar a cabo cada tarea SQL se requieren los scripts correspondientes, create-table.sql e insert.sql. Estos dos scripts se pasan como argumentos a la herramienta de script Ant y ya han sido definidos anteriormente. Ademas para ejecutar estas dos tareas contra la base de datos es necesario, en primer lugar, determinar la *url* correspondiente al SGBDR que se va a utilizar. Por otro lado, es necesario conocer la ubicación del driver JDBC correspondiente, que previamente tuvo que haber sido

desplegado con el resto de paquetes .jar en el directorio /lib del servidor de aplicaciones JBoss. Este paso tuvo que haber sido realizado durante la instalación de la base de datos MySQL. Finalmente, ambas tareas SQL necesitan de un nombre de usuario y un password correspondientes a un usuario con privilegios suficientes como para realizar las operaciones que requiere la aplicación.

Aparte de estas dos tareas nuevas, jboss-build.xml define básicamente las mismas tareas que las de los ejemplos anteriores. La ejecución de cada una de estas tareas por la herramienta Ant se llevará a cabo en posteriores apartados de acuerdo a la fase de desarrollo en la que se encuentre la aplicación.

Nota de seguimiento: pros y contras de querer comentarlo todo

En general siempre es bueno dudar[7] y discutir, hablar y llegar a acuerdos, negociar, aclarar, discrepar, preguntar, debatir, y en el caso de la ingenieria del software, ante todo comentar. Siempre resulta una ventaja el comentar el código cuando se desarrollan aplicaciones. Pero cuidado, que una práctica ventajosa como esta puede convertirse en una desventaja si no se tienen en cuenta las limitaciones del entorno en el que se mueven esos comentarios.

Un buen comentario debidamente acentuado en el fichero jboss-build.xml puede generar una salida como esta:

```
Buildfile: jboss-build.xml

BUILD FAILED
Error reading project file C:\Documents and Settings\abarrera.MPDESARROLLO\Deskt
op\GestorCuentas\jboss-build.xml: Invalid byte 2 of 4-byte UTF-8 sequence.
```

Este podría ser el resultado de la aparición de un acento dentro de un comentario en un fichero XML que debe ser parseado por una herramienta que requiere un formato de tipos específico.

7.4.2.5. Preparando los ficheros

En este apartado se mostrará el código de todos los ficheros implicados en el desarrollo de la aplicación de Gestión de Cuentas. Como es habitual para cada uno de estos ficheros se hará una breve explicación del papel que desempeñan dentro del sistema, se explicarán las principales características del código relativas a la

[7] Dice un célebre autor que la duda es uno de los nombres de la inteligencia.

tecnología de EJB, así como su ubicación dentro de la estructura del directorio de trabajo.

7.4.2.5.1. Interfaces Remoto y Local

En primer lugar se mostrará el interfaz remoto perteneciente al bean de sesión sin estado GestorCuentas que se va a encargar de gestionar la interación del cliente de la aplicación (el propio banco) con el sistema.

El interfaz remoto se guardará en un fichero GestorCuentas.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz remoto para el Enterprise JavaBean: GestorCuentas
*/

package banca;

/**
 * Interfaz remota GestorCuentas.
 * Los clientes operan con esta interfaz
 */
public interface GestorCuentas extends javax.ejb.EJBObject {

    /**
     * Método que utiliza el cliente para crear y operar con una cuenta,
     * realizando distintas operaciones básicas sobre ella
     */
    public void operarCuenta(String titular) throws
                                                   java.rmi.RemoteException;

}
```

Como se puede apreciar en el código anterior, este interfaz únicamente proporciona un método operarCuenta( ) que le permite al cliente de la aplicación interactuar con la cuenta del titular que se pasa como parámetro.

Por su parte el bean de entidad CuentaCorriente requiere de un interfaz local. Este interfaz al igual que lo hacia el interfaz remoto, define cada uno de los métodos de la lógica de negocio que el bean finalmente va a publicar para que sean invocados por los beans que sean sus clientes. En este caso particular, el cliente del bean de entidad será el bean de sesión.

El interfaz local se guardará en un fichero CuentaCorrienteLocal.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz local para el Enterprise JavaBean: CuentaCorriente
*/

package banca;

public interface CuentaCorrienteLocal extends javax.ejb.EJBLocalObject
{
        /*
         * Método que decrementa el saldo en la cantidad pasada en argumento
        */
        public void reintegro(double cantidad);

        /*
         * Método que incrementa el saldo en la cantidad pasada en argumento
        */
        public void ingreso(double cantidad);

        /*
         * Método que devuelve el saldo de la cuenta
        */
        public double getBalance();

        /*
         * Método que devuelve el nombre del titular de la cuenta
        */
        public String getNombreTitular();

        /*
         * Método que asigna el nombre del titular de la cuenta
        */
        public void setNombreTitular(String nombreTitular);

        /*
         * Método que devuelve el identificador de la cuenta
        */
        public String getCodigoCuenta();

        /*
         * Método que asigna el identificador de la cuenta
        */
        public void setCodigoCuenta(String codigoCuenta);
}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

1. El interfaz local define el conjunto de métodos típico que se necesita para llevar a cabo la funcionalidad requerida. Es decir, un par de métodos de negocio que permiten ingresar y reintegrar dinero y los métodos típicos get( ) y set( ) sobre los atributos que gestionará el bean.

2. Hay que destacar que al utilizarse un interfaz local, el interfaz definido como CuentaCorrienteLocal extiende del interfaz javax.ejb.EJBLocalObject en lugar de hacerlo de interfaz javax.ejb.EJBObject como lo hacian los interfaces remotos. Además, como se puede apreciar en la signatura de todos los métodos, al estar definidos como métodos que serán llamados mediante llamadas locales, dichos métodos no lanzarán excepciones remotas como java.rmi.RemoteException.

7.4.2.5.2. Interfaces Home y Local Home

Al igual que en la definición de todos los interfaces de los ejemplos anteriores se necesitará en este caso la definición de los interfaces Home correspondientes. En primer lugar, el interfaz Home del bean de sesión se guardará en un fichero GestorCuentasHome.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz Home para el Enterprise JavaBean: GestionCuentas
 */

package banca;

public interface GestorCuentasHome extends javax.ejb.EJBHome {

   /*
    * Crea la instancia del bean
    */
    GestorCuentas create() throws java.rmi.RemoteException,
                                                    javax.ejb.CreateException;
}
```

El interfaz local Home de un bean de entidad BMP debe contener los métodos de creación, destrucción y búsqueda del objeto EJB asociado. La implementación de este interfaz también es responsabilidad del propio contenedor y se materializa en el objeto local Home.

El interfaz local Home del bean de entidad CuentaCorriente se guardará en un fichero CuentaCorrienteLocalHome.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz local Home para el Enterprise JavaBean: CuentaCorriente
 */

package banca;

import java.util.Collection;

public interface CuentaCorrienteLocalHome extends javax.ejb.EJBLocalHome
{

        /*
         * Método de creación de la instancia del bean que recibe como
         * argumento el código de cuenta y el nombre del titular
         */
        public CuentaCorrienteLocal create(String codigoCuenta, String
                         titular) throws javax.ejb.CreateException;


        /*
         * Método de búsqueda de una instancia a partir de su primary key.
         * Este método devuelve un objeto CuentaCorrienteLocal
         */
        public CuentaCorrienteLocal findByPrimaryKey(CuentaCorrientePK
                                    clave) throws javax.ejb.FinderException;


        /*
         * Método de búsqueda de una instancia a partir del nombre del
         * titular. Este método devuelve un objeto Collection ya que puede
         * darse el caso de que un titular tenga varias cuentas.
         */
        public Collection findByNombreTitular(String titular) throws
                                                   javax.ejb.FinderException;


        /*
         * Método que permite conocer la cantidad de dinero total que hay en
         * las cuentas
         */
        public double getTotalEnCuentas();

}
```

Las cuestiones más importantes a destacar del código de este interfaz se enumeran a continuación:

1. En primer lugar hay que decir que el interfaz local Home define un método de creación que recibe como parámetros el código de la cuenta corriente y el nombre del titular de la misma. Este método creará en la base de datos una

representación de una cuenta corriente en forma de registro o tupla dentro de la tabla correspondiente.

2. Además, el interfaz define un par de métodos de búsqueda que lanzan la excepción javax.ejb.FinderException. El primero de esos métodos busca en la base de datos para ver si ya existe un registro para esa cuenta a través de su *primery key*, que no es otra que el identificador de la cuenta. Por su parte, el segundo método es un método de lógica de negocio que busca en la base de datos y puede devolver una colección de cuentas ya que busca por el nombre del titular de la cuenta. Debido a que se están utilizando beans BMP se deberán implementar estos métodos en la clase del bean. Si por el contrario, se estuviesen utilizando beans CMP, estos métodos deberían, estar también declarados en el interfaz local Home, pero ahora quien los implementaría sería el propio contendor EJB.

3. Finalmente, el interfaz define un métodos de lógica de negocio llamado getTotalEnCuentas( ). Este método se define en el interfaz local Home ya que es un método que no afecta a un único registro de la base de datos sino a toda la tabla en general. Por lo tanto, se considera un método global e independiente de una instancia o registro particular del bean de entidad. Este método deberá ser implementado en la clase del bean como el método ejbHomeGetTotalEnCuentas( ).

4. También es importante destacar, que en el caso de los bean de entidad existe otra forma de interactuar con ellos sin necesidad de recurrir al objeto Home. Esta forma es modificando directamente la base de datos en las que están almacenados los datos del bean. Esto quiere decir que, si existe un bean en memoria que representa los datos de un registro de la base de datos y ese registro es eliminado accediendo directamente a la base de datos, por ejemplo, mediante una aplicación que lanza sentencias SQL de forma directa a la base de datos, ese bean desaparece de memoria.

7.4.2.5.3. La clase primary key

Tal y como se adelantó, cuando se estudiaron las generalidades de los beans de entidad, la especificación EJB impone la definición de una clase denominada *primary key* que se detalla a continuación. Esta clase será la encargada de distinguir las distintas instancias de los beans de forma única y será la clase que, por lo tanto, facilitará realizar la búsqueda y recuperación de toda la información de una determinada cuenta corriente que previamente ha tenido que haberse hecho persistente.

Esta clase clave o *primary key* debe ser serializable incluso en el caso de que sea una clase que gestione una clave compuesta o en el caso de que dicha clave no sea de un tipo primitivo. Además, tal y como se podrá comprobar a continuación, dicha clase debe sobrescribir una serie de métodos como equals( ), toString( ) y hashCode( ) ya que serán estos métodos los que se utilicen para realizar las búsquedas de las instancias de los beans de entidad.

La clase *primary key* se guardará en un fichero denominado CuentaCorrientePK.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Primary key para el Enterprise JavaBean: CuentaCorriente
 */

package banca;

public class CuentaCorrientePK implements java.io.Serializable
{
	/*
	 * Atributo identificador y unico para la clase Primary key
	 */
	private String codigoCuenta;


	/*
	 * Constructor de la clase Primary key
	 */
	public CuentaCorrientePK(String codigoCuenta)
	{
		this.codigoCuenta = codigoCuenta;
	}

	/*
	 * Constructor sin argumentos de la clase Primary key
	 */
	public CuentaCorrientePK()
	{
```

```
        }

        /*
         * Método sobrescrito de la clase Object
         */
        public String toString()
        {
                return codigoCuenta;
        }

        /*
         * Método sobrescrito de la clase Object
         */
        public int hashCode()
        {
                return codigoCuenta.hashCode();
        }

        /*
         * Método sobrescrito de la clase Object
         */
        public boolean equals(Object acc)
        {
                return ((acc instanceof CuentaCorrientePK) &&
                        ((CuentaCorrientePK)acc).getCodigoCuenta().
                          equals(this.getCodigoCuenta())
                       );
        }

}
```

Las cuestiones mas importantes a destacar del código de esta clase se enumeran a continuación:

1. En este caso la clase *primary key* contiene simplemente un String que representa el identificador de la cuenta. Este identificador debe ser único para cada cuenta y el trabajo de creación de ese código único se deja en manos del cliente de la aplicación que debe proporcionar efectivamente un código único para cada cuenta abierta en el sistema. Para el bean de entidad que se está desarrollando basta con este identificador ya que la gestión de las cuentas se realizará utilizando una única tabla. Sin embargo, puede darse el caso de que desarrollen beans que necesiten que su clase *primary key* requiera de mas atributos. Este caso podría darse, si por ejemplo, el bean de entidad representa conceptos de base de datos que son persistentes en más de una tabla. En este caso, sería probable que cada uno de los distintos atributos de la *primary key* del bean representaran a su vez distintas claves primarias de las distintas tablas que representan el concepto.

2. Es necesario sobrescribir el método toString( ). El contenedor EJB llamará a este método para devolver el valor de la clave primaria asociada al bean. Para este caso se devuelve simplemente el valor del atributo codigoCuenta que es de tipo String. Para claves más complejas se deberá devolver una combinación de ellas también como String.

3. El método hashCode( ) también se sobrescribe. Sobrescribiendo este método se consigue que la clase *primary key* pueda ser almacenada en una tabla Hash. Con esta particularidad se consigue que el contenedor EJB, que utiliza este tipo de estructuras para su gestión de recursos interna, pueda almacenar el bean que se está desarrollando, en su lista, por ejemplo, de beans cargados en memoria.

4. El método equals( ) de la clase java.lang.Object también se sobrescribe en la clase *primary key*. El contenedor llamará a este método para comparar la *primary key* del bean con otras *primary keys* y determinar así si dos objetos cacheados en memoria representan a los mismos datos almacenados en la base de datos.

5. Finalmente, el método getCodigoCuenta( ) devuelve el atributo identificador de la cuenta corriente.

7.4.2.5.4. Las clases de los beans de sesión y entidad

En primer lugar, se verá la clase del bean de sesión que se encarga de controlar toda la lógica de gestión de las cuentas corrientes que maneja la aplicación del ejemplo. Esta clase del bean GestorCuentas se guardará en un fichero denominado GestorCuentasBean.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Implementacion de la clase del Enterprise JavaBean: GestorCuentas
*/

package banca;

import java.util.*;
```

```
import javax.naming.*;
import banca.CuentaCorrientePK;
import banca.CuentaCorrienteLocalHome;
import banca.CuentaCorrienteLocal;

/*
 * GestorCuentas es un bean de sesion sin estado responsable de la
 * creacion y manejo de las cuentas de los clientes
 */
public class GestorCuentasBean implements javax.ejb.SessionBean {

        private javax.ejb.SessionContext ctx;

        /*
        * Utiliza el EJB de entidad CuentaCorriente
        */
        public void operarCuenta(String titular)throws
                                                java.rmi.RemoteException{

          try{
                /*
                 * Busqueda del objeto Home en JNDI
                 */

                // Se accede al contexto por defecto
                Context ctx = new InitialContext();
                // Se busca el objeto publicado con JNDI
                Object obj = ctx.lookup
                                 ("java:comp/env/ejb/RefCuentaCorriente");
                // Se hace un cast normal ya que son interfaces locales
                CuentaCorrienteLocalHome home = (CuentaCorrienteLocalHome)
                                                                obj;

                // Se consulta el total acumulado en todas las cuentas
                System.out.println("Total acumulado en cuentas: "
                                        +home.getTotalEnCuentas());


                /*
                 * Creacion de nueva cuenta usuario
                 */

                CuentaCorrienteLocal cuentaPrueba = null;
                // Se crea una nueva cuenta
                home.create("2005a31d1973",titular);
                // Se consulta la cuenta buscandola por el nombre del
                // titular
                Iterator it = home.findByNombreTitular(titular).iterator();
                if (it.hasNext()) {
                        cuentaPrueba = (CuentaCorrienteLocal)it.next();
                } else {
                        System.out.println("No existe la cuenta de "+
                                                                titular);
                }

                /*
                 * Operaciones basicas con la nueva cuenta
                 */

                // Se pide el saldo de esa cuenta
                System.out.println("Saldo actual: "+
                                        cuentaPrueba.getBalance());
                // Se ingresa de esa cuenta
                cuentaPrueba.ingreso(200);
                System.out.println("Saldo actual: "+
                                        cuentaPrueba.getBalance());
                // Se reintegra de esa cuenta
                cuentaPrueba.reintegro(150);
```

```
            System.out.println("Saldo actual: "+
                                 cuentaPrueba.getBalance());
            // Se pide el acumulado de las cuentas con lo ultimos
            // movimientos
            System.out.println("Total acumulado en cuentas: "+
                                 home.getTotalEnCuentas());

            /*
             * Recuperacion del objeto a partir de su PK
             */

            // Se pide la PK para realizar una busqueda con ella
            CuentaCorrientePK pk = (CuentaCorrientePK)
                                 cuentaPrueba.getPrimaryKey();
            // Se destruye el objeto para crearlo de nuevo con la PK
            cuentaPrueba = null;
            // Se realiza la busqueda y se devuelve el saldo
            cuentaPrueba = home.findByPrimaryKey(pk);
            System.out.println ("Encontrada con ID "+pk+" cuenta con
                          saldo " + cuentaPrueba.getBalance());

            /*
             * Se elimina el cliente de prueba para dejar el sistema
             * igual
             */

            // Finalmente se destruye la entidad permanentemente de la
            // BBDD
            if (cuentaPrueba != null){
                cuentaPrueba.remove();
            }

            /*
             * Se produce el desfalco
             */
            // Se consulta la cuenta buscandola por el nombre del
            // titular
            CuentaCorrienteLocal cuentaGerente = null;
            it = home.findByNombreTitular("Gerente").iterator();
            if (it.hasNext()) {
                    cuentaGerente = (CuentaCorrienteLocal)it.next();
                    cuentaGerente.ingreso(1000);
            } else {
                    System.out.println("Imposible que un banco donde "+
                    "realizan este tipo de practicas fraudulentas "+
                    "no tenga creada aun la cuenta del Gerente");
            }

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



//---------------------------------------------------
// Metodos requeridos e invocados por el contenedor,
// nunca desde codigo cliente.
//---------------------------------------------------

public void ejbCreate() throws javax.ejb.CreateException {
            System.out.println("GestorUsuariosBean.ejbCreate()
                                                  invocado.");
     }

     public void ejbRemove() {
            System.out.println("GestorUsuariosBean.ejbRemove()
                                                  invocado.");
```

```
        }

        public void ejbActivate() {
                System.out.println("GestorUsuariosBean.ejbActivate()
                                                        invocado.");
        }

        public void ejbPassivate() {
                System.out.println("GestorUsuariosBean.ejbPassivate()
                                                        invocado.");
        }

        public void setSessionContext(javax.ejb.SessionContext ctx) {
                System.out.println("GestorUsuariosBean.setSessionContext()
                                                        invocado.");
                this.ctx = ctx;
        }

}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

1. En primer lugar, hay que fijarse que el bean de sesión GestorCuentas llama al bean de entidad CuentaCorriente. Lo importante a destacar aquí es que cuando se utilizan distintos beans que colaboran entre si y se da la situación de que un bean (en este caso el bean de sesión) llama a otro bean (el de entidad que utiliza los interfaces locales) no es necesario suministrar parámetros de inicializacion JNDI en el bean llamado. De hecho, el fichero jndi.properties con el que siempre se han pasado los parámetros de inicialización JNDI solo tiene sentido en la parte cliente. Es alli, en el código del cliente (seguramente remoto) donde se necesita conocer el proveedor y el protocolo JNDI para encontrar el bean que brinda la funcionalidad. A partir de ese momento, los beans llamados en una aplicación EJB, pueden recurrir al contexto inicial por defecto JNDI a través de la llamada al objeto InitialContext sin argumentos. Será responsabilidad del contenedor EJB, donde residen los beans llamados, el configurar siempre ese contexto inicial por defecto. Además, cuando colaboran distintos beans es inviable codificar dentro de un bean la información JDNI, de hecho, no se debe conocer nunca el proveedor de servicios JNDI de un bean, para que éste sea relmente independiente del contenedor.

2. También es importante destacar que debido a que se han utilizado interfaces locales para implementar el bean de entidad CuentaCorriente, a la hora de ejecutar el metodo lookup( ) sobre el contexto inicial, el objeto que devuelve dicho método no precisa ser modificado mediante el cast que realiza el metodo narrow( ) de la clase javax.rmi.PortableRemoteObject . Esto es debido a que lo que devuelve lookup( ), en este caso, no es un objeto remoto sino un objeto implementado a partir de interfaces locales. Por lo tanto, se puede hacer el proceso de cast normal que proporciona Java que lo convierte en el tipo adecuado. En definitiva, la búsqueda de un objeto Home y la creación del bean asociado con interfaces locales se puede resumir en la siguiente secuencia de pasos:

```
// Se crea el contexto inicial
Context ctx = new InitialContext();
// Se busca el interfaz Home
Object result = ctx.lookup("java:comp/env/ejb/XXXXLocalHome");
// Se convierte el resultado al tipo apropiado sin necesidad de hacer
un cast al utilizar interfaces locales
XXXXLocalHome home = (XXXXLocalHome) result;
// Se crea el bean
XXXXLocal c = home.create(...);
```

3. Las operaciones que proporciona el bean de sesión en su método operarCuenta( ) básicamente ya estan explicadas en el código de la clase que se acaba de presentar. Sin embargo, si que hay que mencionar, a modo de resumen, que el objetivo último del metodo operarCuenta( ) es el de verificar que el sistema funciona correctamente y dejarlo en el mismo estado en el que se encontrada. Es decir, se crean cuentas, se realizan una serie de operaciones sobre esas cuentas, se comprueba que todo se ejecuta correctamente viendo el log del servidor, y finalmente, se debe volver al estado inicial borrando siempre todas las instancias de los beans creados.

4. Otro detalle importante es que para ejecutar el método que devuelve el saldo de todas las cuentas, en este caso, el banco ha accedido en primer lugar al objeto Home del bean CuentaCorriente que es quien exporta ese método getTotalEnCuentas( ) sin necesidad de crear aun una instancia de

CuentaCorriente. Posteriormente, si que se ha hecho una llamada el método create( ) sobre ese objeto Home, para crear una nueva cuenta con los datos suministrados en el código de la clase y poder trabajar ya sobre una instancia en particular.

5. Sin embargo, a la hora de trabajar con beans de entidad, la creación de sus instancias no debería hacerse de forma descontrolada como práctica habitual. Por el contrario, resulta mucho más eficiente seguir cierta lógica a la hora de crear un nuevo bean de entidad. Por ejemplo, para utilizar un bean de entidad primero se debería intentar encontar la instancia de ese bean llamando al método findByPrimaryKey( ). Si no se encuentra esa instancia, si que se llamaría al método create( ) del interfaz Home del bean que realmente crearía una nueva instancia. Por lo tanto, el contenedor EJB intercepta el método findByPrimaryKey( ), que intenta recuperar un registro de la base de datos con la clave primaria que en el caso del ejemplo es el identificador de la cuenta corriente. Si lo encuentra lo aprovecha y si no lo encuentra se crea el bean llamando al método create( ) del objeto Home.

6. Cuando un cliente realiza un llamada a un metodo create( ) o a un método *finder* hay que aclarar que el contenedor EJB se comporta de distinta manera. En primer lugar, cuando se produce una llamada a create( ), el contenedor EJB busca una nueva instancia dentro del *pool* de instancias disponibles y la extrae de ese *pool* para llamar el metodo ejbCreate( ). Finalmente, el contenedor asocia la instancia con un objeto EJB y devuelve ese objeto EJB al cliente. Por su parte, cuando un cliente invoca un metodo *finder* se realiza la misma busqueda en el *pool* pero en este caso no se extrae la instancia del *pool* sino que por el contrario esa instancia del bean, aun dentro del *pool*, se puede utilizar para dar servicio a otros metodos *finder*.

