ACADEMIA CARTAGENA99
C/ Cartagena 99, Bº C , 28002 Madrid
91 51 51 321
academia@cartagena99.com

CURSO PROGRAMACION AVANZADA JAVA

J2EE

JAVA PARA EMPRESAS

Tema 1: Introducción a los servidores de aplicaciones

Aplicaciones de empresa

El concepto de servidor de aplicaciones está relacionado con el concepto de sistema distribuido. Un sistema distribuido permite mejorar tres aspectos fundamentales en una aplicación: la alta disponibilidad, la escalabilidad y el mantenimiento. Estas características se explican con detalle a continuación:
La alta disponibilidad hace referencia a que un sistema debe estar funcionando las 24 horas del día los 365 días al año. Para poder alcanzar esta característica es necesario el uso de técnicas de balanceo de carga y de recuperación ante fallos (failover).
La escalabilidad es la capacidad de hacer crecer un sistema cuando se incrementa la carga de trabajo (el número de peticiones). Cada máquina tiene una capacidad finita de recursos y por lo tanto sólo puede servir un número limitado de peticiones. Si, por ejemplo, tenemos una tienda que incrementa la demanda de servicio, debemos ser capaces de incorporar nuevas máquinas para dar servicio.
El mantenimiento tiene que ver con la versatilidad a la hora de actualizar, depurar fallos y mantener un sistema. La solución al mantenimiento es la construcción de la lógica de negocio en unidades reutilizables y modulares.

¿Qué es un servidor de aplicaciones?

El estándar J2EE permite el desarrollo de aplicaciones de empresa de una manera sencilla y eficiente. Una aplicación desarrollada con las tecnologías J2EE permite ser desplegada en cualquier servidor de aplicaciones o servidor Web que cumpla con el estándar. Un servidor de aplicaciones es una implementación de la especificación J2EE. La arquitectura J2EE es la siguiente:

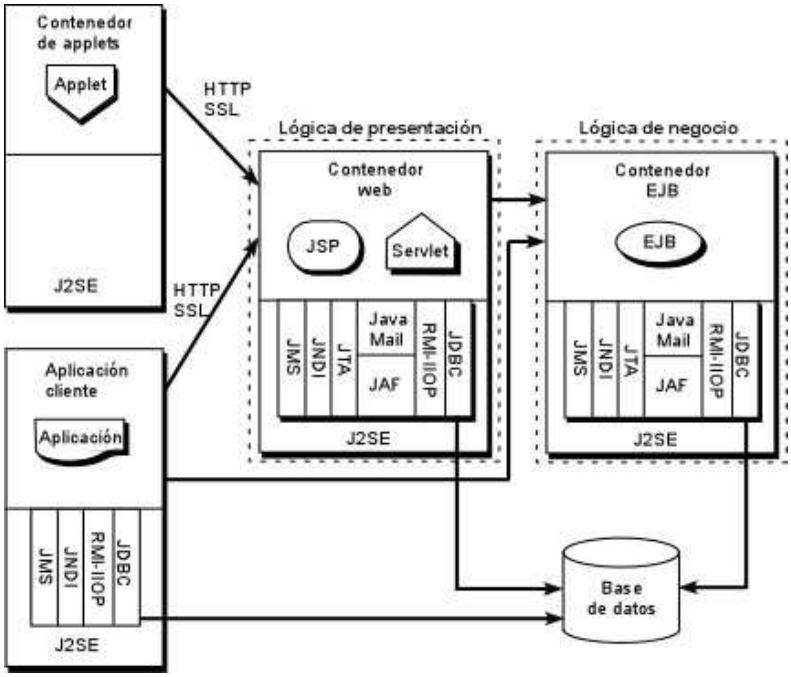


Figura 1. Arquitectura J2EE.

Definimos a continuación algunos de los conceptos que aparecen en la figura 1:

- Cliente Web (contenedor de applets): Es usualmente un navegador e interactúa con el contenedor Web haciendo uso de HTTP. Recibe páginas HTML o XML y puede ejecutar applets y código JavaScript.
- Aplicación cliente: Son clientes que no se ejecutan dentro de un navegador y pueden utilizar cualquier tecnología para comunicarse con el contenedor Web o directamente con la base de datos.
- Contenedor Web: Es lo que comúnmente denominamos servidor web. Es la parte visible del servidor de aplicaciones. Utiliza los protocolos HTTP y SSL (seguro) para comunicarse.
- Servidor de aplicaciones: Proporciona servicios que soportan la ejecución y disponibilidad de las aplicaciones desplegadas. Es el corazón de un gran sistema distribuido.

Frente a la tradicional estructura en dos capas de un servidor web (ver Figura 2) un servidor de aplicaciones proporciona una estructura en tres capas que permite estructurar nuestro sistema de forma más eficiente. Un concepto que debe quedar claro desde el principio es que no todas las aplicaciones de empresa necesitan un servidor de aplicaciones para funcionar. Una pequeña aplicación que acceda a una base de datos no muy compleja y que no sea distribuida probablemente no necesitará un servidor de aplicaciones, tan solo con un servidor web (usando servlets y jsp) sea suficiente.

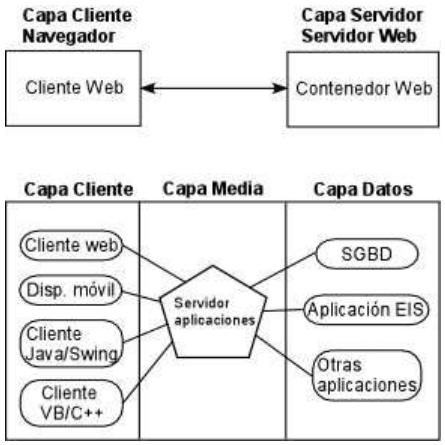


Figura 2. Arquitectura en dos capas frente a tres capas utilizando el servidor de aplicaciones.

Como hemos comentado, un servidor de aplicaciones es una implementación de la especificación J2EE. Existen diversas implementaciones, cada una con sus propias características que la pueden hacer más atractiva en el desarrollo de un determinado sistema. Algunas de las implementaciones más utilizadas son las siguientes:

- BEA WebLogic
- IBM WebSphere
- Sun-Netscape IPlanet
- Sun One
- Oracle IAS
- Borland AppServer
- HP Bluestone

Los dos primeros son los más utilizados en el mercado. En este curso vamos a utilizar el servidor BEA WebLogic. La principal ventaja de WebLogic es que podemos crear un sistema

con varias máquinas con distintos sistemas operativos: Linux, Unix, Windows NT, etc. El sistema funciona sin importarle en qué máquina está corriendo el servidor.

Otros conceptos que aparecerán a lo largo de este módulo:

- Servidor proxy: Centraliza peticiones de los clientes y las reenvía hacia otras máquinas. Puede servir como nivel de indirección y seguridad. También puede ser usado para realizar balanceo de carga.
- Cortafuegos (firewall): Proporciona servicios de filtrado, autorización y autentificación. Puede actuar como proxy y ayuda a manejar los ataques de los hackers.
- Máquina: Representa una unidad física donde reside un servidor. Una máquina se define como tipo Unix o no Unix (Windows NT, etc.).
- Servidor: Un servidor es una instancia de la clase weblogic Server ejecutándose dentro de una máquina virtual de Java. Un servidor está alojado en una máquina, pero una máquina puede contener varios servidores. Si un servidor no lo declaramos en ninguna máquina WLS asume que está en una creada por defecto.
- Dominio: Un dominio es una unidad administrativa. Sirve para declarar varios servidores, aplicaciones, etc. y que todos ellos estén asociados mediante el nombre del dominio.
- Clustering (asociación): Los clusters permiten asociar maquinas y servidores para que actúen de forma conjunta como una única instancia. La creación de un cluster va a permitir el balanceo de carga y la recuperación frente a fallos.
- Balanceo de carga: Es una técnica utilizada para distribuir las peticiones entre varios servidores de tal forma que todos los servidores respondan al mismo número de peticiones.
- Recuperación ante fallos (failover): Permite evitar la caída de un sistema cuando una máquina deja de funcionar o funciona incorrectamente.
- Puerto de escucha: Un servidor tiene varios puertos por los que puede "escuchar" las peticiones. Existen puertos ya asignados a aplicaciones concretas, como por ejemplo el puerto de http que suele ser el 80. Los puertos permiten que varias aplicaciones puedan atender distintas peticiones en la misma máquina. Un puerto en una dirección se especifica de la siguiente manera: http://localhost:7001/direc. Con :7001 indicamos el puerto que estamos atacando. Los puertos del 0 al 1023 son reservados por el sistema. Podemos disponer de los puertos del 1024 al 65536. Hay que tener en cuenta que dos servicios no pueden estar escuchando en el mismo puerto.
- Modo producción y modo desarrollo. Hablaremos muy a menudo de modo desarrollo y modo producción. El modo desarrollo es cuando nos encontramos desarrollando nuestra aplicación y no está disponible exteriormente. El modo producción es cuando está funcionando a pleno rendimiento y tenemos clientes que se encuentran utilizándola. Por defecto, un dominio se arranca en modo desarrollo.

La solución WebLogic Server

El entorno de negocio de hoy en día demanda aplicaciones Web y de comercio electrónico que aceleren nuestra entrada en nuevos mercados, nos ayude a encontrar nuevas formas de llegar y de retener clientes, y nos permita presentar rápidamente productos y servicios. Para construir y desplegar estas nuevas soluciones, necesitamos una plataforma de comercio electrónico probada y creíble que pueda conectar y potenciar a todos los tipos de usuario mientras integra nuestros datos corporativos, las aplicaciones mainframe, y otras aplicaciones empresariales en una solución de comercio electrónico fin-a-fin poderosa y flexible. Nuestra solución debe proporcionar el rendimiento, la escalabilidad, y la alta disponibilidad necesaria para manejar nuestros cálculos de empresa más críticos.

Como una plataforma líder en la industria de comerción electrónico, WebLogic Server nos permite desarrollar y desplegar rápidamente, aplicaciones fiables, seguras, escalables y manejables. Maneja los detalles a nivel del sistema para que podamos concentrarnos en la lógica de negocio y la presentación

Plataforma J2EE

WebLogic Server utiliza tecnologías de la plataforma Java 2, Enterprise Edition (J2EE). J2EE es la plataforma estándard para desarrollar aplicaciones multi-capa basadas en el lenguaje de programación Java. Las tecnologías que componente J2EE fueron desarrolladas colaborativamente entre Sun Microsystems y otros vendedores de software entre los que se incluye BEA Systems.

Las aplicaciones J2EE están basadas en componentes estandarizados y modulares. WebLogic Server proporciona un conjunto completo de servicios para esos componentes y maneja automáticamente muchos detalles del comportamiento de la aplicación, sin requerir programación.

Despliegue de Aplicaciones a través de Entornos Distribuidos y Heterogéneos

WebLogic Server proporciona características esenciales para desarrollar y desplegar aplicaciones críticas de comercio electrónico a través de entornos de computación distribuidos y heterogéneos. Entre estas características están las siguientes:

- Estándars de Liderazgo—Soporte Comprensivo de Java Enterprise para facilitar la implementación y despliegue de componentes de aplicación. WebLogic Server es el primer servidor de aplicaciones independientes desarrollado en Java en conseguir la certificación J2EE.
- Ricas Opciones de Cliente—WebLogic Server soporta nevegadores Web y otros clientes que usen HTTP; los clientes Java que usan RMI (Remote Method Invocation) o IIOP (Internet Inter-ORB Protocol); y dispositivos móviles que usan WAP (Wireless Access Protocol). Los conectores de BEA y otras compañias permiten virtualmente a cualquier cliente o aplicación legal trabajar con el Servidor de aplicaciones WebLogic.
- Escalabilidad de comercio electrónico empresarial—Los recursos críticos se usan eficientemente y la alta disponibilidad está asegurada a través del uso de componentes Enterprise JavaBean y mecanismos como el "clustering" de WebLogic Server para las páginas Web dinámicas, y los almacenes de recursos y las conexiones compartidas.
- Administración Robusta—WebLogic Server ofrece una Consola de Administración basada en Web para configurar y monitorizar los servicios del WebLogic Server. Se hace conveniente para la configuración un interface de línea de comandos para administrar el servidor WebLogic on Scripts.

- Seguridad Lista para el Comercio Electrónico—WebLogic Server proporciona soporte de Secure Sockets Layer (SSL) para encriptar datos transmitidos a través de WebLogic Server, los clientes y los otros servidores. La seguridad de WebLogic permite la autentificación y autorización del usuario para todos los sevicios de WebLogic Server. Los almacenes de seguridad externa, como servidores Lightweight Directory Access Protocol (LDAP), puede adaptarse para WebLogic, nos permiten una sola autentificación para empresas. El "Security Service Provider Interface" hace posible extender los servicios de seguridad de WebLogic e implementar estas características en aplicaciones.
- Máxima flexibilidad en el desarrollo y despliegue—WebLogic Server proporciona una estrecha integración y soporte con las bases de datos más importantes, herramientas de desarrollo y otros entornos.

Arquitectura del Servidor de Aplicaciones WebLogic

WebLogic Server es un servidor de aplicaciones: una plataforma para aplicaciones empresariales multi-capa distribuidas. WebLogic Server centraliza los servicios de aplicación como funciones de servidor webr, componentes del negocio, y acceso a los sistemas "backend" de la empresa. Utiliza tecnologías como el almacenamiento en memoria inmediata y almacenes de conexiones para mejorar la utilización de recursos y el funcionamiento de la aplicación. WebLogic Server también proporciona facilidades a nivel de seguridad empresarial y una administración poderosa.

WebLogic Server funciona en la capa media (o capa "n") de una arquitectura multi-capa. Una arquitectura multi-capa determina dónde se ejecutan los componentes software que crean un sistema de cálculo en relación unos con otros y al hardware, la red y los usuarios. Elegir la mejor localización para cada componente software nos permite desarrollar aplicaciones más rápidamente; facilita el despliegue y la administración; y proporciona un mayor control sobre el funcionamiento, la utilización, la seguridad, el escalabildad, y la confiabilidad.

WebLogic Server implementa J2EE, el estándar para la empresa de Java. Java es un lenguaje de programación, seguro ante la red, orientado a objetos, y J2EE incluye la tecnología de componentes para desarrollar objetos distribuidos. Estas funciones agregan una segunda dimensión architectura del servidor de aplicaciones WebLogic Server-- un capa de lógica de aplicación, con cada capa desplegada selectivamente entre las tecnologías J2EE de WebLogic Server.

Las dos secciones siguientes describen estas dos vistas de la arquitectura de WebLogic Server: capas de software y capas de la lógica de la aplicación.

Capas de Componentes Software

Los componentes software de una arquitectura multi-capa constan de tres capas:

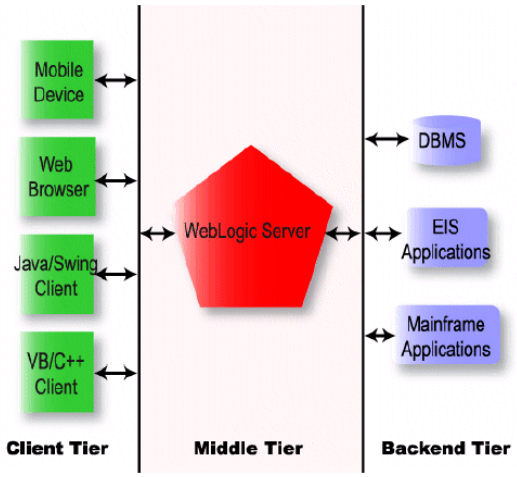
La capa del cliente contiene los programas ejecutados por los usuarios, incluyendo navegadores Web y programas de aplicaciones de red. Estos programas se pueden escribir virtualmente en cualquier lenguaje de programación.

La capa media contiene el servidor WebLogic y otros servidores que son direccionados directamente por los clientes, como servidores web existentes o servidores proxy.

La capa backend contiene recursos de empresa, como sistemas de base de datos, aplicaciones de unidad central y legales, y aplicaciones de plannings de recursos de empresa empaquetados (ERP).

Las aplicaciones del cliente tienen acceso al servidor WebLogic directamente, o a través de un servidor web o un proxy. El servidor WebLogic conecta con servicios backend por cuenta de los clientes, pero los clientes no tienen acceso directamente a los servicios backend.

La figura 1-1 ilustra estas tres capas de la arquitectura del servidor WebLogic Server.



Componentes de la Capa Cliente

Los clientes del servidor WebLogic utilizan interfaces estándares para acceder a servicios del servidor WebLogic. El servidor WebLogic tiene una completa funcionalidad de servidor web, así que un navegador web puede solicitar páginas al servidor WebLogic usando el protocolo estándar de la Web, HTTP. Los servlets de WebLogic Server y las JavaServer Pages (JSPs) producen páginas Web dinámicas, personalizadas requeridas para las aplicaciones avanzadas de comercio electrónico. Los programas del cliente escritos en Java pueden incluir interfaces gráficos de usuario altamente interactivos construidos con las clases de Java Swing. También se puede tener acceso a servicios del servidor WebLogic usando los APIs estándar del J2EE.

Todos estos servicios también están disponibles para los clientes de navegadores web desplegando servlets y páginas JSP en el servidor WebLogic. Los programas del cliente compatibles con CORBA escritos en Visual Basic, C++, Java, y otros lenguajes de programación pueden ejecutar JavaBeans Enterprise y RMI en el servidor WebLogic usando WebLogic RMI-IIOP. Las aplicaciones del cliente escritas en cualquier lenguaje que soporten el protocolo HTTP pueden acceder a cualquier servicio del WebLogic Server a través de un servlet.

Componentes de la Capa Media

La capa media incluye el servidor WebLogic y otros servidores Web, cortafuegos, y servidores proxy que median en el tráfico entre los clientes y el servidor WebLogic. El servidor WAP de Nokia, parte de la solución de comercio móvil de BEA, es un ejemplo de otro servidor de la capa media que proporciona una conectividad entre los dispositivos inhalámbricos y el servidor WebLogic. Las aplicaciones basadas en una arquitectura multi-capa requieren confiabilidad, escalabilidad, y un alto rendimiento en la capa media. El servidor de aplicaciones que seleccionemos para la capa media es, por lo tanto, crítico para el éxito de nuestro sistema.

La opción Cluster del servidor WebLogic permite que distribuyamos peticiones de cliente y servicios backend entre varios servidores WebLogic cooperantes. Los programas en la capa del cliente acceder al cluster como si fuera un solo servidor WebLogic. Cuando la carga de trabajo aumenta, podemos agregar otros servidores WebLogic al cluster para compartir el

trabajo. El cluster utiliza un algoritmo de balance de capa seleccionable para elegir el servidor WebLogic del cluster que es capaz de manejar la petición.

Cuando una petición falla, otro servidor WebLogic que proporciona el servicio solicitado puede asumir el control. Los fallos son transparentes siempre que sea posible, lo que reduce al mínimo la cantidad de código que se debe escribir para recuperar incidentes. Por ejemplo, el estado de la sesión de un servlet se puede replicar en un servidor secundario WebLogic de modo que si el servidor WebLogic que está manejando una petición falla, la sesión del cliente se pueda reanudar de forma ininterrumpida desde el servidor secundario. Todos los servicios de WebLogic, EJB, JMS, JDBC, y RMI están implementados con capacidades de clustering.

Componentes de la Capa Backend

La capa backend contiene los servicios que son accesibles a los clientes sólo a través del servidor WebLogic. Las aplicaciones en la capa backend tienden a ser los recursos más valiosos y de misiones críticas para empresa. El servidor WebLogic los protege de accesos directos de usuarios finales. Con tecnologías tales como almacenes de conexiones y caches, el servidor WebLogic utiliza eficientemente los recursos backend y mejora la respuesta de la aplicación.

Los servicios backend incluyen bases de datos, sistemas de hojas de operación (planning) de recursos de la empresa (ERP), aplicaciones mainframe, aplicaciones legales de la empresa, y monitores de transacciones. Las aplicaciones existentes de la empresa se pueden integrar en la capa backend usando la especificación de configuración del conector Java (JCA) de Sun Microsystems. El servidor WebLogic hace fácil agregar un interface Web a una aplicación backend integrada. Un sistema de control de base de datos es el servicio backend más común, requerido por casi todas las aplicaciones del servidor WebLogic. WebLogic EJB y WebLogic JMS normalmente almacena datos persistentes en una base de datos en la capa backend.

Un almacen de conexiones JDBC, definido en el servidor WebLogic, abre un número predefinido de conexiones a la base de datos. Una vez que estén abiertas, las conexiones a la base de datos son compartidas por todas las aplicaciones del servidor WebLogic que necesiten acceder a esa base de datos. Sólo se incurre una sóla vez en la costosa sobrecarga asociada con el establecimiento de conexiones para cada conexión del almacén, por cada petición de cliente. El servidor WebLogic vigila las conexiones a la base de datos, refrescándolas cuando es necesario y asegurándose de la fiabilidad de los servicios de la base de datos para las aplicaciones.

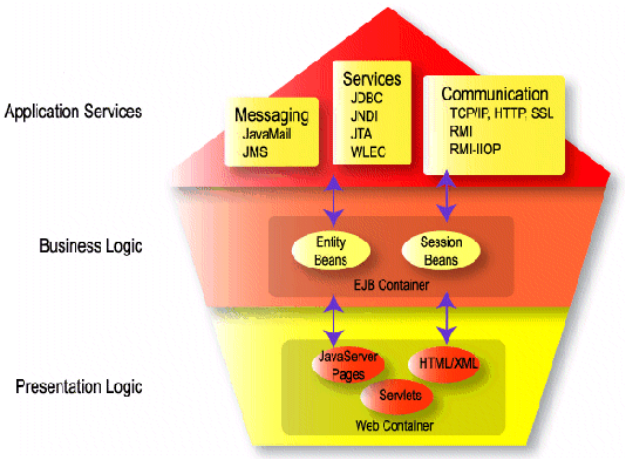
WebLogic Enterprise Connectivity, que proporciona acceso a BEA WebLogic Enterprisesystems, y Jolt® para WebLogic Server que proporciona acceso a los sistemas Tuxedo® de BEA, también utilizan almacenes de conexiones para mejorar el funcionamiento del sistema.

Capas Lógicas de Aplicación

El servidor WebLogic implementa tecnologías de componentes y servicios J2EE. Las tecnologías de componentes J2EE incluyen servlets, páginas JSP, y JavaBeans Enterprise. Los servicios J2EE incluyen el acceso a protocolos de red, a sistemas de base de datos, y a sistemas estándares de mensajería. Para construir una aplicación de servidor WebLogic, debemos crear y ensamblar componentes, usando los APIs de sevicio cuando sean necesarios. Los componentes se ejecutan en contenedor Web del servidor WebLogic o el contenedor de EJB. Los contenedores proporcionan soporte para ciclo vital y los servicios definidos por las especificaciones J2EE de modo que los componentes que construyamos no tengan que manejar los detalles subyacentes.

Los componentes Web proporcionan la lógica de presentación para las aplicaciones J2EE basadas en navegador. Los componentes EJB encapsulan objetos y procesos del negocio. Las aplicaciones Web y los EJBs se construyen sobre servicios de aplicación de J2EE, como JDBC, JMS (servicio de mensajería de Java), y JTA (API de Transaciones de Java).

La Figura 1-2 ilustra los contenedores de componentes y los servicios de aplicación de WebLogic Server.



Capa Lógica de Presentación

La capa de presentación incluye una lógica de interface de usuario y de visualización de aplicaciones. La mayoría de las aplicaciones J2EE utilizan un navegador web en la máquina del cliente porque es mucho más fácil que programas de cliente que se despliegan en cada ordenador de usuario. En este caso, la lógica de la presentación es el contenedor Web del servidor WebLogic. Sin embargo, los programas del cliente escritos en cualquier lenguaje de programación, sin embargo, deben contener la lógica para representar el HTML o su propia lógica de la presentación.

Clientes de Navegador Web

Las aplicaciones basadas en Web construidas con tecnologías web estándar son fáciles de acceder, de mantener, y de portar. Los clientes del navegador web son estándares para las aplicaciones de comercio electrónico. En aplicaciones basadas en Web, el interface de usuario esrá representado por los documentos HTML, las páginas JavaServer (JSP), y los servlets. El navegador web contiene la lógica para representar la página Web en el ordenador del usuario desde la descripción HTML.

Las JavaServer Pages (JSP) y los servlets están muy relacionados. Ambos producen el contenido dinámico de la Web ejecutando el código Java en el servidor WebLogic cada vez que se les invoca. La diferencia entre ellos es que JSP está escrito con una versión extendida de HTML, y los servlets se escriben con el lenguaje de programación Java.

JSP es conveniente para los diseñadores Web que conocen HTML y están acostumbrados al trabajo con un editor o diseñador de HTML. Los Servlets, escritos enteramente en Java, están más pensados para los programadores Java que para los diseñadores Web. Escribir un servlet requiere un cierto conocimiento del protocolo HTTP y de la programación de Java. Un servlet recibe la petición HTTP en un objeto request y escribe el HTML (generalmente) en un objeto result.

Las páginas JSP se convierten en servlets antes de que se ejecuten en el servidor WebLogic, por eso en última instancia las páginas JSP y los servlets son distintas representaciones de la misma cosa. Las páginas JSP se despliegan en el servidor WebLogic la misma forma que se despliega una página HTML. El fichero .jsp se copia en un directorio servido por WebLogic Server. Cuando un cliente solicita un fichero jsp, el servidor WebLogic controla si se ha compilado la página o si ha cambiado desde la última vez que fie compilada. Si es necesario llama el compilador WebLogic JSP, que genera el código del servlet Java del fichero jsp, y entonces compila el código Java a un fichero de clase Java.

Clientes No-Navegadores

Un programa cliente que no es un navegador web debe suminstrar su propio código para represetnar el interface de usuario. Los clientes que no son navegadores generalmente contienen su propia presentación y lógica de la representación, dependiendo del servidor WebLogic solamente para la lógica y el acceso al negocio o a los servicios backend. Esto los hace más difícil de desarrollar y de desplegar y menos convenientes para las aplicaciones basadas en Internet de comercio electrónico que los clientes basados en navegador.

Los programas cliente escritos en Java pueden utilizar cualquier servicio del servidor WebLogic sobre Java RMI (Remote Method Invocatio). RMI permite que un programa cliente opere sobre un objeto del servidor WebLogic la misma forma que operaría sobre un objeto local en el cliente. Como RMI oculta los detalles de las llamadas a tavés de la red, el código del cliente J2EE y el código del lado del servidor son muy similares.

Los programas Java pueden utilizar las clases Swing de Java para crear interfaces de usuario poderosas y portables. Aunque usando Java podemos evitar problemas de portabilidad, no podemos utilizar los servicios del servidor WebLogic sobre RMI a menos que las clases del servidor WebLogic estén instaladas en el cliente. Esto significa que lo clientes RMI de Java no son adecuados para el comercio electrónico. Sin embargo, pueden usarse con eficacia en aplicaciones de empresariales en las cuales una red interna hace vieables la instalación y el mantenimiento. Los programas cliente escritos en lenguajes distintos a Java y los programas clientes Java que no utilizan objetos del servidor WebLogic sobre RMI pueden tener acceso al servidor WebLogic usando HTTP o RMI-IIOP.

HTTP es el protocolo estándar para la Web. Permite que un cliente haga diversos tipos de peticiones a un servidor y pase parámetros al servidor. Un servlet en el servidor WebLogic puede examinar las peticiones del cliente, extraer los parámetros de la petición, y preparar una respuesta para el cliente, usando cualquier servicio del servidor WebLogic. Por ejemplo, un servlet puede responder a un programa cliente con un documento de negocio en XML. Así una aplicación puede utilizar servlets como gateways a otros servicios del servidor WebLogic.

WebLogic RMI-IIOP permite que los programas compatibles con CORBA ejecuten Beans Enterprise del servidor WebLogic y clases RMI como objetos CORBA. El servidor RMI de WebLogic y los compiladores de EJB pueden generar IDL (Interface Definition Language) para las clases RMI y los Beans Enterprise. El IDL generado de esta manera se compila para crear los esqueletos para un ORB (Object Request Broker) y los trozos para el programa cliente. El servidor WebLogic analiza peticiones entrantes IIOP y las envía al sistema de ejecución RMI.

Capa de Lógica de Negocio

Los JavaBeans Enterprise son componentes d ela lógica de negocio para aplicaciones J2EE. El contenedor EJB de WebLogic Server almacenca beans enterprise, proporcionan el control del ciclo de vida y servicios como el caheo, la persistenca, y el control de transaciones. Aquí tenemos tres tipos de beans enterprise: beans de entidad, beans de sesión y beans dirigidos por mensajes. Las siguientes secciones describen cada tipo en más detalle.

Beans de Entidad

Un bean de entidad representa un objeto que contiene datos, como un cliente, una cuenta, o un item de inventario. Los beans de entidad contienen los valores de datos y los métodos que se pueden invocar sobre esos valores. Los valores se salvan en una base de datos (que usa JDBC) o algún otro almacén de datos. Los beans de entidad pueden participar en transacciones que implican otros beans enterprise y servicios transaccionales.

Los beans de entidad se asocian a menudo a objetos en bases de datos. Un bean de entidad puede representar una fila en un vector, una sola columna en una fila, o un resultado completo del vector o de la consulta. Asociado con cada bean de entidad hay una clave primaria única usada para encontrar, extraer, y grabar el bean.

Un bean de entidad puede emplear uno de los siguientes:

Persistencia controlada por el bean. El bean contiene código para extraer y grabar valores persistentes.

Persistencia controlada por el contenedor. El contenedor EJB carga y graba valores en nombre del bean.

Cuando se utiliza la persistencia controlada por el contenedor, el compilador WebLogic EJB puede generar clases de soporte de JDBC para asociar un bean de entidad a una fila de una base de datos. Hay disponibles otros mecanismos de persistencia controlada por el contenedor. Por ejemplo, TOPLink para BEAWebLogic Server, de "The Object People" (http://www.objectpeople.com), proporciona persistencia para una base de datos de objetos relacionales.

Los beans de entidad pueden ser compartidos por muchos clientes y aplicaciones. Un ejemplar de un bean de entidad se puede crear a petición de cualquier cliente, pero no desaparece cuando ese cliente desconecta. Continúa viviendo mientras cualquier cliente lo esté utilizando activamente. Cuando el bean ya no se usa, el contenedor EJB puede pasivizarlo: es decir, puede eliminar el ejemplar vivo del servidor.

Beans de Sesión

Un bean de sesión es un ejemplar transitorio de EJB que sirve a un solo cliente. Los beans de sesión tienden a implementar lógica de procedimiento; incorporan acciones en vez de datos. El contenedor EJB crea un bean de sesión en una petición del cliente. Entonces mantiene el bean mientras el cliente mantiene su conexión al bean. Los beans de sesión no son persistentes, aunque pueden salvar datos a un almacén persistente si lo necesitan.

Un bean de sesión puede ser con o sin estado. Los beans de sesión sin estado no mantienen ningún estado específico del cliente entre llamadas y pueden ser utilizados por cualquier cliente. Pueden ser utilizados para proporcionar acceso a los servicios que no dependen del contexto de una sesión, como enviar un documento a una impresora o extraer datos de sólo lectura en una aplicación. Un bean de sesión con estado mantiene el estado en nombre de un cliente específico.

Los beans de sesión con estado pueden ser utilizados para manejar un proceso, como ensamblar una orden o encaminar un documento con un flujo de proceso. Como pueden acumular y mantener el estado con interacciones múltiples con un cliente, los beans de sesión son a menudo la introducción a los objetos de una aplicación. Como no son persistentes, los beans de sesión deben terminar su trabajo en una sola sesión y utilizar JDBC, JMS, o beans de entidad para registrar el trabajo permanentemente.

Beans Dirigidos por Mensajes

Los beans dirigidos por Mensaje, introducidos en la especificación EJB 2,0, son los beans enterprise que manejan los mensajes asíncronos recibidos de colas de mensaje JMS. JMS encamina mensajes a un bean dirigido por mensaje, que selecciona un ejemplar de un almacen para procesar el mensaje.

Los beans dirigidos por Mensajes se manejan en el contenedor del servidor EJB de WebLogic. Como las aplicaciones dirigidas al usuario no los llaman directamente, no pueden ser accedidas desdee una aplicación usando un EJB home. Sin embargo, una aplicación dirigida al usuario si puede ejemplarizar un bean dirigidopor mensajes indirectamente, enviando un mensaje a la cola de bean JMS.

Servicios de la Capa de Aplicación

El servidor WebLogic proprociona los servicios fundamentales que permiten que los componentes se concentren en lógica del negocio sin la preocupación por detalles de implementación de bajo nivel. Maneja el establecimiento de red, la autentificación, la autorización, la persistencia, y el acceso a objetos remotos para EJBs y servlets. Los APIs Java estándar proporcionan acceso portable a otros servicios que una aplicación puede utilizar, por ejemplo base de datos y servicios de mensajería.

Las aplicaciones cliente de tecnologías de comunicación en la red conectan con el servidor WebLogic usando protocolos de establecimiento de una red estándar sobre TCP/IP. El servidor WebLogic escucha peticiones de conexión en una dirección de red que se pueda especificar como parte de un identificador de recuros uniforme (URI). Un URI es una cadena estandardizada que especifica un recurso en una red, incluyendo Internet. Contiene un especificador de protocolo llamado un esquema, la dirección de red del servidor, el nombre del recurso deseado, y los parámetros opcionales. La URL que introducimos en un navegador web, por ejemplo, http://www.bea.com/index.html, es el formato más familiar de URI.

Los clientes basados en Web se comunican con el servidor WebLogic usando el protocolo HTTP. Los clientes Java conectan usando Java RMI, que permite que un cliente Java ejecute objetos en servidor WebLogic. Los clientes CORBA tienen acceso a objetos RMI desde el servidor WebLogic usando RMI-IIOP, que le permite que ejecutar objetos del servidor WebLogic usando protocolos estándar de CORBA.

Esquema	Protocolo
HTTP	HyperText Transfer Protocol. Utilizado por los navegadores Web y los programas compatibles HTTP.
HTTPS	HyperText Transfer Protoco over Secure Layers (SSL). Utilizado por programas de navegadores Web y clientes compatibles HTTPS.
T3	Protocolo T3 de WebLogic para las conexiones de Java-a-Java, que multiplexa JNDI, RMI, EJB, JDBC, y otros servicios de WebLogic sobre una conexión de red.
T3S	Protocolo T3S de WebLogic sobre sockets seguros (SSL).
IIOP	Protocolo Internet de IIOP Inter-ORB, usado por los clientes de Java compatibles con CORBA para ejecutar objetos WebLogic RMI sobre IIOP. Otros clientes de CORBA conectan con el servidor WebLogic con un CORBA que nombra contexto en vez de un URI para las capas de la lógica de WebLogic Server.

El esquema en un URI determina el protocolo para los intercambios de la red entre un cliente y el servidor WebLogic. Los protocolos de red de la tabla 1-1.