7. Finalmente hay que recordar, que el método create( ) añadirá siempre un nuevo registro a la base de datos que utilizará como clave primaria el identificador de la cuenta corriente. Del mismo modo la invocación del método delete( ) producirá el borrado del registro correspondiente en la base de datos.

Todos los puntos anteriores hacen relación a la clase del bean de sesión. Pero aún queda hablar de la clase del bean de entidad que también se necesita desarrollar para la aplicación propuesta. La implementación de cualquier bean de entidad con persistencia gestionada por el propio bean es bastante extensa, ya que, entre otras cosas, la clase del bean aporta todo el código JDBC necesario para gestionar la persistencia del bean.

Por este motivo, el código que se presenta a continuación, se debe dividir conceptualmente en tres partes distintas que permitirán al lector tener una idea más clara y estructurada de todo el código que se va a encontrar a partir de este momento.

1. Atributos de estado gestionados por el bean. En primer lugar, la clase del bean debe proporcionar los atributos que deben ser persistentes en la base de datos. La instancia del bean cargará y almacenará la información de la base de datos en dichos atributos. De esto se deduce que existirá una estrecha relación entre los atributos definidos en la clase del bean y los campos definidos en las tablas de la base de datos.

2. Métodos propios de la lógica de negocio de la aplicación. Estos son los métodos que proporcionan la funcionalidad propia de la aplicación y que brindan servicio a los clientes. En el caso de la aplicación propuesta se trata de operaciones como el ingreso o el reintegro de dinero en una cuenta corriente. Es decir, en la clase del bean estarán, por un lado, todos aquellos métodos definidos, y por lo tanto publicados, en el interfaz CuentaCorrienteLocal, así como el constructor por defecto del bean. Y por otro lado estarán todos aquellos métodos pertenecientes a la lógica de negocio aunque estén definidos en el interfaz Home, como puede ser el caso del método que devuelve el saldo general de las cuentas.

3. Métodos requeridos por la especificación EJB. Finalmente, en la clase del bean se encuentran los métodos que el contenedor EJB llama para realizar la gestión del ciclo de vida del bean como son los métodos ejbLoad( ) o ejbStore( ). Además de estos métodos, la clase del bean implementa todos los métodos

definidos en el interfaz CuentaCorrienteLocalHome. Dentro de estos métodos se encuentran el constructor definido en ese interfaz, así como los métodos *finder* también definidos allí.

La clase del bean se guardará en un fichero denominado CuentaCorrienteBean.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Clase del bean para el Enterprise JavaBean: CuentaCorriente
*/

package banca;

import javax.naming.*;
import java.util.Collection;

public class CuentaCorrienteBean implements javax.ejb.EntityBean
{

        /***********************************
         * Atributos
        ************************************/

        /*
         * Atributo del bean no persistente
        */
        protected javax.ejb.EntityContext ctx;
        /*
         * Atributo del bean que representa el codigo de la cuenta
         * corriente
        */
        private String codigoCuenta;
        /*
         * Atributo del bean que representa el nombre del titular de
         * la cuenta corriente
        */
        private String nombreTitular;
        /*
         * Atributo del bean que representa el saldo de la cuenta
         * corriente
        */
        private double saldo;


        /***********************************
         * Metodos de logica de negocio
        ************************************/

        /*
         * Constructor por defecto del bean
        */
        public CuentaCorrienteBean()
        {
                System.out.println("Nuevo EJB Entidad CuentaCorrienteBean
construido por contenedor");
        }

        /*
         * Método que devuelve el saldo
        */
```

```
        public double getBalance()
        {
                System.out.println("Invocado getBalance");
                return saldo;
        }

        /*
         * Método que devuelve el nombre del titular
        */
        public String getNombreTitular()
        {
                System.out.println("Invocado getNombreTitular");
                return nombreTitular;
        }

        /*
         * Método que modifica el nombre del titular
        */
        public void setNombreTitular(String nombreTitular)
        {
                System.out.println("Invocado setNombreTitular");
                this.nombreTitular = nombreTitular;
        }

        /*
         * Método que devuelve el codigo de la cuenta
        */
        public String getCodigoCuenta()
        {
                System.out.println("Invocado getCodigoCuenta");
                return codigoCuenta;
        }

        /*
         * Método que modifica el codigo de la cuenta
        */
        public void setCodigoCuenta(String codigoCuenta)
        {
                System.out.println("Invocado setCodigoCuenta");
                this.codigoCuenta = codigoCuenta;
        }

        /*
         * Método que modifica el contexto del bean de entidad
        */
        public void setEntityContext (javax.ejb.EntityContext ctx)
        {
                System.out.println("EntityContext establecido");
                this.ctx=ctx;
        }

        /*
         * Método que modifica el contexto del bean de entidad
        */
        public void unsetEntityContext ()
        {
                System.out.println("EntityContext eliminado");
                this.ctx=null;
        }

        /*
         * Método de reintegro de la cuenta corriente
        */
        public void reintegro(double cantidad)
        {
                System.out.println("Reintegro de "+cantidad+" euros
solicitado");
                if(saldo > cantidad)
```

```
                {
                        saldo -= cantidad;
                        System.out.println("Reintegro realizado con
exito");
                }
        }

        /*
         * Método de ingreso
        */
        public void ingreso(double cantidad)
        {
                // Uhmmm!!! que raro...
                if (!(nombreTitular.toLowerCase()).equals("gerente"))
                        System.out.println("Ingreso de "+cantidad+" euros
solicitado");
                saldo += cantidad;
        }


        /*
         * Método de negocio general que devuelve el saldo de todas las
cuentas
         * declarado en el interfaz Home
        */
        public double ejbHomeGetTotalEnCuentas()
        {
                double saldoTotal = 0.0;
                java.sql.PreparedStatement consulta = null;
                java.sql.Connection conexion = null;

                System.out.println("Invocado getTotalEnCuentas");
                try {
                        conexion = getConnection();
                        consulta = conexion.prepareStatement("select
sum(saldo) as total "+
                                                                    "from
cuenta");

                        java.sql.ResultSet resultado =
consulta.executeQuery();
                        if (resultado.next()){
                                saldoTotal = resultado.getDouble("total");
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                } finally {
                        try { if (consulta != null) consulta.close(); }
catch (Exception e) {}
                        try { if (conexion != null) conexion.close(); }
catch (Exception e) {}
                       return saldoTotal;
                }
        }


        /*
         * Método privado que devuelve una conexion a la base de datos
        */
        private java.sql.Connection getConnection() throws Exception
        {
                try {
                        Context ctx = new InitialContext();
                        javax.sql.DataSource ds =
(javax.sql.DataSource)ctx.lookup

        ("java:/comp/env/jdbc/DefaultDS");
                        return ds.getConnection();
```

```
		} catch (Exception e)
		{
			System.err.println("No se pudo obtener la fuente de
datos");
			e.printStackTrace();
			throw e;
		}
	}



	/*****************************************************
	 * Metodos de ciclo de vida gestionados por el contenedor
	*****************************************************/


	/*
	 * Método llamado por el contenedor de creacion una instancia del
bean
	*/
	public CuentaCorrientePK ejbCreate(String codigoCuenta, String
nombreTitular)
							throws
javax.ejb.CreateException{

		System.out.println("ejbCreate");

		java.sql.PreparedStatement consulta = null;
		java.sql.Connection conexion = null;
		try {
			conexion = getConnection();
			this.codigoCuenta = codigoCuenta;
			this.nombreTitular = nombreTitular;
			this.saldo=0;
			consulta = conexion.prepareStatement("insert into
cuenta "+
								"(id, titular,
saldo) "+
							"values (?, ?, ?)");
			consulta.setString(1,codigoCuenta);
			consulta.setString(2,nombreTitular);
			consulta.setDouble(3,saldo);
			consulta.executeUpdate();

		} catch (Exception e) {
			throw new javax.ejb.CreateException(e.toString());
		} finally {
			try { if (consulta != null) consulta.close(); }
catch (Exception e) {}
			try { if (conexion != null) conexion.close(); }
catch (Exception e) {}

			return new CuentaCorrientePK(this.codigoCuenta);
		}

	}


	/*
	* Método llamado por el contenedor de eliminacion de una instancia
del bean
	*/
	public void ejbRemove()
	{
		System.out.println("ejbRemove");

		CuentaCorrientePK clave = (CuentaCorrientePK)
ctx.getPrimaryKey();
```

```
                String codigo = clave.getCodigoCuenta();

                java.sql.PreparedStatement consulta = null;
                java.sql.Connection conexion = null;
                try {
                        conexion = getConnection();
                        consulta = conexion.prepareStatement("delete from
cuenta "+
                                                                    "where id =
?");
                        consulta.setString(1,codigo);
                        if (consulta.executeUpdate() == 0)
                        {
                                System.err.println("No se pudo borrar la
cuenta de la BBDD");
                        }
                } catch (Exception e) {
                        throw new javax.ejb.EJBException(e.toString());
                } finally {
                        try { if (consulta != null) consulta.close(); }
catch (Exception e) {}
                        try { if (conexion != null) conexion.close(); }
catch (Exception e) {}
                }
        }


        /*
         * Método de carga de la informacion de una instancia guardada en la
         * base de datos
        */
        public void ejbLoad()
        {
                System.out.println("ejbLoad");

                CuentaCorrientePK clave = (CuentaCorrientePK)
ctx.getPrimaryKey();
                String codigo = clave.getCodigoCuenta();

                java.sql.PreparedStatement consulta = null;
                java.sql.Connection conexion = null;
                try {
                        conexion = getConnection();
                        consulta = conexion.prepareStatement("select
titular, saldo, id "+
                                                                    "from cuenta
where id = ?");
                        consulta.setString(1,codigo);
                        java.sql.ResultSet resultado =
consulta.executeQuery();
                        if (resultado.next())
                        {
                                this.nombreTitular =
resultado.getString("titular");
                                this.saldo = resultado.getDouble("saldo");
                                this.codigoCuenta =
resultado.getString("id");
                        }
                } catch (Exception e) {
                        throw new javax.ejb.EJBException(e.toString());
                } finally {
                        try { if (consulta != null) consulta.close(); }
catch (Exception e) {}
                        try { if (conexion != null) conexion.close(); }
catch (Exception e) {}
                }

        }
```

```
        /*
         * Método de almacenamiento y actualizacion de los datos de un bean
en la base de
         * datos.
        */
        public void ejbStore()
        {
                System.out.println("ejbStore");

                java.sql.PreparedStatement consulta = null;
                java.sql.Connection conexion = null;
                try {
                        conexion = getConnection();
                        consulta = conexion.prepareStatement("update cuenta
set "+
                                                                "titular =
?, saldo = ? "+
                                                             "where id =
?");
                        consulta.setString(1,nombreTitular);
                        consulta.setDouble(2,saldo);
                        consulta.setString(3,codigoCuenta);
                        consulta.executeUpdate();

                } catch (Exception e) {
                        throw new javax.ejb.EJBException(e.toString());
                } finally {
                        try { if (consulta != null) consulta.close(); }
catch (Exception e) {}
                        try { if (conexion != null) conexion.close(); }
catch (Exception e) {}
                }
        }


        /*
         * Método de busqueda de una instancia a partir de su PK
        */
        public CuentaCorrientePK ejbFindByPrimaryKey(CuentaCorrientePK
clave)
                                                           throws
javax.ejb.FinderException
        {
                System.out.println("ejbFindByPrimaryKey");

                java.sql.PreparedStatement consulta = null;
                java.sql.Connection conexion = null;
                try {
                        conexion = getConnection();
                        consulta = conexion.prepareStatement("select id
from cuenta "+
                                                             "where id =
?");
                        consulta.setString(1,clave.toString());
                        java.sql.ResultSet resultado =
consulta.executeQuery();

                        // Si no se ha encontrado la clave y no ha habido
errores
                        if (!resultado.next()){
                                throw new javax.ejb.FinderException("No se
encuentra la cuenta");
                        }
                } catch (Exception e) {
                        throw new javax.ejb.FinderException(e.toString());
                }  finally {
```

```
                        try { if (consulta != null) consulta.close(); }
catch (Exception e) {}
                        try { if (conexion != null) conexion.close(); }
catch (Exception e) {}

                        // Devuelvo la clave y si ha habido un error se
habra lanzado excepcion
                        return clave;
                }

        }


        /*
         * Método de busqueda de una instancia a partir del nombre del
titular
        */
        public Collection ejbFindByNombreTitular(String nombre) throws
javax.ejb.FinderException
        {
                System.out.println("ejbFindByNombreTitular");

                java.sql.PreparedStatement consulta = null;
                java.sql.Connection conexion = null;
                java.util.Vector coleccion = new java.util.Vector();
                try {
                        conexion = getConnection();
                        consulta = conexion.prepareStatement("select id
from cuenta "+
                                                                  "where titular
= ?");
                        consulta.setString(1,nombre);
                        java.sql.ResultSet resultado =
consulta.executeQuery();
                        while (resultado.next())
                        {
                                String codigo = resultado.getString("id");
                                coleccion.addElement(new
CuentaCorrientePK(codigo));
                        }

                } catch (Exception e) {
                        throw new javax.ejb.FinderException(e.toString());
                } finally {
                        try { if (consulta != null) consulta.close(); }
catch (Exception e) {}
                        try { if (conexion != null) conexion.close(); }
catch (Exception e) {}
                        // Devuelvo la coleccion y si ha habido un error se
habra lanzado excepcion
                        return coleccion;
                }

        }


        /*
         * Métodos de gestion de ciclo de vida que no necesitan ser
         * implementados y que unicamente dejan su huella en el log cuando
         * son invocados
        */
        public void ejbPostCreate(String codigoCuenta, String nombreTitular)
                {System.out.println("ejbPostCreate");}
        public void ejbActivate()
                {System.out.println("ejbActivate");}
        public void ejbPassivate()
                {System.out.println("ejbPassivate");}
```

```
}
```

Las cuestiones mas importantes a destacar del código de esta clase se enumeran a continuación:

1. Los beans de entidad implementan el interfaz javax.ejb.EntityBean que define un conjunto de métodos de notificación o métodos *call-back* que el bean usa para interactuar con el contenedor EJB.

2. La clase del bean define además una serie de atributos que serán persistentes en la base de datos. Esta clase también define, al igual que sucedía con los beans de sesión, un atributo que representa el contexto. En este caso será un atributo de tipo EntityContext que al igual que en los beans de sesión no será un campo persistente.

3. Los métodos de la lógica de negocio tienden a modificar lo valores de los atributos en memoria a través de la instancia del bean que está en ese momento cargada en memoria. Por lo tanto, esos métodos no suelen requerir código JDBC a pesar de que modifiquen atributos que finalmente serán hechos persistentes en la base de datos.

4. El método que presta la funcionalidad de reintegro de dinero de la cuenta corriente, por simplicidad, solo lleva a cabo su tarea si el saldo es mayor que la cantidad solicitada por el cliente.

5. El método ejbHomeGetTotalEnCuentas( ) es un método de lógica de negocio que tiene código JDBC ya que es un método que no es específico de una instancia de un bean particular, sino que afecta a todos las cuentas de la base de datos. Para dar esta funcionalidad el bean ejecuta una sentencia SQL a través de una conexión. Dicha sentencia SQL es de tipo PreparedStatement. Este tipo de sentencias SQL aportan una serie de ventajas frente a las sentencias SQL normales como puede ser el hecho de recibir parámetros. Sin embargo, normalmente estas sentencias siguen resultando

computacionalmente bastante más pesadas que las sentencias CallableStatement. Estas últimas directamente residen en el sistema gestor de base de datos y que no deben ser constantemente compiladas cuando se envían a la base de datos.

6. A pesar de los problemas de rendimiento planteados, JDBC, a través del *driver* cargado en el contenedor EJB, aporta sin embargo cierta lógica que permite que la ejecución de estas sentencias SQL se vea mejorada. Antes de que una sentencia de este tipo sea enviada a la base de datos para ser compilada y ejecutada allí, el contenedor EJB la parsea para determina la mejor forma de ejecutarla basándose para ello en una serie de estadísticas que el propio contenedor EJB mantiene. Este proceso es lento pero se puede ver mejorado cuando el contenedor descubre que una instancia de una PreparedStatement, igual a la que ahora se quiere ejecutar, ha sido ejecutada sobre una conexión previamente. En este caso el contenedor reutilizará la versión de la sentencia SQL preparada anteriormente sobre la conexión con lo que se mejora el rendimiento.

7. El método ejbHomeGetTotalEnCuentas( ) también hace uso de un método privado llamado getConnection( ). Este método devuelve una conexión buscándola vía JNDI. Para poder llevar a cabo esta búsqueda es necesario que las conexiones estén publicadas mediante JNDI como recursos que va a gestionar el propio contenedor EJB. Esta publicación se lleva a cabo definiendo en el fichero ejb-jar.xml las conexiones a la base de datos como recursos gestionables por el contenedor EJB. El mecanismo concreto de publicación de conexiones mediante JNDI se verá posteriormente cuando se vean los ficheros descriptores de despliegue, tanto los estándar como los propios del contenedor EJB.

8. Por otro lado, hay que destacar que en cada uno de los métodos que hacen uso getConnection( ) al final se cierra la conexión. Este detalle es muy importante ya que de esta forma se facilita que el contenedor EJB pueda gestionar unos recursos, como son las conexiones a la base de datos, mediante un *pool*. De esta forma, cada vez que se necesita una conexión se pide una al contenedor

EJB a través del InitialContext. Si por el contrario no se cerrasen las conexiones, después de su utilización en cada método, sería imposible que un bean aprovechara una conexión creada por otro cuando ese otro no la estuviera utilizando.

9. Antes se ha comentado que la mayoría de los métodos de la propia lógica de negocio de la aplicación no contienen código JDBC ya que atacan directamente a los atributos del bean en memoria. Sin embargo, los métodos que si que tienen código JDBC son todos los métodos *call-back* que llama el contenedor EJB durante la gestión del ciclo de vida de bean como pueden ser ejbCreate( ), ejbLoad( ), ejbStore( ), ejbRemove( ) o las distintas búsquedas de los datos asociados a una instancia particular.

10. Los métodos ejbRemove( ) y ejbLoad( ) requieren en su ejecución del uso del método getPrimaryKey( ) perteneciente al objeto contexto y que devuelve la clave primaria de la instancia que se quiere borrar o recuperar. En el primero de los casos, ejbRemove( ) requiere la clave primaria para saber cual es el registro que hay que borrar de la base de datos, mientras que con ejbLoad( ) se necesita para saber de que registro de la base de datos se tiene que reestablecer el estado del bean.

11. Es conveniente recordar que los métodos de lógica de negocio son invocados por el cliente y que el servicio solicitado lo prestará bien, una nueva instancia del bean, o bien, el contenedor EJB gestionará el *pool* de instancias para elegir una cualquiera y recuperar el estado de la misma. Por lo tanto, el contenedor gestiona un pequeño grupo de instancias de acuerdo al tamaño de su *pool* y las distintas clases *primary keys* que le permiten recuperar el estado de los beans. Para recuperar y almacenar ese estado el contenedor EJB deberá llamar a los métodos ejbLoad( ) y ejbStore( ) que son los que le permiten con las claves primarias llevar a cabo un ciclo de activación/desactivación de la instancias asociadas al bean.

12. La llamada al método ejbActivate( ) se produce, como ya se sabe, cuando un cliente llama a un método sobre un objeto EJB y ese objeto no está unido a

ninguna instancia de un bean de entidad. Es en ese momento cuando el contenedor EJB debe tomar una instancia de bean del pool y llevarlo al estado de activo. Sin embargo hay que tener claro, tal y como se aprecia en el código mostrado, que en esta llamada no se realiza la lectura de los datos de la base de datos. Esa lectura se realiza justo después de llamar al método ejbActivate( ) durante la llamada el método ejbLoad( ). Como siempre ejbActivate( ) deberá apropiarse de todos los recursos que necesite el bean cuando pase al estado activo.

13. El contendor EJB llama al método ejbPassivate( ) cuando quiere devolver una instancia de un bean al pool. De nuevo, se puede apreciar en el código de este método como no es aquí donde se guarda la información del bean de entidad en la base de datos sino que eso se lleva a cabo en el método ejbStore( ) que es llamado justo antes del método ejbPassivate( ).

14. La clase del bean puede definir un método ejbPostCreate( ) por cada método ejbCreate( ). Cada par de métodos debe aceptar los mismos parámetros. El contenedor EJB llamará a ejbPostCreate( ) justo después de llamar a ejbCreate( ). El contenedor llama a este método después de que haya asociado una instancia del bean con un objeto EJB. Ese es el momento en el que se puede completar la inicialización haciendo todo aquello que requiera ese objeto EJB, como puede ser pasar la referencia del objeto EJB del bean a otro bean distinto.

Nota de seguimiento: en todos sitios cuecen habas

La verdad es que en el código que se ha mostrado hasta este momento hay un par de cosas que pueden considerarse por lo menos sospechosas. En el método operarCuenta( ) del bean GestorCuenta hay una última llamada a un método ingreso( ) sobre una cuenta que se ha creado a partir de una búsqueda con el método findByNombreTitular( ) y argumento “gerente”. Además, en el propio método ingreso( ) hay una condición para mostrar en el log la ejecución de dicho método que también involucra al “gerente”.

Muchas veces aparece el concepto “gerente” en este código. Vamos, que no hay que tener complejo de Sherlock Holmes para concluir, en pocas palabras, que el famoso gerente está metido en un negocio turbio. El famoso gerente, en “colaboración” seguramente con algún ingeniero de sistemas del banco, lo que realmente están desarrollando es una aplicación que no deja el sistema en el mismo estado inicial en el que se encontraba antes de realizar la ejecución, como era uno de los requisitos de la aplicación.

Por el contrario, la aplicación del ejemplo deja el sistema intacto, excepto en que cada vez que se realiza una ejecución, se incrementa el saldo de la cuenta del gerente en 1000 euros. Un pequeño detalle si se quiere, igual sin importancia, y que incluso se puede achacar a un error en desarrollo. Pero es que además, todo esto se hace sin dejar ningún rastro en el *log* de la parte servidora. Y eso ya no es un error y resulta más curioso. Lo de eliminar el rastro será por si quien tiene que utilizar la aplicación para comprobar su funcionamiento no es el propio gerente. Lógico.

Si señor, aunque cueste reconocerlo, un auténtico robo es lo que se está implementado en esta aplicación. Angelitos.

7.4.2.5.5. Descriptores de despliegue estándar

A continuación se describirán los detalles fundamentales relacionados con los ficheros descriptores de despliegue estándar que se utilizarán en la aplicación de ejemplo propuesta. Como siempre en estos ficheros se definirán y especificarán las necesidades que requieren los beans del contenedor EJB, como pueden ser, en este caso, la declaración de los recursos que van a utilizar los beans en forma de conexiones a la base de datos. Para la aplicación de ejemplo planteada se utilizarán en la aplicación dos ficheros descriptores de despliegue. En primer lugar, el correspondiente a los componentes EJB dentro del fichero ejb-jar.xml, y en segundo lugar, el fichero opcional de la tecnología EJB llamado application-client.xml, que se usará para el cliente *stand-alone*.

En primer lugar, el fichero descriptor de despliegue de los componentes EJB se guardará en el fichero ejb-jar.xml en el directorio /dd/ejb del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>

    <session>

      <ejb-name>GestorCuentas</ejb-name>
      <home>banca.GestorCuentasHome</home>
      <remote>banca.GestorCuentas</remote>
      <ejb-class>banca.GestorCuentasBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>

      <ejb-local-ref>
        <ejb-ref-name>ejb/RefCuentaCorriente</ejb-ref-name>
```

```
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>banca.CuentaCorrienteLocalHome</local-home>
        <local>banca.CuentaCorrienteLocal</local>
        <ejb-link>CuentaCorriente</ejb-link>
      </ejb-local-ref>

    </session>

    <entity>

      <ejb-name>CuentaCorriente</ejb-name>
      <local-home>banca.CuentaCorrienteLocalHome</local-home>
      <local>banca.CuentaCorrienteLocal</local>
      <ejb-class>banca.CuentaCorrienteBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>banca.CuentaCorrientePK</prim-key-class>
      <reentrant>false</reentrant>

      <resource-ref>
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>

    </entity>

  </enterprise-beans>

</ejb-jar>
```

Como se puede apreciar en el código del fichero descriptor de despliegue que se acaba de mostrar, la aplicación del ejemplo requiere que en este fichero se incluyan las declaraciones pertenecientes a todos los beans que utiliza la aplicación. Por lo tanto, en este caso, las dos declaraciones relativas a los dos beans de la aplicación estarán bien diferenciadas y se comentarán a continuación. Por un lado, se realizan las declaraciones relativas al bean de sesión llamado GestorCuentas dentro del elemento <session> y por otro lado se realizan las declaraciones del bean de entidad CuentaCorriente dentro del elemento <entity>.

Con respecto al código del bean de sesión GestorCuentas hay que decir hay una serie de elementos que ya fueron utilizados en ejemplos anteriores y que, lógicamente, ya fueron explicados en su momento. Por este motivo no se entrará en más detalles en lo que a esos elementos se refiere. Sin embargo, si que resultaria conveniente entrar con mayor detalle en el elemento <ejb-local-ref> que es nuevo y aparece por primera vez en el fichero ejb-jar.xml.

Para acceder a un determinado bean desde otro bean, como ya se comentó cuando se estudio la colaboración entre componente EJB, la especificación EJB define el concepto de referencia. El concepto de referencias también se tocó con anterioridad, sin embargo, la faceta que aún no se ha estudiado es la utilización de referencias utilizando interfaces locales como se hace en este ejemplo y no con interfaces remotos.

En este caso de interfaces locales, en el fichero descriptor de despliegue, tal y como se puede apreciar en el código mostrado, en lugar de utilizar el elemento <ejb-ref> se utiliza el elemento <ejb-local-ref> y en lugar de los elementos <remote> y <home> se utilizaran los elemento <local> y <local-home>. Como es lógico el método lookup( ) que busca un bean, buscará el objeto local Home en lugar del interfaz Home y las llamadas se realizarán sobre el interfaz local en lugar de hacerlo sobre el remoto.

También es importante el hecho de que se utiliza el elemento opcional <ejb-link> para la referencia local. Con la utilización de este elemento se descarta la utilización del elemento <ejb-local-ref> en el fichero jboss.xml. En su lugar, en este caso el mapeo se consigue directamente en el propio fichero ejb-jar.xml poniendo en el elemento <ejb-link> el nombre del bean referenciado.

Por otro lado, con respecto al código mostrado en el fichero ejb-jar.xml relacionado con el bean de entidad CuentaCorriente hay que destacar, por un lado, que en el elemento <persistence-type> ahora se debe establecer el valor de Bean en lugar de Container y que, por otro lado, en el elemento <prim-key-class> se establece el nombre totalmente cualificado (incluyendo en su nomenclatura todos sus paquetes) de la clase *primary key*.

Además, a parte de los elementos vistos anteriormente para el bean de entidad BMP CuentaCorriente, este descriptor de despliegue ejb-jar.xml también introduce un nuevo elemento denominado <resource-ref> que también se va a tratar mas en profundidad.

Con el elemento <resource-ref> se describe una referencia a una fábrica gestora de recursos. Este elemento le permite a las aplicaciones que lo utilizan el poder

referenciar a una fábrica de recursos usando nombres lógicos. Dicha fábrica gestora de recursos estará definida a través del elemento <resource-ref> en el fichero descriptor de despliegue estándar ejb-jar.xml, tal y como se acaba de mostrar. Este elemento <resource-ref> tiene una serie de elemento hijos que se detallan a continuación:

1. <description>: Es un elemento opcional que permite definir el propósito de la referencia.
2. <res-ref-name>: Es un elemento que especifica el nombre de la referencia relativa al contexto java:comp/env.
3. <res-type>: Es un elemento que especifica el nombre de la clase completamente cualificado de la fábrica gestora de recursos.
4. <res-auth>: Es un elemento que especifica si el código del componente de la aplicación lleva a cabo la tarea de registro en la fábrica de recursos o si por el contrario esa labor a realiza el contenedor EJB a través de la información del *principal* suministrado por el encargado del despliegue de la aplicación. Por lo tanto, los valores posibles para este elemento pueden ser tanto *Application* como *Container*.

Por otro lado, hay que mencionar que la especificación J2EE recomienda que todas las referencias a gestores de recursos sean organizadas en subcontextos de aplicación, usando distintos contextos dependiendo del tipo de gestor de recursos. De esta menera, los contextos recomendados dependiendo del tipo de gestor de recurso son los siguientes:

1. Las referencias JDBC o DataSources deberian ser declaradas en un subcontexto denominado java:comp/env/jdbc.
2. Las conexiones a las fábricas de servicios de mensajes o JMS deberian ser declaradas en un subcontexto denominado java:comp/env/jms.
3. Las conexiones a JavaMail deberian ser declaradas en un subcontexto denominado java:comp/env/mail.
4. Las referencias a las fábricas de conexiones URL deberian ser declaradas en un subcontexto denominado java:comp/env/url.

Como es habitual el encargado de realizar el despliegue de la aplicación enlazará el nombre simbólico del fichero ejb-jar.xml a la ubicación real de la fábrica gestora de recursos que realmente existe en el entorno operacional del servidor. Para conseguir este mapeo, tal y como se hacia en anteriores ocasiones, se usaran los ficheros descriptores de despliegue particulares del servidor de aplicaciones como jboss.xml, como se verá en el siguiente apartado.

En segundo lugar, está el fichero descriptor de despliegue opcional para las aplicaciones *stand-alone*, que también se incluirá en este ejemplo. Este descriptor se guardará en el fichero application-client.xml en el directorio /dd/client del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE application-client PUBLIC
          "-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//EN"
          "http://java.sun.com/dtd/application-client_1_3.dtd">


<application-client>

        <display-name>GestorCuentas</display-name>

        <ejb-ref>
              <description></description>
              <ejb-ref-name>refGestorCuentas</ejb-ref-name>
              <ejb-ref-type>Session</ejb-ref-type>
              <home>banca.GestorCuentasHome</home>
              <remote>banca.GestorCuentas</remote>
        </ejb-ref>

</application-client>
```

Como se puede apreciar en el código presentado, este descriptor también utiliza básicamente los mismos elementos vistos en descriptores de despliegue del ejemplo anterior. Las diferencias radican únicamente en que se utilizan, como es lógico, valores distintos para esos elementos en el ejemplo. Por este motivo tampoco se va a entrar en más consideraciones.

7.4.2.5.6. Ficheros específicos del servidor JBoss

Como se ha comentado anteriormente, la aplicación de Gestor de Cuentas, al igual que cualquier otra aplicación que se quiera desplegar en el servidor de aplicaciones JBoss, requiere que se especifiquen ciertos parámetros en los ficheros descriptores de despliegue específicos como jboss.xml.

De nuevo, como es lógico se necesitará para el ejemplo propuesto un fichero XML por cada uno de los descriptores de despliegue estándar que se hayan utilizado. A continuación se presenta el código y las características de cada uno de esos ficheros descriptores de despliegue específicos.

El código correspondiente al fichero descriptor de despliegue EJB para JBoss, que se guardará en el fichero jboss.xml en el directorio /dd/ejb del directorio de trabajo es el siguiente:

```
<!DOCTYPE jboss PUBLIC
          "-//JBoss//DTD JBOSS 4.0//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">
<jboss>

    <enterprise-beans>

         <session>
             <ejb-name>GestorCuentas</ejb-name>
             <jndi-name>ejb/banca/GestorCuentas</jndi-name>
         </session>

        <entity>
            <ejb-name>CuentaCorriente</ejb-name>
             <resource-ref>
                  <res-ref-name>jdbc/DefaultDS</res-ref-name>
                  <jndi-name>java:/DefaultDS</jndi-name>
             </resource-ref>
        </entity>

    </enterprise-beans>

</jboss>
```

El objetivo fundamental de este fichero, como ya se ha comentado en repetidas ocasiones, es el de realizar el mapeo entre los nombre logicos JNDI definidos en los

ficheros descriptores de despliegue estándar y la ubicaron real de los recursos en el entorno del servidor de aplicaciones.

Para conseguir el objetivo de mapear los recursos, como son en este caso los objetos Home de los beans, se utilizan los mismos mecanismos declarativos vistos hasta este momento. En el caso del bean de sesión GestorCuentas el mapeo se realiza como se ha hecho hasta ahora, haciendo coincidir el elemento <ejb-name> del fichero ejb-jar.xml con el elemento <ejb-name> de este fichero específico de JBoss.

Además del mapeo correspondiente al bean de sesión GestorCuentas en este fichero, también se debe realizar el mapeo al bean de entidad CuentaCorriente a través de la referencia local y el elemento <ejb-local-ref>. Como ya se comentó se ha optado en esta oportunidad por usar el elemento <ejb-link> que no requiere mapeo en este fichero.

Por su parte, para el bean de entidad CuentaCorriente hay que mencionar que al no declarar el elemento <jndi-name> en este fichero, el bean se mapeará utilizando por defecto el nombre definido en el elemento <ejb-name> del fichero ejb-jar.xml. Sin embargo, este fichero si que se define un elemento <resource-ref> que se comenta a continuación.

Cuando un bean tiene que utilizar referencias a recursos habría que introducir un nuevo elemento en los ficheros descriptores de despliegue especificos del servidor de aplicaciones JBoss. Por lo tanto, habría que declarar, bien en el fichero jboss.xml o jboss-web.xml, dependiendo del tipo de aplicación que se este implementando, un elemento denominado <resource-ref>. Como se puede apreciar en el código suministrado este elemento tiene una serie de elementos hijos que son necesarios declarar para llevar a cabo el proceso de mapeo que hace este fichero. Estos elementos se detallan a continuación:

1. <jndi-name>: Es un elemento que especifica el nombre JNDI de la fábrica de recursos tal y como ha sido desplegada en el servidor JBoss.

2. <res-ref-name>: Es el elemento que debe coincidir con el elemento de su mismo nombre incluido en el fichero descriptor de despliegue estándar correspondiente (ejb-jar.xml o web.xml).
3. <res-type>: Es un elemento opcional que especifica el nombre de la clase completamente cualificado de la fábrica gestora de recursos.

Por otro lado, la aplicación también requiere del fichero descriptor de despliegue de las aplicaciones *stand-alone* para el servidor JBoss. A continuación se enseña el código correspondiente, que se guardará en el fichero jboss-client.xml en el directorio /dd/client del directorio de trabajo.

```
<!--<!DOCTYPE jboss-client PUBLIC
       "-//JBoss//DTD Application Client 4.0//EN"
       "http://www.jboss.org/j2ee/dtd/jboss-client_4_0.dtd">-->

<jboss-client>

   <ejb-ref>
      <ejb-ref-name>refGestorCuentas</ejb-ref-name>
      <jndi-name>ejb/banca/GestorCuentas</jndi-name>
   </ejb-ref>

</jboss-client>
```

Como se puede apreciar en el fragmento de código mostrado anteriormente, este descriptor utiliza exactamente los mismos elementos que el descriptor de despliegue de las aplicaciones *stand-alone* para el servidor JBoss utilizado en el ejemplo anterior y que ya han sido explicados.

7.4.2.5.7. Cliente stand-alone

La aplicación propuesta como ejemplo para los beans BMP utiliza tanto los interfaces locales como los interfaces remotos. La utilización de los interfaces locales para los beans de entidad sigue la recomendación de la especificación de EJB en la que aconseja que los beans de entidad no sean accedidos de forma remota directamente por los clientes sino que por el contrario ese acceso se realice a través un bean de sesión.

Por lo tanto, el cliente de la aplicación Gestor de Cuentas lo que hace es acceder de forma remota al bean de sesión GestorCuenta. Por este motivo en este ejemplo sigue siendo necesario, como sucedia en los anteriores, que el cliente acceda al servicio de nombres JNDI recurriendo al fichero jndi.properties. El cliente, asumiendo que se ejecuta en una máquina remota a donde corre el servidor de aplicaciones JBoss, accederá a un contexto cuyos argumentos son proporcionados al cliente a través del fichero jndi.properties.

La clase principal del cliente se guardará en un fichero ClienteCuentaCorriente.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Cliente para el Enterprise JavaBean: CuentaCorriente
 */

import banca.GestorCuentasHome;
import banca.GestorCuentas;
import javax.naming.*;

public class ClienteCuentaCorriente
{
    public static void main(String [] args) throws Exception
    {
         try {
                 // Se accede al contexto por defecto
                 Context ctx = new InitialContext();
                 // Se busca el objeto publicado con JNDI
                 Object obj = ctx.lookup("GestorCuentas/refGestorCuentas");
                 GestorCuentasHome home = (GestorCuentasHome)obj;
                         javax.rmi.PortableRemoteObject.narrow(obj,
                                         GestorCuentasHome.class);

                 GestorCuentas cuenta = home.create();
                 // Se crea una cuenta nueva y se opera con ella
                 cuenta.operarCuenta("Alejandro Barrera");

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

Las cuestiones mas importantes a destacar del código de esta clase se enumeran a continuación:

1. El cliente crea un contexto con los parámetros JNDI que recibe a traves del fichero jndi.properties.

2. Una vez realizada la llamada al método lookup( ) para encontrar el objeto Home se realiza un proceso de cast RMI-IIOP a través de método narrow( ) de la clase javax.rmi.PortableRemoteObject( ). Posteriormente se realiza la creación del bean que permite realizar cualquiera de las operaciones que este proporciona.

3. En este caso, el cliente realiza una llamada al método operarCuenta( ) que crea una nueva cuenta con los datos suministrados en el código. Esta cuenta se utilizará para realizar unas cuantas operaciones sobre ella, comprobar que el funcionamiento es el esperado, y finalmente, se eliminará de nuevo dicha cuenta, para en teoria, dejar el sistema en el mismo estado en el que se encontró.

7.4.2.5.8. Fichero application.xml

El contenido del descriptor de despliegue application.xml, que declara todos los módulos J2EE incluidos dentro de la aplicación, se presenta a continuación y se guardará en el directorio /dd del directorio de trabajo.

```
<!DOCTYPE application PUBLIC
        "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
        "http://java.sun.com/dtd/application_1_3.dtd">

<application>

    <description>Una descripción cualquiera</description>

    <display-name>GestorCuentas</display-name>

    <module>
        <ejb>cuentas-ejb.jar</ejb>
    </module>

    <module>
        <java>app-client.jar</java>
    </module>

</application>
```

7.4.2.6. Utilizando Ant

Es el momento de llevar a cabo todas las tareas, con la ayuda de la herramienta Ant, que se definieron en el fichero jboss-build.xml. Al igual que en el ejemplo anterior, esta vez también se propone aprovechar al máximo la potencia de Ant y ejecutar de forma automática y consecutiva todas las tareas agrupadas en una única tarea. Para este caso particular la tarea se denomina *all* y ya se encuentra definida en el fichero jboss-build.xml.

La tarea *all* para este ejemplo implica la ejecución en primer lugar de la tarea de compilación, luego el empaquetamiento de los componentes EJB creando el fichero cuentas-ejb.jar, mas adelante el empaquetamiento del módulo del cliente originando app-client.jar, luego la tarea de ensamblaje de la aplicación que produce el fichero GestorCuentas.ear y, finalmente, el despliegue de dicho fichero dentro del servidor de aplicaciones. Sin embargo, en la tarea *all* no se ha incluido la ejecución de las dos tareas relacionadas con la base de datos que se ejecutaran de forma independiente en el siguiente apartado.

Para ejecutar esta tarea *all* que engloba a todas las demás hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml all
```

Como es habitual, si todo el procesamiento de las tareas ha ido correctamente al final se enseña un mensaje *Build Successful* y si se ha producido algún error se enseñará el mensaje *Build Failed*.

7.4.2.7. Poblando la base de datos

Ahora es el momento de ejecutar las tareas relacionadas con la base de datos. Para ejecutar la tarea *create-table* que crea las tablas que necesita la aplicación para funcionar hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml create-table
```

Por su parte para ejecutar la tarea *insert-data*, que puebla la tabla recién creada con los distintos datos incluidos en el *script*, hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml insert-data
```

En ambos casos si todo el procesamiento de las tareas ha ido correctamente al final se enseña un mensaje *Build Successful* y si se ha producido algún error se enseñará el mensaje *Build Failed*.

Nota de seguimiento: metiendo mano, y no la pata, en la base de datos

Ya esta todo listo para ejecutar la tercera aplicación de ejemplo. Sin embargo, antes de hacerlo sería conveniente, en primer lugar, comprobar que las operaciones sobre la base de datos realizadas hasta el momento se han ejecutado correctamente. Y en segundo lugar, sería importante conocer una forma rápida de comprobar como va evolucionando la base de datos del sistema a medida que los clientes ejecutan las funcionalidades que prestan los beans.

Por lo tanto, para tener acceso a la información gestionada por la base de datos se utilizará la herramienta de gestión de la base de datos MySQL. Para acceder a esta herramienta hay que arrancar la aplicación MySQL Command Line Client que proporciona un interfaz de consola bastante sencillo que permitirá comprobar la evolución de la base de datos y manipularla directamente.

Si, manipularla directamente. Normalmente se creará, destruirá y se buscará información de los beans de entidad utilizando el objeto Home de bean. Sin embargo, se puede interactuar con la información de los beans modificando directamente el contenido de la base de datos insertando o borrando registros de las tablas correspondientes. Esto debe ser posible si existen sistemas legados que accedan directamente a la base de datos para modificar la información.

Pero cuidado, que lo que se intenta es poder meter mano sin meter la pata. Todas estas actualizaciones externas pueden generar problemas de inconsistencias en *cache*. Hay que recordar que un bean de entidad, al ser un objeto Java o representación en memoria de unos datos de una base de datos, se puede considerar como una *cache* para la base de datos. A su vez, el contenedor EJB ofrece distintos algoritmos de cacheo con el mismo principio subyacente: reducir al mínimo el número de llamadas a los métodos ejbLoad( ) y ejbStore( ). Esta reducción en el número de llamadas efectivamente reduce al mínimo el *overhead* asociado a las actualizaciones que necesita la *cache* y eso, obviamente, aumenta el rendimiento. Pero esa reducción en el número de llamadas va en contra de la consistencia.

Por lo tanto, resulta fundamental la elección de un algoritmo de cacheo que mejore el rendimiento sin sacrificar la consistencia.

7.4.2.8. Ejecutando el Gestor de Cuentas

A la hora de ejecutar el cliente stand-alone de la aplicación Gestor de Cuentas, tambien se recurrirá a la ayuda de la herramienta Ant. Para realizar esta tarea de ejecución del cliente se utiliza el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml run-client
```

Esta tarea se encarga de ejecutar, mediante el comando java, la clase ClienteCuentaCorriente que pertenece al paquete app-client.jar ubicado en el directorio /jar del directorio de trabajo.

Como resultado de la ejecución anterior, en el lado servidor, después de ejecutar dos veces la aplicación, lo que se debe visualizar en la consola de *log* del servidor JBoss debe ser una secuencia similar a la siguiente:

```
21:09:45,237 INFO  GestorUsuariosBean.setSessionContext() invocado.
21:09:45,237 INFO  GestorUsuariosBean.ejbCreate() invocado.
21:09:45,287 INFO  Invocado getTotalEnCuentas
21:09:45,297 INFO  Total acumulado en cuentas: 9135.610000610352
21:09:45,297 INFO  ejbCreate
21:09:45,297 INFO  ejbPostCreate
21:09:45,297 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
21:09:45,297 INFO  EntityContext establecido
21:09:45,297 INFO  ejbStore
21:09:45,297 INFO  ejbFindByNombreTitular
21:09:45,307 INFO  ejbActivate
21:09:45,307 INFO  ejbLoad
21:09:45,307 INFO  Invocado getBalance
21:09:45,307 INFO  Saldo actual: 0.0
21:09:45,307 INFO  Ingreso de 200.0 euros solicitado
21:09:45,307 INFO  Invocado getBalance
21:09:45,307 INFO  Saldo actual: 200.0
21:09:45,307 INFO  Reintegro de 150.0 euros solicitado
21:09:45,307 INFO  Reintegro realizado con exito
21:09:45,307 INFO  Invocado getBalance
21:09:45,307 INFO  Saldo actual: 50.0
21:09:45,307 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
21:09:45,307 INFO  EntityContext establecido
21:09:45,307 INFO  Invocado getTotalEnCuentas
21:09:45,317 INFO  Total acumulado en cuentas: 9135.610000610352
21:09:45,317 INFO  Invocado getBalance
21:09:45,317 INFO  Encontrada con ID 2005a31d19 cuenta con saldo 50.0
21:09:45,317 INFO  ejbStore
21:09:45,317 INFO  ejbRemove
21:09:45,327 INFO  ejbFindByNombreTitular
21:09:45,327 INFO  ejbActivate
21:09:45,327 INFO  ejbLoad
21:09:45,327 INFO  ejbStore
```

```
21:09:47,099 INFO  GestorUsuariosBean.ejbPassivate() invocado.