Datos y Servicios de Acceso

El servidor WebLogic implementa tecnologías estándars J2EE para proporcionar servicos de datos y de acceso a las aplicaciones y a los componentes. Estos servicios incluyen los siguientes APIs:

- Java Naming and Directory Interface (JNDI)
- Java Database Connectivity (JDBC)
- Java Transaction API (JTA)

Las secciones siguientes explican estos servicios más detalladamente.

JNDI

JNDI es un API estándar de Java que permite a las aplicaciones buscar un objeto por su nombre. El servidor WebLogic une los objetos Java que sirve a un nombre en un árbol de nombres. Una aplicación puede buscar objetos, como objetos RMI, JavaBeans Enterprise, colas JMS, y JDBC DataSources, obteniendo un contexto JNDI del servidor WebLogic y después llamando el método de operaciones de búsqueda JNDI con el nombre del objeto. Las operaciones de búsqueda devuelven una referencia al servidor de objetos WebLogic.

WebLogic JNDI soporta el balance de carga del cluster WebLogic. Todo servidor WebLogic en un cluster publica los objetos que sirve en un árbol de nombres replicado en un amplio cluster. Una aplicación puede obtener un contexto inicial JNDI del servidor fromany WebLogic en el cluster, realizar operaciones de búsqueda, y recibir una referencia del objeto desde cualquier servidor del cluster WebLogic que sirve el objeto. Se utiliza un algoritmo de balance de carga configurable separar la carga de trabajo entre los servidores del cluster.

JDBC

La conectividad de la base de datos JDBC Java (JDBC) proporciona acceso a los recursos backend de base de datos. Las aplicaciones Java tienen acceso a JDBC usando un driver JDBC, que es un interface específico del vendedor de la base de datos para un servidor de base de datos. Aunque cualquier aplicación Java puede cargar un driver JDBC, conectar con la base de datos, y realizar operaciones de base de datos, el servidor WebLogic proporciona una ventaja de rendimiento significativa ofreciendo almacenes de conexión JDBC.

Un almacen de conexiones JDBC es un grupo se conexiones JDBC con nombre manejadas a través del servidor WebLogic. En el momento de arranque el servidor WebLogic abre conexiones JDBC y las agrega al almacen. Cuando una aplicación requiere una conexión JDBC, consigue una conexión del almacen, la utiliza, y luego la devulve al almacen para su uso por otras aplicaciones. Establecer una conexión con la base de datos es a menudo una operación que consume mucho tiempo, y recrusos, un almacen de conexiones, que limita el número de operaciones de la conexión, mejora su funcionamiento.

Para registar un almacen de conexiones en el árbol de nombrado JNDIl, definimos un objeto DataSource para él. Las aplicaciones clientes Java pueden entonces conseguir una conexión del almacen realizando operaciones de búsqueda JNDI con el nombre del DataSource.

las clases de Java del lado del Servidor utilizan el driver de conexiones JDBC de WebLogic JDBC, que es un driver genérico de JDBC que llama a través al driver específico del vendedr al driver JDBC. Este mecanismo hace el código de la aplicación más portable, incluso si cambiamos la marca de la base de datos usada en la grada backend.

El driver JDBC del lado del cliente es el driver de WebLogic JDBC/RMI, que es un interface RMI al driver del alamacen. Utilizamos este driver de la misma manera que utilizamos cualquier driver JDBC estándar. Cuando se utiliza el driver JDBC/RMI, los programas Java pueden tener acceso a JDBC de una manera consistente con otros objetos distribuidos en el servidor WebLogic, y pueden mantener las estructuras de datos de la base de datos en la capa media.

WebLogic EJB y WebLogic JMS tatan con conexiones de un almacen de conexiones JDBC para cargar y salvar objetos persistentes. Usando EJB y JMS, podemos conseguir a menudo una abstracción más útil de la que podemos obtener usando JDBC directamente en una aplicación. Por ejemplo, usar un bean enterprise para representar un objeto de datos permite que cambiemmos el almacén subyacente más adelante sin la modificación del código JDBC. Si utilizamos mensajes persistentes JMS en vez de operaciones de base de datos con JDBC, será más fácil adaptar la aplicación a un sistema de mensajería de una tercera persona más adelante.

JTA

El API de transacción de Java (JTA) es el interface estándar para el manejo de transacciones en las aplicaciones Java. Usando transacciones, podemos proteger la integridad de los datos en nuestras bases de datos y manejar el acceso a esos datos por aplicaciones o ejemplares simultáneos de la aplicación. Una vez que una transacción comienza, todas las operaciones transaccionales deben terminar con éxito o todas se deben deshacer.

El servidor WebLogic utiliza las transacciones que incluyen operaciones EJB, JMS, y JDBC. Las transacciones distribuidas, coordinadas con dos fases, pueden espandir múltiples bases de datos que son accedidas con los drivers XA-compliant JDBC, como BEA WebLogic jDriver para Oracle/XA.

La especificación EJB define transaciones controladas por el bean y por el contenedor. Cuando se desarrolla un bean con transaciones controladas por el contenedor, el servidor WebLogic coordina la transacción automáticamente. Si se despliega un bean enterprise con transacciones manejadas por el bean, el programador de EJB debe proporcionar un código de transacción.

El código de aplicación basado en los APIs JMS o JDBC puede iniciar una transacción, o participar en una transacción comenzada anteriormente. Un solo contexto de transacción se asocia con el thread de ejecución de WebLogic Server, que ejecuta una aplicación; todas las operaciones transaccionales realizadas en el thread participan en la transacción actual.

Tecnologías de Mensajería

Las tecnologías de mensajería J2EE proporcionan un APIs estándar que las aplicaciones del servidor WebLogic pueden utilizar para comunicarse con una otra, así como con aplicaciones de servidores no-WebLogic. Los servicios de mensajería incluyen los siguientes APIs:

Java Message Service (JMS)

JavaMail

Las secciones siguientes describen estos APIs con más detalle:

JMS

El servicio de mensajería JMS de Java (JMS) permite a las aplicaciones comunicarse unas con otra intercambiando mensajes. Un mensaje es una petición, un informe, y/o un evento que contiene la información necesaria para coordinar la comunicación entre diversas aplicaciones. Un mensaje proporciona un nivel de abstracción, permitiendo que separemos los detalles sobre el sistema de destino del código de la aplicación.

WebLogic JMS implementa dos modelos de mensajería: punto-a-punto (PTP) y publish/subscribe (pub/sub). El modelo PTP permite que cualquier número de remitentes envíe mensajes a una cola. Cada mensaje en la cola se entrega a un solo programa de lectura. El modelo pub/sub permite que cualquier número de remitentes envíe mensajes a un Topic. Cada mensaje en el Topic se envía a todos los programas de lectura con una suscripción al Topic. Los mensajes se pueden entregar a los programas de lectura síncrona o asíncronamente.

Los mensajes JMS pueden ser persistentes o no-persistentes. Los mensajes persistentes se salvan en una base de datos y no se pierden si se rearranca el servidor WebLogic. Los mensajes no-persistentes se pierden si se rearranca el servidor WebLogic. Los mensajes persistentes enviados a un Topic pueden conservarse hasta que todos los suscriptores interesados los hayan recibido.

JMS soporta varios tipos de mensaje que sonn útiles para diversos tipos de aplicaciones. El cuerpo de mensaje puede contener los texto arbitrario, secuencias del bytes, tipos de datos primitivos de Java, parejas de nombre/valor, objetos serializables Java, o contenido XML.

Javamail

El servidor JavaMail WebLogic incluye implementación de referencia del Sun JavaMail. JavaMail permite que una aplicación cree mensajes de E-mail y los envíe a través de un servidor SMTP a la red.

WebLogic Server como Servidor Web

El servidor WebLogic se puede utilizar como el servidor web primario para aplicaciones web avanzadas. Una aplicación Web de J2EE es una colección de páginas HTML o XML, de páginas JSP, de servlets, de clases Java, de applets, de imágenes, de ficheros multimedia, y de otros tipos de ficheros.

Cómo funciona el servidor WebLogic como un Servidor Web

Una aplicación Web se ejecuta en el contenedor Web de un servidor web. En un entorno de servidor WebLogic, un servidor web es una entidad lógica, desplegada en uno o más servidores WebLogic en un cluster.

Los ficheros de una aplicación Web se graban en una estructura de directorios que, opcionalmente, puede empaquetarse en un solo fichero .war (Web ARchive) usando la utilidad jar de Java. Un conjunto de descriptores de despliegue XML definen los componentes y los parámetros de ejecución de una aplicación, como las configuraciones de seguridad. Los descriptores de despliegue permiten cambiar comportamientos durante la ejecución sin cambiar el contenido de los componentes de la aplicación Web, y hacen fácil desplegar la misma aplicación en varios servidores Web.

Características del Servidor Web

Cuando se usa como un servidor web, WebLogic Server soporta las siguientes funcionalidades:

- Hosting Virtual.
- Soporte para configuraciones de servidores proxy
- Balance de Cargas
- Control de fallos

Esta sección describe cómo es soportada cada una de estas funciones por WebLogic Server.

Hosting Virtual

WebLogic Server soporta almacenamiento virtual, un arreglo que permite a un solo servidor WebLogic o a un Cluster WebLogic contener varios sitos Web. Cada servidor web virtual tiene su propio nombre de host, pero todos los servidores Web están mapeados en la DNS de la misma dirección IP del cluster. Cuando un cliente envía una petición HTTP a la dirección del cluster, se selecciona un servidor WebLogic para servir la petición. El nombre del servidor web se extrae de la cabecera de la petición HTTP y se mantiene en subsecuenes intercambios con el cliente para que el hostname virtual permanezca constante desde la perspectiva del cliente. Múltiples aplicaciones Web pueden desplegarse en un servidor WebLogic, y cada aplicación Web se puede mapear a un host virtual.

Usar Configuraciones de Servidor Proxy

WebLogic server se puede integrar con los servidores web existentes. Las peticiones pueden ser almacenadas desde un servidor WebLogic a otro servidor web o, usando un plug-in nativo provisto del servidor WebLogic, desde otro servidor web al servidor WebLogic. BEA proprociona los plug-ins para Apache Web Server, Netscape Enterprise Server, Microsoft Internet Information Server.

El uso de los servidores proxys entre clientes y un conjunto de servidores independientes WebLogic o de un cluster WebLogic permite realizar el balance de carga y el control de fallos para las peticiones Web. Para el cliente, solo parecerá un servidor web.

Balance de Carga

Podemos instalar varios servidores WebLogic detrás de un servidor proxy para acomodar grandes volúmenes de peticiones. El servidor proxy realiza el balance de cargas, distribuyendo las peticiones a través de los distintos servidores en la capa que hay detrás de él.

El servidor proxy puede ser un servidor WebLogic, o puede ser un servidor Apache, Netscape, o Microsoft. El servidor WebLogic incluye los plugs-in de código nativo para algunas plataformas que permitan estos servidores web de terceras partes a las peticiones del servidor proxy de WebLogic.

El servidor proxy se configura para redirigir ciertos tipos de peticiones a los servidores que hay detrás de él. Por ejemplo, un arreglo común es configurar el servidor proxy para manejar las peticiones para páginas HTML estáticas y redirigir los pedidos de servlets y páginas JSP a clusters WebLogic detrás del proxy.

Control de Fallos

Cuando un cliente web empieza una sesión servlet, el servidor proxy podría enviar las peticiones subsecuentes que son parte de la misma sesión a un servidor WebLogic distinto. El servidor WebLogic proporciona replicación de la sesión para asegurarse de que el estado de la sesión del cliente sigue estando disponible.

Hay dos tipos de réplica de sesión:

- Se puede usar la réplica de sesión JDBC con un cluster WebLogic o con un conjunto de servidores WebLogic independientes. No requiere la opción que CLustering del WebLogic Server.
- La réplica de sesión en-memoria requiere la opción de Clustering del WebLogic Server.
- La réplica de sesión JDBC escribe datos de la sesión en una base de datos. Una vez que se haya comenzado una sesión, cualquier servidor WebLogic que seleccione el servidor proxy puede continuar la sesión recuperando los datos de la sesión desde la base de datos.

Cuando se despliega un Cluster WebLogic detrás de un servidor proxy, las sesiones de servlets se pueden replicar sobre la red a un servidor WebLogic secundario seleccionado por el cluster, para evitar la necesidad de acceder a la base de datos. La replicación en-memoria usa menos recursos y es mucho más rápida que la replicación de sesión JDBC, por eso es la mejor forma para proporcionar control de fallos para servlets cuando tenemos un Cluster WebLogic.

Servicios de Seguridad

WebLogic Server proporciona seguridad para las aplicaciones a través de un "security realm" (reino de seguridad). Un reino de la seguridad proporciona acceso a dos servicios:

- Un servicio de autentificación, que permite que el servidor WebLogic verifique la identidad de los usuarios.
- Un servicio de autorización, que controla el acceso de los usuarios a las aplicaciones.

Autentificación

Un reino tiene acceso a un almacén de usuarios y de grupos y puede autentificar a un usuario comprobando una credencial suminstrada por el usuario (generalmente una password) contra el nombre de usuario y la credencial en el almacén de seguridad. Los navegadores Web utilizan la autentificación solicitando un nombre de usuario y una password cuando un cliente web intenta acceder a un servicio protegido del servidor WebLogic. Otros clientes del servidor WebLogic proporcionan nombres de usuarios y credenciales programaticamente cuando establecen conexiones con el servidor WebLogic.

Autorización

Los servicios de WebLogic Server se protegen con listas del control de acceso (ACLs). Una ACL es una lista de usuarios y grupos que están autorizados para acceder a un servicio. Una vez que se haya autentificado a un usuario, el servidor WebLogic consulta la ACL de un servicio antes de permitir que el usuario tenga acceso al servicio.

Introducción al Servidor de Aplicaciones Tomcat de Apache

Introducción

Tomcat es un contenedor de Servlets con un entorno JSP. Un contenedor de Servlets es un shell de ejecución que maneja e invoca servlets por cuenta del usuario.

Podemos dividir los contenedores de Servlets en:

1. Contenedores de Servlets Stand-alone (Independientes)
   Estos son una parte integral del servidor web. Este es el caso cuando usando un servidor web basado en Java, por ejemplo, el contenedor de servlets es parte de JavaWebServer (actualmente sustituido por iPlanet). Este el modo por defecto usado por Tomcat.
   Sin embargo, la mayoría de los servidores, no están basados en Java, los que nos lleva los dos siguientes tipos de contenedores:
2. Contenedores de Servlets dentro-de-Proceso
   El contenedor Servlet es una combinación de un plugin para el servidor web y una implementación de contenedor Java. El plugind del servidor web abre una JVM (Máquina Virtual Java) dentro del espacio de direcciones del servidor web y permite que el contenedor Java se ejecute en él. Si una cierta petición debería ejecutar un servlet, el plugin toma el control sobre la petición y lo pasa al contenedor Java (usando JNI). Un contenedor de este tipo es adecuado para servidores multi-thread de un sólo proceso y proporciona un buen rendimiento pero está limitado en escalabilidad
3. Contenedores de Servlets fuera-de-proceso
   El contenedor Servlet es una combinación de un plugin para el servidor web y una implementación de contenedor Java que se ejecuta en una JVM fuera del servidor web. El plugin del servidor web y el JVM del contenedor Java se comunican usando algún mecanismo IPC (normalmente sockets TCP/IP). Si una cierta petición debería ejecutar un servlet, el plugin toma el control sobre la petición y lo pasa al contenedor Java (usando IPCs). El tiempo de respuesta en este tipo de contenedores no es tan bueno como el anterior, pero obtiene mejores rendimientos en otras cosas (escalabilidad, estabilidad, etc.).

Tomcat puede utilizarse como un contenedor solitario (principalmente para desarrollo y depuración) o como plugin para un servidor web existente (actualmente se soporan los servidores Apache, IIS y Netscape). Esto significa que siempre que despleguemos Tomcat tendremos que decidir cómo usarlo, y, si seleccionamos las opciones 2 o 3, también necesitaremos instalar un adaptador de servidor web

Arrancar y Parar Tomcat

Arrancamos y paramos Tomcat usando los scripts que hay en el directorio bin:

Para arrancar Tomcat ejecutamos:

- Sobre UNIX:
- bin/startup.sh
- Sobre Win32:
- bin\startup

Para parar Tomcat ejecutamos:

- Sobre UNIX:

- bin/shutdown.sh
- Sobre Win32:
- bin\shutdown

La Estructura de Directorios de Tomcat

Asumiendo que hemos descomprimido la distribución binaria de Tomcat deberíamos tener la siguiente estructura de directorios:

Nombre de Directorio	Descripción
bin	Contiene los scripts de arrancar/parar
conf	Contiene varios ficheros de configuración incluyendo server.xml (el fichero de configuración principal de Tomcat) y web.xml que configura los valores por defecto para las distintas aplicaciones desplegadas en Tomcat.
doc	Contiene varia documentación sobre Tomcat (Este manual, en Inglés).
lib	Contiene varios ficheros jar que son utilizados por Tomcat. Sobre UNIX, cualquier fichero de este directorio se añade al classpath de Tomcat.
logs	Aquí es donde Tomcat sitúa los ficheros de diario.
src	Los ficheros fuentes del API Servlet. ¡No te excites, todavía! Estoa son sólo los interfaces vacíos y las clases abstractas que debería implementar cualquier contenedor de servlets.
webapps	Contiene aplicaciones Web de Ejemplo.

Adicionalmente podemos, o Tomcat creará, los siguientes directorios:

Nombre de Directorio	Descripción
work	Generado automáticamente por Tomcat, este es el sitio donde Tomcat sitúa los ficheros intermedios (como las páginas JSP compiladas) durante su trabajo. Si borramos este directorio mientras se está ejecutando Tomcat no podremos ejecutar páginas JSP.
classes	Podemos crear este directorio para añadir clases adicionales al classpath. Cualquier clase que añadamos a este directorio encontrará un lugar en el classpath de Tomcat.

Los Scripts de Tomcat

Tomcat es un programa Java, y por lo tanto es posible ejcutarlo desde la línea de comandos, después de configuar varias variables de entorno. Sin embargo, configurar cada variable de entorno y seguir los parámetros de la línea de comandos usados por Tomcat es tedioso y propenso a errores. En su lugar, el equipo de desarrollo de Tomcat proporciona unos pocos scripts para arrancar y parar Tomcat fácilmente.

Nota: Los scripts son sólo una forma conveniente de arrancar/parar... Podemos modificarlos para personalizar el CLASSPATH, las variables de entorno como PATH y LD_LIBRARY_PATH, etc., mientras que se genera la línea de comandos correcta para Tomcat.

¿Qué son esos scripts? La siguiente tabla presenta los scripts más importantes para el usuario común:

Nombre del Script	Descripción
tomcat	El script principal. Configura el entorno apropiado, incluyendo CLASSPATH, TOMCAT_HOME y JAVA_HOME, y arranca Tomcat con los parámetros de la línea de comando apropiados.
startup	Arrancar tomcat en segundo plano. Acceso directo para tomcat start
shutdown	Para tomcat (lo apaga). Acceso directo para tomcat stop;

El script más importante para los usuarios es tomcat (tomcat.sh/tomcat.bat). Los otros scripts relacionados con tomcat sirven como un punto de entrada simplificado a una sola tarea (configuran diferentes parámetros de la línea de comandos, etc.).

Una mirada más cercana a tomcat.sh/tomcat.bat nos muestra que realiza las siguientes acciones:

Sistema Operativo	Acciones
Unix	• Averigua donde está TOMCAT_HOME si no se especifica. • Averigua donde está JAVA_HOME si no se especifica. • Configura un CLASSPATH que contiene - • El directorio ${TOMCAT_HOME}/classes (si exsiste). • Todo el contenido de ${TOMCAT_HOME}/lib. • ${JAVA_HOME}/lib/tools.jar (este fichero jar contine la herramienta javac, que necesitamos para compilar los ficheros JSP). • Ejecuta java con los parámetros de la línea de comandos que ha configurado un entorno de sistema Java, llamado tomcat.home, con org.apache.tomcat.startup.Tomcat como la clase de arranque. También procesa los parámetros de la línea de comandos para org.apache.tomcat.startup.Tomcat, como: • La operación a ejecutar start/stop/run/etc. • Un path al fichero server.xml usado por este proceso Tomcat. • Por ejemplo, si server.xml está localizado en /etc/server_1.xml y el usuario quiere arrancar Tomcat en segundo plano, debería introducir la siguiente línea de comandos: • bin/tomcat.sh start -f /etc/server_1.xml
Win32	• Graba las configuraciones actuales para TOMCAT_HOME y CLASSPATH. • Prueba JAVA_HOME para asegurarse de que está configurado. • Prueba si TOMCAT_HOME está configurado y los valores por defecto a "." no lo están. Entonces se usa TOMCAT_HOME para probar la existencia de servlet.jar para asegurarse de que TOMCAT_HOME es válido. • Configura la varibale CLASSPATH que contiene - • %TOMCAT_HOME%\classes (incluso si no existe), • Los ficheros Jar de %TOMCAT_HOME%\lib. Si es posible, todos los ficheros jar en %TOMCAT_HOME%\lib sin incluidos dinámicamente. Si no es posible, se incluyen estáticamente los siguientes ficheros jar: ant.jar, jasper.jar, jaxp.jar, parser.jar, servlet.jar, y webserver.jar • %JAVA_HOME%\lib\tools.jar, si existe (este fichero jar contiene la

herramietna javac necesaria para compilar los ficheros JSP).

- Ejecuta %JAVA_HOME%\bin\java, con los parámetros de la línea de comandos que configuran el entorno de sistema Java, llamado tomcat.home, con org.apache.tomcat.startup.Tomcat como la clase de arranque. También le pasa los parámetros de la líena de comandos a org.apache.tomcat.startup.Tomcat, como:
- La operación a realizar: start/stop/run/etc.
- Un path al fichero server.xml usado por este proceso Tomcat.
- Por ejemplo, si server.xml está localizado en conf\server_1.xml y el usuario quiere arrancar Tomcat en una nueva ventana, debería proporcionar la siguiente línea de comando:
- bin\tomcat.bat start -f conf\server_1.xml
- Restaura las configuraciones de TOMCAT_HOME y CLASSPATH grabadas préviamente.

Como podemos ver, la versión Win32 de tomcat.bat no es tán robusta como la de UNIX. Espnnnte, no se averigua los valores de JAVA_HOME y sólo intenta "." como averiguación de TOMCAT_HOME. Puede construir el CLASSPATH dinámicamente, pero no en todos los casos. No puede construir el CLASSPATH dinámincamente si TOMCAT_HOME contiene espacios, o sobre Win9x, si TOMCAT_HOME contiene nombres de directorios que no son 8.3 caracteres.

Ficheros de Configuración de Tomcat

La configuración de Tomcat se basa en dos ficheros:

`server.xml` - El fichero de configuración golbal de Tomcat.

`web.xml` - Configura los distintos contextos en Tomcat.

Esta sección trata la forma de utilizar estos ficheros. No vamos a cubrir la interioridades de `web.xml`, esto se cubre en profundidad en la especificación del API Servlet. En su lugar cubriremos el contenido de `server.xml` y discutiremos el uso de `web.xml` en el contexto de Tomcat.

server.xml

`server.xml` es el fichero de configuración principal de Tomcat. Sirve para dos objetivos:

Proporcionar configuración inicial para los componentes de Tomcat.

Especifica la estructura de Tomcat, lo que significa, permitir que Tomcat arranque y se construya a sí mismo ejemplarizando los componentes especificados en `server.xml`.

Los elementos más importantes de `server.xml` se describen en la siguiente tabla:

Elemento	Descripción
`Server`	El elemento superior del fichero `server.xml`. Server define un servidor Tomcat. Generalmente no deberíamos tocarlo demasiado. Un elemento `Server` puede contener elementos `Logger` y `ContextManager`.
`Logger`	Este elemento define un objeto `logger`. Cada objeto de este

	tipo tiene un nombre que lo identifica, así como un path para el fichero log que contiene la salida y un `verbosityLevel` (que especifica el nivel de log). Actualmente hay loggeres para los servlets (donde va el `ServletContext.log()`), los ficheros JSP y el sistema de ejecución tomcat.
`ContextManager`	Un `ContextManager` especifica la configuración y la estructura para un conjunto de `ContextInterceptors`, `RequestInterceptors`, `Contexts` y sus `Connectors`. El `ContextManager` tiene unos pocos atributos que le proporcionamos con: Nivel de depuración usado para marcar los mensajes de depuración La localización base para `webapps/`, `conf/`, `logs/` y todos los contextos definidos. Se usa para arrancar Tomcat desde un directorio distinto a TOMCAT_HOME. El nombre del directorio de trabajo. Se incluye una bandera para controlar el seguimiento de pila y otra información de depurado en las respuestas por defecto.
`ContextInterceptor & RequestInterceptor`	Estos interceptores escuchan ciertos eventos que sucenden en el `ContextManager`. Por ejemplo, el `ContextInterceptor` escucha los eventos de arrancada y parada de Tomcat, y `RequestInterceptor` mira las distintas fases por las que las peticiones de usuario necesitan pasar durante su servicio. El administrador de Tomcat no necesita conocer mucho sobre los interceptores; por otro lado, un desarrollador debería conocer que éste es un tipo **global** de operaciones que pueden implementarse en Tomcat (por ejemplo, loggin de seguridad por petición).
`Connector`	El `Connector` representa una conexión al usuario, a través de un servidor Web o directamente al navegador del usuario (en una configuración independiente). El objeto `connector` es el responsable del control de los threads en Tomcat y de leer/escribir las peticiones/respuestas desde los sockets conectados a los distintos clientes. La configuración de los conectores incluye información como: La clase `handler`. El `puerto` TCP/IP donde escucha el controlador. el `backlog` TCP/IP para el server socket del controlador. Describiremos cómo se usa esta configuración de conector más adelante.
`Context`	Cada Context representa un path en el árbol de tomcat donde situanos nuestra aplicación web. Un `Context` Tomcat tiene la siguiente configuración: El path donde se localiza el contexto. Este puede ser un path

completo o relativo al home del `ContextManager`.

Nivel de depuración usado para los mensaje de depuración.

Una bandera `reloadable`. Cuando se desarrolla un servlet es muy conveniente tener que recargar el cambio en Tomcat, esto nos permite corregir errores y hacer que Tomcat pruebe el nuevo código sin tener que parar y arrancar. Para volver a recargar el servlet seleccionamos la bandera `reloadable` a true. Sin embargo, detectar los cambios consume tiempo; además, como el nuevo servlet se está cargando en un nuevo objeto **class-loader** hay algunos casos en los que esto lanza errores de forzado (cast). Para evitar estos problemas, podemos seleccionar la bandera `reloadable` a false, esto desactivará esta característica.

Se puede encontrar información adicional dentro del fichero `server.xml`.

Arrancar Tomcat dese Otros Directorio

Por defecto tomcat usará `TOMCAT_HOME/conf/server.xml` para su configuración. La configuración por defecto usará `TOMCAT_HOME` como la base para sus contextos.

Podemos cambiar esto usando la opción `-f /path/to/server.xml`, con un fichero de configuración diferente y configurando la propiedad home del controlador de contexto. Necesitamos configurar los ficheros requeridos dentro del directorio home:

Un directorio `webapps/` (si hemos creado uno) - todos los ficheros **war** se expanderán y todos sus subdirectorios se añadirán como contextos.

Directorio `conf/` - podemos almacenar `tomcat-users.xml` y otros ficheros de configuración.

`logs/` - todos los logs irán a este directorio en lugar de al principal `TOMCAT_HOME/logs/`.

`work/` - directorio de trabajo para los contextos.

Si la propiedad `ContextManager.home` de `server.xml` es relativa, será relativa al directorio de trabajo actual.

web.xml

Podemos encontar una detallada descripción de `web.xml` y la estructura de la aplicación web (incluyendo la estructura de directorios y su configuración) en los capítulo 9, 10 y 14 de la Servlet API Spec en la site de Sun Microsystems.

Hay una pequeña característica de Tomcat que está relacionada con `web.xml`. Tomcat permite al usuario definir los valores por defecto de `web.xml` para todos los contextos poniendo un fichero `web.xml` por defecto en el directorio `conf`. Cuando construimos un nuevo contexto, Tomcat usa el fichero `web.xml` por defecto como la configuración base y el fichero `web.xml` específico de la aplicación (el localizado en el `WEB-INF/web.xml` de la aplicación), sólo sobreescribe estos valores por defecto.

Configurar Tomcat para Cooperar con Apache Web Server

Hasta ahora no hemos explicado Tomcat como un plugin, en su lugar lo hemos considerado como un contenedor independiente y hemos explicado como usarlo. Sin embargo, hay algunos problemas:

Tomcat no es tan rápido como Apache cuando sirve páginas estáticas.

Tomcat no es tan configurable como Apache.

Tomcat no es tan robusto como Apache.

Hay mucho sites que llavan mucho tiempo de investigación sobre ciertos servidores web, por ejemplo, sites que usan scripts CGI o módulos perl o php... No podemos asumir que todos ellos quieran abandonar dichas tecnologías.

Por todas estas razones es recomendable que las sites del mundo real usen un servidor web, como Apache, para servir el contenido estático de la site, y usen Tomcat como un plugin para Servlets/JSP.

No vamos a cubrir las diferentes configuraciones en profundidad, en su lugar:

Cubriremos el comportamiento fundamental de un servidor web.

Explicaremos la configuración que necesitamos.

Demonstraremos esto sobre Apache.

Operación del Servidor Web

En resumidas cuentas un servidor web está esperando peticiones de un cliente HTTP. Cuando estas peticiones llegan el servidor hace lo que sea necesario para servir las peticiones proporcionando el contenido necesario. Añadirle un contenedor de servlets podría cambiar de alguna forma este comportamiento. Ahora el servidor Web también necesita realizar lo siguiente:

Cargar la librería del adaptador del contenedor de servlets e inicializarlo (antes de servir peticiones).

Cuando llega una petición, necesita chequear para ver si una cierta petición pertenece a un servlet, si es así necesita permitir que el adaptador tome el control y lo maneje.

Por otro lado el adaptador necesita saber qué peticiones va a servir, usualmente basándose en algún patrón de la URL requerida, y dónde dirigir estas peticiones.

Las cosas son incluso más complejas cuando el usuario quiere seleccionar una configuración que use hosts virtuales, o cuando quieren que múltiples desarrolladores trabajen en el mismo servidor web pero en distintos contenedores de Servlets. Cubriremos estos dos casos en las secciones avanzadas.

¿Cuál es la Configuración Necesaria

La configuración más óbvia en la que uno puede pensar es en la identidad de las URLs servlet que están bajo la responsabilidad del contenedor de servlets. Esto está claro, alguién debe conocer qué peticiones transmitir al cotenedor de servlets...
Todavía hay algunos ítems de configuración adicionales que deberíamos proporcionar a la combinación `web-server/servlet-container`:

Necesitamos proporcionar la configuración sobre los procesos Tomcat disponibles y sobre los puertos/host TCP/IP sobre los que éstos están escuchando.

Necesitamos decirle al servidor web la localización de la librería adaptador (para que pueda cargarla en la arrancada).

Necesitamos seleccionar la información interna del adaptador sobre cuando log guardar, etc.

Toda esta información debe aparecer en el fichero de configuración del servidor web, o en un fichero de configuración privado usado por el adaptador. La siguiente sección explicará cómo se puede implementar esta configuración en Apache.

Haciéndolo en Apache

Esta sección nos enseña como configurar Apache para que trabaje con Tomcat; intenta proporcionar explicaciones sobre las directivas de configuración que deberíamos usar. Podemos encontrar información adicional en la página http://java.apache.org/jserv/install/index.html.

Cuando Tomcat arranque generá automáticamente un fichero de configuración para apache en `TOMCAT_HOME/conf/tomcat-apache.conf`. La mayoría de las veces no necesitaremos hacer nada más que incluir es fichero (añadir `Include TOMCAT_HOME/conf/tomcat-apache.conf`) en nuestro fichero **httpd.conf**. Si tenemos necesidades especiales, por ejemplo un puerto AJP distinto de 8007, podemos usar este fichero como base para nuestra configuración personalizada y grabar los resultados en otro fichero. Si manejamos nosotros mismos la configuración de Apache necesitaremos actualizarlo siempre que añadamos un nuevo contexto.

Tomcat: debemos re-arrancar tomcat y apache después de añadir un nuevo contexto; Apache no soporta cambios en su configuración sin re-arrancar. Cuando tomcat arranca, también se genera el fichero `TOMCAT_HOME/conf/tomcat-apache.conf` cuando arrancar tomcat, por eso necesitamos arrancar Tomcat antes que Apache. Tomcat sobreescribirá `TOMCAT_HOME/conf/tomcat-apache.conf` cada arrancada para que se mantenga la configuración peronalizada.

La configuración Apache-Tomcat usa las directivas de configuración principales de Apache así como directivas únicas de Jserv por eso podría ser confuso al principio, sin embargo hay dos cosas que lo simplifican:

En general podemos distinguir dos **familias** de directivas porque las directivas únicas de jserv empiezan con un prefijo `ApJServ`.

Toda la configuración relacionada con Tomcat está concentrada en un sólo fichero de configuración llamado **tomcat.conf**, el automáticamente generado **tomcat-apache.conf**, por eso podemos mirar en un sólo fichero.

Veamos ahora un simple fichero **tomcat.conf**:

```
############################################################
#      A minimalistic Apache-Tomcat Configuration File     #
############################################################

# Note: this file should be appended or included into your httpd.conf

# (1) Loading the jserv module that serves as Tomcat's apache adapter.
LoadModule jserv_module libexec/mod_jserv.so
```

```
# (1a) Module dependent configuration.
<IfModule mod_jserv.c>

# (2) Meaning, Apache will not try to start Tomcat.
ApJServManual on
# (2a) Meaning, secure communication is off
ApJServSecretKey DISABLED
# (2b) Meaning, when virtual hosts are used, copy the mount
# points from the base server
ApJServMountCopy on
# (2c) Log level for the jserv module.
ApJServLogLevel notice

# (3) Meaning, the default communication protocol is ajpv12
ApJServDefaultProtocol ajpv12
# (3a) Default location for the Tomcat connectors.
# Located on the same host and on port 8007
ApJServDefaultHost localhost
ApJServDefaultPort 8007

# (4)
ApJServMount /examples /root
# Full URL mount
# ApJServMount /examples ajpv12://hostname:port/root
</IfModule>
```

Como podemos ver el proceso de configuración está dividio en 4 pasos que explicaremos ahora:

En este paso instruimos a Apache para que carque el objeto compartido **jserv** (o la librería **dll** en NT). Esta es una directiva familiar de Apache. Si la carga fue bien y el módulo vino de un fichero llamado `mod_jserv.c (1a)` podemos arrancar con el resto de la configuración Jserv-Tomcat.

Este paso configura varios parámetros internos de **Jserv**, estos parámetros son:

Instruye a jserv para que no arranque el proceso Tomcat. Esto no está implementado todavía.

Desactiva la clave secreta `challenge/response` entre Apache y Tomcat. De nuevo, esto tampo está implementado aún.

Instruye a jserv para que copie el punto de montaje del servidor base (ver siguiente seccion) en caso de hosting virtual

Instruye a jserv para usar el nivel de log de noticia. Otros niveles de log incluidos son: emerg, alert, crit, error, warn, info y debug.