21:10:06,337 INFO  GestorUsuariosBean.setSessionContext() invocado.
21:10:06,337 INFO  GestorUsuariosBean.ejbCreate() invocado.
21:10:06,367 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
21:10:06,367 INFO  EntityContext establecido
21:10:06,367 INFO  Invocado getTotalEnCuentas
21:10:06,367 INFO  Total acumulado en cuentas: 10135.610000610352
21:10:06,367 INFO  ejbCreate
21:10:06,367 INFO  ejbPostCreate
21:10:06,367 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
21:10:06,367 INFO  EntityContext establecido
21:10:06,367 INFO  ejbStore
21:10:06,377 INFO  ejbFindByNombreTitular
21:10:06,377 INFO  ejbActivate
21:10:06,377 INFO  ejbLoad
21:10:06,377 INFO  Invocado getBalance
21:10:06,377 INFO  Saldo actual: 0.0
21:10:06,377 INFO  Ingreso de 200.0 euros solicitado
21:10:06,377 INFO  Invocado getBalance
21:10:06,387 INFO  Saldo actual: 200.0
21:10:06,387 INFO  Reintegro de 150.0 euros solicitado
21:10:06,387 INFO  Reintegro realizado con exito
21:10:06,387 INFO  Invocado getBalance
21:10:06,387 INFO  Saldo actual: 50.0
21:10:06,387 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
21:10:06,387 INFO  EntityContext establecido
21:10:06,387 INFO  Invocado getTotalEnCuentas
21:10:06,387 INFO  Total acumulado en cuentas: 10135.610000610352
21:10:06,387 INFO  Invocado getBalance
21:10:06,387 INFO  Encontrada con ID 2005a31d19 cuenta con saldo 50.0
21:10:06,397 INFO  ejbStore
21:10:06,397 INFO  ejbRemove
21:10:06,397 INFO  ejbFindByNombreTitular
21:10:06,407 INFO  ejbStore
21:10:07,619 INFO  GestorUsuariosBean.ejbPassivate() invocado.
```

Aquí se puede apreciar como la aplicación en la primera ejecución primero crea una instancia del bean para llamar al método getTotalEnCuentas( ) que le da como resultado 9.135 euros. Luego se crea una nueva cuenta corriente, que inicialmente tiene como saldo 0 euros, se ingresan 200 y luego se retiran 150 quedando un saldo de 50 euros. En ese momento se hace una nueva llamada al método getTotalEnCuentas( ) y se observa como no se ha incrementado el saldo total ya que aún no se ha ejecutado el método store( ) que haría persistentes los cambios en la base de datos. Finalmente, se recupera la cuenta creada y se elimina para dejar el sistema en el mismo estado en el que se encontró.

La segunda vez que se ejecuta la aplicación, se puede apreciar como el comportamiento es exactamente igual que la anterior. Sin embargo, en esta ocasión se observa como el saldo total acumulado en las cuentas se ha incrementado en 1.000 euros. El incremento de estos 1.000 euros es el correspondiente al dinero de más que

cada vez que se ejecuta la aplicación se ingresa en la cuenta corriente del gerente de banco. Con esta segunda ejecución queda demostrado el desfalco que se lleva a cabo aquí.

7.5. Beans CMP (Container Managed Persistence)

En los beans de entidad con persistencia gestionada por el contenedor EJB (beans CMP en adelante), el contenedor EJB es el encargado de manejar la persistencia del bean. Como resultado, el código de acceso a los datos del bean, no está acoplado de forma programática a una fuente de datos específica. Esto libera al programador, por un lado, de tener que escribir código de acceso a los datos almacenados en la base de datos dentro de la propia clase del bean y, por otro, permite que el bean de entidad se pueda desplegar en diferentes contenedores y/o contra diferentes fuentes de datos.

Los beans CMP soportan la persistencia de forma declarativa o implícita gracias a las herramientas que proporciona el contenedor EJB. Es decir, no es necesario implementar los métodos de la interfaz javax.ejb.EntityBean, sino que tan sólo hay que definir de forma correcta el descriptor de despliegue adecuado para que así el contenedor EJB tenga la información necesaria para gestionar la persistencia contra una base de datos relacional. La mayoría de los proveedores EJB soportan la persistencia automática o CMP a una base de datos relacional, mediante el mapeo objeto-relacional (O/R).

Para usar la persistencia gestionada por el contenedor, el proveedor debe proporcionar algun tipo de herramienta propietaria que permita mapear los datos manejados por la instancia del bean a sus correspondientes columnas en una tabla de la base de datos. Una vez que los campos del bean se han mapeado a la base de datos, y se ha desarrollado el bean, el contenedor manejará la creación de registros, el borrado, la carga y la actualización de registros en la tabla en respuesta a los métodos invocados en los interfaces Home y remoto del bean.

El hecho de disponer de herramientas que mapean los campos de un bean a la base de datos es un avance importantísimo ya que permite que dicho bean pueda ser mapeado a cualquier base de datos. Sin embargo, esta independencia de la fuente de los datos implica que los atributos del bean deben ser de algún tipo simple primitivo o en su defecto de un tipo serializable. Estos atributos o campos del bean se llaman

manejados por el contenedor porque el contenedor EJB es el responsable de sincronizar su estado con la base de datos. Por lo tanto, los campos manejados por el contenedor EJB deben tener sus correspondientes columnas en la base de datos a través de mapeo objeto-relacional.

Sin embargo, no todos los atributos de un bean CMP serán manejados por el contenedor EJB. Algunos de ellos serán sólo para el uso temporal e interno del bean y no requieren persistencia. Por lo tanto, un desarrollador de beans distinguirá un atributo manejado por el contenedor de un atributo normal indicando aquellos que deben ser manejados por el contenedor EJB en el descriptor de despliegue durante la fase de desarrollo.

Los beans con persistencia manejada por el contenedor son los más simples para el desarrollador y los más complicados para dar soporte en el servidor EJB. Crear un CMP es realmente sencillo y la cantidad de funciones que lleva a cabo es realmente impresionante. Esto es porque toda la lógica de sincronización del estado del bean con la base de datos es manejada automáticamente por el contenedor EJB. A pesar de que en los principios de los EJB, este tipo de beans de entidad no estaban muy difundidos, ya que los programadores no se fiaban mucho de los contenedores EJB, actualmente son los más utilizados e incluso están recomendados debido a su mayor eficiencia con respecto a los beans BMP.

7.5.1. Arquitectura CMP

Para cualquier proveedor de beans o Independant Software Vendor (ISV), por ejemplo, es fundamental que los beans que implementa y que posteriormente vende, sean independientes de la base de datos que posteriormente se vaya utilizar ya que cada cliente utilizará la que considere más conveniente. Si a esta necesidad por parte de los vendedores, se le une el hecho de que obviamente, los compradores de beans no suelen tener acceso a los códigos fuentes de los componentes que compran por problemas de derechos intelectuales, se está frente a un problema de implementación serio que requiere una solución apropiada.

Y la solución como cabía esperar está en la especificación de EJB. Esta especificación propone una separación clara entre el bean de entidad CMP y su representación persistente. Es decir, una separación clara entre los métodos pertenecientes a la lógica de los datos y el propio código JDBC. Con esta separación se garantiza la independencia de la representación persistente sin afectar la lógica de negocio del bean de entidad.

Para conseguir este objetivo, la arquitectura propuesta a partir de la especificación EJB 2.0, exige que la implementación de la persistencia gestionada por el contenedor, para los beans de entidad, se base en el uso de subclases. De esta forma, la lógica de los datos se escribe en la clase bean mientras que el contenedor EJB generará toda la lógica de persistencia en una subclase de esa clase del bean. En realidad, la subclase heredará de la clase del bean y contendrá todo el código JDBC necesario para lleva a cabo la persistencia.

Para tener una visión más genérica de lo que se está tratando, se muestra a continuación un diagrama que muestra todas las clases involucradas en el desarrollo de un bean de entidad con la persistencia gestionada por el contendor EJB. Este diagrama se puede comparar con el diagrama del mismo tipo que se presentó para los beans BMP. Observando ambos diagramas se pueden apreciar las diferencias que existen entre ambos tipos en cuanto a la necesidad que tienen los beans CMP de una clase más que implementará el contenedor.

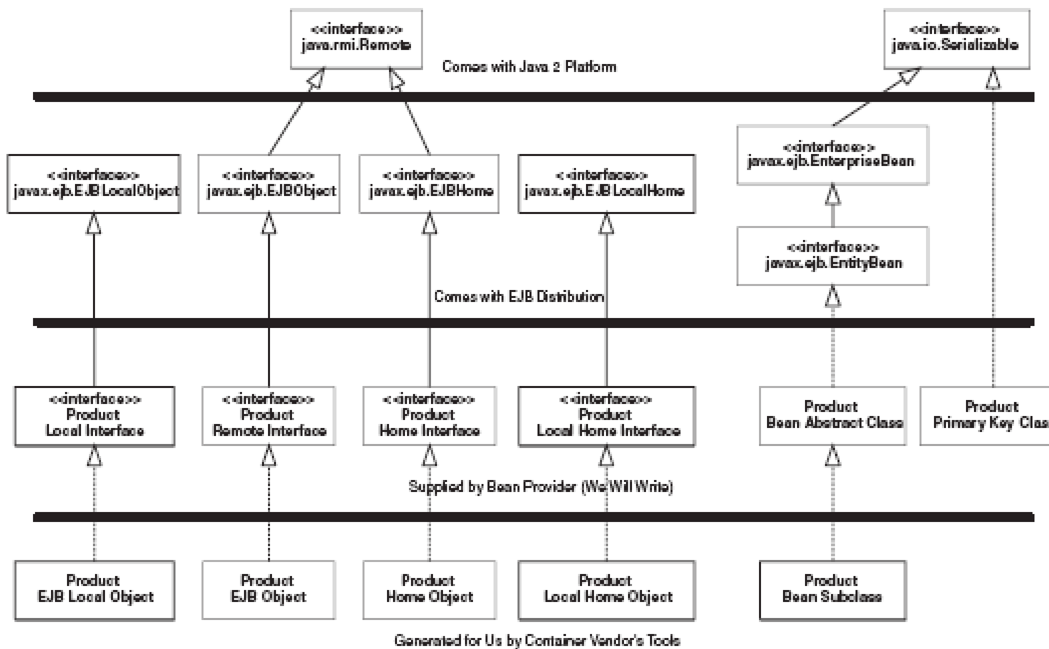


Figura 16: Diagrama de objetos para CMP

Por lo tanto, a la vista de esta arquitectura se puede deducir que cualquier bean de entidad CMP estará compuesto en realidad de dos clases. La primera es una superclase que contendrá toda la lógica de los datos del bean y sin nada de persistencia. Y una segunda clase considerada como subclase de la anterior en la que si que estará toda la lógica de persistencia.

En CMP, la superclase del bean no declara los atributos considerados persistentes. Estos atributos donde realmente están declarados son en la subclase del bean. Para acceder a la información de dichos atributos se accede mediante métodos getXXX( ) y setXXX( ) implementados también en la subclase. En la superclase del bean los métodos están declarados como abstractos, y por lo tanto, están únicamente definidos y no implementados.

La especificación EJB impone que los dichos métodos getXXX( ) y setXXX( ) se definan en la superclase como abstractos porque esos métodos pueden ser necesitados dentro de la lógica de datos que lleva a cabo la superclase. Por lo tanto, esos métodos no solo deben estar definidos e implementados en la subclase, sino que deben definirse en la superclase para que sean accesibles en ella, aunque en realidad los

implemente el contenedor EJB en la clase hija o subclase del bean. Además, resulta lógico pensar que sea en la subclase del bean donde se deben implementar estos métodos ya que es el único sitio donde están declarados los atributos que ellos manejan.

Es muy importante conocer la arquitectura CMP que propone la especificación EJB ya que CMP obliga a tener muy claro cuales son las cosas que puede hacer el contenedor EJB y cuales no. Por ejemplo, cualquier método que realice una operación de cálculo, aunque sea mínimamente complejo, como la suma de dos atributos de un bean es un método que debe estar definido en la superclase del bean, y por lo tanto, fuera del alcance del contenedor EJB. Esa operación aunque sea una simple suma, ya es suficientemente complicada para el contenedor EJB como para no poder realizarla. En cambio, la recuperación de la información de cada uno de los atributos que componen la suma antes mencionada si que son cálculos que se pueden delegar en el contenedor EJB, ya que su trabajo simplemente consiste en recuperar información de la base de datos y en eso si que es un experto.

Efectivamente, no todos los atributos definidos dentro de un bean pueden tener su persistencia manejada por el contenedor EJB. Por ejemplo, el desarrollador puede definir en su superclase atributos que sean mapeados a otras fuentes de datos o que, simplemente, sean fruto del cálculo de otros atributos como se ha visto anteriormente. Sin embargo, a pesar de que en estos casos el contenedor EJB no maneja la persistencia de esos datos, si que informa a la clase del bean de cuando tiene que llevar a cabo esa operación.

Finalmente, se puede afirmar que en general el contenedor EJB no será el responsable de la persistencia de ningún dato definido en la superclase, sino que solo se encargará de gestionar los atributos definidos en la subclase. Atributos definidos en la superclase como las referencias a los contextos de entidad o a los contextos de nombrado nunca deben estar en la subclase ya que no representan nunca información persistente.

7.5.2. Descriptor de despliegue ejb-jar.xml con CMP

Hasta ahora se ha visto como el contenedor EJB se encarga de generar todo el código JDBC de los métodos getXXX( ) y setXXX( ) asociados a los atributos persistentes del bean. Pero la pregunta que hay que plantearse ahora es: como sabe el contenedor EJB qué es exactamente lo que tiene que generar. La respuesta es que el contenedor generará todo aquello que sea declarado en el descriptor de despliegue del bean. El fichero descriptor de despliegue ejb-jar.xml le especificará al contenedor EJB la información sobre cómo el bean gestionará su persistencia o los atributos de transacción. Esta definición de la información que debe ser persistente se conoce como esquema abstracto de persistencia.

Un ejemplo de ese esquema abstracto de persistencia extraído de un descriptor de despliegue ejb-jar.xml de un bean es el siguiente:

```
<!DOCTYPE ejb-jar PUBLIC
        "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
        "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
        <display-name> </display-name>
        <enterprise-beans>

        <entity>
                <display-name>Alumno Entity Bean</display-name>
                <ejb-name>AlumnoEJB</ejb-name>
                <local-home>ejemplo.AlumnoHome</local-home>
                <local>ejemplo.Alumno </local>
                <ejb-class>ejemplo.AlumnoBean</ejb-class>
                <persistence-type>Container</persistence-type>
                <cmp-version>2.x</cmp-version>
                <abstract-schema-name>alumno</abstract-schema-name>
                <prim-key-class>java.lang.Integer</prim-key-class>
                <reentrant>False</reentrant>

                <cmp-field>
                        <field-name>alumnoId</field-name>
                </cmp-field>
                <cmp-field>
                        <field-name>nombre</field-name>
                </cmp-field>
                <cmp-field>
                        <field-name>mote</field-name>
                </cmp-field>
                <primkey-field>alumnoId</primkey-field>
                <!-- ... -->
        </entity>

        </enterprise-beans>

</ejb-jar>
```

Ejemplo de elementos CMP dentro de un ejb-jar.xml. Admin.pdf 373

A continuación se destacan los elementos más importantes del código de este descriptor de despliegue:

1. <persistence-type>: Este elemento es el que determina si se quiere usar una persistencia tipo BMP o CMP.

2. <cmp-version>: Este elemento debe ser 2.x si se quieren aprovechar las ventajas de la especificación EJB 2.0.

3. <abstract-schema-name>: es el nombre que se le quiera dar al esquema de persistencia. Este elemento puede tener cualquier valor, pero hay que tener en cuenta que luego será este nombre el que se referenciará cuando se realicen las queries JDBC.

4. <cmp-field>: Este elemento especifica los atributos que se van a considerar con persistencia gestionada por el contenedor. Cada uno de estos atributos será un atributo que el contenedor generará en la subclase. Cada atributo definido en uno de estos elementos le indicará al contenedor EJB que tiene que crear una columna en la nueva tabla de la base de datos y mapear así cada atributo a su columna particular. Además, es importante destacar que los nombres de estos campos del descriptor deben coincidir con los nombres de los métodos getXXX( ) y setXXX( ) definidos como abstractos excepto la primera letra que no irá en mayúscula. Finalmente, es importante tener en cuenta que dichos nombres tienen que encajar ya que el contenedor EJB también obtendrá el tipo de los atributos definidos en el descriptor de despliegue de los métodos getXXX( ) y setXXX( ).

5. <prim-key-class>: Este elemento especifica el nombre y el tipo de la clave primaria para el bean.

6. <reentrant>: Este elemento especifica si el bean en cuestion puede ser llamado a si mismo a través de otro bean intermedio. Es decir, un bean A es reentrante

si A llama a B y este último a su vez vuelve a llama a A. Con un valor de *true* se consigue este tipo especial de multithreading que obliga a tener mucho cuidado, ya que puede traer consigo comportamientos inesperados durante la ejecución.

7. <resource-ref>: Este elemento permite configurar recursos para el bean, como puede ser un driver JDBC y lo que es mas importante, permite que dichos recursos esten disponibles en un determinada localizacion JNDI para que pueda ser utilizada por los clientes.

7.5.3. El motor CMP2 en JBoss

En esta sección se estudiarán en detalle las particularidades del motor de persistencia CMP2 de JBoss. Este motor, denominado JBossCMP, es el gestor de persistencia por defecto que proporciona el servidor de aplicaciones para cualquier aplicación EJB 2.0 y superiores. JBossCMP se considera una funcionalidad interna del propio servidor, por lo que, viene en la instalación por defecto del servidor y se arrancará por defecto como un servicio más del servidor cuando este se arranque.

7.5.3.1. Descriptor de despliegue JBoss para CMP

La gestión de la persistencia realizada por un servidor de aplicaciones como JBoss requiere de ciertas consideraciones particulares. En este caso, si un bean de entidad utiliza la persistencia gestionada por el contenedor EJB de JBoss, además del fichero descriptor de despliegue estándar ejb-jar.xml, JBoss requiere configurar la persistencia gestionada por el contenedor a través de un fichero con extensión .xml y especifico del servidor.

Este fichero será utilizado para controlar el comportamiento del motor de persistencia JBossCMP del servidor de aplicaciones JBoss. El control de dicho comportamiento se puede realizar de forma global a través del fichero descriptor de despliegue conf/standardjbosscmp-jdbc.xml que se encuentra dentro del conjunto de ficheros de

configuración del servidor. Dicho control también se puede realizar de forma especifica a la aplicación que se esta desarrollando utilizando para ello el fichero de empaquetamiento JAR de los componentes EJB. Para este último caso se debe incluir en ese fichero JAR de la aplicación, en el directorio META-INF, un fichero denominado jbosscmp-jdbc.xml.

A continuación se van a describir los principales elementos incluidos dentro de este fichero propietario de JBoss y que resumen las características básicas que se pueden configurar en el motor de persistencia JBossCMP que proporciona el servidor de aplicaciones JBoss. Los elementos principales de dicho fichero de muestran en la siguiente figura:

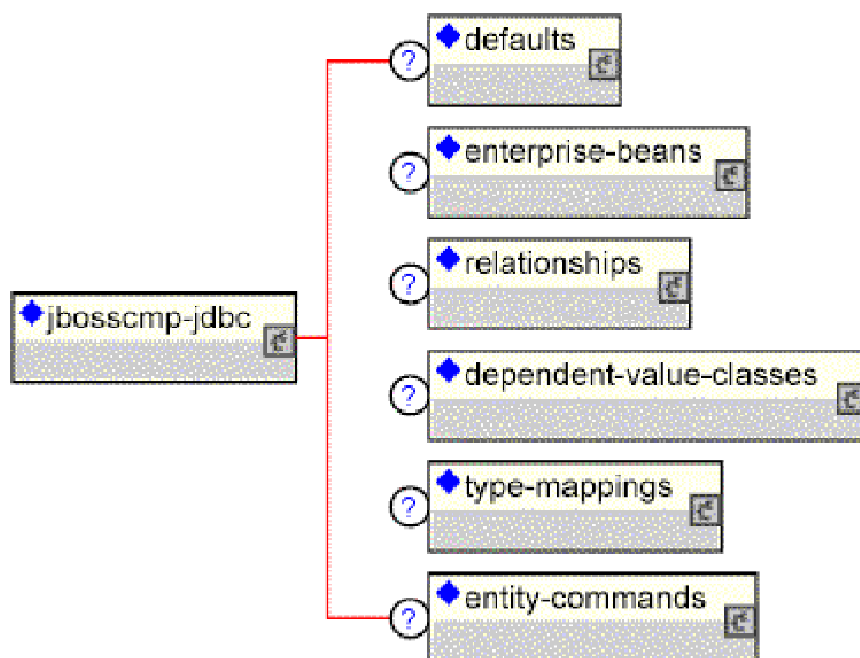


Figura 17: Estructura del fichero descriptor de despliegue jbosscmp-jdbc.xml

1. <default>: Este elemento crea una sección que permite especificar el comportamiento de control que tendrán por defecto los beans de entidad. Con este elemento se permite disminuir la cantidad de información que se debe suministrar a los beans de entidad aprovechando toda la información global definida por defecto para JBossCMP en el fichero global standardjbosscmp-jdbc.xml.

2. <enterprise-bean>: Este elemento permite configurar las características particulares de cada uno de los beans definidos en el fichero descriptor de despliegue ejb-jar.xml.

3. <relationships>: Este elemento permite la configuración de las tablas y definir el comportamiento de la carga de los datos de aquellas entidades involucradas en una relación. Es decir, con esta información declarativa, se delega en el contendor la gestión de las relaciones 1-1, 1-N o N-N con integridad referencial.

4. <type-mappings>: Este elemento permite definir el mapeo de los tipos Java a los tipos SQL de la base de datos.

El DTD entero para el fichero descriptor de despliegue jbosscmp-jdbc.xml se encuentra en la página principal de donde se puede descargar la distribución JBoss, en la ruta /docs/dtd/jbosscmp-jdbc_3_2.dtd. A continuación se muestra la cabecera del DOCTYPE para este DTD:

```
<!DOCTYPE jbosscmp-jdbc PUBLIC
"-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
"http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">
```

7.5.3.2. Configuración del mapping en JBoss

A continuación se hará un estudio más en profundidad de uno de los elementos de alto nivel, como es el elemento <enterprise-bean> visto anteriormente y que es el que permite declarar la mayor parte de la lógica necesaria para hacer el mapeo objeto-relacional en el que se basa la persistencia.

Todos los beans de entidad que deban ser mapeados a un soporte persistente deben aparecer en el fichero jbosscmp-jdbc.xml. Cada uno de estos beans estará dentro de un elemento <entity> y todos ellos estarán agrupados juntos dentro del elemento de alto nivel <enterprise-bean>. A continuación se muestra un sencillo ejemplo que

ilustra la apariencia de un descriptor de despliegue jbosscmp-jdbc.xml con sus elementos básicos:

```
<!DOCTYPE jbosscmp-jdbc PUBLIC
"-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
"http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">

<jbosscmp-jdbc>

    <defaults>
        <datasource>java:/DefaultDS</datasource>
        <datasource-mapping>MySQL</datasource-mapping>
    <defaults>

    <enterprise-beans>

        <entity>
                <ejb-name>AlumnoEJB</ejb-name>
                <table-name>alumno</table-name>
                <cmp-field>
                        <field-name>matricula</field-name>
                        <column-name>id</column-name>
                </cmp-field>
                <cmp-field>
                        <field-name>nombre</field-name>
                        <column-name>nombre</column-name>
                        <jdbc-type>VARCHAR</jdbc-type>
                        <sql-type>VARCHAR(64)</sql-type>
                </cmp-field>
                <cmp-field>
                        <field-name>negativos</field-name>
                        <column-name>nota</column-name>
                </cmp-field>

                <!-- Queries -->
        </entity>

        <entity>
                <ejb-name>ProfesorEJB</ejb-name>
                <table-name>profesor</table-name>
                <cmp-field>
                        <field-name>dni</field-name>
                        <column-name>id</column-name>
                </cmp-field>
                <cmp-field>
                        <field-name>asignatura</field-name>
                        <column-name>asignatura</column-name>
                </cmp-field>
                <query>
                        <query-method>
                                <method-name>findProfesor</method-name>
                                <method-params>
                                        <method-param>int</method-param>
                                </method-params>
                        </query-method>
                        <ejb-ql><![CDATA[
                                SELECT OBJECT(o)
                                FROM profesor AS o
                                WHERE o.id > ?1
                                ]]>
                        </ejb-ql>
                </query>
        </entity>
```