Este paso configura los parámetros de comunicación por defecto. Básicamente dice que el protocolo por defecto utilizado para la comunicación es **ajpv12** que el proceso Tomcat se ejecuta sobre la misma máquina y que escucha en el puerto 8807. Si ejecutamos Tomcat en una máquina distinta a las usada por Apache deberíamos actualizar `ApJServDefaultHost` o usar una URL completa cuando montemos los contextos. Tambien, si configuramos los conectores Tomcat para usar un puerto distinto al 8007, deberíamos actualizar `ApJServDefaultPort` o usar una URL completa cuando montemos los contextos.

Este paso monta un contexto para Tomcat. Básicamente dice que todos los paths del servidor web que empiecen con `/example` irán a Tomcat. Este ejemplo `ApJServMount` es uno muy simple, de hecho, `ApJServMount` también puede proporcionar información sobre el protocolo de comunicación usado y la localización donde está escuchando el proceso Tomcat, por ejemplo:

1. `ApJServMount /examples ajpv12://hostname:port/root`

   monta el contexto `/examples` en un proceso tomcat que se está ejecutando en el host `hostname` y que escucha en el puerto número `port`.

Ahora que hemos entendido las diferentes instrucciones de configuración en el fichero de ejemplo, ¿cómo podríamos añadirla a la configuración de Apache? Un método sencillo es escribir su contenido en **httpd.conf** (el fichero de configuración de Apache), sin embargo, esto puede ser muy desordenado. En su lugar deberíamos usar la directiva **include** de apache. Al final de fichero de configuración de Apache (**httpd.conf**) añadimos la siguiente directiva:

```
include <full path to the Tomcat configuration file>
```

Por ejemplo:

```
include /tome/tomcat/conf/tomcat.conf
```

Esto añadirá nuestra configuración de Tomcat a apache, después de haber copiado el módulo jserv al directorio libexec de Apache (o modules en caso de Win32) y re-arrancar (parar+arrancar) Apache, deberíamos poder conectar con Tomcat.

Obtener el Módulo Jserv (mod_jserv)

Como vimos anteriormente, necesitamos un adaptador de servidor Web para situarlo en Apache y redirigir las peticiones a Tomcat. Para Apache, este adaptador es una versión ligeramente modificada de **mod_jserv**.

Podríamos intentar buscarlo en http://jakarta.apache.org/downloads/binindex.html para ver si hay una versión pre-construida de **mod_jserv** que corresponda con nuestro sistema operativo (Normalmente hay uno para NT), sin embargo, siendo una librería nativa, no deberíamos esperar que esté ya (demasiados sistemas operativos, pocos desarrolladores, la vida es muy corta...) Además, pequeñas variaciones en la forma de construir la variante UNIX de Apache podrían resultar en errores de enlaces dinámicos. Realmente deberíamos intentar construir **mod_jserv** para nuestro sistema (no te asustes, no es tan dificil!).

Construir **mod_jserv** sobre **UNIX**:

Descargar la distribución fuente de Tomcat desde http://jakarta.apache.org/downloads/sourceindex.html.

Descomprimirla en algún directorio.

Construir el módulo:

Mover el directorio a `jakarta-tomcat/src/native/apache/jserv/`

Ejcutar el comando para construirlo:

- **`apxs -c -o mod_jserv.so *.c`**

  `apxs` es parte de la distribución de Apache y debería estar localizado en nuestro `APACHE_HOME/bin`.

Construir **mod_jserv** para **Win32** no es tan sencillo (ya tenemos una dll descargable para Win32). Pero si todavía queremos construirlo deberíamos instalar visual C++ y realizar las siguientes tareas:

Descargar la distribución fuente de Tomcat desde http://jakarta.apache.org/downloads/sourceindex.html.

Descomprimirla en algún directorio.

Construir el módulo:

Mover el directorio a `jakarta-tomcat\src\native\apache\jserv`

Añadir Visual C++ a nuestro entorno ejecutando el script **VCVARS32.BAT**.

Configurar una variable de entorno llamada `APACHE_SRC` que apunte al directorio source de Apache, es decir `SET APACHE_SRC=C:\Program Files\Apache Group\Apache\src`. Observa que el fichero make espera enlazar con `CoreR\ApacheCore.lib` bajo el directorio `APACHE_SRC`. Puedes ver la documentación de Apache para construir ApacheCore.

Ejecutamos el comando para construir:

- **`nmake -f Makefile.win32`**

`nmake` es el programa `make` de Visual C++.

Esto es todo!, ya hemos construido **mod_jserv**...

Hacer que Apache sirva los Ficheros Estáticos del Contexto

El fichero anterior de configuración de Apache-Tomcat era de alguna forma ineficiente, instruye a Apache a enviar cualquier petición que empiece con el prefijo `/examples` para que sea servida por Tomcat. ¿Realmente queremos hacer eso? Hay muchos ficheros estáticos que podrían ser parte de nuestro contexto servlet (por ejemplo imágenes y HTML estático), ¿Por qué debería Tomcat servir esos ficheros?

Realmente tenemos razones para hacer esto, por ejemplo:

Podríamos querer configurar Tomcat basándonos en la seguridad para esos recursos.

Podríamos querer seguir las peticiones de usuarios de recursos estáticos usando interceptores.

En general, sin embargo, este no es ese caso; hacer que Tomcat sirva el contenido estático es sólo malgastar CPU. Deberíamos hacer que Apache sirviera estos ficheros dinámicos y no Tomcat.

Hacer que Apache sirva los ficheros estáticos requiere los siguientes pasos:

Instruir a Apache para que envíe todas la peticiones servlet a Tomcat

Instruir a Apache para que envíe todas las peticiones JSP a Tomcat.

y dejar que Apache maneje el resto. Echemos un vistazo a un fichero de ejemplo **tomcat.conf** que hace exactamente esto:

```
##########################################################################
#               Apache-Tomcat Smart Context Redirection                  #
##########################################################################
LoadModule jserv_module modules/ApacheModuleJServ.dll
```

```
<IfModule mod_jserv.c>
ApJServManual on
ApJServDefaultProtocol ajpv12
ApJServSecretKey DISABLED
ApJServMountCopy on
ApJServLogLevel notice

ApJServDefaultHost localhost
ApJServDefaultPort 8007

#
# Mounting a single smart context:
#
# (1) Make Apache know about the context location.
Alias /examples c:/jakarta-tomcat/webapps/examples
# (2) Optional, customize Apache context service.
<Directory "c:/jakarta-tomcat/webapps/examples">
    Options Indexes FollowSymLinks
# (2a) No directory indexing for the context root.
#    Options -Indexes
# (2b) Set index.jsp to be the directory index file.
#    DirectoryIndex index.jsp
</Directory>
# (3) Protect the WEB-INF directory from tampering.
<Location /examples/WEB-INF/>
    AllowOverride None
    deny from all
</Location>
# (4) Instructing Apache to send all the .jsp files under the context
to the
# jserv servlet handler.
<LocationMatch /examples/*.jsp>
    SetHandler jserv-servlet
</LocationMatch>
# (5) Direct known servlet URLs to Tomcat.
ApJServMount /examples/servlet /examples

# (6) Optional, direct servlet only contexts to Tomcat.
ApJServMount /servlet /ROOT
</IfModule>
```

Como podemos ver, el inicio de este fichero de configuración es el mismo que vimos en el ejemplo anterior. Sin embargo, el último paso (montar el contexto), ha sido reemplazado por una larga serie de directivas de configuración de Apache y ApJServ que ahora explicaremos:

Este paso informa a Apache de la localización del contexto y los asigna a un directorio virtual de Apache. De esta forma Apache puede servir ficheros de este directorio.

Este paso opcional instruye a Apache sobre cómo servir el contexto; por ejemplo podemos decidir si Apache permitirá indexar (listar) el directorio o seleccionar un fichero de indice especial.

Este paso instruye a Apache para proteger el directorio `WEB-INF` de los accesos del cliente. Por razones de seguridad es importante evitar que los visitante vean el contenido del directorio `WEB-INF`, por eemplo `web.xml` podría proporcionar información valiosa a los intrusos. Este paso bloquea el contenido de `WEB-INF` para los visitiantes.

Este paso instruye a Apache para que sirva todas las localizaciones JSP dentro del contexto usando el manejador de servlets **jserv**. El manejador de servlet redirige estas peticiones basándose en el host y puerto por defecto.

Este paso monta las URLs específicas de servelts en Tomcat. Deberíamos observar que deberíamos tener tantas directivas como el número de URLs de servlets especificados.

Este último paso es un ejemplo de adición de un único contexto servlet a Tomcat.

Es facil ver que este fichero de configuración es mucho más complejo y propenso a errores que el primer ejemplo, sin embargo es el precio que debemos pagar (por ahora) para mejorar el rendimiento.

Configurar Varias JVMs Tomcat

Algunas veces es útil tener diferentes contextos manejados por diferentes JVMs (Máquinas Virtuales Java), por ejemplo:

Cuando cada contexto sirve a una tarea específica y diferente y se ejecuta sobre una máquina distinta.

Cuando queremos tener varios desarrolladores trabajando en un proceso Tomcat privado pero usando el mismo servidor web

Implementar dichos esquemas donde diferentes contextos son servidos por diferentes JVMs es muy fácil y el siguiente fichero de configuración lo demuestra:

```
##############################################################################
#             Apache-Tomcat with JVM per Context                             #
##############################################################################
LoadModule jserv_module modules/ApacheModuleJServ.dll
<IfModule mod_jserv.c>
ApJServManual on
ApJServDefaultProtocol ajpv12
ApJServSecretKey DISABLED
ApJServMountCopy on
ApJServLogLevel notice

ApJServDefaultHost localhost
ApJServDefaultPort 8007

# Mounting the first context.
ApJServMount /joe ajpv12://joe.corp.com:8007/joe

# Mounting the second context.
ApJServMount /bill ajpv12://bill.corp.com:8007/bill
</IfModule>
```

Como podemoe ver en el ejemplo anterior, usar varias JVMs (incluso aquellas que se ejecutan en diferentes máquinas) puede conseguirse fácilmente usando una URL completa **ajp** montada. En esta URL completa realmente especificamos el host donde está localizado el proceso Tomcat y su puerto.

Si tuvieramos los dos procesos Tomcat ejecutándose en la misma máquina, Deberíamos configurar cada uno de ellos con un puerto de conector diferente. Por ejemplo, asumiendo que las dos JVMs se ejecutan sobre `localhost`, la configuración **Apache-Tomcat** debería tener algo como esto:

```
##############################################################################
```

```
#      Apache-Tomcat with Same Machine JVM per Context                #
#########################################################################
LoadModule jserv_module modules/ApacheModuleJServ.dll
<IfModule mod_jserv.c>
ApJServManual on
ApJServDefaultProtocol ajpv12
ApJServSecretKey DISABLED
ApJServMountCopy on
ApJServLogLevel notice

ApJServDefaultHost localhost
ApJServDefaultPort 8007

# Mounting the first context.
ApJServMount /joe ajpv12://localhost:8007/joe

# Mounting the second context.
ApJServMount /bill ajpv12://localhost:8009/bill
</IfModule>
```

Mirando al fichero de arriba podemos ver que tenemos dos puntos de montaje ApJServ explícitos, cada uno apuntando a un puerto diferente de la misma máquina. Esta claro que esta configuración requiere soporte desde la configuración encontrada en los ficheros `server.xml`. Necesitamos diferentes configuraciones de `<Connector>` en cada fichero para los diferentes procesos Tomcat. Realmente necesitamos dos ficheros **server.xml** diferentes (llamémosles `server_joe.xml` y `server_bill.xml`) con diferentes entradas `<Connector>` como se ve en los siguientes ejemplos:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Server>
    <!-- Debug low-level events in XmlMapper startup -->
    <xmlmapper:debug level="0" />

    <!--  @@@
        Note, the log files are suffixed with _joe to distinguish
        them from the bill files.
    -->

    <Logger name="tc_log"
            path="logs/tomcat_joe.log"
            customOutput="yes" />

    <Logger name="servlet_log"
            path="logs/servlet_joe.log"
            customOutput="yes" />

    <Logger name="JASPER_LOG"
        path="logs/jasper_joe.log"
            verbosityLevel = "INFORMATION" />

    <!--  @@@
        Note, the work directory is suffixed with _joe to distinguish
        it from the bill work directory.
    -->
    <ContextManager debug="0" workDir="work_joe" >
```

```
        <!-- ==================== Interceptors ==================== --
>

        ...

        <!-- ==================== Connectors ==================== -->

        ...

        <!-- Apache AJP12 support. This is also used to shut down
tomcat.
          -->
        <!-- @@@ This connector uses port number 8007 for it's ajp
communication -->
        <Connector
className="org.apache.tomcat.service.PoolTcpConnector">
            <Parameter name="handler"

value="org.apache.tomcat.service.connector.Ajp12ConnectionHandler"/>
            <Parameter name="port" value="8007"/>
        </Connector>

        <!-- ==================== Special webapps ====================
-->

        <!-- @@@ the /jow context -->
        <Context path="/joe" docBase="webapps/joe" debug="0"
reloadable="true" >
        </Context>
    </ContextManager>
</Server>
```

Cuando miramos a **server_joe.xml** podemos ver que el <Connector> está configurado en el puerto 8007. Por otro lado, en **server_bill.xml** (ver abajo) el conector está configurado para el puerto 8009.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Server>
    <!-- Debug low-level events in XmlMapper startup -->
    <xmlmapper:debug level="0" />

    <!--  @@@
        Note, the log files are suffixed with _bill to distinguish
        them from the joe files.
    -->

    <Logger name="tc_log"
            path="logs/tomcat_bill.log"
            customOutput="yes" />

    <Logger name="servlet_log"
            path="logs/servlet_bill.log"
            customOutput="yes" />

    <Logger name="JASPER_LOG"
        path="logs/jasper_bill.log"
```

```
            verbosityLevel = "INFORMATION" />

    <!--  @@@
        Note, the work directory is suffixed with _bill to distinguish
        it from the joe work directory.
    -->
    <ContextManager debug="0" workDir="work_bill" >

        <!-- ==================== Interceptors ==================== --
>

        ...

        <!-- ==================== Connectors ==================== -->

        ...

        <!-- Apache AJP12 support. This is also used to shut down
tomcat.
          -->
        <!-- @@@ This connector uses port number 8009 for it's ajp
communication -->
        <Connector
className="org.apache.tomcat.service.PoolTcpConnector">
            <Parameter name="handler"

value="org.apache.tomcat.service.connector.Ajp12ConnectionHandler"/>
            <Parameter name="port" value="8009"/>
        </Connector>

        <!-- ==================== Special webapps ====================
-->

        <!-- @@@ the /bill context -->
        <Context path="/bill" docBase="webapps/bill" debug="0"
reloadable="true" >
        </Context>
    </ContextManager>
</Server>
```

La configuración del puerto no es la únia diferencia entre los dos ficheros. Tenemos marcas @@@ en los cuatro lugares de los ficheros xml donde hemos realizado cambios. Como podemos ver, esta diferencia es necesaria para evitar que los dos procesos Tomcat sobreescriban los logs y el espacio de trabajo del otro.

Entonces deberíamos arrancar los dos procesos Tomcat usando el la opción **-f** de la línea de comando:

```
bin\startup -f conf\server_joe.xml

bin\startup -f conf\server_bill.xml
```

y luego accedemos a ellos desde Apache basándonos en los diferentes prefijos de las URLs del path.

Configurar el Hosting Virtual

Es posible soportar host virtuales sobre Tomcat Ver3.2, de hecho la configuración de host virtuales es muy similar a la configuración para múltiples JVM y la razón es sencilla; en Tomcat 3.2 cada host virtual está implementado por un proceso Tomcat diferente.

Con la versión actual de Tomcat (Ver3.2), el hosting virtual sin preocupaciones está proporcionado por el servidor web (Apache/Netscape). El soporte de servidor de host virtual es usado por el adaptador Tomcat para redirigir las peticiones a cierto host virtual a la JVM(s) que conteine los contextos de este host virtual. Esto significa que si (por ejemplo) tenemos dos host virtuales (**vhost1** y **vhost2**), tendremos dos JVMs: una ejecutándose en el contexto de **vhost1** y la otra ejecutándose en el contexto de **vhost2**. Estas JVMs no se preocupan de la existencia de la otra, de hecho, no se preocupan del concepto de host virtual. Toda la lógica del hospedaje virtual está dentro del adaptador del servidor web. Para aclarar las cosas, veamos el siguiente fichero de configuración Apache-Tomcat

```
##############################################################################
#        Apache Tomcat Virtual Hosts Sample Configuration                    #
##############################################################################
LoadModule jserv_module modules/ApacheModuleJServ.dll
<IfModule mod_jserv.c>
ApJServManual on
ApJServDefaultProtocol ajpv12
ApJServSecretKey DISABLED
ApJServMountCopy on
ApJServLogLevel notice

ApJServDefaultHost localhost
ApJServDefaultPort 8007

# 1 Creating an Apache virtual host configuration
NameVirtualHost 9.148.16.139

# 2 Mounting the first virtual host
<VirtualHost 9.148.16.139>
ServerName www.vhost1.com
ApJServMount /examples ajpv12://localhost:8007/examples
</VirtualHost>

# 3 Mounting the second virtual host
<VirtualHost 9.148.16.139>
ServerName www.vhost2.com
ApJServMount /examples ajpv12://localhost:8009/examples
</VirtualHost>
</IfModule>
```

Como podemos ver, los pasos 1, 2 y 3 definen dos host virtuales en Apache y cada uno de ellos monta el contexto `/examples` en cierta URL `ajpv12`. Cada URL `ajpv12` apunta a una JVM que contiene el host virtual. La configuración de las dos JVM es muy similar a la mostrada en la sección anterior, y también necesitaremos usar dos ficheros **server.xml** diferentes (uno por cada host virtual) y necesitaremos arrancar los procesos Tomcat con la opción **-f** de la línea de comandos. Después de hacer esto podremos aproximarnos a Apache, cada vez con un nombre de host diferente, y el adaptador nos redirigirá la JVM apropiada.

La necesidad de mejorar el soporte para hosting virtual
Tener cada host virtual implementado por un JVM diferente es un enorme problema de

escalabilidad. Las siguientes versiones de Tomcat haran posible soportar varios host virtuales en la misma JVM Tomcat.

Trucos de Configuración del Mundo Real

Por defecto la distribución Tomcat viene con una configuración ingenua cuyo objetivo principal es ayudar al usuario recien experimentado y una operación "recién salido de la caja"... Sin embargo, esta configuración no es la mejor forma de desplegar Tomcat en sitios reales. Por ejemplo, los sites reales podrían requerir algún ajuste de rendimiento y configuraciones específicas de la site (elementos de path adicionales, por ejemplo). Esta sección intentará dirigirnos por los primeros pasos que deberíamos realizar antes de publicar una site basada en Tomcat.

Modificar y Personalizar los Ficheros Batch

Como mencionamos en las secciones anteriores, los scripts de arrancada están para nuestra conveniencia. Aunque, algunas veces los scripts que necesitamos para desarrollar deberían ser modificados:

Para configurar los límites de recursos como el máximo número de descriptores.

Para añadir nuevas entradas en el `CLASSPATH` (por ejemplo, drivers JDBC).

Para añadir nuevas entradas en el `PATH/LD_LIBRARY_PATH` (por ejemplo, DLLs de drivers JDBC).

Para modificar las selecciones de la línea de comandos de la JVM.

Para asegurarnos de que estámos usando la JVM adecuada (de las dos o tres que podemos tener instaladas en nuestra máquina).

Para cambiar el usuario de **root** a algún otro usuario usando el comando `"su"` de UNIX.

Por cualquier otra razón.

Algunos de estos cambios se pueden hacer sin cambiar explícitamente los scripts básicos; por ejemplo, el script `tomcat` puede usar una variable de entorno llamada `TOMCAT_OPTS` para seleccionar los parámetros extras de la línea de comando de la JVM (como configuraciones de memoria, etc). Sobre `UNIX` también podemos crear un fichero llamando `".tomcatrc"` en nuestro directorio **home** y Tomcat tomará la información de entorno como `PATH`, `JAVA_HOME`, `TOMCAT_HOME` y `CLASSPATH` desde este fichero. Sin embargo, sobre NT nos veremos forzados a reescrobor algunos de estos scripts de arrancada...

No tengas miedo, sólo hazlo!

Modificar las Configuraciones por Defecto de la JVM

Las configuraciones por defecto de la JVM en el script `tomcat` son muy ingenuas; todo se deja por defecto. Hay algunas cosas que deberíamos considerar para mejorar el rendimiento de Tomcat:

Modificar la configuración de memoria de nuestra JVM. Normalmente la JVM asigna un tamaño inicial para la pila Java y ya está, si necesitamos más memoria de está no podremos obtenerla. Además, en sitios sobrecargados, dar más memoria a la JVM mejora el rendimiento de Tomcat. Deberíamos usar los parámetros de la línea de comandos como `-Xms/-Xmx/-ms/-mx` para seleccionar los tamaños mínimo y máximo de la pila Java (y chequear si mejora el rendimiento).

Modificar nuestra configuración de threading en la JVM. El JDK 1.2.2 para Linux viene con soporte para threads verdes y nativos. En general, los theads nativos son conocidos por proporcionar mejoras de rendimiento para aplicaciones que tratan con I/O, los threads verdes, por otro lado, ponen menos acento en la máquina. Deberíamos experimetnar con estos dos modelos de threads y ver cual es mejor para nuestra site (en general, los threads nativos son mejores).

Seleccionamos la mejor JVM para la tarea. Hay distintos vendedores de JVMs por lo que deberemos decidirnos por la más rápida o la más barata, según nos interese

Modificar nuestros Conectores

Los conectores, según los configura el fichero `server.xml` de Tomcat, contiene dos `Connectors` configurados como en el siguiente fragmento:

```
        <!-- (1) HTTP Connector for stand-alone operation -->
        <Connector
className="org.apache.tomcat.service.PoolTcpConnector">
            <Parameter name="handler"

value="org.apache.tomcat.service.http.HttpConnectionHandler"/>
            <Parameter name="port"
                value="8080"/>
        </Connector>

        <!-- (2) AJPV12 Connector for out-of-process operation -->
        <Connector
className="org.apache.tomcat.service.PoolTcpConnector">
            <Parameter name="handler"

value="org.apache.tomcat.service.connector.Ajp12ConnectionHandler"/>
            <Parameter name="port"
                value="8007"/>
        </Connector>
```

Es un conector que escucha en el puerto 8080 para peticiones HTTP entrantes. Este conector es necesario para operaciones independientes.

Es un conector que escucha en el puerto 8007 para peticiones AJPV12 entrantes. Este conector es necesario para la integración del servidor web (integración de servlets fuera-de-proceso).

El conector AJPV12 es necesario para cerrar Tomcat. Sin embargo, el conector HTTP podría eliminarse si la operación independiente no lo necesitase.

Usar Almacenes de Threads en nuestros Conectores

Tomcat es un contenedor servlet multi-thread lo que significa que cada petición necesita ser ejecutada por algún thread. Anteriomente a Tomcat 3.2, por defecto había que crear un nuevo thread para servir cada petición que llegaba. Este comportamiento era problemático en sitios sobrecargados porque:

Arrancar y parar un thread para cada petición pone en aprietos al sistema operativo y a la JVM.

Es dificil limitar el consumo de recursos. Si llegan 300 peticiones de forma concurrente Tomcat abrirá 300 threads para servirlas y asignará todos los recursos necesarios para servir las 300 peticiones al mismo tiempo. Esto hace que Tomcat asigne muchos más recursos (CPU,

Memoria, Descriptores...) de lo que debiera y puede bajar el rendimiento e incluso colgarse si los recursos están exhaustos.

La solución para estos problemas es usar un **thread pool** (almacen de threads), que se usa por defecto en Tomcat 3.2. Los contenedores Servlets que usan almacenes de threads se liberan a sí mismos de manejar sus treads. En lugar de asignar nuevos threads, cada vez que los necesitan, se los piden al almacen, y cuando todo está hecho, el thread es devuelto al almacen. Ahora el almacen de threads se puede utilizar para implementar técnicas de control de de threads, como:

Mantener threads "abiertos" y reutilizarlos una y otra vez. Esto nos evita el problema asociado con la creación y destrucción continua de threads.

Normalmente el administrador puede instruir al almacen para que no mantenga demasiados threads desocupados, liberándolos si es necesario.

Seleccionando un límite superior en el número de threads usados de forma concurrente. Esto evita el problema de la asignación de recursos asociada con la asignación ilimitada de threads.

Si el contenedor alcanza su límite superior de threads, y llega una nueva petición, esta nueva petición tendrá que esperar hasta que alguna otra petición (anterior) termine y libere el thread que está usando.

Podemos refinar las técnicas descritas arriba de varias formas, pero sólo serán refinamientos. La principal contribución de los almacenes de threads es la reutilización de los thrreads un límite superior que limite el uso de recursos.

Usar un almacen de threads en Tomcat es un sencillo movimiento; todo lo que necesitamos hacer es usar un `PoolTcpConnector` en nuestra configuración de `<Connector>`. Por ejejmplo, el siguiente fragmento de **server.xml** define `ajpv12`, como un conector con almacen:

```
        <!-- A pooled AJPV12 Connector for out-of-process operation --
>
        <Connector
className="org.apache.tomcat.service.PoolTcpConnector">
            <Parameter
                name="handler"

value="org.apache.tomcat.service.connector.Ajp12ConnectionHandler"/>
            <Parameter
                name="port"
                value="8007"/>
        </Connector>
```

Este fragmento es muy simple y el comportamiento (por defecto) del almacen instruido por él es:

Un límite de 50 threads concurrentes..

Cuando el almacen tenga más de 25 threads desocupados empezará a eliminarlos.

El almacen empezará con la creación de 10 threads, y tratará de mantener 10 threads vacantes (mientras no llegue al límite superior)

La configuración por defecto está bien para sites de media carga con un media de 10-40 peticiones concurrentes. Si nuestro site es diferente deberíamos modificar esta configuración (por ejemplo reduciendo el límite superior). La configuración del almacen de threads se puede hacer desde el elemento `<Connector>` en **server.xml** como se demuestra en el siguiene fragmento:

```
        <!-- A pooled AJPV12 Connector for out-of-process operation --
>
        <Connector
className="org.apache.tomcat.service.PoolTcpConnector">
            <Parameter
                name="handler"

value="org.apache.tomcat.service.connector.Ajp12ConnectionHandler"/>
            <Parameter
                name="port"
                value="8007"/>
            <Parameter
                name="max_threads"
                value="30"/>
            <Parameter
                name="max_spare_threads"
                value="20"/>
            <Parameter
                name="min_spare_threads"
                value="5" />
        </Connector>
```

Como se puede ver el almacen tiene 3 parámetros de configuración:

`max_threads` - define el límite superior de concurrencia, el almacen no creará más de este número de threads.

`max_spare_threads` - define el máximo número de threads que el almacen mantendrá inactivos. Si el número de threads inactivos excede este valor los eliminará.

`min_spare_threads` - el almacen intentará asegurarse de que en todo momemto hay al menos este número de threads inactivos esperando que lleguen nuevas peticiones. `min_spare_threads` debe ser mayor que 0.

Deberíamos usar estos parámetros para ajustar el comportamiento del almacen a nuestras necesidades.

Desactivar la Auto-Recarga de Servlets

La auto-recarga de servlets es muy util en el momento del desarrollo. Sin embargo es muy costosa (en términos de degradación del rendimiento) y podría poner a nuestra aplicación en extraños confilctos cuando las clases fueran cargadas y ciertos cargadores de clases no puedieran cooperar con las clases cargadas por el `classloader` actual.

Por eso, a menos que tengamos una necesidad real para recargar las clases durante el despliegue deberíamos desactivar la bandera `reloadable` en nuestros contextos.

Usar el SecurityManager de Java con tomcat

¿Por qué usar un SecurityManager

El `SecurityManager` de Java es el que permite a un navegador ejecutar un applet en su propia caja para evitar que código no firmado acceda a ficheros del sistema local, conectar con un host distinto de donde se cargó el applet, etc.

De la misma forma que el `SecurityManager` nos protege de que se ejecute un applet no firmado en nuestro navegador, el uso de un `SecurityManager` mientras se ejecuta Tomcat puede protejer nuestro servidor de servelts, JSP's, beans JSP, y librerías de etiquetas troyanos. O incluso de errores inadvertidos.

Imagina que alguien que está autorizado a publicar un JSP en nuestro site inadvertidamente incluye esto en su JSP:

```
<% System.exit(1); %>
```

Cada vez que el JSP sea ejecutado por Tomcat, Tomcat se cerrará.

Usar el `SecurityManager` de Java es sólo una línea más de defensa que un administrador de sistemas puede usar para mantener el servidor seguro y fiable.

Requirimientos del Sistema

El uso del `SecurityManager` requiere una JVM que soporte JDK 1.2.

Precacuciones

La implementación de un `SecurityManager` en Tomcat no ha sido completamente probada para asegurar la seguridad de Tomcat. No se han creado `Permissions` especiales para evitar accesos a clases internas de Tomcat por parte de JSPs, aplicaciones web, beans o librerías de etiquetas. Debemos asegurarnos de que estamoa satisfechos con nuestra configuración de `SecurityManager` antes de permitir que los usuarios no creibles publiquen aplicacions web, JSPs, servlets, beans o librerías de etiquetas en nuestra site.

Aún así, ejecutarlo con un `SecurityManager` definitivamente es mejor que hacerlo sin ninguno.

Tipos de Permisos

Las clases `Permission` se usan para definir que clases de **Permisos** tendrán las clases cargadas por Tomcat. Hay varias clases de `Permission` como parte del JDK e incluso podemos crear las nuestras propias para usarlar en nuestras aplicaciones web.

Este es sólo un pequeño sumario de las clases de **System SecurityManager Permission** aplicables a Tomcat. Puedes encontrar más documentación sobre el uso de las clases siguientes en la documentación del JDK.

java.util.PropertyPermission
Controla los accesos de lectura/escritura a las propiedades de JVM como `java.home`.

java.lang.RuntimePermission
Controla el uso de algunas funciones de sistema/ejecución como `exit()` y `exec()`.

java.io.FilePermission
Controla los aceeso de lectura/escritura/ejecución a ficheros y directorios.

java.net.SocketPermission
Controla el uso de sockets de red.

java.net.NetPermission
Controla el uso de conexiones de red multicast.

java.lang.reflect.ReflectPermission
Controla el uso de `reflection` para hacer `introspection` de clase.

java.security.SecurityPermission
Controla el acceso a los métodos de `Security`.

java.security.AllPermission
Permite acceder a todos los permisos, como si se estuviera ejecutando Tomcat sin un `SecurityManager`.

¿Qué sucede cuando el SecurityManager detecta una Violación de Seguridad

La JVM lanzará una `AccessControlException` o una `SecurityException` cuando el `SecurityManager` detecte una violación dela política de seguridad.

Los Workers Tomcat

Un `worker` Tomcat es un ejemplar Tomcat que está esperando para ejecutar servlets por cuenta de algún servidor web. Por ejemplo, podemos tener un servidor web como Apache reenviando peticiones servlets a un proceso Tomcat (el `worker` que se ejecuta detrás de él.

El escenario descrito arriba es uno muy simple; de hecho uno puede configurar múltiples `workers` para servir servlets por cuenta de un cierto servidor web. Las razones para dicha configuración pueden ser:

Queremos que diferentes contextos sean servidos por diferentes `workers` Tomcat para proporcionar un entorno de desarrollo donde todos los desarrolladores compartan el mismo servidor pero con su propio `worker` Tomcat.

Queremos que diferentes host virtuales servidos por diferentes procesos Tomcat proporcionen una clara separación entre los sites pertenecientes a distintas compañías.

Queremos proporcionar un balance de carga, lo que significa ejecutar múltiples `workers` Tomcat cada uno en su propia máquina y distribuir las peticiones entre ellos.

Probablemente haya más razones para tener múltiples `workers` pero creo que esta lista es suficiente...

Los `workers` están definidos en un fichero de propiedades llamado **workers.properties** y está página explica como trabajar con él.

Definir Workers

La definición de `workers` para el plugin Tomcat del servidor web puede hacerse usando un fichero de propiedades (un fichero de ejemplo llamado **workers.properties** está disponible en el directorio `conf/`); el fichero contiene entradas con la siguiente forma:

```
worker.list=<una lista separada por comas de nombres de workers >
```

Por ejemplo:

```
worker.list= ajp12, ajp13
```

Y

```
worker.<nombre de worker>.<property>=<valor de propiedad>
```

Por ejemplo:

```
worker.local.port=8007
```

Cuando arranque, el plugin del servidor web ejemplarizará los `workers` cuyos nombres aparezcan en la propiedad `worker.list`, estos también son los workers a los que podemos mapear peticiones.

Cada `worker` nombrado debería tener una pocas entradas para proporcionar información adicional sobre sí mismo; esta información incluye el tipo de `worker` y otra información relacionada. Actualmente existen estos tipos de `workers` en (Tomcat 3.2-dev):

Tipo de Worker	Description
`ajp12`	Este worker sabe cómo reenviar peticiones a workers Tomcat fuera-de-proceso usando el protocolo **ajpv12**.
`ajp13`	Este worker sabe cómo reenviar peticiones a workers Tomcat fuera-de-proceso usando el protocolo **ajpv13**.
`jni`	Este worker sabe cómo reenviar peticiones a workers Tomcat fuera-de-proceso usando **jni**.
`lb`	Este es un worker de balance de carga, que sabe como proporcionar un balance de carga basado en redondeo con un cierto nivel de tolerancia.

Definir `workers` de un cierto tipo debería hacerse siguiendo este formato de propiedad:

```
worker.<worker name>.type=<worker type>
```

Donde **worker name** es el nombre asignado al `worker` y **worker type** es uno de los cuatro tipos definidos en la tabla. Un nombre de worker podría no contener espacios (una buena convención de nombres sería utilizar las reglas de nombrado para las variables Java).

Por ejemplo:

Definición de Worker	Significado
`worker.local.type=ajp12`	Define un worker llamado "local" que usa el protocolo **ajpv12** para reenviar peticiones a un proceso Tomcat.
`worker.remote.type=ajp13`	Define un worker llamado "remote" que usa el protocolo **ajpv13** para reenviar peticiones a un proceso Tomcat.
`worker.fast.type=jni`	Define un worker llamado "fast" que usa **JNI** para

	reenviar peticiones a un proceso Tomcat.
`worker.loadbalancer.type=lb`	Define un worker llamado "loadbalancer" que hace balance de carga de varios procesos Tomcat de forma transparente.

Configurar Propiedades del Worker

Después de definir los `workers` podemos especificar propiedades para ellos. Las propiedades se pueden especificar de la siguiente manera:

```
worker.<worker name>.<property>=<property value>
```

Cada worker tiene un conjunto de propiedades que podemos configurar según se especifica en las siguientes subsecciones:

Propiedades de un Worker ajp12

Los `workers` del tipo **ajp12** reenvían peticiones a `workers` Tomcat fuera-de-proceso usando el protocolo **ajpv12** sobre sockets TCP/IP. La siguiente tabla especifica las propiedades que puede aceptar un `worker` **ajp12**:

Nombre de Propiedad	Significado	Ejemplo
`port`	El puerto donde el worker Tomcat escucha peticiones ajp12.	`worker.local.port=8007`
`host`	El host donde el worker Tomcat escucha peticiones ajp12.	`worker.local.host=www.x.com`
`lbfactor`	Cuando trabaja con un worker de balance de carga, este es el factor de balance para el worker.	`worker.local.lbfactor=2.5`

Propiedades de un Wroker ajp13

Los `workers` del tipo **ajp13** reenvían peticiones a `workers` Tomcat fuera-de-proceso usando el protocolo **ajpv13** sobre sockets TCP/IP. Las principales diferencias entre ajpv12 y ajpv13 son que:

ajpv13 es un protocolo más binario e intenta comprimir algunos de los datos solicitados codificando los strings más frecuentemente usados en enteros pequeños.

ajpv13 reusa sockets abiertos y los deja abiertos para futuras peticiones.

ajpv13 tiene un tratamiento especial para información SSL por eso el contenedor puede implementar métodos relacionados cono SSL como `isSecure()`.

La siguiente tabla especifica las propiedades que puede aceptar un `worker` **ajp13**:

Nombre de	Significado	Ejemplo

Propiedad		
`port`	El puerto donde el worker Tomcat escucha peticiones ajp12.	`worker.local13.port=8007`
`host`	El host donde el worker Tomcat escucha peticiones ajp12.	`worker.local13.host=www.x.com`
`lbfactor`	Cuando trabaja con un worker de balance de carga, este es el factor de balance para el worker.	`worker.local13.lbfactor=2.5`
`cachesize`	Especifica el número de conexiones sockets abiertas que mantendrá el worker. Por defecto este valor es 1, pero los servidores web multi-thread como Apache2.xx, IIS, y Netscape se beneficiarán si configuramos este valor a un nivel más alto (como una media estimada de los usuarios concurrentes de Tomcat).	`worker.local13.cachesize=30`

Propiedades de un Worker lb

El `worker` de balanceo de cargas realmente no se comunica con otros `workers` Tomcat, en su lugar es el responsable de varios `workers` "reales". Este control incluye:

Ejemplarizar los `workers` en el servidor web.

Usar el factor de balanceo de carga, realizando un balanceo de carga al redondeo de peso donde el `lbfactor` más alto significa una máquina más fuerte (es la que manejará más peticiones).

Seguimiento de las peticiones que pertenecen a la misma sesión y que se ejecutan en el mismo `worker` Tomcat.

Identificar los `workers` Tomcat fallidos, suspender las peticiones a estos `workers` y hacer que otros `workers` las manejen.

El resultado general es que los `workers` manejados por el mismo `worker` **lb** tienen balance de cargas (basándose en su **lbfactor** y la sesión de usuario actual) y tambíen tienen anti-caída por lo que si un proceso Tomcat muere, no "matará" toda la site.

La siguiente tabla especifica las propiedades que puede aceptar un `worker` **lb**:

Nombre de Propiedad	Significado	Ejemplo
`balanced_workers`	Una lista separada por comas de workers que el balanceador de carga necesita manejar. Estos	`worker.loadbalancer.balanced_workers= local13, local12`

	workers no deberían aparecer en la propiedad **worker.list**.

Propiedades de un Worker jni

El `worker` jni abre una JVM dentro del proceso del servidor web y ejecuta Tomcat dentro de ella (es decir en-proceso). Después de esto, los mensajes pasados hacia y desde la JVM son pasados usando llamadas a métodos JNI, esto hace al `worker` jni más rápido que los `workers` fuera-de-proceso que necesitan comunicarse con los `workers` Tomcat escribiendo mensajes AJP sobre sockets TCP/IP.

Nota: como la JVM es multi-thread; el `worker jni` sólo se debería usar dentro de servidores multi-thread como AOLServer, IIS, Netscape y Apache2.0. Deberíamos asegurarnos de que el esquema de threads usado por el servidor web corresponde con el usado para construir el plugin **jk** del servidor web.

Como el `worker jni` abre una JVM puede aceptar tantas propiedades como pueda reenviar a la JVM como el `classpath`, etc. como podemos ver en la siguiente tabla:

Nombre de Propiedad	Significado	Ejemplo
`class_path`	El casspath usado por la JVM en-proceso. Esto debería apuntar a todos los ficheros jar/file de Tomcat así como a cualquier clase u otro fichero jar que queramos añadir a la JVM Deberíamos recordar añadir también **javac** al classpath. Esto se hace en Java2 añadiendo **tools.jar** al classpath. En JDK1.xx deberíamos añadir **classes.zip**. La propiedad `class_path` se puede dividir en múltiples líneas. En este caso el entorno **jk** concatenará todas las entradas classpath poniendo un delimitador (":"/";") entre cada entrada.	`worker.localjni.class_path=path-to-some-jarfile` `worker.localjni.class_path=path-to-class-directory`

`cmd_line`	La línea de comandos que es manejada sobre el código de arranque de Tomcat.
	La propiedad `cmd_line` puede proporcionarse en múltiples
		`worker.localjni.cmd_line=-config`
		`worker.localjni.cmd_line=path-to-tomcats-server.xml-file`
		`worker.localjni.cmd_line=-home`
		`worker.localjni.cmd_line=-path-to-tomcat-home`

	líneas. En este caso el entorno **jk** las concatenará poniéndo espacios entre ellas. **Nota:** El string `cmd_line` no soporta espacios en blanco embebidos. Esto afecta principalmente a las especificaciones de paths. Sobre windows, podemos usar los nombres MS-DOS 8.3 para directorios que de otro modo contendrían un espacio en blanco.	
`jvm_lib`	El path completo a la librería de implementación de la JVM. El worker jni usa este path para cargar la JVM dinámicamente.	`worker.localjni.jvm_lib=full-path-to-jvm.dll`
`stdout`	El path completo donde la JVM escribirá su `System.out`	`worker.localjni.stdout=full-path-to-stdout-file`
`stderr`	El path completo donde la JVM escribirá su `System.err`	`worker.localjni.stderr=full-path-to-stderr-file`
`sysprops`	Propiedades de sistema para la JVM.	`worker.localjni.sysprops=some-property`
`ld_path`	Path a las librerías dinámicas adicionales (similar en naturaleza a `LD_LIBRARY_PATH`).	`worker.localjni.ld_path=some-extra-dynamic-library-path`

Macros en Ficheros de Propiedades

Desde Tomcat3.2 podemos definir **macros** en los ficheros de propiedades. Estas macros nos permite definir propiedades y posteriormente usarlas cuando construyamos otras propiedades. Por ejemplo, el siguiente fragmento:

```
workers.tomcat_home=c:\jakarta-tomcat
workers.java_home=c:\jdk1.2.2
ps=\
worker.inprocess.class_path=$(workers.tomcat_home)$(ps)classes
worker.inprocess.class_path=$(workers.java_home)$(ps)lib$(ps)tools.jar
```

Terminará con los siguientes valores para las propiedades `worker.inprocess.class_path`:

```
worker.inprocess.class_path= c:\jakarta-tomcat\classes
worker.inprocess.class_path=c:\jdk1.2.2\lib\tools.jar
```

El ejemplo worker.properties

Como escribir **worker.properties** por nosotros mismos no es una cosa fácil de hacer; junto con los paquetes de Tomcat3.2 (y superiores) viene un fichero **worker.properties** de ejemplo. Este fichero está pensado para ser tan **genérico** como sea posible y usa extensivamente las macros de propiedades.

El ejemplo **worker.properties** contiene el siguiente nivel de definiciones:

Un worker ajp12 que usa el host `localhost` y el puerto `8007`.

Un worker ajp13 que usa el host `localhost` y el puerto `8009`.

Un worker jni.

Un worker lb que balancea la carga de los workers ajp12 y ajp13.

Las definicioens de los workers ajp12, ajp13 y lb pueden funcionar sin modificar el fichero. Sin embargo, para hacer que funcione el `worker jni` deberemos configurar las siguientes configuraciones en el fichero:

`workers.tomcat_home`
- tiene que apuntar a nuestro `tomcat_home`.

`workers.java_home`
- tiene que apuntar a donde tengamos situado el JDK.

`ps`
- tiene que apuntar al separador de ficheros de nuestro sistema operativo.

Cuando hagamos esto, el worker jni por defecto, **debería** funcionar.

Tema 2: Acceso a Bases de Datos

Acceso a Bases de DatosJava IDL

JDBC fue diseñado para mantener sencillas las cosas sencillas. Esto significa que el API JDBC hace muy sencillas las tareas diarias de una base de datos, como una simple sentencia **SELECT**. Esta sección nos llevará a través de ejemplos que utilizan el JDBC para ejecutar sentencias SQL comunes, para que podamos ver lo sencilla que es la utilización del API JDBC básico.

Esta sección está dividida a su vez en dos secciones.

JDBC Basico que cubre el API JDBC 1.0, que está incluido en el JDK 1.1.

La segunda parte cubre el API JDBC 2.0 API, que forma parte de la versión 1.2 del JDK. También describe brevemente las extensiones del API JDBC, que, al igual que otras extensiones estándard, serán liberadas independientemente.

Al final de esta primera sección, sabremos como utilizar el API básico del JDBC para crear tablas, insertar valores en ellas, pedir tablas, recuperar los resultados de las peticiones y actualizar las tablas. En este proceso, aprenderemos como utilizar las sentencias sencillas y sentencias preparadas, y veremos un ejemplo de un procedimiento almacenado. También aprenderemos como realizar transaciones y como capturar excepciones y avisos. En la última parte de este sección veremos como crear un Applet.

Nuevas Características en el API JDBC 2.0 nos enseña como mover el cursor por una hoja de resultados, cómo actualizar la hoja de resultados utilizando el API JDBC 2.0, y como hacer actualizaciones batch. También conoceremos los nuevos tipos de datos de SQL3 y como utilizarlos en aplicaciones escritas en Java. La parte final de esta lección entrega una visión de la extensión del API JDBC, con caracterísitas que se aprovechan de la tecnologías JavaBeans y Enterprise JavaBeans.

Esta sección no cubre cómo utilizar el API metadata, que se utiliza en programas más sofisticados, como aplicaciones que deban descubrir y presentar dinámicamente la estructura de una base de datos fuente.

Empezar con JDBC

Lo primero que tenemos que hacer es asegurarnos de que disponemos de la configuración apropiada. Esto incluye los siguientes pasos.

Instalar Java y el JDBC en nuestra máquina.

> Para instalar tanto la plataforma JAVA como el API JDBC, simplemente tenemos que seguir las instrucciones de descarga de la última versión del JDK (Java Development Kit). Junto con el JDK también viene el JDBC.

Instalar un driver en nuestra máquina.

Nuestro Driver debe incluir instrucciones para su instalación. Para los drivers JDBC escritos para controladores de bases de datos específicos la instalación consiste sólo en copiar el driver en nuesta máquina; no se necesita ninguna configuración especial.

Instalar nuestro Controlador de Base de Datos si es necesario.

Si no tenemos instalado un controlador de base de datos, necesitaremos seguir las instrucciones de instalación del vendedor. La mayoría de los usuarios tienen un controlador de base de datos instalado y trabajarán con un base de datos establecida.

Seleccionar una Base de Datos

A lo largo de la sección asumiremos que la base de datos **COFFEEBREAK** ya existe. (crear una base de datos no es nada díficil, pero requiere permisos especiales y normalmente lo hace un administrador de bases de datos). Cuando creemos las tablas utilizadas como ejemplos en este tutorial, serán la base de datos por defecto. Hemos mantenido un número pequeño de tablas para mantener las cosas manejables.

Supongamos que nuestra base de datos está siendo utilizada por el propietario de un pequeño café llamado "The Coffee Break", donde los granos de café se venden por kilos y el café liquido se vende por tazas. Para mantener las cosas sencillas, también supondremos que el propietario sólo necesita dos tablas, una para los tipos de café y otra para los suministradores.

Primero veremos como abrir una conexión con nuestro controlador de base de datos, y luego, ya que JDBC puede enviar codigo SQL a nuestro controlador, demostraremos algún código SQL. Después, veremos lo sencillo que es utilizar JDBC para pasar esas sentencias SQL a nuestro controlador de bases de datos y procesar los resultados devueltos.

Este código ha sido probado en la mayoría de los controladores de base de datos. Sin embargo, podríamos encontrar algunos problemas de compatibilidad su utilizamos antiguos drivers ODB con el puente JDBC.ODBC.

Establecer una Conexión

Lo primero que tenemos que hacer es establecer una conexión con el controlador de base de datos que queremos utilizar. Esto implica dos pasos: (1) cargar el driver y (2) hacer la conexión.

Cargar los Drivers

Cargar el driver o drivers que queremos utilizar es muy sencillo y sólo implica una línea de código. Si, por ejemplo, queremos utilizar el puente JDBC-ODBC, se cargaría la siguiente línea de código.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

La documentación del driver nos dará el nombre de la clase a utilizar. Por ejemplo, si el nombre de la clase es **jdbc.DriverXYZ**, cargaríamos el driver con esta línea de código.

```
Class.forName("jdbc.DriverXYZ");
```

No necesitamos crear un ejemplar de un driver y registrarlo con el **DriverManager** porque la llamada a **Class.forName** lo hace automáticamente. Si hubiéramos creado nuestro propio ejemplar, crearíamos un duplicado innecesario, pero no pasaría nada.

Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos.

Hacer la Conexión

El segundo paso para establecer una conexión es tener el driver apropiado conectado al controlador de base de datos. La siguiente línea de código ilustra la idea general.

```
Connection con = DriverManager.getConnection(url, "myLogin",
"myPassword");
```

Este paso también es sencillo, lo más duro es saber qué suministrar para **url**. Si estamos utilizando el puente JDBC-ODBC, el JDBC URL empezará con **jdbc:odbc:**. el resto de la URL normalmente es la fuente de nuestros datos o el sistema de base de datos. Por eso, si estamos utilizando ODBC para acceder a una fuente de datos ODBC llamada "**Fred,**" por ejemplo, nuestro URL podría ser **jdbc:odbc:Fred**. En lugar de "**myLogin**" pondríamos el nombre utilizado para entrar en el controlador de la base de datos; en lugar de "**myPassword**" pondríamos nuestra password para el controlador de la base de datos. Por eso si entramos en el controlador con el nombre "**Fernando**" y la password of "**J8,**" estas dos líneas de código estableceran una conexión.

```
String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernando", "J8");
```

Si estamos utilizando un puente JDBC desarrollado por una tercera parte, la documentación nos dirá el subprotocolo a utilizar, es decir, qué poner despues de **jdbc:** en la URL. Por ejemplo, si el desarrollador ha registrado el nombre "acme" como el subprotocolo, la primera y segunda parte de la URL de JDBC serán **jdbc:acme:**. La documentación del driver también nos dará las guías para el resto de la URL del JDBC. Esta última parte de la URL suministra información para la identificación de los datos fuente.

Si uno de los drivers que hemos cargado reconoce la URL suministada por el método **DriverManager.getConnection**, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase **DriverManager**, como su nombre indica, maneja todos los detalles del establecimiento de la conexión detrás de la escena. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface **Driver**, y el único método de **DriverManager** que realmente necesitaremos conocer es **DriverManager.getConnection**.

La conexión devuelta por el método **DriverManager.getConnection** es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, **con** es una conexión abierta, y se utilizará en los ejemplos posteriores.

Seleccionar una Tabla

Primero, crearemos una de las tablas de nuestro ejemplo. Esta tabla, **COFFEES**, contiene la información esencial sobre los cafés vendidos en "The Coffee Break", incluyendo los nombres de los cafés, sus precios, el número de libras vendidas la semana actual, y el número de libras vendidas hasta la fecha. Aquí puedes ver la tabla **COFFEES**, que describiremos más adelante.

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0

Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

La columna que almacena el nombre del café es **COF_NAME,** y contiene valores con el tipo **VARCHAR** de SQL y una longitud máxima de 32 caracteres. Como utilizamos nombres diferentes para cada tipo de café vendido, el nombre será un único identificador para un café particular y por lo tanto puede servir como clave primaria. La segunda colunma, llamada **SUP_ID**, contiene un número que identifica al suministrador del café; este número será un tipo **INTEGER** de SQL. La tercera columna, llamada **PRICE,** almacena valores del tipo **FLOAT** de SQL porque necesita contener valores decimales. (Observa que el dinero normalmente se almacena en un tipo **DECIMAL** o **NUMERIC** de SQL, pero debido a las diferencias entre controladores de bases de datos y para evitar la incompatibilidad con viejas versiones de JDBC, utilizamos el tipo más estándard **FLOAT**.) La columna llamada **SALES** almacena valores del tipo **INTEGER** de SQL e indica el número de libras vendidas durante la semana actual. La columna final, **TOTAL**,contiene otro valor **INTEGER** de SQL que contiene el número total de libras vendidas hasta la fecha.

SUPPLIERS, la segunda tabla de nuesta base de datos, tiene información sobre cada uno de los suministradores.

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Las tablas **COFFEES** y **SUPPLIERS** contienen la columna **SUP_ID**, lo que significa que estas dos tablas pueden utilizarse en sentencias **SELECT** para obtener datos basados en la información de ambas tablas. La columna **SUP_ID** es la clave primaria de la tabla **SUPPLIERS**, y por lo tanto, es un identificador único para cada uno de los suministradores de café. En la tabla **COFFEES**, **SUP_ID** es llamada clave extranjera. (Se puede pensar en una clave extranjera en el sentido en que es importada desde otra tabla). Observa que cada número **SUP_ID** aparece sólo una vez en la tabla **SUPPLIERS**; esto es necesario para ser una clave primaria. Sin embargo, en la tabla **COFFEES**, donde es una clave extranjera, es perfectamente correcto que haya números duplicados de **SUP_ID** porque un suministrador puede vender varios tipos de café. Más adelante en este capítulo podremos ver cómo utilizar claves primarias y extranjeras en una sentencia **SELECT**.

La siguiente sentencia SQL crea la tabla **COFFEES**. Las entradas dentro de los paréntesis exteriores consisten en el nombre de una columna seguido por un espacio y el tipo SQL que se va a almacenar en esa columna. Una coma separa la entrada de una columna (que consiste en el nombre de la columna y el tipo SQL) de otra. El tipo **VARCHAR** se crea con una longitud máxima, por eso toma un parámetro que indica la longitud máxima. El parámetro debe estar entre paréntesis siguiendo al tipo. La sentencia SQL mostrada aquí, por ejemplo, específica que los nombres de la columna COF-NAME pueden tener hasta 32 caracteres de longitud.

```
CREATE TABLE COFFEES
        (COF_NAME VARCHAR(32),
 SUP_ID INTEGER,
 PRICE FLOAT,
 SALES INTEGER,
 TOTAL INTEGER)
```

Este código no termina con un terminador de sentecia de un controlador de base de datos, que puede variar de un controlador a otro. Por ejemplo, Oracle utiliza un punto y coma (;) para finalizar una sentencia, y Sybase utiliza la plabra **go**. El driver que estamos utilizado proporcionará automáticamente el terminador de setencia apropiado, y no necesitaremos introducirlo en nuestro código JDBC.

Otra cosa que debíamos apuntar sobre las sentencias SQL es su forma. En la sentencia **CREATE TABLE**, las palabras clave se han imprimido en letras máyusculas, y cada ítem en una línea separada. SQL no requiere nada de esto, estas convenciones son sólo para una fácil lectura. El estándard SQL dice que la palabras claves no son sensibles a las mayúsculas, así, por ejemplo, la anterior sentencia **SELECT** pude escribirse de varias formas. Y como ejemplo, estas dos versiones son equivalentes en lo que concierne a SQL.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE "Washington"
select First_Name, Last_Name from Employees where
Last_Name like "Washington"
```

Sin embargo, el material entre comillas si es sensible a las maýusculas: en el nombre "**Washington**", "**W**" debe estar en maýuscula y el resto de las letras en minúscula.

Los requerimientos pueden variar de un controlador de base de datos a otro cuando se trada de nombres de identificadores. Por ejemplo, algunos controladores, requieren que los nombres de columna y de tabla seán exactamente los mismos que se crearon en las sentencias **CREATE** y **TABLE**, mientras que otros controladores no lo necesitan. Para asegurarnos, utilizaremos mayúsculas para identificadores como **COFFEES** y **SUPPLIERS** porque así es como los definimos.

Hasta ahora hemos escrito la sentencia SQL que crea la tabla **COFFEES**. Ahora le pondremos comillas (crearemos un string) y asignaremos el string a la variable **createTableCoffees** para poder utilizarla en nuestro código JDBC más adelante. Como hemos visto, al controlador de base de datos no le importa si las líneas están divididas, pero en el lenguaje Java, un objeto **String** que se extienda más allá de una línea no será compilado. Consecuentemente, cuando estamos entregando cadenas, necesitamos encerrar cada línea entre comillas y utilizar el signo más (+) para concatenarlas.

```
String createTableCoffees = "CREATE TABLE COFFEES " +
                "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, "
+
                "SALES INTEGER, TOTAL INTEGER)";
```

Los tipos de datos que hemos utilizado en nuestras sentencias **CREATE** y **TABLE** son tipos genéricos SQL (también llamados tipos JDBC) que están definidos en la clase **java.sql.Types**. Los controladores de bases de datos generalmente utilizan estos tipos estándards, por eso cuando llegue el momento de probar alguna aplicación, sólo podremos utilizar la aplicación **CreateCoffees.java**, que utiliza las sentencias **CREATE** y **TABLE**. Si tu controlador utiliza sus propios nombres de tipos, te suministrarermos más adelante una aplicación que hace eso.

Sin embargo, antes de ejecutar alguna aplicación, veremos lo más básico sobre el JDBC.

Crear sentencias JDBC

Un objeto **Statement** es el que envía nuestras sentencias SQL al controlador de la base de datos. Simplemente creamos un objeto **Statement** y lo ejecutamos, suministando el método SQL apropiado con la sentencia SQL que queremos enviar. Para una sentencia **SELECT**, el

método a ejecutar es **executeQuery**. Para sentencias que crean o modifican tablas, el método a utilizar es **executeUpdate**.

Se toma un ejemplar de una conexión activa para crear un objeto **Statement**. En el siguiente ejemplo, utilizamos nuestro objeto **Connection**: **con** para crear el objeto **Statement**: **stmt**.

```
Statement stmt = con.createStatement();
```

En este momento **stmt** existe, pero no tiene ninguna sentencia SQL que pasarle al controlador de la base de datos. Necesitamos suministrarle el metodo que utilizaremos para ejecutar **stmt**. Por ejemplo, en el siguiente fragmento de código, suministramos **executeUpdate** con la sentencia SQL del ejemplo anterior.

```
stmt.executeUpdate("CREATE TABLE COFFEES " +
                   "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, "
+
                   "SALES INTEGER, TOTAL INTEGER)");
```

Coma ya habíamos creado un String con la sentencia SQL y lo habíamos llamado **createTableCoffees**, podríamos haber escrito el código de esta forma alternativa.

```
stmt.executeUpdate(createTableCoffees);
```

Ejecutar Sentencias

Utilizamos el método **executeUpdate** porque la sentencia SQL contenida en **createTableCoffees** es una sentencia DDL (data definition language). Las sentencias que crean, modifican o eliminan tablas son todas ejemplos de sentencias DDL y se ejecutan con el método **executeUpdate**. Cómo se podría esperar de su nombre, el método **executeUpdate** también se utiliza para ejecutar sentencias SQL que actualizan un tabla. En la práctica **executeUpdate** se utiliza más frecuentemente para actualizar tablas que para crearlas porque una tabla se crea sólo una vez, pero se puede actualizar muchas veces.

El método más utilizado para ejecutar sentencias SQL es **executeQuery**. Este método se utiliza para ejecutar sentencias **SELECT**, que comprenden la amplia mayoría de las sentencias SQL. Pronto veremos como utilizar este método.

Introducir Datos en una Tabla

Hemos visto como crear la tabla **COFFEES** especificando los nombres de columnas y los tipos de datos almacenados en esas columnas, pero esto sólo configura la estructura de la tabla. La tabla no contiene datos todavía. Introduciremos datos en nuestra tabla una fila cada vez, suministrando la información a almacenar en cada columna de la fila. Observa que los valores insertados en las columnas se listan en el mismo orden en que se declararon las columnas cuando se creó la tabla, que es el orden por defecto.

El siguiente código isnerta una fila de datos con **Colombian** en la columna **COF_NAME**, **101** en **SUP_ID**, **7.99** en **PRICE**, **0** en **SALES**, y **0** en **TOTAL**. (Como acabamos de inaugurar "The Coffee Break", la cantidad vendida durante la semana y la cantidad total son cero para todos los cafés). Al igual que hicimos con el código que creaba la tabla **COFFEES**, crearemos un objeto **Statement** y lo ejecutaremos utilizando el método **executeUpdate**.

Como la sentencia SQL es damasiado larga como para entrar en una sóla línea, la hemos dividido en dos strings concatenándolas mediante un signo más (+) para que puedan compilarse. Presta especial atención a la necesidad de un espacio entre **COFFEES** y **VALUES**.

Este espacio debe estar dentro de las comillas y debe estar después de **COFFEES** y antes de **VALUES**; sin un espacio, la sentencia SQL sería leída erróneamente como "**INSERT INTO COFFEESVALUES . .** ." y el controlador de la base de datos buscaría la tabla**COFFEESVALUES**. Observa también que utilizamos comilla simples alrededor del nombre del café porque está anidado dentro de las comillas dobles. Para la mayoría de controladores de bases de datos, la regla general es alternar comillas dobles y simples para indicar anidación.

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
    "INSERT INTO COFFEES " +
    "VALUES ('Colombian', 101, 7.99, 0, 0)");
```

El siguiente código inserta una segunda línea dentro de la tabla **COFFEES**. Observa que hemos reutilizado el objeto **Statement**: **stmt** en vez de tener que crear uno nuevo para cada ejecución.

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Los valores de las siguientes filas se pueden insertar de esta forma.

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Obtener Datos desde una Tabla

Ahora que la tabla**COFFEES** tiene valores, podemos escribir una sentencia **SELECT** para acceder a dichos valores. El asterisco (*) en la siguiente sentencia SQL indica que la columna debería ser seleccionada. Como no hay claúsula **WHERE** que limite las columas a seleccionar, la siguiente sentencia SQL seleciona la tabla completa.

```
SELECT * FROM COFFEES
```

El resultado, que es la tabla completa, se parecería a esto.

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

El resultado anterior es lo que veríamos en nuestro terminal si introdujeramos la petición SQL directamente en el sistema de la base de datos. Cuando accedemos a una base de datos a través de una aplicación Java, como veremos pronto, necesitamos recuperar los resultados para poder utilizarlos. Veremos como hacer esto en la siguiente página.

Aquí tenemos otro ejemplo de una sentencia **SELECT**, ésta obtiene una lista de cáfes y sus respectivos precios por libra.

```
SELECT COF_NAME, PRICE FROM COFFEES
```

El resultado de esta consulta se parecería a esto.

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

La sentencia **SELECT** genera los nombres y precios de todos los cáfes de la tabla. La siguiente sentencia SQL límita los cafés seleccionados a aquellos que cuesten menos de $9.00 por libra.

```
SELECT COF_NAME, PRICE
FROM COFFEES
WHERE PRICE < 9.00
```

El resultado se parecería es esto.

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Colombian Decaf	8.99

Recuperar Valores desde una Hoja de Resultados

Ahora veremos como enviar la sentencia **SELECT** de la página anterior desde un programa escrito en Java y como obtener los resultados que hemos mostrado.

JDBC devuelve los resultados en un objeto **ResultSet**, por eso necesitamos declarar un ejemplar de la clase **ResultSet** para contener los resultados. El siguiente código presenta el objeto **ResultSet**: **rs** y le asigna el resultado de una consulta anterior.

```
ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
```

Utilizar el Método next

La variable **rs**, que es un ejemplar de **ResultSet**, contiene las filas de cafés y sus precios mostrados en el juego de resultados de la página anterior. Para acceder a los nombres y los precios, iremos a la fila y recuperaremos los valores de acuerdo con sus tipos. El método **next** mueve algo llamado cursor a la siguiente fila y hace que esa fila (llamada fila actual) sea con la que podamos operar. Como el cursor inicialmente se posiciona justo encima de la primera fila de un objeto **ResultSet**, primero debemos llamar al método **next** para mover el cursor a la primera fila y convertirla en la fila actual. Sucesivas invocaciones del método **next** moverán el

cursor de línea en línea de arriba a abajo. Observa que con el JDBC 2.0, cubierto en la siguiente sección, se puede mover el cursor hacia atrás, hacia posiciones específicas y a posiciones relativas a la fila actual además de mover el cursor hacia adelante.

Utilizar los métodos getXXX

Los métodos **getXXX** del tipo apropiado se utilizan para recuperar el valor de cada columna. Por ejemplo, la primera columna de cada fila de **rs** es **COF_NAME**, que almacena un valor del tipo **VARCHAR** de SQL. El método para recuperar un valor **VARCHAR** es **getString**. La segunda columna de cada fila almacena un valor del tipo **FLOAT** de SQL, y el método para recuperar valores de ese tipo es **getFloat**. El siguiente código accede a los valores almacenados en la fila actual de **rs** e imprime una línea con el nombre seguido por tres espacios y el precio. Cada vez que se llama al método **next**, la siguiente fila se convierte en la actual, y el bucle continúa hasta que no haya más filas en **rs**.

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
               while (rs.next()) {
               String s = rs.getString("COF_NAME");
               Float n = rs.getFloat("PRICE");
               System.out.println(s + "   " + n);
}
```

La salida se parecerá a esto.

```
Colombian   7.99
French_Roast   8.99
Espresso   9.99
Colombian_Decaf   8.99
French_Roast_Decaf   9.99
```

Veamos cómo funcionan los métodos **getXXX** examinando las dos sentencias **getXXX** de este código. Primero examinaremos **getString**.

```
String s = rs.getString("COF_NAME");
```

El método **getString** es invocado sobre el objeto **ResultSet**: **rs**, por eso **getString** recuperará (obtendrá) el valor almacenado en la columna **COF_NAME** de la fila actual de **rs**. El valor recuperado por **getString** se ha convertido desde un **VARCHAR** de SQL a un **String** de Java y se ha asignado al objeto **String s**. Observa que utilizamos la variable **s** en la expresión **println** mostrada arriba, de esta forma: **println(s + " " + n)**

La situación es similar con el método **getFloat** excepto en que recupera el valor almacenado en la columna **PRICE**, que es un **FLOAT** de SQL, y lo convierte a un **float** de Java antes de asignarlo a la variable **n**.

JDBC ofrece dos formas para identificar la columna de la que un método **getXXX** obtiene un valor. Una forma es dar el nombre de la columna, como se ha hecho arriba. La segunda forma es dar el índice de la columna (el número de columna), con un **1** significando la primera columna, un **2** para la segunda, etc. Si utilizáramos el número de columna en vez del nombre de columna el código anterior se podría parecer a esto.

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

La primera línea de código obtiene el valor de la primera columna de la fila actual de **rs** (columna **COF_NAME**), convirtiéndolo a un objeto **String** de Java y asignándolo a **s**. La segunda línea de código obtiene el valor de la segunda columna de la fila actual de **rs**, lo convierte a un **float** de Java y lo asigna a **n**. Recuerda que el número de columna se refiere al número de columna en la hoja de resultados no en la tabla original.

En suma, JDBC permite utilizar tanto el nombre cómo el número de la columna como argumento a un método **getXXX**. Utilizar el número de columna es un poco más eficiente, y hay algunos casos donde es necesario utilizarlo.

JDBC permite muchas lateralidades para utilizar los métodos **getXXX** para obtener diferentes tipos de datos SQL. Por ejemplo, el método **getInt** puede ser utilizado para recuperar cualquier tipo numérico de caracteres. Los datos recuperados serán convertidos a un **int**; esto es, si el tipo SQL es **VARCHAR**, JDBC intentará convertirlo en un entero. Se recomienda utilizar el método **getInt** sólo para recuperar **INTEGER** de SQL, sin embargo, no puede utilizarse con los tipos **BINARY**, **VARBINARY**, **LONGVARBINARY**, **DATE**, **TIME**, o **TIMESTAMP** de SQL.

Métodos para Recuperar Tipos SQL muestra qué métodos pueden utilizarse legalmente para recuperar tipos SQL, y más importante, qué métodos están recomendados para recuperar los distintos tipos SQL. Observa que esta tabla utiliza el término "JDBC type" en lugar de "SQL type." Ambos términos se refieren a los tipos genéricos de SQL definidos en **java.sql.Types**, y ambos son intercambiables.

Utilizar el método getString

Aunque el metodo **getString** está recomendado para recuperar tipos **CHAR** y **VARCHAR** de SQL, es posible recuperar cualquier tipo básico SQL con él. (Sin embargo, no se pueden recuperar los nuevos tipos de datoas del SQL3. Explicaremos el SQL3 más adelante).

Obtener un valor con **getString** puede ser muy útil, pero tiene sus limitaciones. Por ejemplo, si se está utilizando para recuperar un tipo numérico, **getString** lo convertirá en un **String** de Java, y el valor tendrá que ser convertido de nuevo a número antes de poder operar con él.

Utilizar los métodos de ResultSet.getXXX para Recuperar tipos JDBC

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR
getByte	X	x	x	x	x	x	x	x	x	x	x
getShort	x	X	x	x	x	x	x	x	x	x	x
getInt	x	x	X	x	x	x	x	x	x	x	x
getLong	x	x	x	X	x	x	x	x	x	x	x
getFloat	x	x	x	x	X	x	x	x	x	x	x
getDouble	x	x	x	x	x	X	X	x	x	x	x
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x
getBoolean	x	x	x	x	x	x	x	x	x	X	x
getString	x	x	x	x	x	x	x	x	x	x	X
getBytes											
getDate											x
getTime											x
getTimestamp											x
getAsciiStream											x
getUnicodeStream											x
getBinaryStream											
getObject	x	x	x	x	x	x	x	x	x	x	x

	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	VARCHAR
getByte	x	x						
getShort	x	x						
getInt	x	x						
getLong	x	x						
getFloat	x	x						
getDouble	x	x						
getBigDecimal	x	x						
getBoolean	x	x						
getString	X	x	x	x	x	x	x	x
getBytes			X	X	x			
getDate	x	x				X		x
getTime	x	x					X	x
getTimestamp	x	x				x	x	X
getAsciiStream	x	X	x	x	x			
getUnicodeStream	x	X	x	x	x			
getBinaryStream			x	x	X			
getObject	x	x	x	x	x	x	x	x

Una "x" indica que el método **getXXX** se puede utilizar legalmente para recuperar el tipo JDBC dado.

Una "X" indica que el método **getXXX** está recomendado para recuperar el tipo JDBC dado.

Actualizar Tablas

Supongamos que después de una primera semana exitosa, el propietario de "The Coffee Break" quiere actualizar la columna **SALES** de la tabla **COFFEES** introduciendo el número de libras vendidas de cada tipo de café. La sentencia SQL para actualizar una columna se podría parecer a esto.