```
    </enterprise-beans>
</jbosscmp-jdbc>
```

Hay que destacar que todos lo elementos pertenecientes a <entity> son opcionales, como se puede consultar en su DTD, a excepción del elemento <ejb-name>. Este elemento debe coincidir con el nombre de la entidad declarada en el fichero ejb-jar.xml ya que es el mecanismo de mapeo utilizado para asignar la configuración de persistencia al bean. Además, todos aquellos valores que no se incluyan en este fichero tomarán los valores por defecto definidos, bien en el propio fichero jbosscmp-jdbc.xml tal y como se puede apreciar en el codigo que se muestra, o bien del fichero conf/standardjbosscmp-jdbc.xml que contiene la configuración por defecto actual para el servidor.

A continuación se hace una breve descripción de cada uno de los elementos que pueden pertenecer al elemento <entity>.

1. <ejb-name>: Es un elemento obligatorio y que define el nombre del bean al que aplica la configuración. Como ya se ha comentado este nombre debe coincidir con el nombre especificado en el elemento <ejb-name> del fichero ejb-jar.xml.

2. <datasource>: Es un elemento opcional que representa el nombre JNDI utilizado para buscar la fuente de datos. Todas las conexiones a la base de datos que usan los bean de entidad son obtenidos a partir de la fuente de datos. El valor por defecto a no ser que se sobrescriba en la sección de *defaults* es java:/DefaultDS.

3. <create-table>: Es un elemento opcional que específica, cuando tiene valor *true*, que el motor JBossCMP debe intentar crear una tabla asociada al bean de entidad. Cuando la aplicación es desplegada, el motor JBossCMP verifica si la tabla ya esta creada antes de hacerlo. El valor por defecto a no ser que se sobrescriba en *default* es *false*.

4. <table-name>: Es un elemento opcional que define le nombre de la tabla que va a mantener los datos del bean de entidad. Cada una de la instancias de ese bean será almacenado en un registro distinto de esa tabla. El valor por defecto de este elemento es <ejb-name>

5. <cmp-field>: Es un elemento opcional que permite definir como un campo definido en el fichero ejb-jar.xml será mapeado dentro del dispositivo de persistencia. Este elemento, como se puede ver en el ejemplo anterior puede presentar a su vez subelementos dentro de los que destacan los que se enumeran a continuación:

    - <field-name>: Es un elemento obligatorio y es el nombre del atributo que esta siendo configurado. Este nombre tiene que coincidir con el nombre definido en el elemento <cmp-field> declarado para ese bean de entidad en el fichero ejb-jar.xml.
    - <column-name>: Es un elemento opcional que representa el nombre de la columna a la que va a ser mapeado el atributo <field-name>. El valor por defecto, en caso de no declararse, será el del atributo <field-name>.
    - <jdbc-type>: Solo es necesario declararlo si también se declara el elemento <sql-type>. Representa el tipo JDBC que se usará cuando se pasen parámetros en una sentencia JDBC PreparedStatement o cuando se carguen datos desde un ResultSet de JDBC. Los tipos validos estarán definidos en java.sql.Types.
    - <sql-type>: es el tipo SQL que se usará en las sentencias de creación de las tablas para el campo en el que se defina. Los tipos válidos SQL estarán limitados, obviamente, por el proveedor de la base de datos que se vaya a utilizar. Este elemento, al igual que el anterior, solo es necesario si se ha definido el elemento <jdbc-type>.

6. <query>: Es un elemento opcional que define, utilizando el lenguaje EJB-QL que se estudiará a continuación, la definición de los métodos *finders* y de selección.

7.5.3.3. Profundizando en la configuración de CMP

Existen otras maneras de afinar el comportamiento del motor CMP aparte de con la utilización del fichero jbosscmp-jdbc.xml que se ha visto anteriormente. Este fichero se utiliza para aportar información básica como puede ser el nombre de la fuente de datos, el nombre de las tablas y las columnas en la base de datos, si dichas tablas deben ser automáticamente creadas en tiempo de despliegue o incluso la configuración de los tipos que se utilizarán en el mapeo.

Sin embargo, tambien existe un fichero estándar de configuración de CMP llamado standardjbosscmp-jdbc.xml que reside en el directorio /conf del servidor que se esté utilizando. Este fichero contiene una serie de valores por defecto para unos cuantos parámetros y una lista de valores por defecto para los tipos de mapeo que utilizan las distintas bases de datos. A continuación se muestra el comienzo de este fichero para ver su apariencia.

```
<jbosscmp-jdbc>

        <defaults>

                <datasource>java:/DefaultDS</datasource>
                <datasource-mapping>MySQL</datasource-mapping>
                <create-table>true</create-table>
                <remove-table>false</remove-table>
                <read-only>false</read-only>
                <read-time-out>300000</read-time-out>
                <row-locking>false</row-locking>
                <pk-constraint>true</pk-constraint>
                <fk-constraint>false</fk-constraint>
                <preferred-relation-mapping>foreign-key</preferred-
                                                        relation-mapping>

                <read-ahead>
                        <strategy>on-load</strategy>
                        <page-size>1000</page-size>
                        <eager-load-group>*</eager-load-group>
                </read-ahead>

                <list-cache-max>1000</list-cache-max>
                ...

        </defaults>

</jbosscmp-jdbc>
```

Como se puede apreciar, entre otras cosas, este fichero impone la creación de la tabla correspondiente al bean si el esquema no ha sido creado previamente. Los elementos <read-only> y <read-time-out> especifican si los datos son de solo lectura y el tiempo en milisegundos durante el que son válidos.

Por otro lado, como se puede apreciar en el código mostrado, con este fichero se puede afinar la forma en la que el motor CMP cargará los datos de acuerdo a lo que se espera que esos datos sean usados. Por ejemplo, usando el elemento <read-ahead> se puede conseguir leer y tener en caché bloques de datos para distintos beans con una unica llamada a una sentencia SQL, aticipandose a futuros accesos. Un grupo de carga impaciente (o eager-loading group) puede especificarse para lograr que solo ciertos campos sean cargados inicialmente desde la fuente de datos al bean. El resto de los campos se considerarán cargados de forma perezosa (lazy-loading) solo en el caso de ser requeridos.

Para este caso particular el elemento <read-ahead> usa la estrategia *on-load* que significa que los datos del bean serán cargados cuando el bean sea accedido y el elemento <page-size> indica que los datos de hasta 1000 entidades serán cargados con una única sentencia SQL.

Esta flexibilidad que se alcanza con los beans CMP es imposible de conseguir con los beans BMP en los que cada bean debe ser cargado con datos procedentes de una única sentencia SQL.

Nota de seguimiento: una de anatomia, cuanto vale un dedo pulgar?

Como se habrán podido dar cuenta muchos, entre otras cosas, el fichero standardjbosscmp-jdbc.xml configura, al igual que lo hacia el fichero jbosscmp-jdbc.xml, el nombre de la fuente de datos y el tipo de mapping que se va a utilizar con la correspondiente base de datos. Es decir, se puede sobrescribir el comportamiento por defecto definido en el fichero standardjbosscmp-jdbc.xml utilizando un fichero jbosscmp-jdbc.xml e incluso añadir en este más información si es necesario.

En definitiva, toda la configuración del motor CMP se puede realizar tanto añadiendo un nuevo fichero descriptor de despliegue que se denominaría jbosscmp-jdbc.xml y que debería ser parte del fichero de despliegue del componente EJB o modificando el fichero global de parametros por defecto en standardjbosscmp-jdbc.xml. Para llevar a cabo esta última solución tan solo habría que editar el fichero en cuestión y modificar el elemento <type-mapping> el valor adecuado.

Cual es la mejor solución?. Depende. Cualquiera de las dos soluciones es válida. Si bien es cierto que la última solución es más simple que la primera, también es justo decir que esa solución tiene también desventajas ya que hay que reiniciar el servidor de aplicacione JBoss para que los cambios tengan efecto.

Por su parte, utilizar la solución de crear un nuevo descriptor de despliegue efectivamente requiere que se tenga que empaquetar de nuevo la aplicación, pero no implica reiniciar el servidor. No está de más recordar que, como siempre, todo lo que se necesita hacer para desplegar un bean en JBoss es crear un fichero .jar con las clases compiladas del bean y los descriptores de desliegue y situarlo en el directorio de despliegue de su servidor particular JBoss ya que el despliegue se hace en caliente.

Además, la elección de un tipo u otro de configuración del motor de CMP, no solo depende de los pros y los contras de cada una de las soluciones. En esta cuestión también influye la granularidad de cada caso. Hay que tener siempre presente que el fichero standardjbosscmp-jdbc.xml sería un fichero con información global a todo el servidor de aplicaciones JBoss, y que por lo tanto aplicaría a cualquier aplicación que corriera dentro del servidor. Sin embargo, la creación de un fichero jbosscmp-jdbc.xml particular permitiría la definición de unos valores por defecto que afectarían solo a la aplicación con la que este empaquetado ese descriptor de despliegue.

Por lo tanto, la mejor respuesta es: depende. Por cuanto te dejarias cortar el dedo pulgar de la mano con la escribes?. Depende. Todos tenemos un precio y el cuanto, lo determinaría nuestra escala de valores.

7.5.4. EJB-QL

Como ya se ha visto los beans de entidad debido a su persistencia pueden ser buscados. Para ello la especificación EJB permite la creación de cualquier número de métodos *finders*. En BMP esos métodos eran definidos en el interfaz Home que a su vez delegaba en la clase del bean que realmente tenia la implementación del método hecha por el desarrollador.

Sin embargo, en CMP el contenedor EJB es el encargado de implementar o generar el código JDBC necesario en los métodos *finders*. Pero como se ha dicho antes, la especificación EJB permite la definición del número de métodos *finders* que se quiera. Por lo tanto, cabría preguntarse como sabe el contenedor cuáles son realmente los métodos *finders* que tiene que implementar y cuál es el código JDBC asociado a ellos. Se necesita entonces una forma estándar y portable que permita definir esos métodos *finders* para no tener que reescribirlos cuando se decida migrar el bean CMP a otro contenedor distinto.

La solución la proporciona la especificación EJB ofreciendo un lenguaje de consultas, llamado EJB-QL. Este lenguaje es utilizado para escribir consultas en beans de entidad manejados por el contenedor (CMP) de forma independiente del sistema de almacenamiento. Dichas consultas escritas en EJB-QL las traducirá el contenedor EJB al lenguaje de consulta SQL propietario del sistema gestor de base de datos en el que serán desplegados los datos del bean.

EJB-QL se utiliza para crear las consultas que se utilizarán en los métodos *find* definidos en el interfaz Home. Las distintas consultas se sitúan dentro del fichero descriptor de despliegue ejb-jar.xml de la aplicación. Dichas consultas EJB-QL contendrán una cláusula SELECT y una cláusula FROM, y opcionalmente, podrán contener una cláusula WHERE.

En EJB-QL, en lugar del nombre de la tabla, se utiliza el nombre del esquema definido en el descriptor de despliegue. Lo mismo ocurre con los nombres de columnas que se utilizarían en SQL, en EJB-QL esos nombres son reemplazados por los correspondientes atributos declarados en el bean.

El siguiente código XML definido dentro del fichero descriptor de despliegue mapea una consulta a un método findByState( ) del interfaz Home de un bean. Una aplicación puede utilizar este método y por lo tanto la consulta asociada, por ejemplo, para buscar el usuario que tiene un determinado estado.

```
<entity>

      ...
      <query>
         <query-method>
            <method-name>findByState</method-name>
            <method-params>
               <method-param>java.lang.String</method-param>
            </method-params>
         </query-method>
         <ejb-ql>
            [!CDATA[
            SELECT DISTINCT OBJECT(u)
            FROM UserEJB AS u
            WHERE u.STATE = ?1]]
         </ejb-ql>
      </query>

</entity>
```

Como se puede apreciar, el elemento ejb-ql define la consulta EJB-QL real a utilizar. La palabra clave OBJECT es necesaria cuando se devuelve un único tipo de objeto. La consulta que se propone en el ejemplo es una consulta parametrizada. Se denominan así ya que son sentencias SQL que reciben parámetros. De esta forma cada número representa un parámetro de la lista de parámetros, empezando con un índice de 1. En el ejemplo, el número 1 será reemplazado por el primer parámetro pasado al método findByState.

Los beans de entidad que emplean CMP pueden definir un método findAll( ) y/o un método findByPrimaryKey( ), que serán implementados y ejecutados de forma transparente por el contenedor EJB.

7.5.5. El método ejbSelect( )

Finalmente, la última gran diferencia entre los beans BMP y CMP es que los beans de entidad CMP pueden disponer de un método especial denominado ejbSelect( ). Este método es muy parecido a los métodos *finders* que permitían a los clientes buscar una determinada instancia de un bean de entidad.

Sin embargo, los métodos ejbSelect( ) no son exportados por los beans a través de sus interfaces remoto o Home como lo eran los métodos *finders*. Por el contrario, los métodos de selección son métodos que son usados normalmente de forma interna dentro del bean como métodos que colaboran en el acceso a los sistemas de almacenamiento. Es decir, son métodos que usa un bean de entidad cuando se relaciona con datos externos así como con otros beans.

En definitiva, un método ejbSelectXXX( ) puede realizar operaciones especificas de base de datos que el bean internamente requiere, pero que no resulta necesario publicar a los clientes de ese bean mediante la definición del método en sus interfaces. Para llevar a cabo esta tarea en CMP no es necesario que el desarrollador escriba en el método el código JDBC correspondiente. En su lugar, el desarrollador le dice al contenedor como quiere que se implemente ese método y el propio contenedor EJB

generará ese código JDBC. Como es lógico, la forma de indicarle al contendor la implementación del método ejbSelect( ) es utilizando el lenguaje EJB-QL y definiendo la sentencia que se necesita para que posteriormente el contenedor EJB realice la traducción.

Los métodos ejbSelectXXX( ) se definirán como métodos abstractos en la superclase del bean y se implementarán al igual que los métodos getXXX( ) y setXXX( ) en la subclase generada por el contenedor EJB.

7.5.6. Ejemplo CMP: gestión de cuentas con CMP

7.5.6.1. Introducción

Como ejemplo práctico para este tema se propone la implemantación de la misma aplicación que se utilizó en el ejemplo de beans BMP, pero lógicamente adaptado al tipo de beans que ahora se estan estudiando. Con esta propuesta se pretende no romper con la filosofía que se ha seguido hasta el momento en lo que se lleva de libro. Es decir, se busca la implementacion de una aplicación que cumpla un fin claramente didáctico y que permita ver las caracteristicas principales de la tecnología que en ese momento se esta estudiando.

Con la implementacion de la misma aplicación que en el ejemplo anterior se podrán ver claramente las diferencias que distinguen BMP de CMP, sus similitudes y las desventajas o facilidades que aporta una sobre la otra. Todo esto, además, sin perder tiempo en el análisis y diseño de una nueva aplicación y aprovechando toda la infraestructura de base de datos del ejemplo anterior.

Con respecto a la gestión de la base de datos en esta aplicación, hay que mencionar que en este ejemplo se intentará aprovechar al máximo la potencia que ofrece el motor CMP del servidor de aplicaciones JBoss. Es decir, este ejemplo no solo se centrará en mostrar la funcionalidad típica que aporta el servidor de aplicaciones JBoss al trabajar con beans CMP, en lo que a la gestión de persistencia de los datos que maneja este tipo de beans se refiere. Por el contrario, se intenta ir mas allá y ver también como se

puede minimizar al máximo la realización, por parte del programador, de todas las tareas de gestión de las base de datos.

Por lo tanto, en este ejemplo, se ilustrará al lector en los mecanismos que aporta la tecnología EJB, a través de su especificación, para que el desarrollador pueda delegar en el propio contenedor EJB del servidor de aplicaciones JBoss gran parte de las tareas relacionadas con la base de datos. Por ejemplo, la creación de la tabla donde se almacenan las cuentas corrientes y que da soporte presistente al bean, la llevará a cabo el propio contenedor EJB y no la tendrá que realizar el propio programador ejecutando una tarea *ant* que contenia el código SQL necesario como en el ejemplo anterior.

En definitiva, la aplicación de ejemplo propuesta se denominará Gestor de Cuentas CMP y será básicamente una aplicación *stand-alone* exactamente igual que la del ejemplo anterior. De nuevo se permitirá al cliente de la aplicación realizar una serie de operaciones básicas sobre las distintas cuentas corrientes que gestiona el banco. La única diferencia radica en que en este caso el bean de entidad que se va a utilizar será un bean CMP en lugar de BMP.

7.5.6.2. Requisitos para ejecutar el ejemplo

El sistema gestor de base de datos que se va a utilzar en este ejemplo lógicamente vuelve a ser MySQL con la configuración que se llevó a cabo para el ejemplo anterior. La base de datos sigue siendo la de banco que ya fue creada en el ejemplo anterior.

En cuanto a las tablas que requiere el ejemplo de la base de datos banco, como ya se ha comentado, se creará una nueva tabla denominada cuentaCMP para que no interfiera con la tabla cuenta creada para el ejemplo anterior y que será responsabilidad del contenedor EJB.

Finalmente, hay que aclarar que en este caso no se van a necesitar scripts de base de datos. El primero de los *scripts* utilizados en el ejemplo anterior, denominado create-

table.sql, se encargaba de crear la tabla de soporte de las cuentas corrientes y esa labor la hará el contenedor. Por su parte, el segundo de esos *scripts*, denominado insert.sql, insertaba una serie de cuentas que no son absolutamente imprescindibles para que el ejemplo funcione.

Por esta razón se ha optado por no insertar en principio dichos datos en la base de datos. Sin embargo, se deja a elección del lector el que quiera ejecutar en cualquier momento el script de inserción de cuentas del ejemplo anterior con la única modificación del cambio en el nombre de la tabla.

7.5.6.3. Preparando el fichero jboss-build.xml de Ant

A continuación se muestra el fichero de configuración jboss-build.xml que se utilizará en el desarrollo de la aplicación de ejemplo Gestor de Cuentas CMP. Como siempre, este fichero ha sido construido utilizando como plantilla el fichero jboss-build.xml del ejemplo anterior. En este caso simplemente se han modificado un par de nombres de ficheros que genera la herramienta y se han eliminado las tareas relacionadas con los scripts de creación de tabla y el de inserción.

En definitiva el fichero jboss-build.xml se guardará en el directorio raiz del directorio de trabajo y su código se muestra a continuación:

```
<project name="GestorCuentas" default="all" basedir=".">

    <property file="jboss-build.properties"/>

    <property name="lib.dir"   value="../../libs"/>
    <property name="src.dir"   value="${basedir}/src"/>
    <property name="build.dir" value="${basedir}/build"/>

    <!-- The classpath for running the client -->
    <path id="client.classpath">
        <fileset dir="${jboss.home}/client">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <!-- The build classpath -->
    <path id="build.classpath">
        <path refid="client.classpath"/>
        <fileset dir="${jboss.server}/lib/">
        </fileset>
    </path>
```

```
<!-- Hypersonic SQL classpath -->
<path id="hsql.classpath">
    <pathelement location="${jboss.server}/lib/hsqldb.jar"/>
</path>

<!-- ================================================================ -->
<!-- Directorio de creacion                                           -->
<!-- ================================================================ -->
<target name="prepare">
    <mkdir dir="${build.dir}"/>
</target>

<!-- ================================================================ -->
<!-- Compilacion del codigo fuente                                    -->
<!-- ================================================================ -->
<target name="compile" depends="prepare">
    <javac destdir="${build.dir}" classpathref="build.classpath"
                                                        debug="on">
        <src path="${src.dir}"/>
    </javac>
</target>

<target name="package-ejb" depends="compile">
    <mkdir dir="jar" />
    <delete file="jar/cuentasCMP-ejb.jar"/>
    <jar jarfile="jar/cuentasCMP-ejb.jar">
        <metainf dir="dd/ejb" includes="**/*.xml" />
        <fileset dir="${build.dir}">
            <include name="**/*.class"/>
            <exclude name="*.class"/>
        </fileset>
    </jar>
</target>

<target name="package-client" depends="compile">
    <mkdir dir="jar" />
     <delete file="jar/app-client.jar"/>
    <jar jarfile="jar/app-client.jar">
        <metainf dir="dd/client" includes="*.xml"/>
        <fileset dir="${build.dir}">
            <include name="**/*.class"/>
        </fileset>
         <fileset dir="dd/client">
            <include name="jndi.properties"/>
        </fileset>
     </jar>
</target>

<!--Crea un fichero ear que contiene los ejbjars y el webclient war. -->
<target name="assemble-app">
    <delete file="jar/GestorCuentasCMP.ear"/>
    <ear destfile="jar/GestorCuentasCMP.ear"
                                      appxml="dd/application.xml">
        <fileset dir="jar" includes="*.jar"/>
    </ear>
</target>

<!--Despliega el fichero EAR copiandolo en el directorio deploy JBoss-->
<target name="deploy" depends="assemble-app">
    <copy file="jar/GestorCuentasCMP.ear"
                                      todir="${jboss.server}/deploy"/>
</target>


<!--Ejecuta el cliente standalone -->
<target name="run-client">
    <java classname="ClienteCuentaCorriente" fork="yes">
```

```
            <classpath>
                <pathelement path="jar/app-client.jar"/>
                 <path refid="client.classpath"/>
            </classpath>
        </java>
    </target>

    <target name="clean">
       <delete dir="${build.dir}" />
       <mkdir dir="${build.dir}"  />
    </target>

    <target name="all" depends="compile,package-ejb,package-client,assemble-
                                                            app,deploy" />

</project>
```

7.5.6.4. Preparando los ficheros

En este apartado se mostrará el código de todos los ficheros implicados en el desarrollo de la aplicación de Gestión de Cuentas CMP. Sin embargo, antes de empezar a mostrar el código de dichos ficheros sería conveniente aclarar que todos aquellos ficheros relacionados con el bean de sesión sin estado GestorCuentas serán exactamente los mismos que los del ejemplo anterior. Esto se debe a que la unica diferencia entre el ejemplo anterior y este radica en que la gestion del bean de entidad se realizará utilizando CMP y no BMP.

Esta modificación del bean de entidad, como resulta evidente, no afecta en absoluto a la forma de implementar el bean de sesión utilizado en la aplicación del ejemplo anterior. En cualquier caso, para no perder la lógica seguida hasta este momento, y a pesar de que algunos ficheros ya son conocidos, se mostrará el contenido de todos los ficheros (tanto los del bean de sesión como los de estado) implicados en el desarrollo de la aplicación CMP.

Esto ayudará a que el lector siga teniendo un hilo conductor, al que ya está familiarizado, que le guie de forma clara por las particularidades de este tipo de beans CMP. Finalmente hay que decir, que el número de comentarios relacionados con los ficheros del bean de sesión serán los menos debido a que ya fueron tratados en el ejemplo anterior.

7.5.6.4.1. Interfaces Remoto y Local

En primer lugar, el interfaz remoto perteneciente al bean de sesión sin estado GestorCuentas será prácticamente el mismo que el del ejemplo anterior. La única pequeña diferencia es que en este ejemplo todos los ficheros que componen el código de los beans se agrupan dentro del paquete denominado bancaCMP.

Esta mínima diferencia se repetirá durante todo el ejemplo. Este cambio resulta imprescindible para evitar problemas de solapamiento con el ejemplo anterior durante el despliegue de los beans CMP dentro del servidor de aplicaciones. Hay que tener en cuenta que los nombres de los ficheros no se van a cambiar para que resulte lo más parecido al ejemplo anterior. Esta circunstacia, unida a que los mecanismos declarativos que utilizan los descritores de despliegue, en los que se requieren los nombres de los ficheros completamente cualificados (incluyendo el nombre de los paquetes a los que pertenecen los ficheros), hacen que sea necesario el agrupar los ficheros en un paquete distinto.

El interfaz remoto se guardará en un fichero GestorCuentas.java en el directorio /src del directorio de trabajo utilizado para esta aplicación y su código se muestra a continuación:

```
/*
 * Interfaz remoto para el Enterprise JavaBean: GestorCuentas
*/

package bancaCMP;

/**
 * Interfaz remota GestorCuentas.
 * Los clientes operan con esta interfaz
 */
public interface GestorCuentas extends javax.ejb.EJBObject {

    /**
     * Método que utiliza el bean entidad BMP CuentaCorriente para
     * realizar operaciones sobre una cuenta
     */
    public void operarCuenta(String titular) throws
                                                    java.rmi.RemoteException;

}
```

Por su parte el interfaz local, define cada uno de los métodos de la lógica de negocio que el bean finalmente va a publicar para que sean invocados por los clientes.

El interfaz local se guardará en un fichero CuentaCorrienteLocal.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz local para el Enterprise JavaBean: CuentaCorriente
*/

package bancaCMP;

public interface CuentaCorrienteLocal extends javax.ejb.EJBLocalObject
{
        /*
         * Método que decrementa el saldo en la cantidad pasada en argumento
        */
        public void reintegro(double cantidad);

        /*
         * Método que incrementa el saldo en la cantidad pasada en argumento
        */
        public void ingreso(double cantidad);

        /*
         * Método que devuelve el saldo de la cuenta
        */
        public double getBalance();

        /*
         * Método que modifica el saldo de una cuenta
        */
        public void setBalance(double balance);

        /*
         * Método que devuelve el nombre del titular de la cuenta
        */
        public String getNombreTitular();

        /*
         * Método que asigna el nombre del titular de la cuenta
        */
        public void setNombreTitular(String nombreTitular);

        /*
         * Método que devuelve el identificador de la cuenta
        */
        public String getCodigoCuenta();

        /*
         * Método que asigna el identificador de la cuenta
        */
        public void setCodigoCuenta(String codigoCuenta);
}
```

Las cuestiones mas importantes a destacar del código de este interfaz se enumeran a continuación:

1. Este interfaz local define, básicamente, el mismo conjunto de métodos que lo hacía el interfaz del ejemplo anterior y con las mismas caracteristicas ya explicadas. Este interfaz define métodos que se necesitan para llevar a cabo la funcionalidad requerida: un par de métodos de negocio que permiten ingresar y reintegrar dinero y los métodos típicos get( ) y set( ) sobre los atributos que gestionará el bean.