```
String updateString = "UPDATE COFFEES " +
        "SET SALES = 75 " +
        "WHERE COF_NAME LIKE 'Colombian'";
```

Uitlizando el objeto **stmt**, este código JDBC ejecuta la sentencia SQL contenida en **updateString**.

```
stmt.executeUpdate(updateString);
```

La tabla **COFFEES** ahora se parecerá a esto.

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Observa que todavía no hemos actualizado la columna **TOTAL**, y por eso tiene valor **0**.

Ahora seleccionaremos la fila que hemos actualizado, recuperando los valores de las columnas **COF_NAME** y **SALES**, e imprimiendo esos valores.

```
String query = "SELECT COF_NAME, SALES FROM COFFEES " +
               "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
               String s = rs.getString("COF_NAME");
               int n = rs.getInt("SALES");
               System.out.println(n + " pounds of " + s +
                                                  " sold this week.")
}
```

Esto imprimira lo siguiente.

```
75 pounds of Colombian sold this week.
```

Cómo la claúsula **WHERE** límita la selección a una sóla línea, sólo hay una línea en la **ResultSet**: **rs** y una línea en la salida. Por lo tanto, sería posible escribir el código sin un bucle **while**.

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println(n + " pounds of " + s + " sold this week.")
```

Aunque hay una sóla línea en la hoja de resultados, necesitamos utilizar el método **next** para acceder a ella. Un objeto **ResultSet** se crea con un cursor apuntando por encima de la primera fila. La primera llamada al método **next** posiciona el cursor en la primera fila (y en este caso, la única) de **rs**. En este código, sólo se llama una vez a **next**, si sucediera que existiera una línea, nunca se accedería a ella.

Ahora actualizaremos la columna **TOTAL** añadiendo la cantidad vendida durante la semana a la cantidad total existente, y luego imprimiremos el número de libras vendidas hasta la fecha.

```
String updateString = "UPDATE COFFEES " +
                      "SET TOTAL = TOTAL + 75 " +
                      "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT COF_NAME, TOTAL FROM COFFEES " +
                                   "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
               String s = rs.getString(1);
               int n = rs.getInt(2);
               System.out.println(n + " pounds of " + s + " sold to
date.")
}
```

Observa que en este ejemplo, utilizamos el índice de columna en vez del nombre de columna, suministrando el índice **1** a **getString** (la primera columna de la hoja de resultados es **COF_NAME**), y el índice **2** a **getInt** (la segunda columna de la hoja de resultados es **TOTAL**). Es importante distinguir entre un índice de columna en la tabla de la base de datos como opuesto al índice en la tabla de la hoja de resultados. Por ejemplo, **TOTAL** es la quinta columna en la tabla **COFFEES** pero es la segunda columna en la hoja de resultados generada por la petición del ejemplo anterior.

Utilizar Sentencias Preparadas

Algunas veces es más conveniente o eficiente utilizar objetos **PreparedStatement** para enviar sentencias SQL a la base de datos. Este tipo especial de sentencias se deriva de una clase más general, Statement, que ya conocemos.

Cuándo utilizar un Objeto PreparedStatement

Si queremos ejecutar muchas veces un objeto **Statement**, reduciremos el tiempo de ejecución si utilizamos un objeto **PreparedStatement**, en su lugar.

La caracterísitca principal de un objeto **PreparedStatement** es que, al contrario que un objeto **Statement**, se le entrega una sentencia SQL cuando se crea. La ventaja de esto es que en la

mayoría de los casos, esta sentencia SQL se enviará al controlador de la base de datos inmediatamente, donde será compilado. Como resultado, el objeto **PreparedStatement** no sólo contiene una sentencia SQL, sino una sentencia SQL que ha sido precompilada. Esto significa que cuando se ejecuta la **PreparedStatement**, el controlador de base de datos puede ejecutarla sin tener que compilarla primero.

Aunque los objetos **PreparedStatement** se pueden utilizar con sentencias SQL sin parámetros, probablemente nosotros utilizaremos más frecuentemente sentencias con parámetros. La ventajA de utilizar sentencias SQL que utilizan parámetros es que podemos utilizar la misma sentencia y suministrar distintos valores cada vez que la ejecutemos. Veremos un ejemplo de esto en las página siguientes.

Crear un Objeto PreparedStatement

Al igual que los objetos **Statement**, creamos un objeto **PreparedStatement** con un objeto **Connection**. Utilizando nuestra conexión **con** abierta en ejemplos anteriores, podríamos escribir lo siguiente para crear un objeto **PreparedStatement** que tome dos parámetros de entrada.

```
PreparedStatement updateSales = con.prepareStatement(
                        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME
LIKE ?");
```

La variable **updateSales** contiene la sentencia SQL, **"UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?"**, que también ha sido, en la mayoría de los casos, enviada al controlador de la base de datos, y ha sido precompilado.

Suministrar Valores para los Parámetros de un PreparedStatement

Necesitamos suministrar los valores que se utilizarán en los luegares donde están las marcas de interrogación, si hay alguno, antes de ejecutar un objeto **PreparedStatement**. Podemos hacer esto llamado a uno de los métodos **setXXX** definidos en la clase **PreparedStatement**. Si el valor que queremos sustituir por una marca de interrogación es un **int** de Java, podemos llamar al método **setInt**. Si el valor que queremos sustituir es un **String** de Java, podemos llamar al método **setString**, etc. En general, hay un método **setXXX** para cada tipo Java.

Utilizando el objeto **updateSales** del ejemplo anterior, la siguiente línea de código selecciona la primera marca de interrogación para un **int** de Java, con un valor de 75.

```
updateSales.setInt(1, 75);
```

Cómo podríamos asumir a partir de este ejemplo, el primer argumento de un método **setXXX** indica la marca de interrogación que queremos seleccionar, y el segundo argumento el valor que queremos ponerle. El siguiente ejemplo selecciona la segunda marca de interrogación con el string "**Colombian**".

```
updateSales.setString(2, "Colombian");
```

Después de que estos valores hayan sido asignados para sus dos parámetros, la sentencia SQL de **updateSales** será equivalente a la sentencia SQL que hay en string **updateString** que utilizando en el ejemplo anterior. Por lo tanto, los dos fragmentos de código siguientes consiguen la misma cosa.

Código 1.

```
String updateString = "UPDATE COFFEES SET SALES = 75 " +
                      "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

Código 2.

```
PreparedStatement updateSales = con.prepareStatement(
        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate().
```

Utilizamos el método **executeUpdate** para ejecutar ambas sentencias **stmt updateSales**. Observa, sin embargo, que no se suministran argumentos a **executeUpdate** cuando se utiliza para ejecutar **updateSales**. Esto es cierto porque **updateSales** ya contiene la sentencia SQL a ejecutar.

Mirando esto ejemplos podríamos preguntarnos por qué utilizar un objeto **PreparedStatement** con parámetros en vez de una simple sentencia, ya que la sentencia simple implica menos pasos. Si actualizáramos la columna **SALES** sólo una o dos veces, no sería necesario utilizar una sentencia SQL con parámetros. Si por otro lado, tuvieramos que actualizarla frecuentemente, podría ser más fácil utilizar un objeto **PreparedStatement**, especialmente en situaciones cuando la utilizamos con un bucle while para seleccionar un parámetro a una sucesión de valores. Veremos este ejemplo más adelante en esta sección.

Una vez que a un parámetro se ha asignado un valor, el valor permanece hasta que lo resetee otro valor o se llame al método **clearParameters**. Utilizando el objeto **PreparedStatement**: **updateSales**, el siguiente fragmento de código reutiliza una sentencia prepared después de resetar el valor de uno de sus parámetros, dejando el otro igual.

```
updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();
// changes SALES column of French Roast row to 100
updateSales.setString(2, "Espresso");
updateSales.executeUpdate();
// changes SALES column of Espresso row to 100 (the first
// parameter stayed 100, and the second parameter was reset
// to "Espresso")
```

Utilizar una Bucle para asignar Valores

Normalmente se codifica más sencillo utilizando un bucle **for** o **while** para asignar valores de los parámetros de entrada.

El siguiente fragmento de código demuestra la utilización de un bucle **for** para asignar los parámetros en un objeto **PreparedStatement**: **updateSales**. El array **salesForWeek** contiene las cantidades vendidas semanalmente. Estas cantidades corresponden con los nombres de los cafés listados en el array **coffees**, por eso la primera cantidad de **salesForWeek** (175) se aplica al primer nombre de café de **coffees** ("**Colombian**"), la segunda cantidad de **salesForWeek** (150) se aplica al segundo nombre de café en **coffees** ("**French_Roast**"), etc. Este fragmento de código demuestra la actualización de la columna **SALES** para todos los cafés de la tabla **COFFEES**

```
PreparedStatement updateSales;
String updateString = "update COFFEES " +
```

```
                    "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);int [] salesForWeek =
{175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
                    "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
               updateSales.setInt(1, salesForWeek[i]);
               updateSales.setString(2, coffees[i]);
               updateSales.executeUpdate();
       }
```

Cuando el propietario quiera actualizar las ventas de la semana siguiente, puede utilizar el mismo código como una plantilla. Todo lo que tiene que haces es introducir las nuevas cantidades en el orden apropiado en el array **salesForWeek**. Los nombres de cafés del array **coffees** permanecen constantes, por eso no necesitan cambiarse. (En una aplicación real, los valores probablemente serían introducidos por el usuario en vez de desde un array inicializado).

Valores de retorno del método executeUpdate

Siempre que **executeQuery** devuelve un objeto **ResultSet** que contiene los resultados de una petición al controlador de la base datos, el valor devuelto por **executeUpdate** es un **int** que indica cuántas líneas de la tabla fueron actualizadas. Por ejemplo, el siguiente código muestra el valor de retorno de **executeUpdate** asignado a la variable **n**.

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it
```

La tabla **COFFEES** se ha actualziado poniendo el valor **50** en la columna **SALES** de la fila correspondiente a **Espresso**. La actualización afecta sólo a una línea de la tabla, por eso **n** es igual a **1**.

Cuando el método **executeUpdate** es utilizado para ejecutar una sentecia DDL, como la creación de una tabla, devuelve el **int**: **0**. Consecuentemente, en el siguiente fragmento de código, que ejecuta la sentencia DDL utilizada pra crear la tabla **COFFEES**, **n** tendrá el valor **0**.

```
int n = executeUpdate(createTableCoffees); // n = 0
```

Observa que cuando el valor devuelto por **executeUpdate** sea **0**, puede significar dos cosas: (1) la sentencia ejecutada no ha actualizado ninguna fila, o (2) la sentencia ejecutada fue una sentencia DDL.

Utilizar Uniones

Algunas veces necesitamos utilizar una o más tablas para obtener los datos que queremos. Por ejemplo, supongamos que el propietario del "The Coffee Break" quiere una lista de los cafés que le compra a Acme, Inc. Esto implica información de la tabla **COFFEES** y también de la que vamos a crear **SUPPLIERS**. Este es el caso en que se necesitan los "joins" (unión). Una unión es una operación de base de datos que relaciona dos o más tablas por medio de los valores que comparten. En nuestro ejemplo, las tablas **COFFEES** y **SUPPLIERS** tienen la columna **SUP_ID**, que puede ser utilizada para unirlas.

Antes de ir más allá, necesitamos crear la tabla **SUPPLIERS** y rellenarla con valores.

El sigueinte código crea la tabla **SUPPLIERS**.

```
String createSUPPLIERS = "create table SUPPLIERS " +
                         "(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
                         "STREET VARCHAR(40), CITY VARCHAR(20), " +
                         "STATE CHAR(2), ZIP CHAR(5))";
stmt.executeUpdate(createSUPPLIERS);
```

El siguiente código inserta filas para tres suministradores dentro de **SUPPLIERS**.

```
stmt.executeUpdate("insert into SUPPLIERS values (101, " +
                "'Acme, Inc.', '99 Market Street', 'Groundsville', " +
"'CA', '95199'");
stmt.executeUpdate("Insert into SUPPLIERS values (49," +
                "'Superior Coffee', '1 Party Place', 'Mendocino', 'CA',
" + "'95460'");
stmt.executeUpdate("Insert into SUPPLIERS values (150, " +
                "'The High Ground', '100 Coffee Lane', 'Meadows', 'CA',
" + "'93966'");
```

El siguiente código selecciona la tabla y nos permite verla.

```
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
```

El resultado sería algo similar a esto.

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Ahora que tenemos las tablas **COFFEES** y **SUPPLIERS**, podremos proceder con el escenario en que el propietario quería una lista de los cafés comprados a un suministrador particular. Los nombres de los suminstradores están en la tabla **SUPPLIERS**, y los nombres de los cafés en la tabla **COFFEES**. Como ambas tablas tienen la columna **SUP_ID**, podemos utilizar esta columna en una unión. Lo siguiente que necesitamos es la forma de distinguir la columna **SUP_ID** a la que nos referimos. Esto se hace precediendo el nombre de la columna con el nombre de la tabla, "**COFFEES.SUP_ID**" para indicar que queremos referirnos a la columna **SUP_ID** de la tabla **COFFEES**. En el siguiente código, donde **stmt** es un objeto **Statement**, seleccionamos los cafés comprados a Acme, Inc..

```
String query = "
SELECT COFFEES.COF_NAME " +
   "FROM COFFEES, SUPPLIERS " +
   "WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.'" +
   "and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";

ResultSet rs = stmt.executeQuery(query);
System.out.println("Coffees bought from Acme, Inc.: ");
while (rs.next()) {
```

```
                String coffeeName = getString("COF_NAME");
                System.out.println("     " + coffeeName);
}
```

Esto producirá la siguiente salida.

```
Coffees bought from Acme, Inc..
     Colombian
     Colombian_Decaf
```

Utilizar Transaciones

Hay veces que no queremos que una sentencia tenga efecto a menos que otra también suceda. Por ejemplo, cuando el propietario del "The Coffee Break" actualiza la cantidad de café vendida semanalmente, también querrá actualizar la cantidad total vendida hasta la fecha. Sin embargo, el no querrá actualizar una sin actualizar la otra; de otro modo, los datos serían inconsistentes. La forma para asegurarnos que ocurren las dos acciones o que no ocurre ninguna es utilizar una transación. Una transación es un conjunto de una o más sentencias que se ejecutan como una unidad, por eso o se ejecutan todas o no se ejecuta ninguna.

Desactivar el modo Auto-entrega

Cuando se crea una conexión, está en modo auto-entrega. Esto significa que cada sentencia SQL individual es tratada como una transación y será automáticamente entregada justo después de ser ejecutada. (Para ser más preciso, por defecto, una sentencia SQL será entregada cuando está completa, no cuando se ejecuta. Una sentencia está completa cuando todas sus hojas de resultados y cuentas de actualización han sido recuperadas. Sin embargo, en la mayoría de los casos, una sentencia está completa, y por lo tanto, entregada, justo después de ser ejecutada).

La forma de permitir que dos o más sentencia sean agrupadas en una transación es desactivar el modo auto-entrega. Esto se demuestra en el siguiente código, donde **con** es una conexión activa.

```
con.setAutoCommit(false);
```

Entregar una Transación

Una vez que se ha desactivado la auto-entrega, no se entregará ninguna sentencia SQL hasta que llamemos explícitamente al método **commit**. Todas las sentencias ejecutadas después de la anterior llamada al método **commit** serán incluidas en la transación actual y serán entregadas juntas como una unidad. El siguiente código, en el que **con** es una conexión activa, ilustra una transación.

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
                        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME
LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
        "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
```

```
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

En este ejemplo, el modo auto-entrega se desactiva para la conexión **con**, lo que significa que las dos sentencias prepared **updateSales** y **updateTotal** serán entregadas juntas cuando se llame al método **commit**. Siempre que se llame al método **commit** (bien automáticamente, cuando está activado el modo auto-commit o explícitamente cuando está desactivado), todos los cambios resultantes de las sentencias de la transación serán permanentes. En este caso, significa que las columnas **SALES** y **TOTAL** para el café Colombian han sido cambiadas a **50** (si **TOTAL** ha sido **0** anteriormente) y mantendrá este valor hasta que se cambie con otra sentencia de actualización.

La línea final del ejemplo anterior activa el modo auto-commit, lo que significa que cada sentencia será de nuevo entregada automáticamente cuando esté completa. Volvemos por lo tanto al estado por defecto, en el que no tenemos que llamar al método **commit**. Es bueno desactivar el modo auto-commit sólo mientras queramos estar en modo transación. De esta forma, evitamos bloquear la base de datos durante varias sentencias, lo que incrementa los conflictos con otros usuarios.

Utilizar Transaciones para Preservar al Integridad de los Datos

Además de agrupar las sentencias para ejecutarlas como una unidad, las transaciones pueden ayudarnos a preservar la integridad de los datos de una tabla. Por ejemplo, supongamos que un empleado se ha propuesto introducir los nuevos precios de los cafés en la tabla **COFFEES** pero lo retrasa unos días. Mientras tanto, los precios han subido, y hoy el propietario está introduciendo los nuevos precios. Finalmente el empleado empieza a intrudir los precios ahora desfasados al mismo tiempo que el propietario intenta actualizar la tabla. Después de insertar los precios desfasados, el empleado se da cuenta de que ya no son válidos y llama el método **rollback** de la **Connection** para deshacer sus efectos. (El método **rollback** aborta la transación y restaura los valores que había antes de intentar la actualziación. Al mismo tiempo, el propietario está ejecutando una sentencia **SELECT** e imprime los nuevos precios. En esta situación, es posible que el propietario imprima los precios que más tarde serían devueltos a sus valores anteriores, haciendo que los precio impresos sean incorrectos.

Esta clase de situaciones puede evitarse utilizando Transaciones. Si un controlador de base de datos soporta transaciones, y casi todos lo hacen, proporcionará algún nivel de protección contra conflictos que pueden surgir cuando dos usuarios acceden a los datos a la misma vez.

Para evitar conflictos durante una transación, un controlador de base de datos utiliza bloqueos, mecanismos para bloquear el acceso de otros a los datos que están siendo accedidos por una transación. (Observa que en el modo auto-commit, donde cada sentencia es una transación, el bloqueo sólo se mantiene durante una sentencia). Una vez activado, el bloqueo permanece hasta que la transación sea entregada o anulada. Por ejemplo, un controlador de base de datos podría bloquear una fila de una tabla hasta que la actualización se haya entregado. El efecto de este bloqueo es evitar que usuario obtenga una lectura sucia, esto es, que lea un valor antes de que sea permanente. (Acceder a un valor actualizado que no haya sido entregado se considera una lectura sucia porque es posible que el valor sea devuelto a su valor anterior. Si leemos un valor que luego es devuelto a su valor antiguo, habremos leído un valor nulo).

La forma en que se configuran los bloqueos está determinado por lo que se llama nivel de aislamiento de transación, que pude variar desde no soportar transaciones en absoluto a soportar todas las transaciones que fuerzan una reglas de acceso muy estrictas.

Un ejemplo de nivel de aislamiento de transación es **TRANSACTION_READ_COMMITTED**, que no permite que se acceda a un valor hasta que haya sido entregado. En otras palabras, si nivel de aislamiento de transación se selecciona a **TRANSACTION_READ_COMMITTED**, el controlador de la base de datos no permitirá que ocurran lecturas sucias. El interface **Connection** incluye cinco valores que representan los niveles de aislamiento de transación que se pueden utilizar en JDBC.

Normalmente, no se necesita cambiar el nivel de aislamiento de transación; podemos utilizar el valor por defecto de nuestro controlador. JDBC permite averiguar el nivel de aislamiento de transación de nuestro controlador de la base de datos (utilizando el método **getTransactionIsolation** de **Connection**) y permite configurarlo a otro nivel (utilizando el método **setTransactionIsolation** de **Connection**). Sin embargo, ten en cuenta, que aunque JDBC permite seleccionar un nivel de aislamiento, hacer esto no tendrá ningún efecto a no ser que el driver del controlador de la base de datos lo soporte.

Cuándo llamar al método rollback

Como se mencionó anteriormente, llamar al método **rollback** aborta la transación y devuelve cualquier valor que fuera modificado a sus valores anteriores. Si estamos intentando ejecutar una o más sentencias en una transación y obtenemos una **SQLException**, deberíamos llamar al método **rollback** para abortar la transación y empezarla de nuevo. Esta es la única forma para asegurarnos de cuál ha sido entregada y cuál no ha sido entregada. Capturar una **SQLException** nos dice que hay algo erróneo, pero no nos dice si fue o no fue entregada. Como no podemos contar con el hecho de que nada fue entregado, llamar al método **rollback** es la única forma de asegurarnos.

Procedimientos Almacenados

Un procedimiento almacenado es un grupo de sentencias SQL que forman una unidad lógica y que realizan una tarea particular. Los procedimientos almacenados se utilizan para encapsular un conjunto de operaciones o peticiones para ejecutar en un servidor de base de datos. Por ejemplo, las operaciones sobre una base de datos de empleados (salarios, despidos, promociones, bloqueos) podrían ser codificados como procedimientos almacenados ejecutados por el código de la aplicación. Los procedimientos almacenados pueden compilarse y ejecutarse con diferentes parámetros y resultados, y podrían tener cualquier combinación de parámtros de entrada/salida.

> Los procedimientos almacenados están soportados por la mayoría de los controladores de bases de datos, pero existe una gran cantidad de variaciones en su síntaxis y capacidades. Por esta razón, sólo mostraremos un ejemplo sencillo de lo que podría ser un procedimiento almacenado y cómo llamarlos desde JDBC, pero este ejemplo no está diseñado para ejecutarse.

Utilizar Sentencias SQL

Esta página muestra un procedimiento almacenado muy sencillo que no tiene parámetros. Aunque la mayoría de los procedimientos almacenados hacen cosas más complejas que este ejemplo, sirve para ilustrar algunos puntos básicos sobre ellos. Como paso prévio, la sintaxis para definir un procedimiento almacenado es diferente de un controlador de base de datos a otro. Por ejemplo, algunos utilizan **begin . . . end** u otras palabras clave para indicar el principio y final de la definición de procedimiento. En algunos controladores, la siguiente sentencia SQL crea un procedimiento almacenado.

```
create procedure SHOW_SUPPLIERS
as
```

```
select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
from SUPPLIERS, COFFEES
where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
order by SUP_NAME
```

El siguiente código pone la sentencia SQL dentro de un string y lo asigna a la variable **createProcedure**, que utilizaremos más adelante.

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
                         "as " +
                         "select SUPPLIERS.SUP_NAME,
COFFEES.COF_NAME " +
                         "from SUPPLIERS, COFFEES " +
                         "where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
" +
                         "order by SUP_NAME";
```

El siguiente fragmento de código utiliza el objeto **Connection**, **con** para crear un objeto **Statement**, que es utilizado para enviar la sentencia SQL que crea el procedimiento almacenado en la base de datos.

```
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

El procedimiento **SHOW_SUPPLIERS** será compilado y almacenado en la base de datos como un objeto de la propia base y puede ser llamado, como se llamaría a cualquier otro método.

Llamar a un Procedimiento Almacenado desde JDBC

JDBC permite llamar a un procedimiento almacenado en la base de datos desde una aplicación escrita en Java. El primer paso es crear un objeto **CallableStatement**. Al igual que con los objetos **Statement** y **PreparedStatement**, esto se hace con una conexión abierta, **Connection**. Un objeto **CallableStatement** contiene una llamada a un procedimiento almacenado; no contiene el propio procedimiento. La primera línea del código siguiente crea una llamada al procedimiento almacenado **SHOW_SUPPLIERS** utilizando la conexión **con**. La parte que está encerrada entre corchetes es la sintaxis de escape para los precedimientos almacenados. Cuando un controlador encuentra **"{call SHOW_SUPPLIERS}"**, traducirá esta sintaxis de escape al SQL nativo utilizado en la base de datos para llamar al procedimiento almacenado llamado **SHOW_SUPPLIERS**.

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

La hoja de resultados de **rs** será similar a esto.

```
SUP_NAME                      COF_NAME
----------------              ----------------------
Acme, Inc.                    Colombian
Acme, Inc.                    Colombian_Decaf
Superior Coffee               French_Roast
Superior Coffee               French_Roast_Decaf
The High Ground               Espresso
```

Observa que el método utilizado para ejecutar **cs** es **executeQuery** porque **cs** llama a un procedimiento almacenado que contiene una petición y esto produce una hoja de resultados. Si el procedimiento hubiera contenido una sentencia de actualziación o una sentencia DDL, se

hubiera utilizado el método **executeUpdate**. Sin embargo, en algunos casos, cuando el procedimiento almacenado contiene más de una sentencia SQL producirá más de una hoja de resultados, o cuando contiene más de una cuenta de actualizaciónm o alguna combinación de hojas de resultados y actualizaciones. en estos casos, donde existen múltiples resultados, se debería utilizar el método **execute** para ejecutar **CallableStatement**.

La clase **CallableStatement** es una subclase de **PreparedStatement**, por eso un objeto **CallableStatement** puede tomar parámetros de entrada como lo haría un objeto **PreparedStatement**. Además, un objeto **CallableStatement** puede tomar parámetros de salida, o parámetros que son tanto de entrada como de salida. Los parámetros INOUT y el método **execute** se utilizan raramente.

Crear Aplicaciones JDBC Completas

Hasta ahora sólo hemos visto fragmentos de código. Más adelante veremos programas de ejemplo que son aplicaciones completas que podremos ejecutar.

El primer código de ejemplo crea la tabla **COFFEES**; el segundo inserta valores en la tabla e imprime los resultados de una petición. La terecera aplicación crea la tabla **SUPPLIERS**, y el cuarto la rellena con valores. Después de haber ejecutado este código, podemos intentar una petición que una las tablas **COFFEES** y **SUPPLIERS**, como en el quinto código de ejemplo. El sexto ejemplo de código es una aplicación que demuestra una transación y también muestra como configurar las posiciones de los parámetros en un objeto **PreparedStatement** utilizando un bucle **for**.

Como son aplicaciones completas, incluyen algunos elementos del lenguaje Java que no hemos visto en los fragmentos anteriores. Aquí explicaremos estos elementos brevemente.

Poner Código en una Definición de Clase

En el lenguaje Java, cualquier código que querramos ejecutar debe estar dentro de una definición de clase. Tecleamos la definición de clase en un fichero y a éste le damos el nombre de la clase con la extensión **.java**. Por eso si tenemos una clase llamada **MySQLStatement**, su definición debería estar en un fichero llamado **MySQLStatement.java**

Importar Clases para Hacerlas Visibles

Lo primero es importar los paquetes o clases que se van a utilizar en la nueva clase. Todas las clases de nuestros ejemplos utilizan el paquete **java.sql** (el API JDBC), que se hace visible cuando la siguiente línea de código precede a la definición de clase.

```
import java.sql.*;
```

El asterisco (*****) indica que todas las clases del paquete **java.sql** serán importadas. Importar una clase la hace visible y significa que no tendremos que escribir su nombre totalmente cualificado cuando utilicemos un método o un campo de esa clase. Si no incluimos "**import java.sql.*;**" en nuestro código, tendríamos que escribir "**java.sql.**" más el nombre de la clase delante de todos los campos o métodos JDBC que utilicemos cada vez que los utilicemos. Observa que también podemos importar clases individuales selectivamente en vez de importar un paquete completo. Java no requiere que importemos clases o paquetes, pero al hacerlo el código se hace mucho más conveniente.

Cualquier línea que importe clases aparece en la parte superior de los ejemplos de código, que es donde deben estar para hacer visibles las clases importadas a la clase que está siendo definida. La definición real de la clase sigue a cualquier línea que importe clases.

Utilizar el Método main()

Si una clase se va a ejecutar, debe contener un método **static public main**. Este método viene justo después de la línea que declara la clase y llama a los otros métodos de la clase. La palabra clave **static** indica que este método opera a nivel de clase en vez sobre ejemplares individuales de la clase. La palabra clave **public** significa que los miembros de cualquier clase pueden acceder a este método. Como no estamos definiendo clases sólo para ser ejecutadas por otras clases sino que queremos ejecutarlas, las aplicaciones de ejemplo de este capítulo incluyen un método **main**.

Utilizar bloques try y catch

Algo que también incluyen todas las aplicaciones de ejemplo son los bloques **try** y **catch**. Este es un mecanismo del lenguaje Java para manejar excepciones. Java requiere que cuando un método lanza un excepción exista un mecanismo que la maneje. Generalmente un bloque **catch** capturará la excepción y especificará lo que sucederá (que podría ser no hacer nada). En el código de ejemplo, utilizamos dos bloques **try** y dos bloques **catch**. El primer bloque **try** contiene el método **Class.forName,** del paquete **java.lang**. Este método lanza una **ClassNotFoundException**, por eso el bloque **catch** que le sigue maneja esa excepción. El segundo bloque **try** contiene métodos JDBC, todos ellos lanzan **SQLException**, por eso el bloque **catch** del final de la aplicación puede manejar el resto de las excepciones que podrían lanzarse ya que todas serían objetos **SQLException**.

Recuperar Excepciones

JDBC permite ver los avisos y excepciones generados por nuestro controlador de base de datos y por el compilador Java. Para ver las excepciones, podemos tener un bloque **catch** que las imprima. Por ejemplo, los dos bloques **catch** del siguiente código de ejemplo imprimen un mensaje explicando la excepción.

```
try {
        // Aquí va el código que podría generar la excepción.
        // Si se genera una excepción, el bloque catch imprimirá
        // información sobre ella.
} catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
}

try {
        Class.forName("myDriverClassName");
} catch(java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
}
```

Si ejecutarámos **CreateCOFFEES.java** dos veces, obtendríamos un mensaje de error similar a éste.

```
SQLException: There is already an object named 'COFFEES' in the
database.
Severity 16, State 1, Line 1
```

Este ejemplo ilustra la impresión del componente mensaje de un objeto **SQLException**, lo que es suficiente para la mayoría de las situaciones.

Sin embargo, realmente existen tres componentes, y para ser completos, podemos imprimirlos todos. El siguiente fragmento de código muestra un bloque **catch** que se ha completado de dos formas. Primero, imprime las tres partes de un objeto **SQLException**: el mensaje (un string que describe el error), el SQLState (un string que identifica el error de acuerdo a los convenciones X/Open de SQLState), y un código de error del vendedor (un número que es el código de error del vendedor del driver). El objeto **SQLException**, **ex** es capturado y se accede a sus tres componentes con los métodos **getMessage**, **getSQLState**, y **getErrorCode**.

La segunda forma del siguiente bloque **catch** completo obtiene todas las exepciones que podrían haber sido lanzada. Si hay una segunda excepción, sería encadenada a **ex**, por eso se llama a **ex.getNextException** para ver si hay más excepciones. Si las hay, el bucle **while** continúa e imprime el mensaje de la siguiente excecpción, el SQLState, y el código de error del vendedor. Esto continúa hasta que no haya más excepciones.

```
try {
        // Aquí va el código que podría generar la excepción.
        // Si se genera una excepción, el bloque catch imprimirá
        // información sobre ella.
} catch(SQLException ex) {
        System.out.println("\n--- SQLException caught ---\n");
        while (ex != null) {
                System.out.println("Message:   " + ex.getMessage ());
                System.out.println("SQLState:  " + ex.getSQLState ());
                System.out.println("ErrorCode: " + ex.getErrorCode ());
                ex = ex.getNextException();
                System.out.println("");
        }
}
```

Si hubieramos sustituido el bloque **catch** anterior en el Código de ejemplo 1 (**CreateCoffees**) y lo hubieramos ejecutado después de que la tabla **COFFEES** ya se hubiera creado, obtendríamos la siguiente información.

```
--- SQLException caught ---
Message:  There is already an object named 'COFFEES' in the database.
Severity 16, State 1, Line 1
SQLState: 42501
ErrorCode:   2714
```

SQLState es un código definido en X/Open y ANSI-92 que identifica la excepción. Aquí podemos ver dos ejemplos de códigos SQLState.

```
08001 -- No suitable driver
HY011 -- Operation invalid at this time
```

El código de error del vendedor es específico de cada driver, por lo que debemos revisar la documentación del driver buscando una lista con el significado de estos códigos de error.

Recuperar Avisos

Los objetos **SQLWarning** son una subclase de **SQLException** que trata los avisos de accesos a bases de datos. Los Avisos no detienen la ejecución de una aplicación, como las excepciones; simplemente alertan al usuario de que algo no ha salido como se esperaba. Por ejemplo, un aviso podría hacernos saber que un privilegio que queriamos revocar no ha fue revocado. O un aviso podría decirnos que ha ocurrido algún error durante una petición de desconexión.

Un aviso puede reportarse sobre un objeto **Connection**, un objeto **Statement** (incluyendo objetios **PreparedStatement** y **CallableStatement**), o un objeto **ResultSet**. Cada una de esas clases tiene un método **getWarnings**, al que debemos llamar para ver el primer aviso reportado en la llamada al objeto. Si **getWarnings** devuelve un aviso, podemos llamar al método **getNextWarning** de **SQLWarning** para obtener avisos adicionales. Al ejecutar una sentencia se borran automáticamente los avisos de la sentencia anterior, por eso no se apilan. Sin embargo, esto significa que si queremos recuperar los avisos reportados por una sentencia, debemos hacerlo antes de ejecutar otra sentencia.