2. Sin embargo, hay que mencionar que para este ejemplo se ha tenido que incluir un método *set* que no estaba en el ejemplo anterior. Ese método se denomina setBalance(double balance). Este método no estaba incluido porque realmente no se utilizaba ninguna llamada en la aplicación del ejemplo anterior, aunque al ser un método *set* relacionado con un atributo de la clase del bean (o propiedad del componente) debería haber estado. Por ese motivo y por que, como se verá cuando se vea el código de la clase del bean de entidad CMP, ahora si que es necesario hacer la llamada a ese metodo *set*, se ha incluido aquí.

3. Finalmente, y como consecuencia de la modificación en el interfaz anterior, hay que aclarar lo siguiente. Las mayores modificaciones a la hora de migrar de un bean BMP a uno CMP están en la clase del bean y en los descriptores de despliegue. Por el contrario, los interfaces Remote y Home (o sus interfaces locales respectivos, como en este caso) y la clase *primary key* permanecen siempre invariantes. Por lo tanto, hay que tener claro que la recompilación en este caso del interfaz local es mas fruto de una deficiencia anterior que de una necesidad propia de la migración.

7.5.6.4.2. Interfaces Home y Local Home

Al igual que en los interfaces anteriores se necesitará en primer lugar del interfaz Home del bean de sesión que no presenta tampoco ningún cambio con respecto al ejemplo anterior.

Este interfaz Home se guardará en un fichero GestorCuentasHome.java en el directorio /src del directorio de trabajo correspondiente al ejemplo y su código se muestra a continuación:

```
/*
 * Interfaz Home para el Enterprise JavaBean: GestionCuentas
 */

package bancaCMP;

public interface GestorCuentasHome extends javax.ejb.EJBHome {

   /*
    * Crea la instancia del bean
    */
    GestorCuentas create() throws java.rmi.RemoteException,
                                                   javax.ejb.CreateException;
}
```

El interfaz local Home también se mantiene invariante (salvo la pequeña excepción del nombre del paquete ya reseñada) respecto al ejemplo anterior y se guardará en un fichero CuentaCorrienteLocalHome.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Interfaz local Home para el Enterprise JavaBean: CuentaCorriente
 */

package bancaCMP;

import java.util.Collection;

public interface CuentaCorrienteLocalHome extends javax.ejb.EJBLocalHome
{

         /*
          * Método de creación de la instancia del bean que recibe como
          * argumento el código de cuenta y el nombre del titular
          */
         public CuentaCorrienteLocal create(String codigoCuenta, String
                          titular) throws javax.ejb.CreateException;


         /*
          * Método de búsqueda de una instancia a partir de su primary key.
          * Este método devuelve un objeto CuentaCorrienteLocal
          */
         public CuentaCorrienteLocal findByPrimaryKey(CuentaCorrientePK
                                    clave) throws javax.ejb.FinderException;


         /*
          * Método de búsqueda de una instancia a partir del nombre del
```

```
         * titular. Este método devuelve un objeto Collection ya que puede
         * darse el caso de que un titular tenga varias cuentas.
         */
        public Collection findByNombreTitular(String titular) throws
                                                  javax.ejb.FinderException;


        /*
         * Método que permite conocer la cantidad de dinero total que hay en
         * las cuentas
         */
        public double getTotalEnCuentas();

}
```

Las cuestiones más importantes a destacar del código de este interfaz ya se enumeraron en el ejemplo anterior.

7.5.6.4.3. La clase primary key

La clase *primary key* se guardará en un fichero denominado CuentaCorrientePK.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Primary key para el Enterprise JavaBean: CuentaCorriente
 */

package bancaCMP;

public class CuentaCorrientePK implements java.io.Serializable
{
        /*
         * Atributo identificador y unico para la clase Primary key
         */
        public String codigoCuenta;


        /*
         * Constructor de la clase Primary key
         */
        public CuentaCorrientePK(String codigoCuenta)
        {
                this.codigoCuenta = codigoCuenta;
        }

        /*
         * Constructor sin argumentos de la clase Primary key
         */
        public CuentaCorrientePK()
        {
        }

        /*
         * Método sobrescrito de la clase Object
         */
        public String toString()
```

```
        {
                return codigoCuenta;
        }

        /*
         * Método sobrescrito de la clase Object
         */
        public int hashCode()
        {
                return codigoCuenta.hashCode();
        }

        /*
         * Método sobrescrito de la clase Object
         */
        public boolean equals(Object acc)
        {
                return ((acc instanceof CuentaCorrientePK) &&
                        ((CuentaCorrientePK)acc).getCodigoCuenta().
                          equals(this.getCodigoCuenta())
                       );
        }

}
```

Si se compara este fichero con el del ejemplo anterior se aprecia como solo hay una pequeña y sutil modificación. Esta diferencia está en el modificador del atributo códigoCuenta que actúa como clave primaria del bean CuantaCorriente. En el ejemplo anterior ese atributo era privado, mientras que en CMP es necesario que el atributo sea público para que el contenedor EJB pueda determinar quien compone la clave primaria a la hora de crear la tabla.

7.5.6.4.4. Las clases de los beans de sesión y entidad

En primer lugar, se mostrará la clase del bean de sesión encargada de controlar toda la gestión de las cuentas corrientes que maneja la aplicación del ejemplo y que permanece invariante respecto al ejemplo anterior.

Esta clase del bean GestorCuentas se guardará en un fichero denominado GestorCuentasBean.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Implementacion de la clase del Enterprise JavaBean: GestorCuentas
*/

package bancaCMP;

import java.util.*;
import javax.naming.*;
```

```
import bancaCMP.CuentaCorrientePK;
import bancaCMP.CuentaCorrienteLocalHome;
import bancaCMP.CuentaCorrienteLocal;

/*
 * GestorCuentas es un bean de sesion sin estado responsable de la
 * creacion y manejo de las cuentas de los clientes
 */
public class GestorCuentasBean implements javax.ejb.SessionBean {

        private javax.ejb.SessionContext ctx;

        /*
        * Utiliza el EJB de entidad CuentaCorriente
        */
        public void operarCuenta(String titular)throws
                                                java.rmi.RemoteException{

          try{
                /*
                 * Busqueda del objeto Home en JNDI
                 */

                // Se accede al contexto por defecto
                Context ctx = new InitialContext();
                // Se busca el objeto publicado con JNDI
                Object obj =
                        ctx.lookup("java:comp/env/ejb/RefCuentaCorriente");
                // Se hace un cast normal ya que son interfaces locales
                CuentaCorrienteLocalHome home =
                                        (CuentaCorrienteLocalHome)obj;

                // Se consulta el total acumulado en todas las cuentas
                System.out.println("Total acumulado en cuentas: "
                                        +home.getTotalEnCuentas());

                /*
                 * Creacion de nueva cuenta usuario
                 */

                CuentaCorrienteLocal cuentaPrueba = null;
                // Se crea una nueva cuenta
                home.create("2005a31d1973",titular);
                // Se consulta la cuenta buscandola por el nombre del
                // titular
                Iterator it = home.findByNombreTitular(titular).iterator();
                if (it.hasNext()) {
                        cuentaPrueba = (CuentaCorrienteLocal)it.next();
                } else {
                        System.out.println("No existe la cuenta de "+
                                                                titular);
                }

                /*
                 * Operaciones basicas con la nueva cuenta
                 */

                // Se pide el saldo de esa cuenta
                System.out.println("Saldo actual: "
                                        +cuentaPrueba.getBalance());
                // Se ingresa de esa cuenta
                cuentaPrueba.ingreso(200);
                System.out.println("Saldo actual: "
                                        +cuentaPrueba.getBalance());
                // Se reintegra de esa cuenta
                cuentaPrueba.reintegro(150);
                System.out.println("Saldo actual: "
                                        +cuentaPrueba.getBalance());
```

```
            // Se pide el acumulado de las cuentas con lo ultimos
            // movimientos
            System.out.println("Total acumulado en cuentas: "
                                  +home.getTotalEnCuentas());

            /*
             * Recuperacion del objeto a partir de su PK
             */

            // Se pide la PK para realizar una busqueda con ella
            CuentaCorrientePK pk = (CuentaCorrientePK)
                                  cuentaPrueba.getPrimaryKey();
            // Se destruye el objeto para crearlo de nuevo con la PK
            cuentaPrueba = null;
            // Se realiza la busqueda y se devuelve el saldo
            cuentaPrueba = home.findByPrimaryKey(pk);
            System.out.println ("Encontrada con ID "+pk+
                    " cuenta con saldo " + cuentaPrueba.getBalance());

            /*
             * Se elimina el cliente de prueba para dejar el sistema
             * igual
             */

            // Finalmente se destruye la entidad permanentemente de la
            // BBDD
            if (cuentaPrueba != null){
                cuentaPrueba.remove();
            }

            /*
             * Se produce el desfalco
             */
            // Se consulta la cuenta buscandola por el nombre del
            // titular
            CuentaCorrienteLocal cuentaGerente = null;
            it = home.findByNombreTitular("Gerente").iterator();
            if (it.hasNext()) {
                    cuentaGerente = (CuentaCorrienteLocal)it.next();
                    cuentaGerente.ingreso(1000);
            } else {
                    System.out.println("Imposible que un banco donde "+
                    "realizan este tipo de practicas fraudulentas "+
                    "no tenga creada aun la cuenta del Gerente");
            }


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



//------------------------------------------------------
// Metodos requeridos e invocados por el contenedor,
// nunca desde codigo cliente.
//------------------------------------------------------

public void ejbCreate() throws javax.ejb.CreateException {
            System.out.println("GestorUsuariosBean.ejbCreate()
                                                invocado.");
     }

     public void ejbRemove() {
            System.out.println("GestorUsuariosBean.ejbRemove()
                                                invocado.");
     }
```

```
        public void ejbActivate() {
                System.out.println("GestorUsuariosBean.ejbActivate()
                                                        invocado.");
        }

        public void ejbPassivate() {
                System.out.println("GestorUsuariosBean.ejbPassivate()
                                                        invocado.");
        }

        public void setSessionContext(javax.ejb.SessionContext ctx) {
                System.out.println("GestorUsuariosBean.setSessionContext()
                                                        invocado.");
                this.ctx = ctx;
        }

}
```

Las cuestiones mas importantes a destacar del código anterior ya se expusierón en el ejemplo anterior.

Por otro lado, ahora se verá la clase del bean de entidad CMP que se necesita desarrollar para la aplicación propuesta. La clase del bean se guardará en un fichero denominado CuentaCorrienteBean.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Clase del bean para el Enterprise JavaBean: CuentaCorrienteCMP
*/

package bancaCMP;

import javax.naming.*;
import java.util.Collection;

public abstract class CuentaCorrienteBean implements javax.ejb.EntityBean
{

        /************************************
         * Atributos
        ************************************/

        /*
         * Los únicos atributos que se definen en esta clase
         * son los atributos no persistente del bean. El resto
         * de ellos los implementará el contenedor en la
         * subclase tomándolos de el descriptor de despliegue
        */
        protected javax.ejb.EntityContext ctx;



        /************************************
         * Metodos de logica de negocio
        ************************************/
```

```
        /*
         * Constructor por defecto del bean
        */
        public CuentaCorrienteBean()
        {
                System.out.println("Nuevo EJB Entidad CuentaCorrienteBean
                                        construido por contenedor");
        }

        /*
         * Métodos get/set definidos como abstractos
        */
        public abstract double getBalance();
        public abstract void setBalance(double balance);
        public abstract String getNombreTitular();
        public abstract void setNombreTitular(String nombreTitular);
        public abstract String getCodigoCuenta();
        public abstract void setCodigoCuenta(String codigoCuenta);

        /*
         * Métodos select definido como abstracto para ser implementado
         * a través de EJB-QL y ser llamado como implementación de los
         * métodos del interfaz Home
        */
        public abstract double ejbSelectTotalCuentas() throws
                                                javax.ejb.FinderException;


        /*
         * Métodos de la lógica de negocio de la aplicación
        */
        public void setEntityContext (javax.ejb.EntityContext ctx)
        {
                System.out.println("EntityContext establecido");
                this.ctx=ctx;
        }

        /*
         * Método que modifica el contexto del bean de entidad
        */
        public void unsetEntityContext ()
        {
                System.out.println("EntityContext eliminado");
                this.ctx=null;
        }

        /*
         * Método de reintegro de la cuenta corriente
        */
        public void reintegro(double cantidad)
        {
                double saldo = getBalance();
                System.out.println("Reintegro de "+cantidad+
                                                " euros solicitado");
                if(saldo > cantidad)
                {
                        saldo -= cantidad;
                        setBalance(saldo);
                        System.out.println("Reintegro realizado con
                                                                exito");
                }
        }

        /*
         * Método de ingreso
        */
        public void ingreso(double cantidad)
        {
```

```
        double saldo = getBalance();
        // Uhmmm!!! que raro...
        if (!(getNombreTitular().toLowerCase()).equals("gerente"))
                System.out.println("Ingreso de "+cantidad+
                                        " euros solicitado");
        saldo += cantidad;
        setBalance(saldo);
}



/*******************************************************
 * Metodos de ciclo de vida gestionados por el contenedor
*******************************************************/


/*
 * Método llamado por el contenedor de creacion una instancia del
 * bean
*/
public CuentaCorrientePK ejbCreate(String codigoCuenta,
                                   String nombreTitular)
                                   throws javax.ejb.CreateException{

        System.out.println("ejbCreate");
        setNombreTitular(nombreTitular);
        setCodigoCuenta(codigoCuenta);
        setBalance(0);
        return new CuentaCorrientePK(codigoCuenta);
}


/*
 * Método definido en el interfaz Home
*/
public double ejbHomeGetTotalEnCuentas(){
        double saldoTotal;
        try{
                saldoTotal=ejbSelectTotalCuentas();
        }catch(Exception e){
                saldoTotal=0;
        }
        return saldoTotal;
}


/*
* Método llamado por el contenedor de eliminacion de una instancia
* del bean Puede estar a vacio ya que será re-escrito
* por la subclase

*/
public void ejbRemove()
{
        System.out.println("ejbRemove");
}


/*
 * Método de carga de la informacion de una instancia guardada en la
 * base de datos. Puede estar a vacio ya que será re-escrito por
 * la subclase
*/
public void ejbLoad()
{
        System.out.println("ejbLoad");
}
```

```
        /*
         * Método de almacenamiento y actualizacion de los datos de un bean
         * en la base de datos. Puede estar a vacio ya que será re-escrito
         * por la subclase
         *
        */
        public void ejbStore()
        {
                System.out.println("ejbStore");
        }

        /*
         * Resto de métodos de gestion de ciclo de vida que tampoco
         * necesitan ser implementados y que solo dejan su huella en el log
         * cuando son invocados
        */
        public void ejbPostCreate(String codigoCuenta, String nombreTitular)
                {System.out.println("ejbPostCreate");}
        public void ejbActivate()
                {System.out.println("ejbActivate");}
        public void ejbPassivate()
                {System.out.println("ejbPassivate");}

}
```

Las cuestiones mas importantes a destacar del código de esta clase se enumeran a continuación:

1. La implementación de cualquier bean de entidad con persistencia gestionada por el contenedor EJB es bastante menos extensa que la implementación vista para un bean BMP. Esto es asi ya que, entre otras cosas, la clase del bean ya no aporta todo el código JDBC necesario para gestionar su persistencia. Además, en la clase del bean CMP se declaran una serie de métodos como abstractos, para que sean implementados mediante una subclase, que hacen que la clase del bean ahora sea una clase abstracta.

2. En los beans CMP la clase del bean ya no declara los atributos de estado gestionados por el bean. Ahora ese papel recae en los descriptores de despliegue que son los que deben proporcionar los atributos que deben ser persistentes en la base de datos. Finalmente, será una subclase de esta clase del bean, creada por el contenedor EJB, la que se encargará de declarar esos atributos para que una instancia de esa clase cargue y almacene la información de la base de datos en dichos atributos.

3. En cuanto a los métodos propios de la lógica de negocio de la aplicación, la clase de un bean CMP tiene también un comportamiento distinto al de los beans BMP. Para esta aplicación, en la clase del bean se pueden definir tres tipos distintos de métodos de lógica de negocio. Por un lado, estarán todos aquellos métodos definidos en el interfaz CuentaCorrienteLocal y relacionados con las propiedades del bean, como son todos los métodos *get* y *set*. Estos métodos se definirán en la clase del bean CMP como abstractos, para que todos ellos sean realmente implementados por la subclase que generará el contenedor EJB. Por otro lado, estarán todos aquellos métodos de operaciones como el ingreso o el reintegro de dinero en una cuenta corriente, que estarán implementados en la propia clase utilizando los métodos *get* y *set* definidos anteriormente.

4. Como ya se comentó en el caso de los beans BMP y como de hecho aún sucede en el caso de un bean CMP, la mayoría de los métodos de la propia lógica de negocio de la aplicación no contienen código JDBC ya que atacan directamente a los atributos del bean en memoria. Uno de esos metodos era el metodo denominado ingreso( ) que modificaba el atributo saldo directamente en memoria. Sin embargo, al no disponer en el caso de los beans CMP de atributos declarados en la propia clase del bean resulta abosolutamente imprescindible la implementación del método setBalance( ) al que ya se ha hecho referencia anteriormente.

5. El último tipo de métodos de la logica de negocio será el que agrupa a todos aquellos métodos pertenecientes a la lógica de negocio aunque estén definidos en el interfaz Home, como puede ser el caso del método que devuelve el saldo general de las cuentas. Este método denominado getTotalEnCuentas( ) tiene una llamada a un método ejbSelect( ) dentro de su código. Ese método ejbSelect( ) se definirá como abstracto en la clase del bean y se resolverá su funcionalidad a través de la ejecución de una sentencia SQL definida mediante EJB-QL en el descriptor de despliegue. Los métodos ejbSelect( ) son métodos de ayuda que ejecutan sentencias SQL que internamente necesita un bean pero que nunca son accesibles desde un cliente. Para mas información acerca de las

particularidades de este método se recomienda leer la siguiente nota de seguimiento que trata este tema.

6. Finalmente en la clase del bean de entidad se deben encontrar implementados todos los métodos definidos en su interfaz Home. Es decir, en la clase del bean se deben encontrar los métodos requeridos por la especificación EJB y que son métodos que el contenedor llama para realizar la gestión del ciclo de vida como los métodos ejbLoad( ) o ejbStore( ). El método ejbLoad( ) es llamado por el contenedor EJB para cargar los datos de la base de datos en la instancia del bean. No debe llevar código asociado de lectura de la base de datos ya que esa es tarea del contenedor. Sin embargo, al no ser un método sobrescrito por la subclase si que puede contener el código necesario después de leerse los datos de la base de datos. Por su parte, el método ejbStore( ) es llamado por el contenedor para actualizar la base de datos y sincronizarla con los valores de los atributos en memoria. Al igual que el método anterior no necesita código de actualización a pesar de que puede aportar código necesario para preparar los campos que se van a almacenar.

7. Dentro de los métodos definidos en el interfaz Home se encuentran el constructor, los métodos de lógica de negocio así como los métodos *finder*. En cuanto al constructor hay que recordar que cuando un cliente llama al método create( ) de un objeto Home, este a su vez llama al método ejbCreate( ) de la instancia del bean. El método ejbCreate( ) será el responsable de la creación de nuevos datos en la base de datos y de inicializar el bean. En este caso, el constructor debe llamar a los métodos *get* y *set* para inicializar los parámetros de la sblcase generada por el contenedor. Posteriormente será el contenedor a través de esa subclase quien cree los datos en la base de datos. Además, si una aplicación no está interesada en la creación de datos en la base de datos (ya que los inserta atacándola directamente, por ejemplo) no sería necesaria la implementación del método ejbCreate( ).

8. Cuando un cliente llama a un método remove( ) este a su vez llama al método ejbRemove( ) para destruir la información de la base de datos. Hay que tener claro que una llamada a remove( ) no elimina objetos Java (es decir, instancias

de un bean) ya que estos pueden ser vueltos a poner en el *pool* y ser de nuevo reutilizados. No debe llevar código asociado al borrado de datos de la base de datos ya que esa es tarea del contenedor EJB que la realizará justo después de que se invoque al método ejbRemove( ). Este método puede contener el código necesario que se ejecutaría justo antes de eliminar los datos de la base de datos.

9. A pesar de que los métodos *finder* se encuentran definidos en el interfaz Home, en el caso de los beans CMP estos métodos nunca se implementan en la clase del bean. El contenedor EJB se encargará de todo lo relativo a las búsquedas de datos en la base de datos. Para conseguir esto el contenedor EJB se basa en las sentencias SQL definidas mediante EJB-QL en el descriptor de despliegue.

10. Por último, destacar que en BMP los métodos que tienen código JDBC son todos los métodos *call-back* que llama el contenedor EJB durante la gestión del ciclo de vida de bean como pueden ser ejbCreate( ), ejbLoad( ), ejbStore( ) o ejbRemove( ). Por su parte en CMP, en la clase del bean se añaden esos métodos *call-back* pero, bien implementados a vacío, o con el código que se necesite para la preparación o recepción de los datos que se gestionarán en la subclase.

Nota de seguimiento: y otra de anatomia, por qué los hombres tienen pezones?

Algunas veces se necesitan métodos en un bean de entidad que no sean específicos de una determinada instancia de un bean. Es decir, métodos que no sean propios de una determinada tupla de la base de datos. Por ejemplo, muchas veces es necesario saber el número de filas o tuplas que hay en una tabla de la base de datos. Ese tipo de métodos pueden incluirse dentro de un bean como métodos ejbHome que realizan ese tipo de operaciones.

Es el caso particular, en el ejemplo propuesto, del método getTotalEnCuentas( ). Este método es un método de lógica de negocio especial porque es llamado desde un bean del *pool* de instancias antes de que dicho bean haya sido asociado con algún conjunto de datos específico. Los clientes llamarán a ese metodo bien sea desde un interfaz home o desde un interfaz local home como es el caso.

La implementación de dicho método getTotalEnCuentas( ) utilizando beans BMP ya fué mostrada anteriormente, y se basaba simplemente en la utilización de código JDBC desde la

clase del bean. La consulta SQL consultaba todas las filas almacenadas en la tabla correspondiente para hallar el dinero total acumulado.

Ahora bien, ese mismo método para el caso de CMP abre dos alternativas que pueden solucionar su problema de implementación. La forma mas rápida y fácil de conseguir su implemantación sería usando el mismo código JDBC que se uso en BMP. Sin embargo, si se pretende, como se supone que es el objetivo fundamental que persigue CMP, evitar en la medida de lo posible la inclusión de código JDBC dentro de la clase del bean, esta no parece la opción más elegante.

En su lugar, se podría utilizar una segunda alternativa que llevase todo ese código SQL a los ficheros descriptores de despliegue a través de EJB-QL. Esta forma, mucho mas limpia y acorde con la filosofia CMP, usa, tal y como se puede apreciar en el código de la clase del bean que se presenta, una llamada a un método ejbSelect( ) dentro del código del método getTotalEnCuentas( ). Ese método ejbSelect( ) se definirá como abstracto en la clase del bean y resolverá su funcionalidad a través de la ejecución de una sentencia SQL definida mediante EJB-QL en el descriptor de despliegue.

Cual de las dos soluciones es mejor?. Definitivamente hay preguntas que resultan más difíciles de responder que otras. O hay alguien que ya sabe por qué los hombres tienen pezones?

7.5.6.4.5. Descriptores de despliegue estándar

A continuación se describirán los elementos fundamentales que deben incluir los ficheros descriptores de despliegue estándar que se utilizarán en la aplicación de ejemplo propuesta. Esta aplicación, como siempre, utilizará dos ficheros descriptores de despliegue estándar. En primer lugar, el correspondiente a los componentes EJB (o parte servidora) dentro del fichero ejb-jar.xml, y en segundo lugar, el fichero opcional de la tecnología EJB llamado application-client.xml, que se usará para el cliente *stand-alone*.

Sin embargo, antes de continuar hay que recordar que lo que se propone en el ejemplo actual, básicamente, consiste en migrar el bean de entidad del tipo BMP al CMP. Esta modificación no debe incluir ningún tipo de modificación en el código del cliente ya que la migración solo afecta a la forma que tiene el bean de llevar a cabo su propia persistencia. Por este motivo se puede deducir que todos los ficheros descriptores de despliegue que pertenezcan a la parte cliente no sufriran ningun tipo de variación con respecto al ejemplo anterior.

Por su parte, todos los ficheros descriptores de despliegue de la parte servidora si que se verán modificados ya que son todos estos ficheros los que, a través de una forma

declarativa, darán soporte a toda la información de persistencia que requiere ahora el bean para comportarse como un componente CMP.

Por lo tanto, y como siempre, los ficheros descriptores de despliegue estándar de la parte servidora definirán los requisitos que necesita el bean, como pueden ser la declaración de recursos que va a tener que utilizar, como una conexión a la base de datos. Sin embargo, en el caso de los beans CMP, aparte de la declaración de los recursos y de los propios parámetros de identificación del bean de entidad, es necesario incluir en este fichero, por ejemplo, los atributos cuya persistencia debe gestionar el contenedor EJB o todo el código EJB-QL necesario para la implementación de los métodos *find*.

En primer lugar, el fichero descriptor de despliegue de los componentes EJB se guardará en el fichero ejb-jar.xml en el directorio /dd/ejb del directorio de trabajo y su código se muestra a continuación:

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

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN"
                         "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>

    <session>

      <ejb-name>GestorCuentas</ejb-name>
      <home>bancaCMP.GestorCuentasHome</home>
      <remote>bancaCMP.GestorCuentas</remote>
      <ejb-class>bancaCMP.GestorCuentasBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>

      <ejb-local-ref>
         <ejb-ref-name>ejb/RefCuentaCorriente</ejb-ref-name>
         <ejb-ref-type>Entity</ejb-ref-type>
         <local-home>bancaCMP.CuentaCorrienteLocalHome</local-home>
         <local>bancaCMP.CuentaCorrienteLocal</local>
      </ejb-local-ref>

    </session>

    <entity>

      <ejb-name>CuentaCorriente</ejb-name>
      <local-home>bancaCMP.CuentaCorrienteLocalHome</local-home>
      <local>bancaCMP.CuentaCorrienteLocal</local>
      <ejb-class>bancaCMP.CuentaCorrienteBean</ejb-class>
      <persistence-type>Container</persistence-type>
```

```
      <prim-key-class>bancaCMP.CuentaCorrientePK</prim-key-class>
      <cmp-version>2.x</cmp-version>
      <reentrant>false</reentrant>

      <abstract-schema-name>cuentacorriente</abstract-schema-name>
      <cmp-field>
        <field-name>codigoCuenta</field-name>
      </cmp-field>
      <cmp-field>
                <field-name>nombreTitular</field-name>
      </cmp-field>
      <cmp-field>
                <field-name>balance</field-name>
      </cmp-field>

      <query>
            <query-method>
               <method-name>findByNombreTitular</method-name>
               <method-params>
                          <method-param>java.lang.String</method-param>
                </method-params>
            </query-method>
            <ejb-ql><![CDATA[
               select OBJECT(p) from cuentacorriente as p where
                                                   p.nombreTitular = ?1
            ]]></ejb-ql>
      </query>

      <query>
            <query-method>
               <method-name>ejbSelectTotalCuentas</method-name>
               <method-params></method-params>
            </query-method>
            <ejb-ql><![CDATA[
               select sum(p.balance) from cuentacorriente as p
            ]]></ejb-ql>
      </query>

      <resource-ref>
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>

    </entity>

  </enterprise-beans>

</ejb-jar>
```

Como se puede apreciar en el código del fichero descriptor de despliegue que se acaba de mostrar, las declaraciones relativas al bean de sesión llamado GestorCuentas realizadas dentro del elemento <session> son exactamente iguales a las del ejemplo anterior.

Por otro lado, con respecto al código del bean de entidad CuentaCorriente las cuestiones mas importantes a destacar del código de esta clase se enumeran a continuación:

1. En este fichero descriptor de despliegue hay que destacar, en primer lugar, que en el elemento <persistence-type> ahora se debe establecer el valor de Container y no Bean.

2. Este descriptor empieza prácticamente igual que el descriptor del bean BMP hasta que empieza a definir los atributos gestionados por el contenedor EJB que deben coincidir con los métodos abstractos *get* y *set* definidos en la clase del bean.

3. La mayor cantidad de código que presenta este descriptor de despliegue corresponde normalmente con el código de las sentencias SQL correspondiente a lso método *finder* del bean. Estas sentencias SQL, definidas a través de EJB-QL, le informan al contenedor EJB de cómo se debe implementar la funcionalidad de esos métodos. Cuando el contenedor EJB interpreta el código EJB-QL generá automáticamente el codigo de acceso JDBC a la base de datos necesario para llevar a cabo la funcionalidad requerida en el método *finder*.

4. Un último detalle importante a destacar en el código que se acaba de mostrar es la inclusión del siguiente elemento: <![CDATA[XXX]]>. Este elemento es importante ya que instruye al *parser* XML del contenedor EJB para que ignore todo el texto que se encuentra encerrado entre los corchetes. Esto es muy importante ya que evita que el *parser* del contenedor puede interpretar un símbolo menor que (“<”) incluido, por ejemplo, dentro de una sentencia SQL, como un símbolo de apertura de una nueva etiqueta que rompería la estructura lógica del fichero.

Nota de seguimiento: la razón contra la moral

El siguiente test pone a prueba la razón contra la moral.

Dilema para la razón:

Hay que implementar un bean de entidad y se debe elegir su *primary key* pudiendo:
1. Usar una clase *primary key* y en el descriptor de despliegue usar el elemento:
<prim-key-class>bancaCMP.CuentaCorrientePK</prim-key-class>
2. Usar uno de los propios campos gestionados por el contenedor EJB como clave primaria y en el descriptor de despliegue usar los siguientes elementos:
<prim-key-class>java.lang.String</prim-key-class>
<primkey-field>codigoCuenta</primkey-field>

Dilema para la moral:

Regalan 1 millon de euros y hay que elegir entre:
1. Comprar para el propio uso y disfrute el mejor coche deportivo que valga ese dinero (que los hay), sin posibilidad de venderlo posteriormente y con el mantenimiento cubierto.
2. Donarlo a la entidad benéfica que se quiera.

Resultados del test:

Opinión de la razón:

En cuanto al dilema para la razón, la propia razón opina que en la medida de lo posible se deben evitar usar los propios campos como claves primarias. Esto es debido a que con la utilización de una clase particular que funcione como un *wrapper* se independiza al código del bean ante cambios en los identificadores únicos que requiere el dispositivo de almacenamiento. De esta forma un posible cambio en el modelo de datos de la aplicación sería más fácil de solucionar con una clase *primary key*.

En cuanto al dilema para la moral, la razón opina que entiende que haya dudas debido a la condición humana y a la escala de valores de cada uno. Cree además la razón, que ante la misma pregunta pero planteada con 10 veces menos de dinero la respuesta es probable que fuese completamente distinta. Sin embargo, planteada la pregunta tal y como está y tratándose de un caso claramente llevado al extremo, cree la razón que lo más razonable es optar por la donación.

Opinión de la moral:

En cuanto al dilema para la moral, la propia moral opina que debe ser muy complicado sentarse a conducir un coche de 1 millon de euros, sabiendo que ese dinero se podía haber invertido en algo bastante más razonable.

En cuanto al dilema para la razón, la moral opina que no entiende nada y que no es cuestión suya.

En segundo lugar, se presenta el fichero descriptor de despliegue opcional para las aplicaciones *stand-alone*, que permanece invariante respecto al ejemplo anterior y también se incluirá en este ejemplo.

Este descriptor se guardará en el fichero application-client.xml en el directorio /dd/client del directorio de trabajo y su código se muestra a continuación:

```
<!DOCTYPE application-client PUBLIC
          "-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//EN"
          "http://java.sun.com/dtd/application-client_1_3.dtd">


<application-client>

        <display-name>GestorCuentas</display-name>

        <ejb-ref>
              <description></description>
              <ejb-ref-name>refGestorCuentasCMP</ejb-ref-name>
              <ejb-ref-type>Session</ejb-ref-type>
              <home>bancaCMP.GestorCuentasHome</home>
              <remote>bancaCMP.GestorCuentas</remote>
        </ejb-ref>

</application-client>
```

7.5.6.4.6. Ficheros específicos del servidor JBoss

Como ya se ha comentado cualquier aplicación que se quiera desplegar en el servidor de aplicaciones JBoss, requiere que se especifiquen ciertos parámetros en los ficheros descriptores de despliegue específicos como jboss.xml. Como es lógico, el caso de los beans CMP no es una excepción.

Para la aplicación de ejemplo planteada Gestor de Cuentas CMP se utilizarán en esta ocasión tres ficheros descriptores de despliegue. En primer lugar, el correspondiente a los componentes EJB en el fichero jboss.xml, en segundo lugar, se requiere la utilización de un fichero denominado jbosscmp-jdbc.xml que permite que todos los beans de entidad sean mapeados al soporte de persistencia correspondiente, y finalmente, el fichero opcional de la tecnología EJB llamado jboss-client.xml, que se usará para el cliente *stand-alone*.

A continuación se muestra el código correspondiente al fichero descriptor de despliegue EJB para JBoss, que presenta unos ligeros cambios respecto al mismo fichero del ejemplo anterior y que se guardará en el fichero jboss.xml en el directorio /dd/ejb del directorio de trabajo:

```
<!DOCTYPE jboss PUBLIC
          "-//JBoss//DTD JBOSS 4.0//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">
<jboss>

    <enterprise-beans>

         <session>
             <ejb-name>GestorCuentas</ejb-name>
             <jndi-name>ejb/bancaCMP/GestorCuentasCMP</jndi-name>
             <ejb-local-ref>
                 <ejb-ref-name>ejb/RefCuentaCorriente</ejb-ref-name>
                 <local-jndi-name>java:/CuentaCorrienteCMP</local-jndi-name>
             </ejb-local-ref>
         </session>

        <entity>
            <ejb-name>CuentaCorriente</ejb-name>
             <local-jndi-name>java:/CuentaCorrienteCMP</local-jndi-name>
             <resource-ref>
                  <res-ref-name>jdbc/DefaultDS</res-ref-name>
                  <jndi-name>java:/DefaultDS</jndi-name>
             </resource-ref>
        </entity>

    </enterprise-beans>

</jboss>
```

A diferencia del ejemplo anterior y para evitar que el bean CuentaCorriente fuese mapeado con igual nombre que el mismo bean del ejemplo anterior, produciendo asi inconsistencias en el despliegue, en esta oportunidad se ha utilizado para el mapeo de la referencia local el elemento <ejb-local-ref>. Este elemento ya se ha explicado anteriormente y lleva a cabo el mapeo del bean en cuestión utilizando los mismos mecanismos de siempre. Es decir, imponiendo la igualdad entre los elementos <ejb-ref-name> de ambos ficheros, ejb-jar.xml y jboss.xml.

También hay que destacar que el bean GestorCuenta, al tener una referencia local al bean de entidad CuentaCorriente (definida con el elemento <ejb-local-ref>), impone que los elementos <local-jndi-name>, tanto el de dicha referencia como el del mapeo del propio bean de entidad tengan un valor que empiece por el prefijo *java:*. Esto es asi debido a que son referencias locales.

Además del fichero específico de JBoss visto anteriormente, en el caso de los beans CMP es necesaria la utilización de un fichero XML que permita el mapeo de los

atributos persistentes por el contenedor EJB a los campos particulares del dispositivo de almacenamiento.

Bien es cierto que toda esta configuración del mapping de JBoss se puede hacer, tal y como ya se explicó anteriormente, bien mediante la creación de un nuevo fichero descriptor de despliegue llamado jbosscmp-jdbc.xml, o bien, modificando el fichero de configuración del servidor del aplicaciones JBoss llamado standardjbosscmp-jdbc.xml. Sin embargo, para la aplicación del ejemplo se ha considerado mas didactica la creación del fichero jbosscmp-jdbc.xml que es la que se va a mostrar a continuación.

El código correspondiente al fichero descriptor de despliegue EJB para JBoss que permite el mapeo al dispositivo de almacenamiento se guardará en el fichero jbosscmp-jdbc.xml en el directorio /dd/ejb del directorio de trabajo:

```
<!DOCTYPE jbosscmp-jdbc PUBLIC
          "-//JBoss//DTD JBOSSCMP-JDBC 4.0//EN"
          "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_4_0.dtd">

<jbosscmp-jdbc>

    <defaults>
        <datasource>java:/DefaultDS</datasource>
        <datasource-mapping>mySQL</datasource-mapping>
        <create-table>true</create-table>
    </defaults>

    <enterprise-beans>

        <entity>

            <ejb-name>CuentaCorriente</ejb-name>
            <table-name>cuentaCMP</table-name>

            <cmp-field>
                <field-name>codigoCuenta</field-name>
                <column-name>id</column-name>
                <jdbc-type>VARCHAR</jdbc-type>
                <sql-type>VARCHAR(15)</sql-type>
            </cmp-field>
            <cmp-field>
                <field-name>nombreTitular</field-name>
                <column-name>titular</column-name>
            </cmp-field>
            <cmp-field>
                <field-name>balance</field-name>
                <column-name>saldo</column-name>
            </cmp-field>

        </entity>

    </enterprise-beans>
```

```
</jbosscmp-jdbc>
```

Las cuestiones mas importantes a destacar del código de este descriptor de despliegue se enumeran a continuación:

1. En el elemento <defaults>, en primer lugar, se define cual es la base de datos asociada a la persistencia CMP, utilizando la fuente de datos que se publicó como recurso a través de JNDI. Dentro de este elemento también se define a través del elemento <datasource-mapping> cual va a ser el tipo de mapping que se va a utilizar. Finalmente, se impone la creación de una tabla con el nombre declarado en el elemento <table-name> como soporte a las propiedades que maneja el bean.

2. El elemento <ejb-name> tiene que coincidir con el resto de elementos con el mismo nombre definidos en el resto de ficheros descriptores de despliegue de la parte servidora.

3. El elemento <cmp-field> es el que permite realizar el mapeo entre el nombre lógico declarado en el fichero ejb-jar.xml con el nombre del campo real de la tabla creada en el dispositivo de almacenamiento. El <field-name> tiene que coincidir con el nombre declarado en el ficher ejb-jar.xml y el elemento <column-name> determina el nombre real en la tabla. Además en este fichero se pueden declarar los tipos que se quiera que tengan esos campos en la base de datos a través de los elementos <jdbc-type> y <sql-type> tal y como se muestra.

Finalmente, la aplicación también requiere del fichero descriptor de despliegue de las aplicaciones *stand-alone* para el servidor JBoss que es otro de los que permanece igual al ejemplo anterior.

A continuación se enseña el código correspondiente, que se guardará en el fichero jboss-client.xml en el directorio /dd/client del directorio de trabajo.

```
<!--<!DOCTYPE jboss-client PUBLIC
       "-//JBoss//DTD Application Client 4.0//EN"
       "http://www.jboss.org/j2ee/dtd/jboss-client_4_0.dtd">-->

<jboss-client>

   <ejb-ref>
      <ejb-ref-name>refGestorCuentasCMP</ejb-ref-name>
      <jndi-name>ejb/bancaCMP/GestorCuentasCMP</jndi-name>
   </ejb-ref>

</jboss-client>
```

7.5.6.4.7. Cliente stand-alone

Es hora de mostrar el código correspondiente a la parte cliente de la aplicación denominada Gestor de Cuentas CMP. Hay que recordar que en este caso la aplicación utiliza beans CMP y que sigue utilizando tanto los interfaces locales como los interfaces remotos. El interfaz remoto lo utiliza el bean de sesión GestorCuenta que no ha cambiado en este ejemplo, mientras que el interfaz local lo utiliza el bean de entidad CuentaCorriente. Es este último bean el que ha pasado de ser de tipo BMP a CMP.

Por lo tanto, el cliente de la aplicación Gestor de Cuentas CMP lo que hace es acceder de forma remota al bean de sesión GestorCuenta, que no ha sufrido ningún cambio del ejemplo anterior a este. Por este motivo en este ejemplo el código correspondiente al cliente tampoco no ha sufrido ninguna modificación.

La clase principal del cliente se guardará en un fichero ClienteCuentaCorriente.java en el directorio /src del directorio de trabajo y su código se muestra a continuación:

```
/*
 * Cliente para el Enterprise JavaBean: CuentaCorriente
 */

import banca.GestorCuentasHome;
import banca.GestorCuentas;
import javax.naming.*;

public class ClienteCuentaCorriente
{
```

```
    public static void main(String [] args) throws Exception
    {
        try {
                // Se accede al contexto por defecto
                Context ctx = new InitialContext();
                // Se busca el objeto publicado con JNDI
                Object obj =
                        ctx.lookup("GestorCuentas/refGestorCuentasCMP");
                GestorCuentasHome home = (GestorCuentasHome)obj;
                        javax.rmi.PortableRemoteObject.narrow(obj,
                                                GestorCuentasHome.class);

                GestorCuentas cuenta = home.create();
                // Se crea una cuenta nueva y se opera con ella
                cuenta.operarCuenta("Alejandro Barrera");

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

Las cuestiones mas importantes a destacar del código de esta clase ya se enumeráron anteriormente.

7.5.6.4.8. Fichero application.xml

El contenido del descriptor de despliegue application.xml, que declara todos los módulos J2EE incluidos dentro de la aplicación, se presenta a continuación y se guardará en el directorio /dd del directorio de trabajo.

```
<!DOCTYPE application PUBLIC
        "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
        "http://java.sun.com/dtd/application_1_3.dtd">

<application>

    <description>Una descripción cualquiera</description>

    <display-name>GestorCuentas</display-name>

    <module>
        <ejb>cuentasCMP-ejb.jar</ejb>
    </module>

    <module>
        <java>app-client.jar</java>
    </module>

</application>
```

7.5.6.5. Utilizando Ant

Al igual que en el ejemplo anterior, esta vez también se propone aprovechar al máximo la potencia de Ant y ejecutar de forma automática y consecutiva todas las tareas agrupadas en una única tarea. La tarea a ejecutar se denomina *all* y ya se encuentra definida en el fichero jboss-build.xml.

Para ejecutar esta tarea *all* que engloba a todas las demás hay que invocar dicha tarea con el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml all
```

Como es habitual, si todo el procesamiento de las tareas ha ido correctamente al final se enseña un mensaje Build Successful y si se ha producido algún error se enseñará el mensaje Build failed.

7.5.6.6. Poblando la base de datos

En el caso del ejemplo anterior de beans BMP este era el momento de ejecutar las tareas relacionadas con la base de datos. Era en este momento cuando se debian ejecutar las tareas de creación de tablas a través de la tarea create-table y de poblar dicha tabla con los distintos datos incluidos en el *script* ejecutando la tarea denominada insert-data.

Sin embargo, tal y como se explicó al comienzo del ejemplo, se ha intentado aprovechar al máximo la potencia del motor CMP del servidor de aplicaciones JBoss y de esta forma facilitar que el propio contenedor EJB cree la tablas que deben dar soporte a los datos que maneja el bean de entidad. Por lo tanto, en este ejemplo no se tendrá que ejecutar ninguna tarea de creación de tablas ya que estan se crearán de forma automática cuando se realice el despliegue de la aplicación.

Por su parte la ejecución de la tarea insert-data que puebla de datos la tabla creada por el contenedor EJB, se deja como opcional para el lector. Si se decide ejecutarla habrá

que cambiar ligeramente el *script* para adaptarlo al nombre de la tabla creada y el resultado de la ejecución del ejemplo debería ser exactamente igual que la ejeución del ejemplo anterior. Si por el contrario, se decide no ejecutarla, ya que no es una tarea imprescindible para el funcionamiento de la aplicación, el resultado será el que se va a enseñar a continuación.

7.5.6.7. Ejecutando el Gestor de Cuentas

A la hora de ejecutar el cliente stand-alone de la aplicación Gestion de Cuentas CMP, tambien se recurrirá a la ayuda de la herramienta Ant. Para realizar esta tarea de ejecución del cliente se utiliza el siguiente comando:

```
C:\XXXX\ant -f jboss-build.xml run-client
```

Como resultado de la ejecución anterior, en el lado servidor, después de ejecutar dos veces la aplicación, lo que se debe visualizar en la consola de *log* del JBoss debe ser una secuencia similar a la siguiente:

```
17:29:44,111 INFO  GestorUsuariosBean.setSessionContext() invocado.
17:29:44,131 INFO  GestorUsuariosBean.ejbCreate() invocado.
17:29:44,181 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
17:29:44,191 INFO  EntityContext establecido
17:29:44,191 INFO  Total acumulado en cuentas: 0.0
17:29:44,201 INFO  ejbCreate
17:29:44,211 INFO  ejbPostCreate
17:29:44,211 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
17:29:44,211 INFO  EntityContext establecido
17:29:44,211 INFO  ejbStore
17:29:44,221 INFO  Saldo actual: 0.0
17:29:44,221 INFO  Ingreso de 200.0 euros solicitado
17:29:44,221 INFO  Saldo actual: 200.0
17:29:44,221 INFO  Reintegro de 150.0 euros solicitado
17:29:44,221 INFO  Reintegro realizado con exito
17:29:44,221 INFO  Saldo actual: 50.0
17:29:44,221 INFO  ejbStore
17:29:44,231 INFO  Total acumulado en cuentas: 50.0
17:29:44,231 INFO  Encontrada con ID 05abc73CMP cuenta con saldo 50.0
17:29:44,251 INFO  ejbStore
17:29:44,251 INFO  ejbRemove
17:29:44,261 INFO  Es imposible que en un banco donde se realizan este
tipo de practicas fraudulentas no este creada aun la cuenta del Gerente
17:29:45,993 INFO  GestorUsuariosBean.ejbPassivate() invocado.