El siguiente fragmento de código ilustra como obtener información completa sobre los avisos reportados por el objeto **Statement**, **stmt** y también por el objeto **ResultSet**, **rs**.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select COF_NAME from COFFEES");
while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        System.out.println("Coffees available at the Coffee Break:  ");
        System.out.println("    " + coffeeName);
        SQLWarning warning = stmt.getWarnings();
        if (warning != null) {
                System.out.println("\n---Warning---\n");
                while (warning != null) {
                        System.out.println("Message: " +
warning.getMessage());
                        System.out.println("SQLState: " +
warning.getSQLState());
                        System.out.print("Vendor error code: ");
                        System.out.println(warning.getErrorCode());
                        System.out.println("");
                        warning = warning.getNextWarning();
                }
        }
        SQLWarning warn = rs.getWarnings();
        if (warn != null) {
                System.out.println("\n---Warning---\n");
                while (warn != null) {
                        System.out.println("Message: " +
warn.getMessage());
                        System.out.println("SQLState: " +
warn.getSQLState());
                        System.out.print("Vendor error code: ");
                        System.out.println(warn.getErrorCode());
                        System.out.println("");
                        warn = warn.getNextWarning();
                }
        }
}
```

Los avisos no son muy comunes, De aquellos que son reportados, el aviso más común es un **DataTruncation**, una subclase de **SQLWarning**. Todos los objetos **DataTruncation** tienen un SQLState **01004**, indicando que ha habido un problema al leer o escribir datos. Los métodos de **DataTruncation** permiten encontrar en que columna o parámetro se truncaron los datos, si la ruptura se produjo en una operación de lectura o de escritura, cuántos bytes deberían haber sido transmitidos, y cuántos bytes se transmitieron realmente.

Ejecutar la aplicación de Ejemplo

Ahora estamos listos para probar algún código de ejemplo. El directorio **book.html**, contiene una aplicación completa, ejecutable, que ilustra los conceptos presentados en este capítulo y el siguiente. Puedes descargar este codigo de ejemplo del site de JDBC situado en http://www.javasoft.com/products/jdbc/book.html

Antes de poder ejecutar una de esas aplicaciones, necesitamos editar el fichero sustituyendo la información apropiada para las siguientes variables.

url
: La URL JDBC, las partes uno y dos son suministradas por el driver, y la tercera parte especifica la fuente de datos.

myLogin
: Tu nombre de usuario o login.

myPassword
: Tu password para el controlador de base de datos.

myDriver.ClassName
: El nombre de clase suministrado con tu driver

La primera aplicación de ejemplo es la clase **CreateCoffees**, que está en el fichero llamado **CreateCoffees.java**. Abajo tienes las instrucciones para ejecutar **CreateCoffees.java** en las dos plataformas principales.

La primera línea compila el código del fichero **CreateCoffees.java**. Si la compilación tiene éxito, se producirá un fichero llamado **CreateCoffees.class**, que contendrá los bytecodes traducidos desde el fichero **CreateCoffees.java**. Estos bytecodes serán interpretados por la máquina virtual Java, que es la que hace posible que el código Java se pueda ejecutar en cualquier máquina que la tenga instalada.

La segunda línea de código ejecuta el código. Observa que se utiliza el nombre de la clase, **CreateCoffees**, no el nombre del fichero **CreateCoffees.class**.

UNIX

```
javac CreateCoffees.java
java CreateCoffees
```

Windows 95/NT

```
javac CreateCoffees.java
java CreateCoffees
```

Crear un Applet desde una Aplicación

Supongamos que el propietario de "The Coffee Break" quiere mostrar los precios actuales de los cafés en un applet en su página Web. Puede asegurarse de que está mostrando los precios actuales haciendo que applet obtenga los precios directamente desde su base de datos.

Para hacer esto necesita crear dos ficheros de código, uno con el código del applet y otro con el código HTML. El código del applet contiene el código JDBC que podría aparecer en una aplicación normal más el código adicional para ejecutar el applet y mostrar el resultado de la petición a la base d edatos. En nuestro ejemplo el código del applet está en el fichero **OutputApplet.java**. Para mostrar el applet en una página HTML, el fichero **OutputApplet.html** le dice al navegador qué mostrar y dónde mostrarlo.

El resto de esta página explicará algunos elementos que se encuentran en el código del applet y que no están presentes en el código de las aplicaciones. Algunos de estos ejemplos involucran aspectos avanzados del lenguaje Java. Daremos alguna explicación básica y racional, ya que una explicación completa va más allá de este turorial. El propósito de este applet es dar una idea general, para poder utilizarlo como plantilla, sustituyendo nuestras consultas por las del applet.

Escribir el Código del Applet

Para empezar, los applets importan clases que no son utilizadas por las aplicaciones. Nuestro applet importa dos clases que son especiales para los applets: la clase **Applet**, que forma parte del paquete **java.applet**, y la clase **Graphics**, que forma parte del paquete **java.awt**. Este applet también importa la clase de propósito general **java.util.Vector** que se utiliza para acceder a un contenedor tipo array cuyo tamaño puede ser modificado. Este código utiliza objetos **Vector** para almacenar los resultados de las peticiones para poder mostrarlas después.

Todos los applets descienden de la clase **Applet**; es decir, son subclases de **Applet**. Por lo tanto, toda definición de applet debe contener las palabras **extends Applet**; como se vé aquí.

```
public class MyAppletName extends Applet {
    . . .
}
```

En nuestro ejemplo, esta línea también incluye las palabras **implements Runnable**, por lo que se parece a esto.

```
public class OutputApplet extends Applet implements Runnable {
    . . .
}
```

Runnable es un interface que hace posible ejecutar más de un thread a la vez. Un thread es un flujo secuencial de control, y un programa puede tener muchos threads haciendo cosas diferentes concurrentemente. La clase **OutputApplet** implementa el interface **Runnable** definiendo el método **run**; el único método de **Runnable**. En nuestro ejemplo el método **run** contiene el código JDBC para abrir la conexión, ejecutar una petición, y obtener los resultados desde la hoja de resultados. Como las conexiones a las bases de datos pueden ser lentas, y algunas veces pueden tardar varios segundos, es una buena idea estructurar un applet para que pueda manejar el trabajo con bases de datos en un thread separado.

Al igual que las aplicaciones deben tener un método **main**, un applet debe implementar al menos uno de estos métodos **init**, **start**, o **paint**. Nuestro ejemplo define un método **start** y un método **paint**. Cada vez que se llama a **start**, crea un nuevo thread (**worker**) para re-evaluar la petición a la base de datos. Cada vez que se llama a **paint**, se muestra o bien el resultado de la petición o un string que describe el estado actual del applet.

Como se mencionó anteriormente, el método **run** definido en **OutputApplet** contiene el código JDBC, Cuando el thread **worker** llama al método **start**, se llama automáticamente al método **run**, y éste ejecuta el código JDBC en el thread **worker**. El código de **run** es muy similar al código que hemos visto en otros ejemplos con tres excepciones. Primero, utiliza la clase **Vector** para almacenar los resultados de la petición. Segundo, no imprime los resultados, sino que los añade al **Vector**, **results** para mostrarlos más tarde. Tercero, tampoco muestra ninguna excepción, en su lugar almacena los mensajes de error para mostrarlos más tarde.

Los applets tienen varias formas de dibujar, o mostrar, su contenido. Este applet, es uno muy simple que sólo utiliza texto, utiliza el método **drawString** (una parte de la clase **Graphics**) para mostrar texto. El método **drawString** tiene tres argumentos: (1) el string a mostrar, (2) la

coordenada **x**, indicando la posición horizontal de inicio del string, y (3) la coordenada **y**, indicando la posición vertical de inicio del string (que está en la parte inferior del texto).

El método **paint** es el que realmente dibuja las cosas en la pantalla, y en **OutputApplet.java**, está definido para contener llamadas al método **drawString**. La cosa principal que muestra **drawString** es el contenido del **Vector**, **results** (los resultados almacenados). Cuando no hay resultados que mostrar, **drawString** muestra el estado actual contenido en el **String**, **message**. Este string será "Initializing" para empezar. Será "Connecting to database" cuando se llame al método **start**, y el método **setError** pondrá en él un mensaje de error cuando capture una excepción. Así, si la conexión a la base de datos tarda mucho tiempo, la persona que está viendo el applet verá el mensaje "Connecting to database" porque ese será el contenido de **message** en ese momento. (El método **paint** es llamado por el AWT cuando quiere que el applet muestre su estado actual en la pantalla).

Al menos dos métodos definidos en la clase **OutputApplet**, **setError** y **setResults** son privados, lo que signfica que sólo pueden ser utilizados por **OutputApplet**. Estos métodos llaman el método **repaint**, que borra la pantalla y llama a **paint**. Por eso si **setResults** llama a **repaint**, se mostrarán los resultados de la petición, y si **setError** llama a **repaint**, se mostrará un mensaje de error.

Otro punto es hacer que todos los métodos definidos en **OutputApplet** excepto **run** son **synchronized**. La palabra clave **synchronized** indica que mientras que un método esté accediendo a un objeto, otros métodos **synchronized** están bloqueados para acceder a ese objeto. El método **run** no se declara **synchronized** para que el applet pueda dibujarse en la pantalla mientras se produce la conexión a la base de datos. Si los métodos de acceso a la base de datos fueran **synchronized**, evitarían que el applet se redibujara mientras se están ejecutando, lo que podría resultar en retrasos sin acompañamiento de mensaje de estado.

Para sumarizar, en un applet, es una buena práctica de programación es hacer algunas cosas que no necesitaríamos hacer en una aplicación.

Poner nuestro código JDBC en un thread separado.

Mostrar mensajes de estado durante los retardos, como cuando la conexión a la base de datos tarda mucho tiempo.

Mostrar los mensajes de error en la pantalla en lugar de imprimirlos en **System.out** o **System.err**.

Ejecutar un Applet

Antes de ejecutar nuestro applet, necesitamos compilar el fichero **OutputApplet.java**. Esto crea el fichero **OutputApplet.class**, que es referenciado por el fichero **OutputApplet.html**.

La forma más fácil de ejecutar un applet es utilizar el appletviewer, que se incluye en el JDK. Sólo debemos seguir las instrucciones para nuestra plataforma.

UNIX

```
javac OutputApplet.java
appletviewer OutputApplet.html
```

Windows 95/NT

```
javac OutputApplet.java
appletviewer OutputApplet.html
```

Los applets descargados a través de la red están sujetos a distintas restricciones de seguridad. Aunque esto puede parecer molesto, es absolutamente necesario para la seguridad de la red, y la seguridad es una de las mayores ventajas de utilizar Java. Un applet no puede hacer

conexiones en la red excepto con el host del que se descargó a menos que el navegador se lo permita. Si uno puede tratar con applets instalados localmente como applets "trusted" (firmados) también dependen de restricciones de seguridad impuestas por el navegador. Un applet normalmente no puede leer o escribir ficheros en el host en el que se está ejecuando, y no puede cargar librerías ni definir métodos nativos.

Los applets pueden hacer conexiones con el host del que vinieron, por eso pueden trabajar muy bien en intranets.

El driver puente JDBC-ODBC es de alguna forma un caso especial. Puede utilizarse satisfactoriamente para acceder a intranet, pero requiere que ODBC, el puente, la librería nativa, y el JDBC esten instalados en cada cliente. Con esta configuración, los accesos a intranet funcionan con aplicaciones Java y con applets firmados. Sin embargo, como los puentes requieren configuraciones especiales del cliente, no es práctico para ejecutar applets en Internet con el puente JDBC-ODBC. Observa que esta limitaciones son para el puente JDBC-ODBC, no para JDBC. Con un driver JDBC puro Java, no se necesita ninguna configuración especial para ejecutar applets en Internet.

El API de JDBC 2..0

El paquete java.sql que está incluido en la versión JDK 1.2 (conocido como el API JDBC 2.0) incluye muchas nuevas características no incluidas en el paquete java.sql que forma parte de la versión JDK 1.1 (referenciado como el API JDBC 1.0).

Con el API JDBC 2.0, podremos hacer las siguientes cosas:

Ir hacia adelante o hacia atrás en una hoja de resultados o movernos a un fila específica.

Hacer actualizaciones de las tablas de la base datos utilizando métodos Java en lugar de utilizar comandos SQL.

Enviar múltiples secuencias SQL a la base de datos como una unidad, o batch.

Uitlizar los nuevos tipos de datos SQL3 como valores de columnas.

Inicialización para Utilizar JDBC 2.0

Si queremos ejecutar código que emplee alguna de las características del JDBC 2.0, necesitamos hacer lo siguiente.

Descargar el JDK 1.2 siguiendo las instrucciones de descarga

Instalar un Driver JDBC que implemente las características del JDBC 2.0 utilizadas en el código.

Acceder a un controlador de base de datos que implemente las características del JDBC utilizadas en el código.

En el momento de escribir esto, no se había implementado ningún driver que soportara las nuevas características, pero hay muchos en desarrollo. Como consecuencia, no es posible probar el código de desmostración de las características del JDBC 2.0. Podemos aprender de los ejemplos, pero no se asegura que éstos funcionen.

Mover el Cursor por una Hoja de Resultados

Una de las nuevas características del API JDBC 2.0 es la habilidad de mover el cursor en una hoja de resultados tanto hacia atrás como hacia adelante. También hay métodos que nos permiten mover el cursor a una fila particular y comprobar la posición del cursor. La hoja de resultados Scrollable hace posible crear una herramienta GUI (Interface Gráfico de Usuario) para navegar a través de ella, lo que probablemente será uno de los principales usos de esta característica. Otro uso será movernos a una fila para actualizarla.

Antes de poder aprovechar estas ventajas, necesitamos crear un objeto **ResultSet** Scrollable.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
```

Este código es similar al utilizado anteriormente, excepto en que añade dos argumentos al método **createStatement**. El primer argumento es una de las tres constantes añadidas al API **ResultSet** para indicar el tipo de un objeto **ResultSet**: **TYPE_FORWARD_ONLY**, **TYPE_SCROLL_INSENSITIVE**, y **TYPE_SCROLL_SENSITIVE**. El segundo argumento es una de las dos constantes de **ResultSet** para especificar si la hoja de resultados es de sólo lectura o actualizable:**CONCUR_READ_ONLY** y **CONCUR_UPDATABLE**. Lo que debemos recordar aquí es que si especificamos un tipo, también debemos especificar si es de sólo lectura o actualizable. También, debemos especificar primero el tipo, y como ambos parámetros son **int**, el compilador no comprobará si los hemos intercambiado.

Especificando la constante **TYPE_FORWARD_ONLY** se crea una hoja de resultados no desplazable, es decir, una hoja en la que el cursor sólo se mueve hacia adelante. Si no se especifican constantes para el tipo y actualización de un objeto **ResultSet**, obtendremos automáticamente una **TYPE_FORWARD_ONLY** y **CONCUR_READ_ONLY** (exactamente igual que en el API del JDBC 1.0).

Obtendremos un objeto **ResultSet** desplazable si utilizamos una de estas constantes:**TYPE_SCROLL_INSENSITIVE** o **TYPE_SCROLL_SENSITIVE**. La diferencia entre estas dos es si la hoja de resultados refleja los cambios que se han hecho mientras estaba abierta y si se puede llamar a ciertos métodos para detectar estos cambios. Generalmente hablando, una hoja de resultados **TYPE_SCROLL_INSENSITIVE** no refleja los cambios hechos mientras estaba abierta y en una hoja **TYPE_SCROLL_SENSITIVE** si se reflejan. Los tres tipos de hojas de resultados harán visibles los resultados si se cierran y se vuelve a abrir. En este momento, no necesitamos preocuparnos de los puntos delicados de las capacidades de un objeto **ResultSet**, entraremos en más detalle más adelante. Aunque deberíamos tener en mente el hecho de que no importa el tipo de hoja de resultados que especifiquemos, siempre estaremos limitados por nuestro controlador de base de datos y el driver utilizados.

Una vez que tengamos un objeto **ResultSet** desplazable, **srs** en el ejemplo anterior, podemos utilizarlo para mover el cursor sobre la hoja de resultados. Recuerda que cuando creabamos un objeto **ResultSet** anteriormente, tenía el cursor posicionado antes de la primera fila. Incluso aunque una hoja de resultados se seleccione desplazable, el cursor también se posiciona inicialmente delante de la primera fila. En el API JDBC 1.0, la única forma de mover el cursor era llamar al método **next**. Este método todavía es apropiado si queremos acceder a las filas una a una, yendo de la primera fila a la última, pero ahora tenemos muchas más formas para mover el cursor.

La contrapartida del método **next**, que mueve el cursor una fila hacia delante (hacia el final de la hoja de resultados), es el nuevo método **previous**, que mueve el cursor una fila hacia atrás (hacia el inicio de la hoja de resultados). Ambos métodos devuelven **false** cuando el cursor se

sale de la hoja de resultados (posición antes de la primera o después de la última fila), lo que hace posible utilizarlos en un bucle **while**. Ye hemos utilizado un método **next** en un bucle while, pero para refrescar la memoria, aquí tenemos un ejemplo que mueve el cursor a la primera fila y luego a la siguiente cada vez que pasa por el bucle while. El bucle termina cuando alcanza la última fila, haciendo que el método **next** devuelva **false**. El siguiente fragmento de código imprime los valores de cada fila de **srs**, con cinco espacios en blanco entre el nombre y el precio.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
while (srs.next()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
}
```

La salida se podría parecer a esto.

```
Colombian     7.99
French_Roast     8.99
Espresso     9.99
Colombian_Decaf     8.99
French_Roast_Decaf     9.99
```

Al igual que en el fragmento anterior, podemos procesar todas las filas de **srs** hacia atrás, pero para hacer esto, el cursor debe estar detrás de la última fila. Se puede mover el cursor explícitamente a esa posicón con el método **afterLast**. Luego el método **previous** mueve el cursor desde la posicón detrás de la última fila a la última fila, y luego a la fila anterior en cada iteracción del bucle **while**. El bucle termina cuando el cursor alcanza la posición anterior a la primera fila, cuando el método **previous** devuelve **false**.

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
srs.afterLast();
while (srs.previous()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
}
```

La salida se podría parecer a esto.

```
French_Roast_Decaf     9.99
Colombian_Decaf     8.99
Espresso     9.99
French_Roast     8.99
Colombian     7.99
```

Como se puede ver, las dos salidas tienen los mismos valores, pero las filas están en orden inverso.

Se puede mover el cursor a una fila particular en un objeto **ResultSet**. Los métodos **first**, **last**, **beforeFirst**, y **afterLast** mueven el cursor a la fila indicada en sus nombres. El método **absolute** moverá el cursor al número de fila indicado en su argumento. Si el número es positivo, el cursor se mueve al número dado desde el principio, por eso llamar a **absolute(1)** pone el cursor en la primera fila. Si el número es negativo, mueve el cursor al número dado desde el final, por eso llamar a **absolute(-1)** pone el cursor en la última fila. La siguiente línea de código mueve el cursor a la cuarta fila de **srs**.

```
srs.absolute(4);
```

Si **srs** tuviera 500 filas, la siguiente línea de código movería el cursor a la fila 497.

```
srs.absolute(-4);
```

Tres métodos mueven el cursor a una posición relativa a su posición actual. Como hemos podido ver, el método **next** mueve el cursor a la fila siguiente, y el método **previous** lo mueve a la fila anterior. Con el método **relative**, se puede especificar cuántas filas se moverá desde la fila actual y también la dirección en la que se moverá. Un número positivo mueve el cursor hacia adelante el número de filas dado; un número negativo mueve el cursor hacia atrás el número de filas dado. Por ejemplo, en el siguiente fragmente de código, el cursor se mueve a la cuarta fila, luego a la primera y por último a la tercera.

```
srs.absolute(4); // cursor está en la cuarta fila
. . .
srs.relative(-3); // cursor está en la primera fila
. . .
srs.relative(2); // cursor está en la tercera fila
```

El método **getRow** permite comprobar el número de fila donde está el cursor. Por ejemplo, se puede utilizar **getRow** para verificar la posición actual del cursor en el ejemplo anterior.

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum debería ser 4
srs.relative(-3);
int rowNum = srs.getRow(); // rowNum debería ser 1
srs.relative(2);
int rowNum = srs.getRow(); // rowNum debería ser 3
```

Existen cuatro métodos adicionales que permiten verificar si el cursor se encuentra en una posición particular. La posición se indica en sus nombres:**isFirst**, **isLast**, **isBeforeFirst**, **isAfterLast**. Todos estos métodos devuelven un **boolean** y por lo tanto pueden ser utilizados en una sentencia condicional. Por ejemplo, el siguiente fragmento de código comprueba si el cursor está después de la última fila antes de llamar al método **previous** en un bucle **while**. Si el método **isAfterLast** devuelve **false**, el cursor no estará después de la última fila, por eso se llama al método **afterLast**. Esto garantiza que el cursor estára después de la última fila antes de utilizar el método **previous** en el bucle **while** para cubrir todas las filas de **srs**.

```
if (srs.isAfterLast() == false) {
        srs.afterLast();
}
while (srs.previous()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
}
```

En la siguiente página, veremos cómo utilizar otros dos métodos de **ResultSet** para mover el cursor, **moveToInsertRow** y **moveToCurrentRow**. También veremos ejemplos que ilustran por qué podríamos querer mover el cursor a ciertas posiciones.

Hacer Actualizaciones en una Hoja de Resultados

Otra nueva característica del API JDBC 2.0 es la habilidad de actualizar filas en una hoja de resultados utilizando métodos Java en vez de tener que enviar comandos SQL. Pero antes de poder aprovechar esta capacidad, necesitamos crear un objeto **ResultSet** actualizable. Para hacer esto, suministramos la constante **CONCUR_UPDATABLE** de **ResulSet** al método **createStatement**, como se ha visto en ejemplos anteriores. El objeto **Statement** creado producirá un objeto **ResultSet** actualizable cada vez que se ejecute una petición. El siguiente fragmento de código ilustra la creacción de un objeto **ResultSet** actualizable, **uprs**. Observa que el código también lo hace desplazable. Un objeto **ResultSet** actualizable no tiene porque ser desplazable, pero cuando se hacen cambios en una hoja de resultados, generalmente queremos poder movernos por ella. Con una hoja de resultados desplazable, podemos movernos a las filas que queremos cambiar, y si el tipo es **TYPE_SCROLL_SENSITIVE**, podemos obtener el nuevo valor de una fila después de haberlo cambiado.

```
Connection con =
DriverManager.getConnection("jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
```

El objeto **ResultSet**, **uprs** resultante se podría parecer a esto.

```
COF_NAME              PRICE
------------------     -----
Colombian             7.99
French_Roast          8.99
Espresso              9.99
Colombian_Decaf       8.99
French_Roast_Decaf    9.99
```

Podemos utilizar los nuevos métodos del JDBC 2.0 en el interface **ResultSet** para insertar una nueva fila en **uprs**, borrar una fila de **uprs**, o modificar un valor de una columna de **uprs**.

Actualizar una Hoja de Resultados Programáticamente

Una actualización es la modificación del valor de una columna de la fila actual. Supongamos que queremos aumentar el precio del café "French Roast Decaf" a 10.99. utilizando el API JDBC 1.0, la actualización podría ser algo como esto.

```
stmt.executeUpdate("UPDATE COFFEES SET PRICE = 10.99" +
                   "WHERE COF_NAME = FRENCH_ROAST_DECAF");
```

El siguiente fragmento de código muesta otra forma de realizar la actualización, esta vez utilizando el API JDBC 2.0.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
```

Las operaciones de actualización en el API JDBC 2.0 afectan a los valores de columna de la fila en la que se encuentra el cursor, por eso en le primera línea se llama al método **last** para mover el cursor a la última fila (la fila donde la columna **COF_NAME** tiene el valor **FRENCH_ROAST_DECAF**). Una vez situado el cursor, todos los métodos de actualización que llamemos operarán sobre esa fila hasta que movamos el cursor a otra fila. La segunda línea de código cambia el valor de la columna **PRICE** a 10.99 llamando al método **updateFloat**. Se utiliza este método porque el valor de la columna que queremos actualizar es un **float** Java.

Los métodos **updateXXX** de **ResultSet** toman dos parámetros: la columna a actualizar y el nuevo valor a colocar en ella. Al igual que en los métodos **getXXX** de **ResultSet.**, el parámetro que designa la columna podría ser el nombre de la columna o el número de la columna. Existe un método **updateXXX** diferente para cada tipo (**updateString**, **updateBigDecimal**, **updateInt**, etc.)

En este punto, el precio en **uprs** para "French Roast Decaf" será 10.99, pero el precio en la tabla **COFFEES** de la base de datos será todavía 9.99. Para que la actualización tenga efecto en la base de datos y no sólo en la hoja de resultados, debemos llamar al método **updateRow** de **ResultSet**. Aquí está el código para actualizar tanto **uprs** como **COFFEES**.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.updateRow();
```

Si hubiéramos movido el cursor a una fila diferente antes de llamar al método **updateRow**, la actualización se habría perdido. Si, por el contrario, nos damos cuenta de que el precio debería haber sido 10.79 en vez de 10.99 podríamos haber cancelado la actualización llamando al método **cancelRowUpdates**. Tenemos que llamar al método **cancelRowUpdates** antes de llamar al método **updateRow**; una vez que se llama a **updateRow**, llamar a **cancelRowUpdates** no hará nada. Observa que **cancelRowUpdates** cancela todas las actualizaciones en una fila, por eso, si había muchas llamadas a método **updateXXX** en la misma fila, no podemos cancelar sólo una de ellas. El siguiente fragmento de código primero cancela el precio 10.99 y luego lo actualiza a 10.79.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
uprs.updateRow();
```

En este ejemplo, sólo se había actualizado una columna, pero podemos llamar a un método **updateXXX** apropiado para cada una de las columnas de la fila. El concepto a recordar es que las actualizaciones y las operaciones relacionadas se aplican sobre la fila en la que se encuentra el cursor. Incluso si hay muchas llamadas a métodos **updateXXX**, sólo se hace una llamada al método **updateRow** para actualizar la base de datos con todos los cambios realizados en la fila actual.

Si también queremos actualizar el precio de **COLOMBIAN_DECAF**, tenemos que mover el cursor a la fila que contiene ese café. Cómo la fila de **COLOMBIAN_DECAF** precede inmediatamente a la fila de **FRENCH_ROAST_DECAF**, podemos llamar al método **previous** para posicionar el cursor en la fila de **COLOMBIAN_DECAF**. El siguiente fragmento de código cambia el precio de esa fila a 9.79 tanto en la hoja de resultados como en la base de datos.

```
uprs.previous();
uprs.updateFloat("PRICE", 9.79);
uprs.updateRow();
```

Todos los movimientos de cursor se refieren a filas del objeto **ResultSet**, no a filas de la tabla de la base de datos. Si una petición selecciona cinco filas de la tabla de la base de datos, habrá cinco filas en la hoja de resultados, con la primera fila siendo la fila 1, la sengunda siendo la fila 2, etc. La fila 1 puede ser identificada como la primera, y, en una hoja de resultados con cinco filas, la fila 5 será la última.

El orden de las filas en la hoja de resultados no tiene nada que ver con el orden de las filas en la tablas de la base de datos. De hecho, el orden de las filas en la tabla de la base de datos es indeterminado. El controlador de la base de datos sigue la pista de las filas seleccionadas, y hace las actualizaciones en la fila apropiada, pero podrían estar localizadas en cualquier lugar de la tabla. Cuando se inserta una fila, por ejemplo, no hay forma de saber donde será insertada dentro de la tabla.

Insertar y Borrar filas Programáticamente

En la sección anterior hemos visto cómo modificar el valor de una columna utilizando métodos del API JDBC 2.0 en vez de utilizar comandos SQL. Con el API JDBC 2.0 también podemos insertar una fila en una tabla o borrar una fila existente programáticamente.

Supongamos que nuestro propietario del café ha obtenido una nueva variedad de café de uno de sus suministradores. El "High Ground", y quiere añadirlo a su base de datos. Utilizando el API JDBC 1,0 podría escribir el código que pasa una sentencia insert de SQL al controlador de la báse de datos. El siguiente fragmento de código, en el que **stmt** es un objeto **Statement**, muestra esta aproximación.

```
stmt.executeUpdate("INSERT INTO COFFEES " +
                   "VALUES ('Kona', 150, 10.99, 0, 0)");
```

Se puede hacer esto mismo sin comandos SQL utilizando los métodos de **ResultSet** del API JDBC 2.0. Básicamente, después de tener un objeto **ResultSet** con los resultados de la tabla **COFFEES**, podemos constuir una nueva fila insertándola tanto en la hoja de resultados como en la tabla **COFFEES** en un sólo paso. Se construye una nueva fila en una llamada "fila de inserción", una fila especial asociada con cada objeto **ResultSet**. Esta fila realmente no forma parte de la hoja de resultados; podemos pensar en ella como un buffer separado en el que componer una nueva fila.

El primer paso será mover el cursor a la fila de inserción, lo que podemos hacer llamando al método **moveToInsertRow**. El siguiente paso será seleccionar un valor para cada columna de la fila. Hacemos esto llamando a los métodos **updateXXX** apropiados para cada valor. Observa que estos son los mismos métodos **updateXXX** utilizados en la página anterior para cambiar el valor de una columna. Finalmente, podemos llamar al método **insertRow** para insertar la fila que hemos rellenado en la hoja de resultados. Este único método inserta la fila simultáneamente tanto en el objeto **ResultSet** como en la tabla de la base de datos de la que la hoja de datos fue seleccionada.

El siguiente fragmento de código crea un objeto **ResultSet** actualizable y desplazable, **uprs**, que contiene todas las filas y columnas de la tabla **COFFEES**.

```
Connection con =
DriverManager.getConnection("jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
```

El siguiente fragmento de código utiliza el objeto **ResultSet**, **uprs** para insertar la fila para "kona", mostrada en el ejemplo SQL. Mueve el cursor a la fila de inserción, selecciona los cinco valores de columna e inserta la fila dentro de **uprs** y **COFFEES**.

```
uprs.moveToInsertRow();
uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);
uprs.insertRow();
```

Como podemos utilizar el nombre o el número de la columna para indicar la columna seleccionada, nuestro código para seleccionar los valores de columna se podría parecer a esto.

```
uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);
```

Podríamos habernos preguntado por qué los métodos **updateXXX** parecen tener un comportamiento distinto a como lo hacían en los ejemplos de actualización. En aquellos ejemplos, el valor seleccionado con un método **updateXXX** reemplazaba inmediatamente el valor de la columna en la hoja de resultados. Esto era porque el cursor estaba sobre una fila de la hoja de resultados. Cuando el cursor está sobre la fila de inserción, el valor seleccionado con un método **updateXXX** también es automáticamente seleccionado, pero lo es en la fila de inserción y no en la propia hoja de resultados. Tanto en actualizaciones como en inserciones, llamar a los métodos **updateXXX** no afectan a la tabla de la base de datos. Se debe llamar al método **updateRow** para hacer que las actualizaciones ocurran en la base de datos. Para el inserciones, el método **insertRow** inserta la nueva fila en la hoja de resultados y en la base de datos al mismo tiempo.

Podríamos preguntarnos que sucedería si insertáramos una fila pero sin suministrar los valores para cada columna. Si no suministramos valores para una columna que estaba definida para aceptar valores **NULL** de SQL, el valor asignado a esa columna es **NULL**. Si la columna no acepta valores null, obtendremos una **SQLException**. Esto también es cierto si falta una columna de la tabla en nuestro objeto **ResultSet**. En el ejemplo anterior, la petición era **SELECT * FROM COFFEES**, lo que producía una hoja de resultados con todas las columnas y todas las filas. Cuando queremos insertar una o más filas, nuestra petición no tiene porque seleccionar todas las filas, pero sí todas las columnas. Especialmente si nuestra tabla tiene cientos o miles de filas, querremos utilizar una claúsula **WHERE** para límitar el número de filas devueltas por la sentencia **SELECT**.

Después de haber llamado al método **insertRow**, podemos construir otra fila, o podemos mover el cursor de nuevo a la hoja de resultados. Por ejemplo, podemos, llamar a cualquier método que ponga el cursor en una fila específica, como **first**, **last**, **beforeFirst**, **afterLast**, y **absolute**. También podemos utilizar los métodos **previous**, **relative**, y **moveToCurrentRow**. Observa que sólo podemos llamar a **moveToCurrentRow** cuando el cursor está en la fila de inserción.

Cuando llamamos al método **moveToInsertRow**, la hoja de resultados graba la fila en la que se encontraba el cursor, que por definición es la fila actual. Como consecuencia, el método **moveToCurrentRow** puede mover el cursor desde la fila de inserción a la fila en la que se encontraba anteriormente. Esto también explica porque podemos utilizar los métodos **previous** y **relative**, que requieren movimientos relativos a la fila actual.

Insertar una Fila

El siguiente ejemplo de código es un programa completo que debería funcionar si tenemos un driver JDBC 2.0 que implemente una hoja de resultados desplazable.

Hay algunas cosas que podríamos observar sobre el código.

El objeto **ResultSet**, **uprs** es actualizable, desplazable y sensible a los cambios hechos por ella y por otros. Aunque es **TYPE_SCROLL_SENSITIVE**, es posible que los métodos **getXXX** llamados después de las inserciones no recuperen los valores de la nuevas filas. Hay métodos en el interface **DatabaseMetaData** que nos dirán qué es visible y qué será detectado en los diferentes tipos de hojas de resultados para nuestro driver y nuestro controlador de base de datos.

Después de haber introducido los valores de una fila con los métodos **updateXXX**, el código inserta la fila en la hoja de resultados y en la base de datos con el método **insertRow**. Luego, estándo todavía en la "fila de inserción", selecciona valores para otra nueva fila.

```
import java.sql.*;

public class InsertRows {
        public static void main(String args[]) {
                String url = "jdbc:mySubprotocol:myDataSource";
                Connection con;
                Statement stmt;
        try {
                Class.forName("myDriver.ClassName");
        } catch(java.lang.ClassNotFoundException e) {
                System.err.print("ClassNotFoundException: ");
                System.err.println(e.getMessage());
        }
        try {
                con = DriverManager.getConnection(url, "myLogin",
"myPassword");
                stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                                ResultSet.CONCUR_UPDATABLE);
                ResultSet uprs = stmt.executeQuery("SELECT * FROM
COFFEES");
                uprs.moveToInsertRow();
                uprs.updateString("COF_NAME", "Kona");
                uprs.updateInt("SUP_ID", 150);
                uprs.updateFloat("PRICE", 10.99f);
                uprs.updateInt("SALES", 0);
                uprs.updateInt("TOTAL", 0);
                uprs.insertRow();
                uprs.updateString("COF_NAME", "Kona_Decaf");
                uprs.updateInt("SUP_ID", 150);
                uprs.updateFloat("PRICE", 11.99f);
                uprs.updateInt("SALES", 0);
                uprs.updateInt("TOTAL", 0);
                uprs.insertRow();
                uprs.beforeFirst();
                System.out.println("Table COFFEES after insertion:");
                while (uprs.next()) {
                        String name = uprs.getString("COF_NAME");
                        int id = uprs.getInt("SUP_ID");
                        float price = uprs.getFloat("PRICE");
                        int sales = uprs.getInt("SALES");
```

```
                        int total = uprs.getInt("TOTAL");
                        System.out.print(name + "   " + id + "   " +
price);
                        System.out.println("   " + sales + "   " +
total);
                }

                uprs.close();
                stmt.close();
                con.close();

        } catch(SQLException ex) {
                System.err.println("SQLException: " + ex.getMessage());
        }
}
```

Borrar una Fila

Hasta ahora, hemos visto cómo actualizar un valor y cómo insertar una nueva fila. Borrar una fila es la tercera forma de modificar un objeto **ResultSet**, y es la más simple. Todo lo que tenemos que hacer es mover el cursor a la fila que queremos borrar y luego llamar al método **deleteRow**. Por ejemplo, si queremos borrar la cuarta fila de la hoja de resultados **uprs**, nuestro código se parecería a esto.

```
uprs.absolute(4);
uprs.deleteRow();
```

La cuarta fila ha sido eliminada de **uprs** y de la base de datos.

El único problema con las eliminaciones es lo que **ResultSet** realmente hace cuando se borra una fila. Con algunos driver JDBC, una línea borrada es eliminada y ya no es visible en una hoja de resultados. Algunos drives JDBC utilizan una fila en blanco en su lugar pone (un "hole") donde la fila borrada fuera utilizada. Si existe una fila en blanco en lugar de la fila borrada, se puede utilizar el método **absolute** con la posición original de la fila para mover el cursor, porque el número de filas en la hoja de resultados no ha cambiado.

En cualquier caso, deberíamos recordar que los drivers JDBC manejan las eliminaciones de forma diferente. Por ejemplo, si escribimos una aplicación para ejecutarse con diferentes bases de datos, no deberíamos escribir código que dependiera de si hay una fila vacía en la hoja de resultados.

Usar Tipos de Datos de SQL3

Los tipos de datos comunmente referidos como tipos SQL3 son los nuevos tipos de datos que están siendo adoptados en la nueva versión del estándard ANSI/ISO de SQL. El JDBC 2.0 proporciona interfaces que representan un mapeado de estos tipos de datos SQL3 dentro del lenguaje Java. Con estos nuevos interfaces, podremos trabajar con tipos SQL3 igual que con otros tipos.

Los nuevos tipos SQL3 le dan a una base de datos relacional más flexibilidad en lo que pueden utiizar como tipos para una columna de una tabla. Por ejemplo, una columna podría ahora almacenar el nuevo tipo de dato **BLOB** (Binary Large Object), que puede almacenar grandes cantidades de datos como una fila de bytes. Una columna también puede ser del tipo **CLOB**

(Character Large Object), que es capaz de almacenar grandes cantidades de datos en formato caracter. El nuevo tipo **ARRAY** hace posible el uso de un array como un valor de columna. Incluso las nuevas estructuras de tipos-definidos-por-el-usuario (UDTs) de SQL pueden almacenarse como valores de columna.

La siguiente lista tiene los interfaces del JDBC 2.0 que mapean los tipos SQL3. Los explicaremos en más detalle más adelante.

Un ejemplar **Blob** mapea un **BLOB** de SQL.

Un ejemplar **Clob** mapea un **CLOB** de SQL.

Un ejempar **Array** mapea un **ARRAY** de SQL.

Un ejemplar **Struct** mapea un tipo Estructurado de SQL.

Un ejemplar **Ref** mapea un **REF** de SQL.

Uitlizar tipos de datos SQL3

Se recuperan, almacenan, y actualizan tipos de datos SQL3 de la misma forma que los otros tipos. Se utilizan los métodos **ResultSet.getXXX** o **CallableStatement.getXXX** para recuperarlos, los métodos **PreparedStatement.setXXX**para almacenarlos y **updateXXX** para actualizarlos. Probablemente el 90% de las operacuines realizadas con tipos SQL3 implican el uso de los métodos **getXXX**, **setXXX**, y **updateXXX**. La siguiente tabla muestra qué métodos utilizar.

Tipo SQL3	Método getXXX	Método setXXX	Método updateXXX
BLOB	**getBlob**	**setBlob**	updateBlob
CLOB	**getClob**	**setClob**	updateClob
ARRAY	**getArray**	**setArray**	updateArray
Tipo Structured	**getObject**	**setObject**	updateObject
REF(Tipo Structured)	**getRef**	**setRef**	updateRef

Por ejemplo, el siguiente fragmento de código recupera un valor **ARRAY** de SQL. Para este ejemplo, la columna **SCORES** de la tabla **STUDENTS** contiene valores del tipo **ARRAY**. La variable **stmt** es un objeto **Statement**.

```
ResultSet rs = stmt.executeQuery("SELECT SCORES FROM STUDENTS
        WHERE ID = 2238");
rs.next();
Array scores = rs.getArray("SCORES");
```

La variable **scores** es un puntero lógico al objeto **ARRAY** de SQL almacenado en la tabla **STUDENTS** en la fila del estudiante 2238.

Si queremos almacenar un valor en la base de datos, utilizamos el método **setXXX** apropiado. Por ejemplo, el siguiente fragmento de código, en el que **rs** es un objeto **ResultSet**, almacena un objeto **Clob**.

```
Clob notes = rs.getClob("NOTES");
PreparedStatement pstmt = con.prepareStatement("UPDATE MARKETS
                                                SET COMMENTS = ? WHERE
SALES < 1000000",
```

```
        ResultSet.TYPE_SCROLL_INSENSITIVE,

        ResultSet.CONCUR_UPDATABLE);
pstmt.setClob(1, notes);
```

Este código configura **notes** como el primer parámetro de la sentencia de actualización que está siendo enviada a la base de datos. El valor **CLOB** designado por **notes** será almacenado en la tabla **MARKETS** en la columna **COMMENTS** en cada columna en que el valor de la columna **SALES** sea menor de un millón.

Objetos Blob, Clob, y Array

Una característica importante sobre los objetos **Blob**, **Clob**, y **Array** es que se pueden manipular sin tener que traer todos los datos desde el servidor de la base de datos a nuestra máquina cliente. Un ejemplar de cualquiera de esos tipos es realmente un puntero lógico al objeto en la base de datos que representa el ejemplar. Como los objetos SQL **BLOB**, **CLOB**, o **ARRAY** pueden ser muy grandes, esta característica puede aumentar drásticamente el rendimiento.

Se pueden utilizar los comandos SQL y los API JDBC 1.0 y 2.0 con objetos **Blob**, **Clob**, y **Array** como si estuvieramos operando realmente con el objeto de la base de datos. Sin embargo, si queremos trabajar con cualquiera de ellos como un objeto Java, necesitamos traer todos los datos al cliente, con lo que la referencia se materializa en el objeto. Por ejemplo, si queremos utilizar un **ARRAY** de SQL en una aplicación como si fuera un array Java, necesitamos materializar el objeto **ARRAY** en el cliente para convertirlo en un array de Java. Entonces podremos utilizar los métodos de arrays de Java para operar con los elementos del array. Todos los interfaces **Blob**, **Clob**, y **Array** tienen métodos para materializar los objetos que representan.

Tipos Struct y Distinct

Los tipos estructurados y distinct de SQL son dos tipos de datos que el usuario puede definir. Normalmente nos referimos a ellos como UDTs (user-defined types), y se crean con un sentencia **CREATE TYPE** de SQL.

Un tipo estructurado de SQL se parece a los tipos estructurados de Java en que tienen miembros, llamados atributos, que puede ser cualquier tipo de datos. De echo, un atributo podría ser a su vez un tipo estructurado. Aquí tienes un ejemjplo de una simple definición de un nuevo tipo de dato SQL.

```
CREATE TYPE PLANE_POINT
(
               X FLOAT,
               Y FLOAT
)
```

Al contrario que los objetos **Blob**, **Clob**, y **Array**, un objeto **Struct** contiene valores para cada uno de los atributos del tipo estructurado de SQL y no es sólo un puntero lógico al objeto en la base de datos. Por ejemplo, supongamos que un objeto **PLANE_POINT** está almacenado en la columna **POINTS** de la tabla **PRICES**.

```
ResultSet rs = stmt.executeQuery("SELECT POINTS FROM PRICES  WHERE
PRICE > 3000.00");
while (rs.next()) {
        Struct point = (Struct)rs.getObject("POINTS");
```

```
        // do something with point
}
```

Si el objeto **PLANE_POINT** recuperado tiene un valor X de 3 y un valor Y de -5, el objeto **Struct**, **point** contendrá los valores 3 y -5.

Podríamos haber observador que **Struct** es el único tipo que no tiene métodos **getXXX** y **setXXX** con su nombre como **XXX**. Debemos utilizar **getObject** y **setObject** con ejemplares **Struct**. Esto significa que cuando recuperemos un valor utilizando el método **getObject**, obtendremos un **Object** Java que podremos forzar a **Struct**, como se ha hecho en el ejemplo de código anterior.

El segundo tipo SQL que un usuario puede definir con una sentencia **CREATE TYPE** de SQL es un tipo Distinct. Este tipo se parece al **typedef** de C o C++ en que es un tipo basado en un tipo existente. Aquí tenemos un ejemplo de creacción de un tipo distinct.

```
CREATE TYPE MONEY AS NUMERIC(10, 2)
```

Esta definición crea un nuevo tipo llamado **MONEY**, que es un número del tipo **NUMERIC** que siempre está en base 10 con dos dígitos después de la coma decimal. **MONEY** es ahora un tipo de datos en el esquema en el que fue definido, y podemos almacenar ejemplares de **MONEY** en una tabla que tenga una columna del tipo **MONEY**.

Un tipo distinct SQL se mapea en Java al tipo en el que se mapearía le tipo original. Por ejemplo, **NUMERIC** mapea a **java.math.BigDecimal**, porque el tipo **MONEY** mapea a **java.math.BigDecimal**. Para recuperar un objeto **MONEY**, utilizamos **ResultSet.getBigDecimal** o **CallableStatement.getBigDecimal**; para almacenarlo utilizamos **PreparedStatement.setBigDecimal**.

Características Avanzadas de SQL3

Algunos aspectos del trabajo con los tipos SQL3 pueden parecer bastante complejos. Mencionamos alguna de las características más avanzadas para que las conozcas, pero una explicación profunda no es apropiada para un tutorial básico.

El interface **Struct** es el mapeo estándard para un tipo estructurado de SQL. Si queremos trabajar fácilmente con un tipo estructurado de Java, podemos mapearlo a una clae Java. El tipo structurado se convierte en una clase, y sus atributos en campos. No tenemos porque utilizar un mapeado personalizado, pero normalmente es más conveniente.

Algunas veces podríamos querer trabajar con un puntero lógico a un tipo estructurado de SQL en vez de con todos los valores contenidos en el propio tipo, Esto podría suceder, por ejemplo, si el tipo estructurado tiene muchos atributos o si los atributos son muy grandes. Para referenciar un tipo estructurado, podemos declarar un tipo **REF** de SQL que represente un tipo estructurado particular. Un objeto **REF** de SQL se mapea en un objeto **Ref** de Java, y podemos operar con ella como si lo hicieramos con el objeto del tipo estructurado al que representa.

Tema 3: JSF - Java Server Faces (y comparación con Struts)

Introducción

JSF (Java Server Faces) es un framework de desarrollo basado en el patrón MVC (Modelo Vista Controlador).

Al igual que Struts, JSF pretende normalizar y estandarizar el desarrollo de aplicaciones web. Hay que tener en cuenta JSF es posterior a Struts, y por lo tanto se a nutrido de la experiencia de este, mejorando algunas sus deficiencias. De hecho el creador de Struts (Craig R. McClanahan) también es líder de la especificación de JSF.

A continuación presento algunos de los puntos por los que JSF me parece una tecnología muy interesante (incluso más que Struts ;)

- Hay una serie de especificaciones que definen JSF:

  JSR 127 (http://www.jcp.org/en/jsr/detail?id=127)
  JSR 252 (http://www.jcp.org/en/jsr/detail?id=252)
  JSR 276 (http://www.jcp.org/en/jsr/detail?id=276)

- JSF trata la vista (el interfaz de usuario) de una forma algo diferente a lo que estamos acostumbrados en aplicaciones web. Sería más similar al estilo de Swing, Visual Basic o Delphi, donde la programación del interfaz se hace a través de componentes y basada en eventos (se pulsa un botón, cambia el valor de un campo, ...).
- JSF es muy flexible. Por ejemplo nos permite crear nuestros propios componentes, o crear nuestros propios “render” para pintar los componentes según nos convenga.
- Es más sencillo.

JSF no puede competir en madurez con Struts (en este punto Struts es claro ganador), pero si puede ser una opción muy recomendable para nuevos desarrollos, sobre todo si todavía no tenemos experiencia con Struts.

Si queréis ver una comparativa más a fondo entre Struts y JSF os recomiendo el articulo: http://websphere.sys-con.com/read/46516.htm?CFID=61124&CFTOKEN=FD559D82-11F9-B3B2-738E901F37DDB4DE

Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil Ahtex Signal X-9500M (Centrino 1.6 GHz, 1024 MB RAM, 60 GB HD).
- Sistema Operativo: GNU / Linux, Debian Sid (unstable), Kernel 2.6.12, KDE 3.4
- Máquina Virtual Java: JDK 1.5.0_03 de Sun Microsystems
- Eclipse 3.1

- Tomcat 5.5.9
- MyFaces 1.0.9

Instalación de MyFaces

Una de las ventajas de que JSF sea una especificación es que podemos encontrar implementaciones de distintos fabricantes. Esto nos permite no atarnos con un proveedor y poder seleccionar el que más nos interesa según: número de componentes que nos proporciona, rendimiento, soporte, precio, política, ...

En nuestro caso vamos a usar el proyecto de Apache: MyFaces. Esta es una implementación de JSF de Software Libre que, además de cumplir con el estándar, también nos proporciona algunos componentes adicionales.

Descargamos myfaces-*.tgz de http://myfaces.apache.org/binary.cgi

Una vez descargado lo descomprimimos en un directorio.

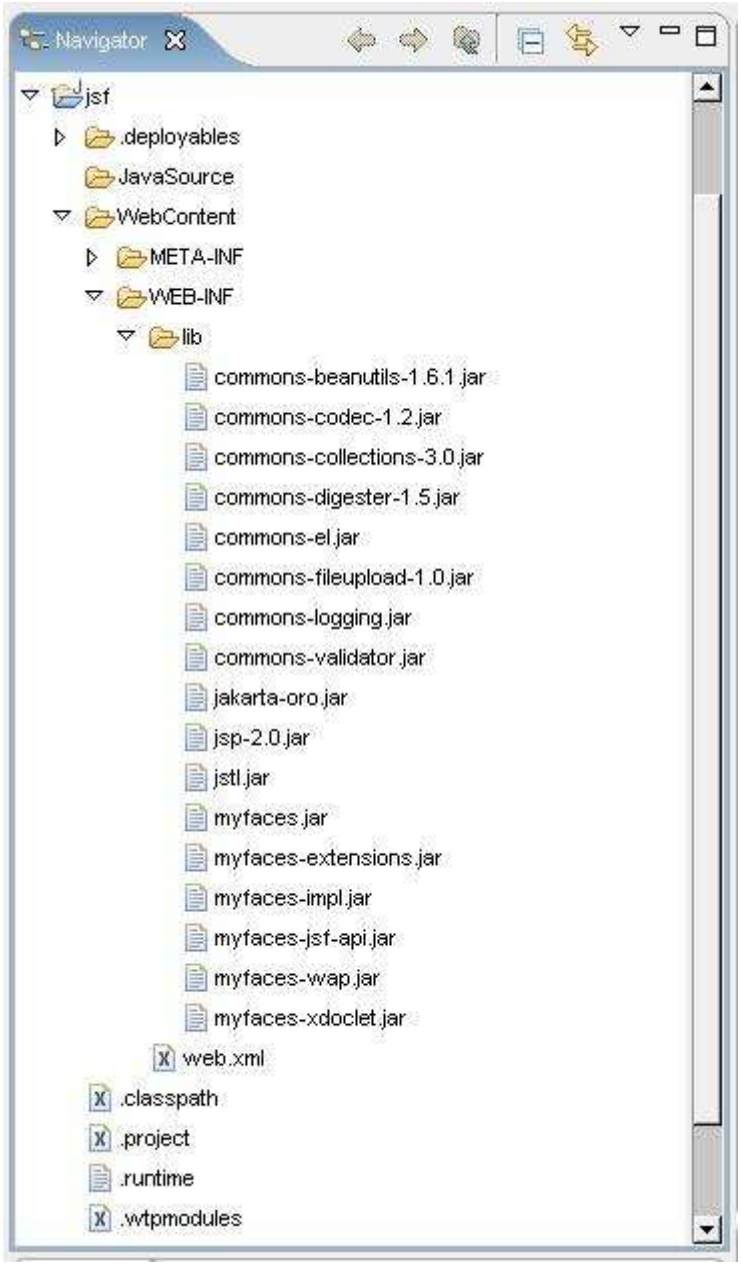
Creamos un proyecto web en nuestro Eclipse: File --> New --> Project... --> Dynamic Web Project

Copiamos las librerías necesarias para usar JSF

```
$ cp <MYFACES_HOME>/lib/* <MYPROJECT_HOME>/WebContent/WEB-
INF/lib
```

donde <MYFACES_HOME> es el directorio donde hemos descomprimido myfaces-*.tgz y <MYPROJECT_HOME> es el directorio de nuestro proyecto de Eclipse.

Nuestro proyecto tendrá el siguiente aspecto:



También modificamos nuestro web.xml para que quede de la siguiente manera:

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

<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

<context-param>
    <description>
        State saving method: "client" or "server" (= default)
        See JSF Specification 2.5.2
    </description>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
```

```
    <param-value>client</param-value>
</context-param>

<context-param>
    <description>
        This parameter tells MyFaces if javascript code
should be allowed in the
        rendered HTML output.
        If javascript is allowed, command_link anchors will
have javascript code
        that submits the corresponding form.
        If javascript is not allowed, the state saving info
and nested parameters
        will be added as url parameters.
        Default: "true"
    </description>
    <param-name>org.apache.myfaces.ALLOW_JAVASCRIPT</param-
name>
    <param-value>true</param-value>
</context-param>

<context-param>
    <description>
        If true, rendered HTML code will be formatted, so
that it is "human readable".
        i.e. additional line separators and whitespace will
be written, that do not
        influence the HTML code.
        Default: "true"
    </description>
    <param-name>org.apache.myfaces.PRETTY_HTML</param-name>
    <param-value>true</param-value>
</context-param>

<context-param>
    <param-name>org.apache.myfaces.DETECT_JAVASCRIPT</param-
name>
    <param-value>false</param-value>
</context-param>

<context-param>
    <description>
        If true, a javascript function will be rendered that
is able to restore the
        former vertical scroll on every request. Convenient
feature if you have pages
        with long lists and you do not want the browser page
to always jump to the top
        if you trigger a link or button action that stays on
the same page.
        Default: "false"
    </description>
    <param-name>org.apache.myfaces.AUTO_SCROLL</param-name>
    <param-value>false</param-value>
</context-param>

<!-- Listener, that does all the startup work (configuration,
init). -->
<listener>
```

```
    <listener-
class>org.apache.myfaces.webapp.StartupServletContextListener
</listener-class>
</listener>

<!-- Faces Servlet -->
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>

<!-- Welcome files -->
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Se puede ver como el servlet de JSF recogerá todas las peticiones que acaben en ".jsf".

Vamos al lío

Ya tenemos instalada una implementación de JSF. Ahora vamos a hacer una pequeña aplicación de ejemplo, para demostrar como funcionan las principales características de JSF (el movimiento se demuestra andando).

Primero vamos a describir nuestra aplicación de ejemplo (este ejemplo está basado en los ejemplos del tutorial Core-Servlets JSF Tutorial by Marty Hall – http://www.coreservlets.com/JSF-Tutorial), presentando cada uno de los ficheros que forman parte de ella, junto con una captura de las ventana correspondiente (si tiene representación visual), y junto a su código fuente.

Después, en cada capítulo, explicaremos concretamente cada uno de los puntos de la aplicación. Al final del tutorial habremos visto:

- Internacionalización (i18n)
- Recuperación de los datos del formulario
- Validaciones
- Gestión de eventos
- Navegación

4.1 customize.jsp

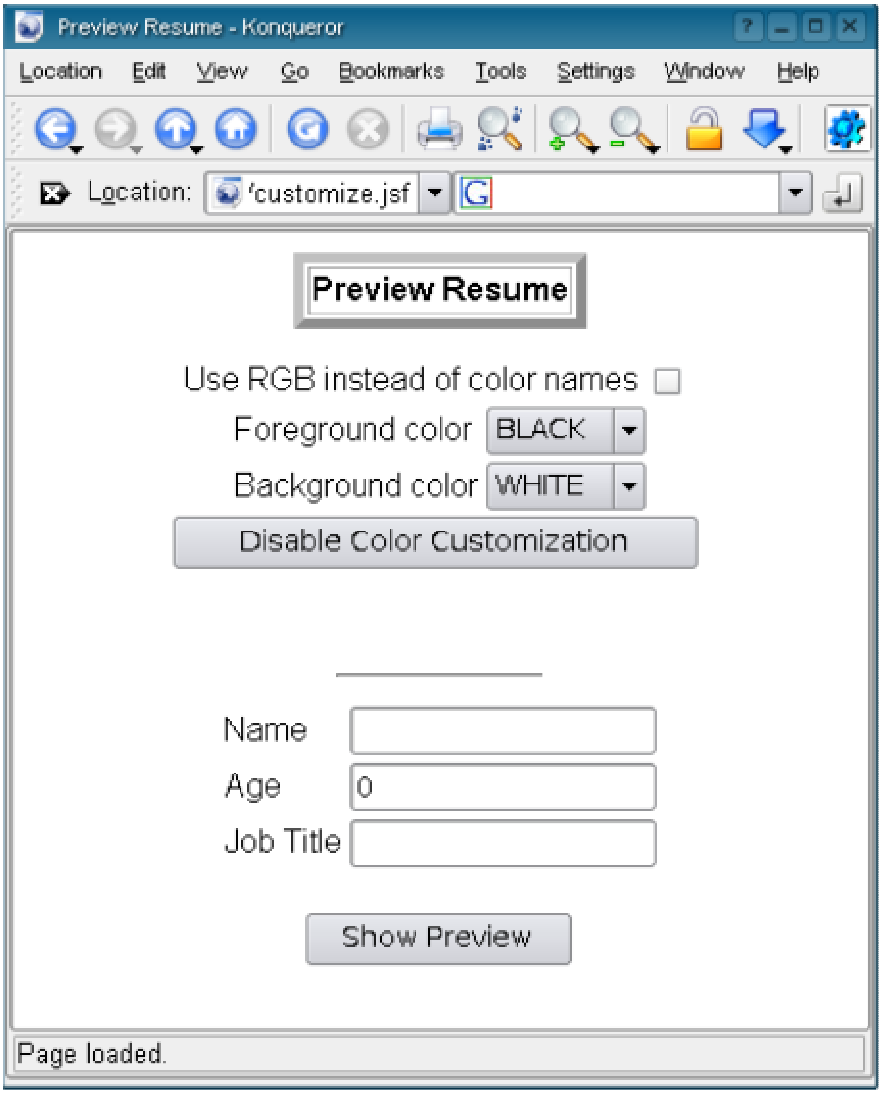
Dentro de la carpeta WebContent.

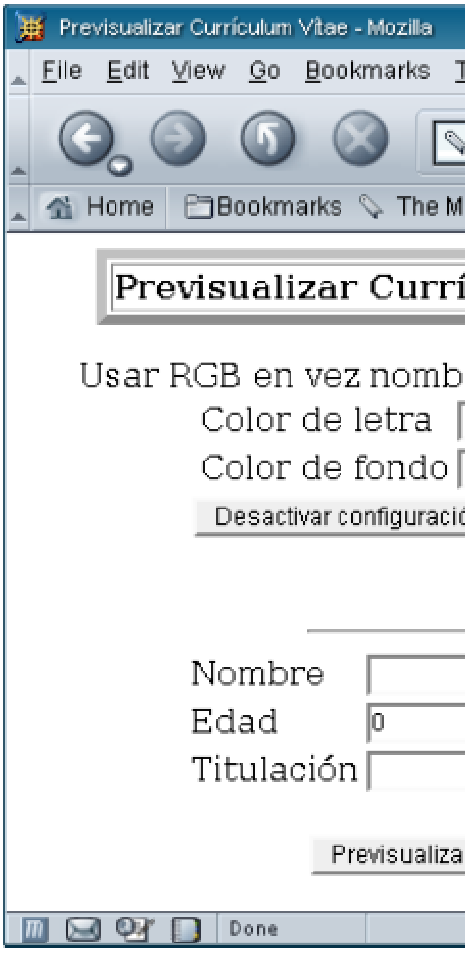
Esta será la ventana inicial de la aplicación. Es un simple formulario de entrada donde podemos personalizar los colores de nuestro CV ("Resume" en ingles), introducir nuestro nombre, la edad, y nuestra profesión.

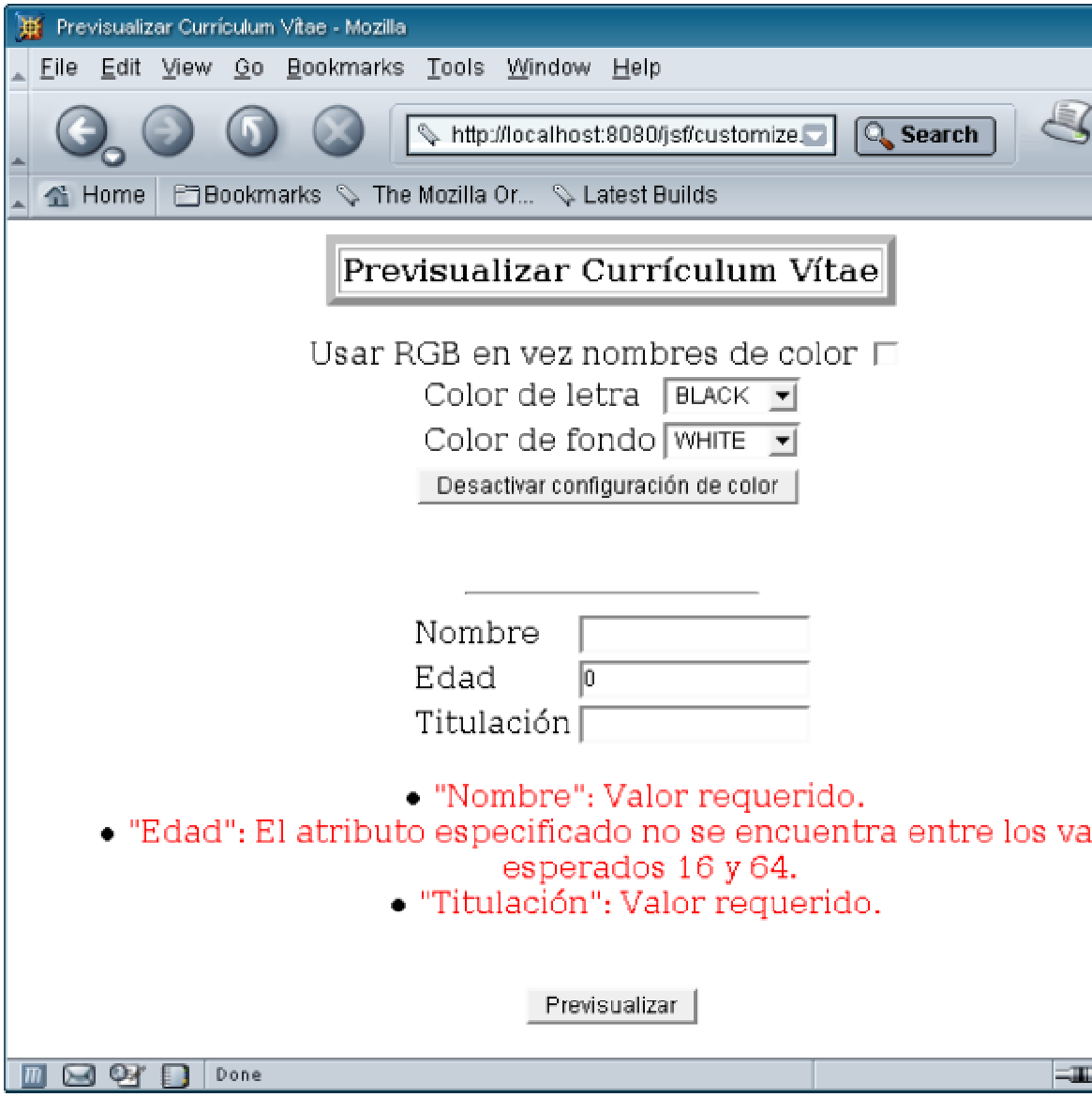
En las siguientes imágenes se puede apreciar la aplicación en distintos lenguajes, y como se muestran los errores de validación.

En el código podemos ver como se utilizan librerías de etiquetas para "pintar" los distintos componentes. En este sentido JSF es muy similar a Struts, ya que el interfaz de usuario se implementa con JSPs, la ventaja de JSF frente a Struts es que JSF desde un principio está pensado para que las herramientas de desarrollo faciliten la elaboración del interfaz de forma visual. Dentro de las este tipo de herramientas (también permiten modificar los ficheros de configuración de JSF de forma visual) podemos encontrar, entre otras muchas:

- Exadel Studio Pro y Exadel Studio (gratuito), http://www.exadel.com/products_exadelstudiopro.htm
- FacesIDE (Software Libre bajo CPL) http://amateras.sourceforge.jp/cgi-bin/fswiki_en/wiki.cgi?page=FacesIDE








```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://myfaces.apache.org/extensions"
prefix="x"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">

<html>
<f:view>
<f:loadBundle basename="MessageResources" var="msg"/>
<head>
    <title>${msg.customize_title}</title>
</head>

<body>
<center>
```

```
<table BORDER=5>
    <tr><th>${msg.customize_title}</th></tr>
</table>
<p/>
<h:form>
    <h:outputLabel for="changeColorMode"
value="#{msg.changeColorMode}" />
    <h:selectBooleanCheckbox id="changeColorMode"

valueChangeListener="#{resumeBean.changeColorMode}"
            immediate="true"
            onchange="submit()" />
    <br/>
    <h:panelGrid columns="2">
        <h:outputLabel for="fgColor" value="#{msg.fgColor}"
/>
        <h:selectOneMenu id="fgColor"
                value="#{resumeBean.fgColor}"
                disabled="#{!resumeBean.colorSupported}">
            <f:selectItems
value="#{resumeBean.availableColors}" />
        </h:selectOneMenu>

        <h:outputLabel for="bgColor" value="#{msg.bgColor}"
/>
        <h:selectOneMenu id="bgColor"
                value="#{resumeBean.bgColor}"
                disabled="#{!resumeBean.colorSupported}">
            <f:selectItems
value="#{resumeBean.availableColors}" />
        </h:selectOneMenu>
    </h:panelGrid>
    <h:commandButton value="#{resumeBean.colorSupportLabel}"
            actionListener="#{resumeBean.toggleColorSupport}"
            immediate="true" />
    <br/>
    <br/>
    <br/>
    <hr width="25%"/>
    <h:panelGrid id="i5" columns="2">
        <h:outputLabel for="name" value="#{msg.name}" />
        <h:inputText id="name" value="#{resumeBean.name}"
required="true" />

        <h:outputLabel for="age" value="#{msg.age}" />
        <h:inputText id="age" value="#{resumeBean.age}"
required="true">
            <f:validateLongRange minimum="16" maximum="64"/>
        </h:inputText>

        <h:outputLabel for="jobTitle" value="#{msg.jobTitle}"
/>
        <h:inputText id="jobTitle"
value="#{resumeBean.jobTitle}" required="true" />
    </h:panelGrid>
    <x:messages showSummary="false" showDetail="true"
style="color: red" />
    <br/>
```

```
    <h:commandButton value="#{msg.showPreview}"
action="#{resumeBean.showPreview}" />
</h:form>

</center>
</f:view>
</body>
</html>
```

4.2 same-color.jsp

Dentro de la carpeta WebContent.

Esta pantalla muestra un mensaje de error cuando se selecciona el mismo color para el fondo y para las letras.

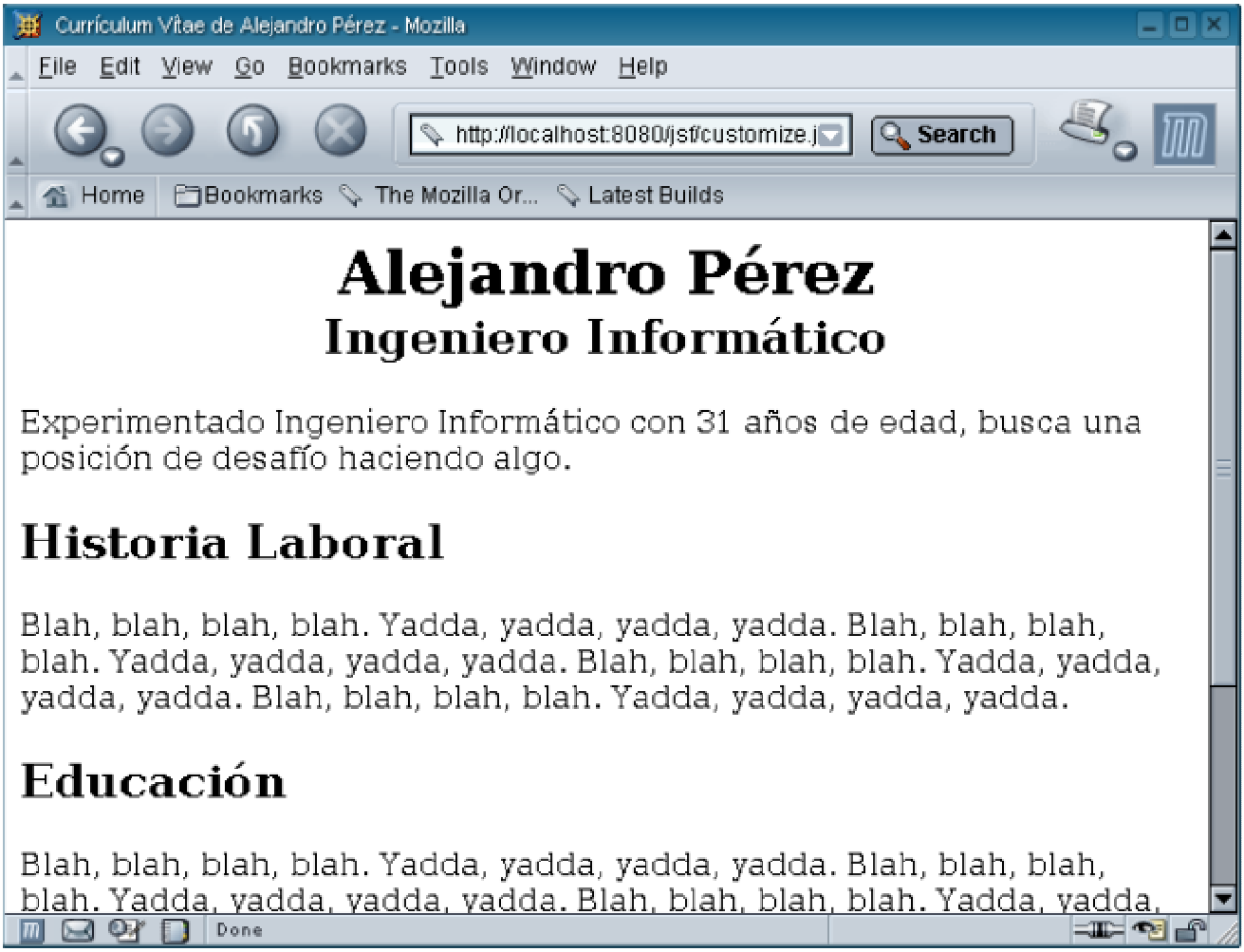

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://myfaces.apache.org/extensions"
prefix="x"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">

<html>
<f:view>
<f:loadBundle basename="MessageResources" var="msg"/>
<head>
        <title>${msg.sameColor_title}</title>
```

```
</head>

<body>
<center>

<table border=5>
        <tr><th>${msg.sameColor_title}</th></tr>
</table>
<p/>
<h:outputFormat value="#{msg.sameColor_message}"
escape="false">
        <f:param value="#{resumeBean.fgColor}"/>
</h:outputFormat>

</center>
</f:view>
</body>
</html>
```

4.3 show-preview.jsp

Dentro de la carpeta WebContent.

Muestra un borrador del CV según lo que hemos puesto en el formulario de la ventana inicial.