17:30:36,676 INFO  GestorUsuariosBean.setSessionContext() invocado.
17:30:36,676 INFO  GestorUsuariosBean.ejbCreate() invocado.
17:30:36,706 INFO  Total acumulado en cuentas: 0.0
17:30:36,716 INFO  ejbCreate
```

```
17:30:36,716 INFO  ejbPostCreate
17:30:36,716 INFO  Nuevo EJB Entidad CuentaCorrienteBean construido por cont
17:30:36,726 INFO  EntityContext establecido
17:30:36,726 INFO  ejbStore
17:30:36,726 INFO  Saldo actual: 0.0
17:30:36,726 INFO  Ingreso de 200.0 euros solicitado
17:30:36,736 INFO  Saldo actual: 200.0
17:30:36,736 INFO  Reintegro de 150.0 euros solicitado
17:30:36,736 INFO  Reintegro realizado con exito
17:30:36,736 INFO  Saldo actual: 50.0
17:30:36,736 INFO  ejbStore
17:30:36,746 INFO  Total acumulado en cuentas: 50.0
17:30:36,746 INFO  Encontrada con ID 05abc73CMP cuenta con saldo 50.0
17:30:36,746 INFO  ejbStore
17:30:36,746 INFO  ejbRemove
17:30:36,756 INFO  Es imposible que en un banco donde se realizan este
tipo de practicas fraudulentas no este creada aun la cuenta del Gerente
17:30:38,449 INFO  GestorUsuariosBean.ejbPassivate() invocado.
```

Aquí se puede apreciar como la aplicación en la primera ejecución primero crea una instancia del bean para llamar al método getTotalEnCuentas( ) que le da como resultado 0 euros. Luego se crea una nueva cuenta corriente, que inicialmente tiene como saldo 0 euros, se ingresan 200 y luego se retiran 150 quedando un saldo de 50 euros. Finalmente, se recupera la cuenta creada y se elimina para dejar el sistema en el mismo estado en el que se encontró.

Sin embargo, la gran diferencia con la ejecución del ejemplo anterior radica en que en esta ocasión se observa como el saldo total acumulado en las cuentas se no ha incrementado en 1.000 euros. Esto es debido a que al no ejecutar la tarea asociada con el *script* de inserción de datos, no se ha incluido aún la cuenta del Gerente y por ese motivo aparece en el *log* del servidor el mensaje que avisa de esa situación poco corriente.

La segunda vez que se ejecuta la aplicación, se puede apreciar como el comportamiento es exactamente igual que la anterior.

Nota de seguimiento: de la importancia de estar bien informados… y del log4j

Uno de los principales problemas que aportan los beans CMP, frente a los beans BMP, es que su depuración ante errores de ejecucion es bastante más difícil de realizar que en el caso de BMP. Esto es debido a que en CMP el contendor EJB genera su propio código recurriendo a mecanismos como las subclases que no son controladas por el desarrollador.

De esta forma, resultaría más difícil encontrar el origen del problema ante un fallo en la creación de una tabla para una persistencia CMP, por ejemplo. Cuanto mayor es la cantidad de código generada por el desarrollador, mayor facilidad a la hora de depurarlo. Si por el

contrario, gran parte del código lo genera el contenedor EJB, puede llegar a haber mas fiabildad (con el paso del tiempo los servidores de aplicaciones brindan cada vez un mejor rendimiento), pero indiscutiblemente menor será la capacidad de depuración.

Por lo tanto, cuando se utilizan beans CMP es fundamental saber siempre lo que se está haciendo. En estos casos resulta imprescindible recurrir a generadores de información como lo son el log4j. El log4j nos permite conocer con bastante precisión las cosas que pasan en el servidor de aplicaciones.

Por ejemplo, si se echa un vistazo al *log* del servidor de aplicaciones JBoss, que se encuentra en el fichero server.log dentro del directorio /log del servidor que se este utilizando se puede ver una gran cantidad de información que puede resultar de mucho interés y que puede ayudar bastante a la hora de solucionar problemas o errores que se presenten en tiempo de despliegue o ejecución.

A continuación se muestra un fragmento de dicho *log* que muestra unas cuantas cosas interesantes:

2005-08-19 17:37:27,567 TRACE Returning connection to pool org.jboss.resource.connectionmanager.TxConnectionManager$TxConnectionEventListener@1ef45e0[state=NORMAL handles=0 lastUse=1124465847567 permit=false trackByTx=false mcp=org.jboss.resource.connectionmanager.JBossManagedConnectionPool$OnePool@1f4cdd2] [InUse/Available/Max]: [0/20/20]

2005-08-19 17:37:27,567 DEBUG Executing SQL: CREATE TABLE cuentaCMP (id VARCHAR(10) NOT NULL, titular VARCHAR(250) BINARY, saldo DOUBLE NOT NULL, CONSTRAINT pk_cuenta PRIMARY KEY (id))

La primera parte del *log* se corresponde con una traza relativa a una petición de una conexión hecha por el contenedor EJB al pool de conexiones que maneja. Una vez obtenida la conexión, el contenedor EJB ejecuta una sentencia SQL que él mismo ha generado para crear la tabla que se le ha impuesto a través de los ficheros descriptores de despliegue.

Es importante destacar que los tipos que se utilizan en la creación de la tabla se pueden ver aquí. De esta forma, se aprecia como en el caso de campo id se utilizará un tipo SQL VARCHAR(10) que fué el definido en el descriptor de despliegue jbosscmp-jdbc.xml, mientras que el campo titular utilizará un tipo SQL VARCHAR(250) que es el definido por defecto para los tipos String.

Finalmente, se propone un ejercicio que demuestra la utilidad de estar bien informados a través del log. Edítese la clase *primary key* del bean CuentaCorriente y cámbiese el modificador del atributo codigoCuenta a privado. Elimínese ahora la tabla cuentacmp de la base de datos para obligar de nuevo a su creación durante el siguiente despliegue. Y finalmente, ejecutese la tarea *all* con la herramienta Ant. Lógicamente esto debería producir un nuevo despliegue del los beans de la aplicación.

Sin embargo, se puede comprobar en la consola del servidor JBoss que la operación no se ha realizado con éxito y que se producen excepciones SQL en la creación de la tabla. El problema en la consola no es evidente, a continuación se presenta un fragmento de lo poco que aporta:

18:02:14,229 ERROR [EntityContainer] Starting failed jboss.j2ee:jndiName=local/CuentaCorri service=EJB org.jboss.deployment.DeploymentException: Error while creating table cuentaCMP;- nested throwable: (java.sql.SQLException: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '))' at line 1)

Pero si el lector visita el *log*, allí se podrá encontrar la siguiente información:

2005-08-19 18:31:07,889 DEBUG Executing SQL: CREATE TABLE cuentaCMP (id VARCHAR(10) NOT NULL, titular VARCHAR(250) BINARY, saldo DOUBLE NOT NULL, CONSTRAINT pk_cuenta PRIMARY KEY ( ))

Que falta? Efectivamente, la declaración de la *primary key* es errónea. Falta el atributo que se puso como privado.

7.6. BMP o CMP?

La elección entre un tipo BMP o CMP para un bean de entidad no es una decisión trivial ya que cada tipo tiene sus ventajas e inconvenientes. A continuación se realiza un estudio detallado de las características más importantes que puede afectar a la decisión sobre uno u otro.

1. Reducción de código y rapidez a la hora de desarrollar. La reducción del tamaño del código que presenta CMP frente a BMP, debido en gran parte a que se elimina toda la lógica JDBC, es un detalle importante a tener en cuenta. Además, este hecho lógicamente también reduce bastante el tiempo de desarrollo y depuración de los beans de entidad. Sin embargo, hay que tener en cuenta que, por ejemplo, para lo métodos *finders*, el desarrollador sigue teniendo la necesidad de implementar lógica de acceso. Bien es cierto que en CMP lo hace de forma declarativa y no programática como en BMP, pero hay que seguir haciéndolo.

2. Rendimiento. Los beans CMP configurados apropiadamente presentan mucho mejor rendimiento que los beans BMP. El número de sentencias SQL que implementa un desarrollador siempre es mucho mayor que las mega-sentencias que implementa un contenedor EJB.

3. Depuración. La depuración del código en BMP, a pesar de ser una tarea de mayor duración, debido al tamaño de los beans, siempre resulta más sencilla que en CMP ya que el desarrollador tiene el control de todo el código JDBC mientras que en CMP ese código lo genera el contenedor. Además, el recurrir

a mecanismos de carácter descriptivo en CMP puede convertirse en una fuente constante de errores. Por ejemplo, habría que saber que sucede si no coinciden los datos del bean con los del fichero descriptor de despliegue. Ante este tipo de situaciones el contenedor no tiene un comportamiento siempre determinista, y puede desde generar simplemente un código erróneo hasta producir un fallo.

4. Independencia de base de datos. Esta es quizás una de las características más importantes a tener en cuenta a la hora de elegir un tipo u otro de bean de entidad. CMP aporta una total independencia de la base de datos ya que los beans de entidad no contienen todo el código JDBC relacionado con la persistencia. Esta circunstancia resulta fundamental para los proveedores de componentes que no deben atarse a un sistema de almacenamiento ni proporcionar su código fuente.

5. Independencia del servidor de aplicaciones. El hecho de ser independiente del servidor de aplicaciones resulta mucho mas complicado en el caso de los beans CMP. No existe actualmente un estándar que defina el mapeo objeto-relacional en el que se basa la persistencia. Cada servidor de aplicaciones tiene sus propias herramientas de mapeo de atributos a columnas de la base de datos por lo que resulta necesario realizar de nuevo la especificación de ese mapeo cada vez que se quiere cambiar de servidor de aplicaciones. Es decir, ante un cambio de servidor de aplicaciones habrá que volver a generar los ficheros descriptores de despliegue propietarios del servidor. Incluso puede darse el caso de que un servidor de aplicaciones soporte operaciones de persistencia que otro servidor no soporta.

Se acaban de ver todos los posibles condicionantes que pueden afectar la elección de uno u otro tipo de bean de entidad. Sin embargo, como se ha podido comprobar, las mayores diferencias entre los BMP y los CMP están en la clase del bean y los descriptores de despliegue. Por el contrario, los interfaces Remote y Home (o sus interfaces locales respectivos) y la clase *primary key* permanecen invariantes. Este hecho produce una consecuencia fundamental y es el hecho de no tener que cambiar

para nada el código del cliente en caso de cambiar un bean de BMP a CMP o viceversa.

A pesar de que el cambio de un bean BMP a uno CMP no sea crítico de cara a un cliente hay que mencionar, sin embargo, que existe una forma de trabajar con beans que permite aprovechar de cierta forma las ventajas de ambos tipos. De entrada, se puede considerar como una buena practica el trabajar siempre con beans CMP. La mayor parte de los desarrolladores se encuentran a gusto utilizando esos beans CMP y lo único que temen es no ser lo suficientemente flexibles en el futuro. Por lo tanto, para conseguir esta flexibilidad se recomienda que todos los beans de entidad sean del tipo CMP y que a medida que se vaya necesitando beans BMP se vayan creando subclases de los CMP para ir introduciendo la lógica necesaria.

Esta modificación en el tipo de un bean CMP a uno bean BMP no impactará en absoluto en los interfaces home o remoto del bean que ya estarán creados con anterioridad. Esta estrategia es posible debido que los beans de entidad están implementados como clases abstractas. Además, esta estrategia de extender el bean CMP para crear un bean BMP, trae como consecuencias que se permita a un bean ser tanto un bean CMP como uno BMP y que se acorte significativamente la cantidad de código a mostrar.

7.7. Heurística y consejos prácticos en EJB

7.7.1. Beans de sesión con estado o sin estado.

Cuando se tiene que elegir entre un bean de sesión con estado y sin estado es conveniente plantearse las siguientes cuestiones. Hay que saber si el proceso de negocio que realiza el bean abarca varias invocaciones, es decir, si va a requerir de una lógica conversacional. Si la respuesta es positiva claramente el tipo a elegir debe ser un bean de sesión con estado. Si la respuesta en negativa y el proceso del bean se compone de una única llamada a un método, la elección mas apropiada sería la de un bean de sesión sin estado.

En los beans de sesión sin estado, el contenedor EJB es capaz de reutilizar de forma sencilla los beans utilizando el concepto de *pool*. De esta forma, se consigue dar servicio a multitud de clientes con unos pocos beans. Sin embargo, la mayor desventaja de un bean sin estado es que a menudo se necesita invocar métodos que requieren determinados parámetros. En estos casos, en cada invocación del método sería necesario reenviar la información específica del cliente.

En el caso de beans de sesión con estado, la misma idea de reutilización de instancias, en caso de no tener más instancia de beans disponibles en el *pool*, obliga a realizar costosas tareas de pasivación que pueden resultar siendo un cuello de botella. Por otro lado los beans de sesión con estado mantienen el estado conversacional con el cliente en memoria. Por lo tanto, un fallo en un bean que no proporciona mecanismos de recuperación de estado puede generar errores graves. Eso en los beans de sesión sin estado no sucede ya que en este caso, ante un fallo en un bean, su funcionalidad puede ser cubierta por cualquier otro bean ya que cualquier bean puede dar servicio a cualquier cliente.

7.7.2. Recubrir los beans de entidad con un bean de sesion

El valor añadido que proporcionan los beans de sesión como recubridores de los beans de entidad se esfuma si el acceso de los clientes a esos beans de entidad no se hace de forma remota. Es decir, los beans de sesión como *wrappers* de los beans de entidad son útiles únicamente como optimizadotes del rendimiento en las conexiones a través de una red. Por este motivo, en un despliegue de una aplicación web J2EE en el que los servlets y las páginas JSP acceden a componentes EJB´s que residen en el mismo proceso, este recubrimiento no tendría sentido.

Sin embargo, en el acceso remoto a través de una red de una aplicación cliente a un bean de entidad se producen una serie de llamadas que hacen que el rendimiento se vea afectado. Para solucionar esta situación se recubre el bean de entidad. De esta forma es el bean de sesión quien realiza todas las operaciones de persistencia y no el cliente remoto. Es decir, las operaciones de creación, recuperación, actualización y borrado (u operaciones CRUD, Creation, Read, Update and Delete) se ejecutan ahora desde el servidor y no desde en el cliente.

Esta solución además, permite que el control de las transacciones se ejecute también en el propio servidor y no desde el cliente. Así, el bean de sesión actuará como una fachada transaccional. El bean de entidad no será nunca visible desde el cliente, en su lugar el bean de entidad se debe entender como el mecanismo que permite al bean de sesión realizar la persistencia.

7.7.3. Interfaces locales o remotos

Los interfaces locales incluidos desde la especificación EJB 2.0 permiten acceder a los componentes EJB sin necesidad de incluir todos los mecanismos necesarios en las llamadas remotas a través de una red. Al no incluir dichos mecanismos de llamadas remotas a otros espacios de direcciones se permite el paso de parámetros por referencia sin necesidad de que sean parámetros serializables.

Sin embargo, las llamadas remotas dentro de la especificación EJB siguen teniendo una importancia capital ya que son la única forma de realizar llamadas entre componentes que se encuentran en distintos espacio de direcciones, que lógicamente, es uno de los fundamentos del desarrollo de aplicaciones distribuidas. Por lo tanto, las llamadas remotas se utilizan si se necesita tener acceso a los componentes del sistema desde clientes remotos, si es necesario que los contenedores puedan distribuir su carga de trabajo o se requiere prestar servicios de tolerancia a fallos mediante clustering en el lado servidor.

Una vez conocidas las particularidades de cada uno de los dos tipos de accesos, hay que estudiar cual es mas apropiado para cada aplicación a desarrollar. Para un rendimiento óptimo se recomienda que la mayoría de los componentes EJB, principalmente los beans de entidad, se implementen usando los interfaces remotos. En cambio, solo aquellos beans en los que resulta imprescindible el acceso remoto, como pueden ser algunos beans de sesión, estarían obligados a utilizar los interfaces remotos con los que se permite el acceso de los clientes que se encuentran en otro espacio direcciones.

También, es importante destacar que un cambio en el tipo de interfaces implica, obviamente, la re-codificación de dichos interfaces y sobre todo de los clientes. Esta es una desafortunada consecuencia de la utilización de métodos programáticos para la selección del tipo de acceso. Esto se produce porque el código de los interfaces locales y remotos es ligeramente distinto. Los interfaces locales extienden de un interfaz distinto a los remotos y no usan el método PortableRemoteObject.narrow(), a parte de que no se lanzan las excepciones RemoteExceptions debido a que no hay acceso remoto.

Por lo tanto, para limitar las consecuencias negativas de un cambio de acceso, resulta fundamental realizar un estudio previo detallado del tipo de interfaces que va a utilizar la aplicación dependiendo del tipo de acceso que va a soportar el sistema. Por ejemplo, si se esta construyendo una aplicación *web*, hay que decidir a priori, si el sistema será una aplicación completa J2EE que ejecutará en un único proceso, o si por el contrario, la capa *web* puede estar separada de la capa EJB en distintos procesos y por lo tanto, si que requeriría el acceso remoto.

7.7.4. Gestión eficaz de transacciones

Resulta una práctica muy útil el asegurarse de que las transacciones se ejecuten siempre en el servidor, que sean lo más cortas posibles y que encapsulen todas las operaciones realizadas por el bean de entidad que participan en la transacción. Este último detalle es muy importante ya que hay que tener en cuenta que las llamadas JDBC suelen ejecutarse tanto al comienzo como al final de las transacciones. Por lo tanto, es conveniente tratar de ejecutar el mínimo número sentencias SQL que abarquen la mayor cantidad de código posible en cada transacción. Si se realizan transacciones por cada operación *get* o *set* se están realizando demasiadas operaciones SQL en cada llamada a método.

Para los beans de entidad, la mejor forma de realizar la gestión de transacciones es encapsulando todas las llamadas a los beans de entidad dentro de un método del bean de sesión. Esto permitirá crear una transacción en el bean de sesión que encapsulará a todas las llamadas al bean de entidad. Finalmente, para conseguir esta gestión transaccional será necesario realizar el despliegue tanto de los beans de sesión como

de los beans de entidad con un valor "*Required*" para el elemento <container-managed transaction>

El secreto de esta técnica radica en que el bean de entidad debe disponer de transacciones gestionadas por el contenedor. Con este tipo de gestión, el desarrollador esta obligado a especificar algunos atributos de la transacción en el descriptor de despliegue como se ha comentado anteriormente. Además el desarrollador debe lanzar la ejecución de un *roll-back* cuando lo necesite o cuando se lance una excepción a nivel de sistema. Para ello debe llamar el método setRollbackOnly( ) del interfaz EJBContext.

Por otro lado, los beans de sesión pueden tener la gestión de transacciones bien gestionada por el propio bean o gestionada por el contenedor EJB. La gestión de transacción gestionada por el contenedor se trataría de la misma forma que se ha comentado anteriormente para los beans de entidad. Mientras que la gestión de transacciones gestionada por el propio bean presenta dos opciones: a través de JDBC y a través de la API JTA.

Por un lado, las transacciones JDBC se delimitan usando los métodos commit( ) y rollback( ) del interfaz Connection presente en la API Java JDBC. Y por otro lado, las transacciones de la API Java JTA se delimitan invocando a los métodos begin( ), commit( ) y rollback( ) del interfaz UserTransaction presente en la API Java de transacciones.

La siguiente tabla muestra a modo de resumen los distintos tipos de transacciones que existen y los que están permitidos dependiendo del tipo de bean.

Tipo de bean	**Gestionado por el contenedor**	**Gestionado por el bean**	
		JTA	**JDBC**
Entidad	Y	N	N
Sesión	Y	Y	Y

Gestionado por mensaje	Y	Y	Y

7.7.5. Afinando el rendimiento en CMP

Puede afirmarse que normalmente resulta más recomendable por cuestiones de rendimiento usar la persistencia gestionada por el contendor en lugar de la gestionada por el bean. Sin embargo, esta afirmación solo es válida si previamente se ha confirmado que la herramienta que permite el mapeo objeto-relacional de CMP es buena y suficientemente flexible.

Otra buena práctica es si se esta utilizando la persistencia CMP el que se le indique al contenedor que se realice la persistencia de los atributos en bloque (o *bulk*). Esta estrategia permite que la persistencia de los atributos se realice en bloque y por lo tanto se minimice, también de esta forma, el número de sentencias SQL asociadas a dicha persistencia.

Por otra parte, y también relacionado con la persistencia de los atributos, se considera conveniente que si no se va a acceder a todos los atributos de un bean en cada transacción se pueda forzar al contenedor EJB, si lo soporta, a utilizar el patrón de carga perezosa (o lazy-load proxy pattern). Este patrón permite que solo se carguen ciertos atributos y no todos cuando se accede por primera vez al bean. Para forzar al contenedor a utilizar este patrón se recurre a la forma declarativa que utiliza CMP con los descriptores de despliegue.

También, es recomendable, como ya se comentó, que el contenedor realice todas las actualizaciones en la base de datos a la vez y al final de una transacción. De esta forma se aprovecha una única sentencia SQL para realizar distintas acciones a través de la misma conexión a la base de datos.

Finalmente, se considera buena práctica, cuando se utiliza CMP, el forzar a las herramientas del contenedor a que el método *finder* y la propia carga del bean que se realiza con el método load( ) se realice automáticamente en un solo paso. Esta estrategia de ejecución en un solo paso permite que se ejecute una única sentencia

SQL en vez de realizarlo por separado en dos de ellas. La única vez en la que se no se hacen las dos operaciones de forma consecutiva y atómica es cuando no se van a leer datos del bean de entidad, por ejemplo, en una operación *set* de un atributo que no requiere la carga del resto de los atributos.