```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://myfaces.apache.org/extensions"
prefix="x"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">

<html>
<f:view>
<f:loadBundle basename="MessageResources" var="msg"/>
<head>
        <title>
                <h:outputFormat value="#{msg.preview_title}">
                        <f:param value="#{resumeBean.name}"/>
                </h:outputFormat>
        </title>
</head>

<body text="<h:outputText value="#{resumeBean.fgColor}" />"
      bgcolor="<h:outputText value="#{resumeBean.bgColor}"
/>">
<h1 align="CENTER">
```

```
        <h:outputText value="#{resumeBean.name}"/><BR>
        <small><h:outputText
value="#{resumeBean.jobTitle}"/></small>
</h1>
<h:outputFormat value="#{msg.summary}">
        <f:param value="#{resumeBean.age}"/>
        <f:param value="#{resumeBean.jobTitle}"/>
</h:outputFormat>
<h2>${msg.employmentHistory}</h2>
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
<h2>${msg.education}</h2>
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
<h2>${msg.publications}</h2>
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
Blah, blah, blah, blah. Yadda, yadda, yadda, yadda.
</f:view>
</body>
</html>
```

4.4 Ficheros de recursos

Dentro de la carpeta JavaSource.

Estos ficheros son los que contienen la traducción de los textos dependiendo del lenguaje.

MessageResources_es.properties

```
customize_title=Previsualizar Currículum Vítae
changeColorMode=Usar RGB en vez nombres de color
fgColor=Color de letra
bgColor=Color de fondo
name=Nombre
age=Edad
jobTitle=Titulación
colorSupportEnable=Activar configuración de color
colorSupportDisable=Desactivar configuración de color
showPreview=Previsualizar

sameColor_title=Error en el Color
sameColor_message=Has elegido "{0}" tanto para el color de
letra
como para el color de fondo.<p/>No te mereces encontrar un
trabajo,
pero supongo que te dejaremos <a href="customize.jsf">probar
de nuevo</a>.

preview_title=Currículum Vítae de {0}
```

```
summary=Experimentado {1} con {0} años de edad, busca una
posición de
desafío haciendo algo.
employmentHistory=Historia Laboral
education=Educación
publications=Publicaciones y Reconocimientos
```

MessageResources_en.properties

```
customize_title=Preview Resume
changeColorMode=Use RGB instead of color names
fgColor=Foreground color
bgColor=Background color
name=Name
age=Age
jobTitle=Job Title
colorSupportEnable=Enable Colors Customization
colorSupportDisable=Disable Color Customization
showPreview=Show Preview

sameColor_title=Color Error
sameColor_message=You chose "{0}" as both the foreground and
background
color.<p/>You don't deserve to get a job, but I suppose we
will let you
<a href="customize.jsf">try again</a>.

preview_title=Resume for {0}
summary={0} years old experienced {1}, seeks challenging
position doing
something.
employmentHistory=Employment History
education=Education
publications=Publications and Awards
```

4.5 ResumeBean.java

Dentro de la carpeta JavaSource/com/home/jsfexample.

Este bean es el que guardará toda la información recogida en el formulario. El equivalente en Struts sería una mezcla entre la clase de formulario (ActionForm) y la clase acción (Action). Esto lo podríamos ver como una ventaja ya que simplifica el código, aunque si nos gusta más la aproximación de Struts también es posible hacer dos clases distintas en JSF (ejemplo de la mayor flexibilidad de JSF frente a Struts).

También podemos observar que la clase no define ninguna relación de herencia (en Struts extendemos de Action o ActionForm). Esto es una ventaja ya que hay menos acoplamiento entre nuestro modelo y JSF.

```
package com.home.jsfexample;

import java.util.Locale;
import java.util.ResourceBundle;
```

```
import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;

public class ResumeBean {
    private String name = "";
    private int age = 0;
    private String jobTitle = "";
    private String fgColor = "BLACK";
    private String bgColor = "WHITE";
    private SelectItem[] availableColorNames = {
            new SelectItem("BLACK"), new SelectItem("WHITE"),
            new SelectItem("SILVER"), new SelectItem("RED"),
            new SelectItem("GREEN"), new SelectItem("BLUE")
};

    private SelectItem[] availableColorValues = {
            new SelectItem("#000000"), new
SelectItem("#FFFFFF"),
            new SelectItem("#C0C0C0"), new
SelectItem("#FF0000"),
            new SelectItem("#00FF00"), new
SelectItem("#0000FF") };

    private boolean isColorSupported = true;

    private boolean isUsingColorNames = true;

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

    public String getJobTitle() {
        return jobTitle;
    }

    public void setJobTitle(String jobTitle) {
        this.jobTitle = jobTitle;
    }
    public String getFgColor() {
        return fgColor;
    }
    public void setFgColor(String fgColor) {
        this.fgColor = fgColor;
    }
    public String getBgColor() {
        return bgColor;
    }
    public void setBgColor(String bgColor) {
        this.bgColor = bgColor;
    }

    public SelectItem[] getAvailableColors() {
        if (isUsingColorNames) {
```

```
                return (availableColorNames);
            } else {
                return (availableColorValues);
            }
        }

        public boolean isColorSupported() {
            return isColorSupported;
        }

        public void toggleColorSupport(ActionEvent event) {
            isColorSupported = !isColorSupported;
        }

        public String getColorSupportLabel() {
            FacesContext context =
    FacesContext.getCurrentInstance();
            Locale locale = context.getViewRoot().getLocale();
            ResourceBundle bundle =
    ResourceBundle.getBundle("MessageResources", locale);
            String msg = null;
            if (isColorSupported) {
                msg = bundle.getString("colorSupportDisable");
            } else {
                msg = bundle.getString("colorSupportEnable");
            }
            return msg;
        }

        public boolean isUsingColorNames() {
            return isUsingColorNames;
        }

        public void setUsingColorNames(boolean isUsingColorNames)
    {
            this.isUsingColorNames = isUsingColorNames;
        }

        public void changeColorMode(ValueChangeEvent event) {
            boolean flag =
    ((Boolean)event.getNewValue()).booleanValue();
            setUsingColorNames(!flag);
        }

        public String showPreview() {
            if (isColorSupported && fgColor.equals(bgColor)) {
                return "same-color";
            } else {
                return "success";
            }
        }

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }
```

```
}
```

4.6 faces-config.xml

Dentro de la carpeta WebContent/WEB-INF.

Este fichero es donde configuramos JSF. Es como el “pegamento” que une modelo, vista y controlador. El equivalente en Struts sería el fichero struts-config.xml.

En este fichero por un lado declaramos los beans que vamos a utilizar para recoger la información de los formularios (en el struts-config.xml teníamos una sección similar), y por otro lado las reglas de navegación (en el struts-config-xml se podría comparar con la definición de forwards de las acciones).

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>
    <application>
        <locale-config>
            <default-locale>es</default-locale>
            <supported-locale>en</supported-locale>
        </locale-config>
    </application>

    <!--
      - managed beans
      -->
    <managed-bean>
        <managed-bean-name>resumeBean</managed-bean-name>
        <managed-bean-
class>com.home.jsfexample.ResumeBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <!--
      - navigation rules
      -->
    <navigation-rule>
        <from-view-id>/customize.jsp</from-view-id>
        <navigation-case>
            <from-outcome>same-color</from-outcome>
            <to-view-id>/same-color.jsp</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/show-preview.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Internacionalización (i18n)

La internacionalización nos va ha permitir mostrar la aplicación según el idioma del usuario. Para ello en el fichero faces-config.xml vamos a especificar que localizaciones soporta la aplicación, y cual es la localización por defecto:

```
<application>
    <locale-config>
        <default-locale>es</default-locale>
        <supported-locale>en</supported-locale>
    </locale-config>
</application>
```

A continuación mostramos un ejemplo de uso de i18n, que podemos encontrar en el fichero customize.jsp:

```
...
<html>
<f:view>
<f:loadBundle basename="MessageResources" var="msg"/>
<head>
    <title>${msg.customize_title}</title>
</head>

<body>
...
```

En el ejemplo se puede ver como lo primero que se hace es cargar el fichero de recursos. El sistema se encargará de seleccionar el fichero apropiado según el lenguaje, y si se trata de un lenguaje no soportado se seleccionará el idioma por defecto.

Una vez cargado el fichero de recursos es tan facil usarlo como “${msg.customize_title}” donde “msg” es el identificador que le hemos dado al fichero de recursos y “cutomize_title” es la clave del mensaje que queremos mostrar.

Recuperando los valores del formulario

Desde el punto de vista del interfaz (customize.jsp) tenemos líneas del estilo:

```
<h:inputText id="name" value="#{resumeBean.name}"
required="true" />
```

Esto dibujará un campo de entrada y los valores introducidos los guardará en la propiedad “name” del bean “resumeBean”. Este bean es el que hemos definido en el fichero faces-config.xml:

```
    <managed-bean>
        <managed-bean-name>resumeBean</managed-bean-name>
        <managed-bean-
class>com.home.jsfexample.ResumeBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
```

```
</managed-bean>
```

Por supuesto la clase com.home.jsfexample.ResumeBean tendrá métodos get/set públicos para acceder a “name”.

Es importante destacar como en el fichero faces-config.xml se define el scope del bean. JSF buscará el bean en ese scope, y si no lo encuentra lo creará y lo guardará en ese scope.

Ojo porque JSF sólo se encarga de crear el bean, pero no de su destrucción, así que todo lo que guardemos en sesión o aplicación allí se quedará hasta que nosotros lo quitemos. Si queremos quitar un bean de la sesión (muy recomendable cuando el objeto ya no es necesario) podemos hacer algo como:

```
FacesContext context = FacesContext.getCurrentInstance();
context.getExternalContext().getSessionMap().remove("nombreDe
lBeanEnSesion");
```

Validación de los campos de entrada

Especificar las validaciones

En la vista encontramos cosas como:

```
<h:inputText id="name" value="#{resumeBean.name}"
required="true" />
```

Se ve como al definir el campo de entrada se está especificando que es obligatorio “required = true”. Otro ejemplo más sería:

```
        <h:inputText id="age" value="#{resumeBean.age}"
required="true">
            <f:validateLongRange minimum="16" maximum="64"/>
        </h:inputText>
```

Aquí se puede ver como, además de hacer que el campo sea obligatorio, se está fijando un valor mínimo y un valor máximo. Además como la propiedad “age” del bean “resumeBean” es de tipo “int” también se hará comprobación del tipo del valor introducido por el usuario, es decir, si el usuario no introduce un número, dará un error al validar.

Al igual que en Struts disponemos de multitud de validaciones predefinidas. Además también contamos con API que podemos implementar para hacer nuestras propias validaciones.

Esta forma de especificar las validaciones es más sencilla que en Struts, ya que se hace directamente en la definición del campo de entrada, y no en un fichero separado. Además la validación no está asociada a ningún bean en concreto como pasa en Struts, con lo que resulta más flexible.

Actualmente JSF no dispone de validaciones en cliente con JavaScript, pero esto no es un problema ya que si queremos usar esta funcionalidad siempre podemos usar Apache Commons Validator. Podéis encontrar un ejemplo en
http://jroller.com/page/dgeary?entry=added_commons_validator_support_to

Mostrar los errores de la validación

Para mostrar los errores de la validación basta con escribir la siguiente línea donde queremos mostrar los errores:

```
<x:messages showSummary="false" showDetail="true"
style="color: red" />
```

En nuestro caso estamos usando el componente con funcionalidad ampliada que proporciona MyFaces (x:messages). Este funciona igual que el definido por el estándar, pero al mostrar el error, en vez de aparecer el id del campo de entrada, aparece el valor de la etiqueta asociada. Con esto conseguimos que el mensaje sea más legible para el usuario, ya que se tendrá en cuenta el lenguaje en el que se está viendo la página.

Para escribir la etiqueta del campo de entrada usaremos:

```
<h:outputLabel for="name" value="#{msg.name}" />
```

donde el atributo “for” es el identificador del campo de entrada.

Por supuesto, al igual que en Strutrs, podemos redefinir los mensaje de error de las validaciones.

Gestión de eventos y navegación

En este apartado vamos a ver como gestionamos cuando un usuario pulsa un botón o cambia el valor de un campos de entrada, y como afecta ese evento a la navegación de la aplicación.

Con JSF todos los eventos se gestionan en el servidor, pero no hay ninguna restricción para tratar en cliente con JavaScript los eventos que afectan a como se visualiza el interfaz de usuario.

Aceptar el formulario

En nuestro ejemplo encontramos un par de botones, vamos a fijarnos primero en el botón de “Previsualizar”. Este botón sería el que típicamente acepta el formulario, manda los datos al servidor, y nos da paso a la siguiente pantalla.

La línea que define el botón y su comportamiento es:

```
<h:commandButton value="#{msg.showPreview}"
action="#{resumeBean.showPreview}" />
```

Se puede ver como el atributo “actión” define el método que se ha de invocar. En nuestro ejemplo el método “showPreview” del bean “resumeBean”. Lo importante de este tipo de métodos es que tienen que devolver un String indicando a donde hay que saltar.

```
public String showPreview() {
    if (isColorSupported && fgColor.equals(bgColor)) {
        return "same-color";
```

```
        } else {
            return "success";
        }
    }
```

Esto enlaza con las reglas de navegación que tenemos en el fichero faces-config.xml:

```
<navigation-rule>
    <from-view-id>/customize.jsp</from-view-id>
    <navigation-case>
        <from-outcome>same-color</from-outcome>
        <to-view-id>/same-color.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/show-preview.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

Si estamos en la página “customize.jsp” y el método devuelve el String “same-color” saltaremos a la página “same-color.jsp”. Pero si el método devuelve el String “success” saltaremos a la página “show-preview.jsp”. Como ya adelantábamos, es muy similar al funcionamiento de los forward de Struts, tiene la ventaja de que nuestro bean no tiene que extender de ninguna clase en concreto.

No siempre es necesario invocar a un método para activar la regla de navegación. Esto sólo será necesario para una navegación dinámica donde, dependiendo de alguna condición, saltaremos a una página o a otra. Si la navegación es estática, es decir siempre saltamos a la misma página, no haría falta implementar ningún método, y bastaría con poner:

```
<h:commandButton value="#{msg.showPreview}" action="success"
/>
```

Un evento en un botón

Podemos ver este tipo de eventos como acciones que implican cambios sobre el interfaz de usuario, pero todavía no hacemos la aceptación y procesamiento de los datos del formulario.

Tenemos el ejemplo del botón “Desactivar configuración de color”. Cuando el usuario pulse este botón se desactivarán las listas desplegables que hay sobre él, y cambia el texto del propio botón a “Activar configuración de color”.

```
<h:commandButton value="#{resumeBean.colorSupportLabel}"
        actionListener="#{resumeBean.toggleColorSupport}"
        immediate="true" />
```

Para conseguir que cambie le texto se puede ver como, en vez de usar directamente i18n, le pedimos el valor al propio bean con “#{resumeBean.colorSupportLabel}”.

Con “immediate='true'” conseguimos que no se ejecuten las validaciones.

Con el atributo “actionListener” estamos definiendo quien atenderá el evento cuando se pulse el botón. En nuestro ejemplo es el método “toggleColorSupport” del bean “resumeBean”. Este tipo de métodos no devuelven nada y tienen un único parámetro de tipo “ActionEvent”.

```
    public void toggleColorSupport(ActionEvent event) {
        isColorSupported = !isColorSupported;
    }
```

Este método simplemente cambia el valor de la propiedad “isColorSupported”. El valor de esta propiedad es consultado cuando se va a pintar la lista de selección, en el atributo “disabled”:

```
            <h:selectOneMenu id="fgColor"
                    value="#{resumeBean.fgColor}"
                    disabled="#{!resumeBean.colorSupported}">
                <f:selectItems
value="#{resumeBean.availableColors}" />
            </h:selectOneMenu>
```

Un evento en un check box

En nuestro ejemplo tenemos un check box que cuando está seleccionado hace que los en vez de los nombres de los colores veamos su codificación RGB (Red Green Blue).

Este caso es un poquito más complicado, ya que la pulsación de un check box no supone el envío del formulario al servidor. En este caso utilizamos:

```
        <h:selectBooleanCheckbox id="changeColorMode"

valueChangeListener="#{resumeBean.changeColorMode}"
                immediate="true"
                onchange="submit()" />
```

Nuevamente estamos utilizando un método que atiende el evento del cambio del valor del check box, lo definimos con el atributo “valueChangeListener”. Este tipo de métodos no devuelven nada y tienen un único parámetro de tipo “ValueChangeEvent”.

```
    public void changeColorMode(ValueChangeEvent event) {
        boolean flag =
((Boolean)event.getNewValue()).booleanValue();
        setUsingColorNames(!flag);
    }
```

Volvemos a utilizar “immediate” para que no se ejecuten las validaciones.

Es fundamental el uso de “onchange='submit()'”. Esta es la principal diferencia con el caso anterior, y es lo que nos permite que el procesamiento del evento se haga de forma inmediata una vez el usuario pulsa sobre el check box.

Tema 4. OBJETOS DISTRIBUIDOS CON RMI

Sistemas Distribuidos Orientados a Objetos, Arquitectura de RMI

Un sistema distribuido orientado a objetos provee los mecanismos necesarios para un manejo transparente de la comunicación, de modo que el programador se ocupe de la lógica de la aplicación. Esta idea de abstraer la parte de comunicación fue introducida por primera vez con RPC (Remote Procedure Call). RPC no sigue el paradigma de orientación a objetos por lo que el diseño de sistemas distribuidos orientados a objetos debe hacerse desde los cimientos.

Objetivos de RMI

Permitir invocación remota de métodos en objetos que residen en diferentes Máquinas Virtuales

Permitir invocación de métodos remotos por Applets

Integrar el Modelo de Objetos Distribuidos al lenguaje Java de modo natural y preservando en lo posible la semántica de objetos en Java

Permitir la distinción entre objetos locales y remotos

Preservar la seguridad de tipos (type safety) dada por el ambiente de ejecución Java

Permitir diferentes semánticas en las referencias a objetos remotos: no persistentes (vivas), persistentes, de activación lenta.

Mantener la seguridad del ambiente dada por los Security Managers y Class Loaders

Facilitar el desarrollo de aplicaciones distribuidas

Aplicación Genérica:

Servidor:

- Crea objeto remoto
- Crea referencia al objeto remoto
- Espera a que un cliente invoque un método en el objeto remoto

Cliente:

- Obtiene referencia a un objeto remoto en el servidor
- Invoca un método remoto

Funcionalidades requeridas:

Localización de objetos remotos : En RMI se implementa un `Registry` (registro) que actúa como servidor de nombres y permite la localización de objetos remotos

Comunicación con objetos remotos : A cargo del Sistema RMI, es transparente para el programador

Paso de parámetros y resultados (objetos locales y remotos) en métodos remotos : Incluye el manejo de referencias y la carga dinámica de clases

El Sistema RMI se puede analizar en términos de un modelo de capas (ver figura 1). Cada capa tiene una función específica. El programador solo se ocupa de la capa de Aplicación; las demás capas son responsabilidad del sistema RMI.

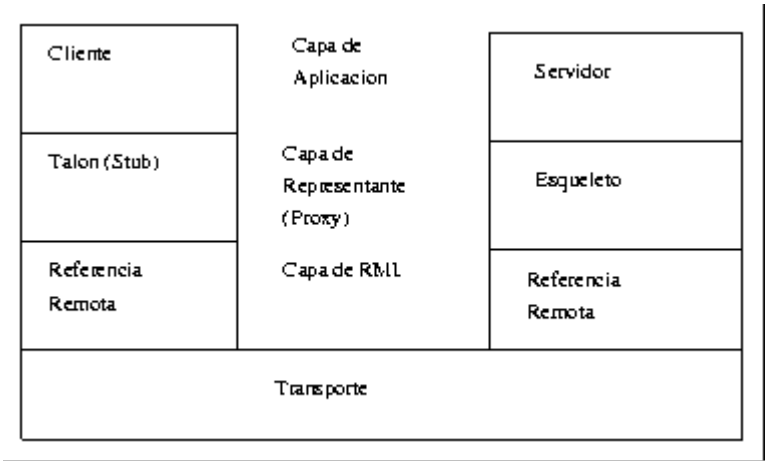


Figura 1. *Modelo de capas RMI*

La función de las distintas capas se explica a continuación:

Capa de Aplicación . En esta capa se encuantran los objetos que implementan a la aplicación. En general en este nivel se distinguen dos tipos de agentes: los clientes, que son objetos que invocan métodos o hacen peticiones a otros objetos remotos y los servidores, que son objetos que reciben peticiones de otros objetos remotos.

Capa de Representantes (Proxy) . En esta capa se encuentran los objetos que actúan como representantes locales de objetos remotos. Se encargan del embalaje y desembalaje (marshalling) de las invocaciones, argumentos y resultados de métodos remotos. En RMI existen dos tipos de representantes: los `talones` (stubs) del lado de los clientes y los `esqueletos` (skeletons) del lado de los servidores.

Capa de Referencias Remotas . En esta capa se realiza la interpretacion de las referencias a objetos remotos, las referencias locales a talones o esqueletos se resuelven a sus contrapartes remotas y estos datos, junto con los paquetes "embalados" (marshalled) que contienen las invocaciones o los resultados de invocaciones se pasan a la Capa de Transporte.

Capa de Transporte . En esta capa se encuentra el protocolo de comunicación, se encarga de transportar los mensajes que intercambian los distintos objetos. RMI utiliza por omisión el protocolo TCP/IP.

Modelo de Objetos Distribuidos en Java

Definición

Objeto Remoto:

Objeto cuyos métodos pueden invocarse desde otras Máquinas Virtuales

Descrito por una o más Interfaces Remotas (las que implementa) en las que se declaran los métodos que pueden ser invocados por objetos desde otras VMs

Difiere de un objeto local en:

Se puede hacer conversión de tipos (Cast) a una interfaz remota que el objeto implemente

Su tipo se puede revisar con el operador `instanceof` : prueba las interfaz remota que un objeto implementa

Es subclase de RemoteObject, clase que redefine métodos de java.lang Object : toString( ), equals( ), hashCode ( ), clone ( ), de modo adecuado para objetos remotos, por ejemplo, dos referencias remotas son iguales si apuntan al mismo objeto remoto, sin importar el contenido o estado de dicho objeto

Definición

Invocación a Método Remoto:

Acción de invocar un método de una interfaz remota en un objeto remoto

Tiene la misma sintaxis de una invocación a un método local

Difiere de una invocación local en:

Argumentos no remotos en invocaciones remotas se pasan por valor (copia)

Objetos remotos se pasan por referencia

Hay Excepciones Remotas. Estas excepciones no son de tipo runtime, por lo que deben ser cachadas o lanzadas de manera adecuada.

Interfaces y Clases en RMI

Implementan el Sistema RMI 5 Paquetes:

java.rmi . Contiene Clases, Interfases y Excepciones vistas por los clientes

java.rmi.server . Contiene Clases, Interfaces y Excepciones vistas por los servidores

java.rmi.registry . Contiene Clases, Interfaces y Excepciones útiles para localizar y registrar objetos remotos

java.rmi.dgc . Contiene Clases, Interfaces y Excepciones para la recolección de basura

java.rmi.activation . Contiene Clases, Interfaces y Excepciones para la activación de objetos remotos

Paso de Parámetros a Métodos Remotos

Hay 3 Mecanismos básicos (paso de argumentos y resultados)

Tipos primitivos : se pasan por valor (copia). Todos son serializables

Objetos Remotos: se pasan por referencia (talones, usados para invocar métodos remotos).

Objetos Locales: se pasan por valor (solo si son serializables), se crea un nuevo objeto en la Máquina Virtual que recibe la copia.

Observaciones

Los objetos remotos no viajan, en cambio se envían referencias (que permiten crear talones o sustitutos) a quienes los reciben, ya sea como argumentos de un método o como resultados de un método

Un talón se debe convertir al tipo (cast) de las interfaces remotas que implemente la clase del objeto remoto al que corresponde. El tipo del talón determina que métodos remotos se pueden invocar en el objeto remoto (aquellos que se declaran en la interfaz correspondiente)

Si un objeto remoto implementa varias interfaces remotas un cliente solo puede convertir el talón a una de ellas (y por lo tanto solo podrá invocar los métodos declarados en esa interfaz)

Dos talones que se refieren al mismo objeto remotos en el mismo servidor se consideran iguales bajo la operacion `equals`.

Tema 5: Introducción a la tecnología EJB

Los contenidos que vamos a ver en el tema son:

Desarrollo basado en componentes

Servicios proporcionados por el contenedor EJB

Funcionamiento de componentes EJB

Tipos de beans

Desarrollo de beans

Clientes de los beans

Roles EJB

Evolución de la especificación EJB

Ventajas de la tecnología EJB

Desarrollo basado en componentes

Con la tecnología J2EE Enterprise JavaBeans es posible desarrollar componentes (*enterprise beans*) que luego puedes reutilizar y ensamblar en distintas aplicaciones que tengas que hacer para la empresa. Por ejemplo, podrías desarrollar un bean `Cliente` que represente un cliente en una base de datos. Podrías usar después ese bean `Cliente` en un programa de contabilidad o en una aplicación de comercio electrónico o virtualmente en cualquier programa en el que se necesite representar un cliente. De hecho, incluso sería posible que el desarrollador del bean y el ensamblador de la aplicación no fueran la misma persona, o ni siquiera trabajaran en la misma empresa.

El desarrollo basado en componentes promete un paso más en el camino de la programación orientada a objetos. Con la programación orientada a objetos puedes reutilizar clases, pero con componentes es posible reutilizar n mayor nivel de funcionalidades e incluso es posible modificar estas funcionalidades y adaptarlas a cada entorno de trabajo particular sin tocar el código del componente desarrollado. Aunque veremos el tema con mucho más detalle, en este momento puedes ver un componente como un objeto tradicional con un conjunto de servicios adicionales soportados en tiempo de ejecución por el contenedor de componentes. El contenedor de componentes se denomina *contenedor EJB* y es algo así como el sistema operativo en el que éstos residen. Recuerda que en Java existe un modelo de programación de objetos remotos denominado RMI. Con RMI es posible enviar peticiones a objetos que están ejecutándose en otra máquina virtual Java. Podemos ver un componente EJB como un objeto remoto RMI que reside en un contenedor EJB que le proporciona un conjunto de servicios adicionales.

Cuando estés trabajando con componentes tendrás que dedicarle tanta atención al despliegue (*deployment*) del componente como a su desarrollo. Entendemos por despliegue la incorporación del componente a nuestro contenedor EJB y a nuestro entorno de trabajo (bases de datos, arquitectura de la aplicación, etc.). El despliegue se define de forma declarativa, mediante un fichero XML (descriptor del despliegue, *deployment descriptor*) en el que se definen todas las características del bean.

El desarrollo basado en componentes ha creado expectativas sobre la aparición de una serie de empresas dedicadas a implementar y vender componentes específicos a terceros. Este mercado de componentes nunca ha llegado a tener la suficiente masa crítica como para crear una industria sostenible. Esto es debido a distintas razones, como la dificultad en el diseño de componentes genéricos capaces de adaptarse a distintos dominios de aplicación, la falta de

estandarización de los dominios de aplicación o la diversidad de estos dominios. Aun así, existe un campo creciente de negocio en esta área (puedes ver, como ejemplo, www.componentsource.com).

Servicios proporcionados por el contenedor EJB

En el apartado anterior hemos comentado que la diferencia fundamental entre los componentes y los objetos clásicos reside en que los componentes *viven* en un contenedor EJB que los envuelve proporcionando una capa de servicios añadidos. ¿Cuáles son estos servicios? Los más importantes son los siguientes:

Manejo de transacciones: apertura y cierre de transacciones asociadas a las llamadas a los métodos del bean.

Seguridad: comprobación de permisos de acceso a los métodos del bean.

Concurrencia: llamada simultánea a un mismo bean desde múltiples clientes.

Servicios de red: comunicación entre el cliente y el bean en máquinas distintas.

Gestión de recursos: gestión automática de múltiples recursos, como colas de mensajes, bases de datos o fuentes de datos en aplicaciones heredadas que no han sido traducidas a nuevos lenguajes/entornos y siguen usándose en la empresa.

Persistencia: sincronización entre los datos del bean y tablas de una base de datos.

Gestión de mensajes: manejo de Java Message Service (JMS).

Escalabilidad: posibilidad de constituir clusters de servidores de aplicaciones con múltiples hosts para poder dar respuesta a aumentos repentinos de carga de la aplicación con sólo añadir hosts adicionales.

Adaptación en tiempo de despliegue: posibilidad de modificación de todas estas características en el momento del despliegue del bean.

Pensad en lo complicado que sería programar una clase "a mano" que implementara todas estas características. Como se suele decir, la programación de EJB es sencilla si la comparamos con lo que habría que implementar de hacerlo todo por uno mismo. Evidentemente, si en la aplicación que estás desarrollando no vas a necesitar estos servicios y va a tener un interfaz web, podrías utilizar simplemente páginas JSP y JDBC.

Funcionamiento de los componentes EJB

El funcionamiento de los componentes EJB se basa fundamentalmente en el trabajo del contenedor EJB. El contenedor EJB es un programa Java que corre en el servidor y que contiene todas las clases y objetos necesarios para el correcto funcionamiento de los enterprise beans.

En la figura siguiente puedes ver una representación de muy alto nivel del funcionamiento básico de los enterprise beans. En primer lugar, puedes ver que el cliente que realiza peticiones al bean y el servidor que contiene el bean están ejecutándose en máquinas virtuales Java distintas. Incluso pueden estar en distintos hosts. Otra cosa a resaltar es que el cliente *nunca* se comunica directamente con el enterprise bean, sino que el contenedor EJB proporciona un *EJBObject* que hace de interfaz. Cualquier petición del cliente (una llamada a un *método de negocio* del enterprise bean) se debe hacer a través del objeto EJB, el cual solicita al contenedor EJB una serie de servicios y se comunica con el enterprise bean. Por último, el bean realiza las peticiones correspondientes a la base de datos.

El contenedor EJB se preocupa de cuestiones como:

¿Tiene el cliente permiso para llamar al método?

Hay que abrir la transacción al comienzo de la llamada y cerrarla al terminar.

¿Es necesario refrescar el bean con los datos de la base de datos?

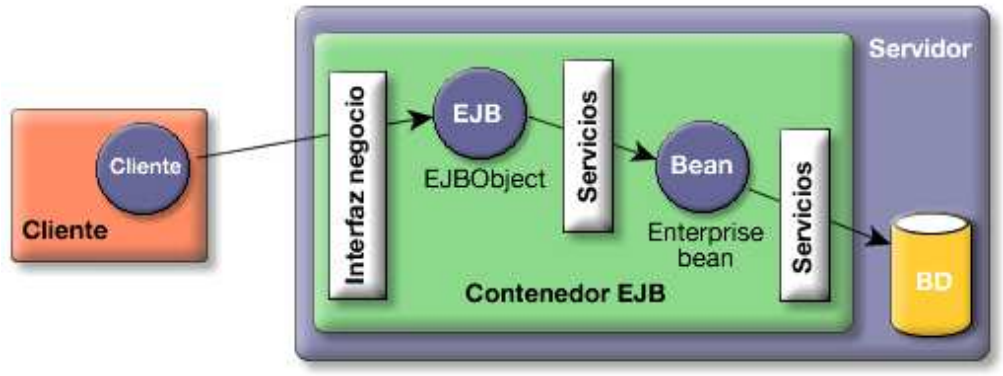


Figura 1.1: representación de alto nivel del funcionamiento de los enterprise beans.

Vamos a ver un ejemplo para que puedas entender mejor el flujo de llamadas. Supongamos que tenemos una aplicación de bolsa y el bean proporciona una implementación de un Broker. La interfaz de negocio del Broker está compuesta de varios métodos, entre ellos, por ejemplo, los métodos `compra` o `vende`. Supongamos que desde el objeto cliente queremos llamar al método `compra`. Esto va a provocar la siguiente secuencia de llamadas:

Cliente: "Necesito realizar una petición de compra al bean Broker."

EJBObject: "Espera un momento, necesito comprobar tus permisos."

Contenedor EJB: "Sí, el cliente tiene permisos suficientes para llamar al método compra."

Contenedor EJB: "Necesito un bean Broker para realizar una operación de compra. Y no olvidéis comenzar la transacción en el momento de instanciaros."

Pool de beans: "A ver... ¿a quién de nosotros le toca esta vez?".

Contenedor EJB: "Ya tengo un bean Broker. Pásale la petición del cliente."

Por cierto, la idea de usar este tipo de diálogos para describir el funcionamiento de un proceso o una arquitectura de un sistema informático es de Kathy Sierra en sus libros *"Head First Java"* y "*Head First EJB"*. Se trata de un tipo de libros radicalmente distintos a los habituales manuales de Java que consiguen que realmente aprendas este lenguaje cuando los sigues. Échales un vistazo si tienes oportunidad.

Tipos de beans

La tecnología EJB define tres tipos de beans: beans de sesión, beans de entidad y beans dirigidos por mensajes.

Los **beans de entidad** representan un objeto concreto que tiene existencia en alguna base de datos de la empresa. Una instancia de un bean de entidad representa una fila en una tabla de la base de datos. Por ejemplo, podríamos considerar el bean Cliente, con una instancia del bean siendo Eva Martínez (ID# 342) y otra instancia Francisco Gómez (ID# 120).

Los **beans dirigidos por mensajes** pueden escuchar mensajes de un servicio de mensajes JMS. Los clientes de estos beans nunca los llaman directamente, sino que es necesario enviar un mensaje JMS para comunicarse con ellos. Los beans dirigidos por mensajes no necesitan

objetos EJBObject porque los clientes no se comunican nunca con ellos directamente. Un ejemplo de bean dirigido por mensajes podría ser un bean ListenerNuevoCliente que se activara cada vez que se envía un mensaje comunicando que se ha dado de alta a un nuevo cliente.

Por último, un **bean de sesión** representa un proceso o una acción de negocio. Normalmente, cualquier llamada a un servicio del servidor debería comenzar con una llamada a un bean de sesión. Mientras que un bean de entidad representa una cosa que se puede representar con un nombre, al pensar en un bean de sesión deberías pensar en un verbo. Ejemplos de beans de sesión podrían ser un carrito de la compra de una aplicación de negocio electrónico o un sistema verificador de tarjetas de crédito.

Vamos a describir con algo más de detalle estos tipos de bean. Comenzamos con los beans de sesión para continuar con los de entidad y terminar con los dirigidos por mensajes.

Beans de sesión

Los beans de sesión representan sesiones interactivas con uno o más clientes. Los bean de sesión pueden mantener un estado, pero sólo durante el tiempo que el cliente interactúa con el bean. Esto significa que los beans de sesión no almacenan sus datos en una base de datos después de que el cliente termine el proceso. Por ello se suele decir que los beans de sesión no son persistentes.

A diferencia de los bean de entidad, los beans de sesión no se comparten entre más de un cliente, sino que existe una correspondencia uno-uno entre beans de sesión y clientes. Por esto, el contenedor EJB no necesita implementar mecanismos de manejo de concurrencia en el acceso a estos beans.

Existen dos tipos de beans de sesión: con estado y sin él.

Beans de sesión sin estado

Los beans de sesión sin estado no se modifican con las llamadas de los clientes. Los métodos que ponen a disposición de las aplicaciones clientes son llamadas que reciben datos y devuelven resultados, pero que no modifican internamente el estado del bean. Esta propiedad permite que el contenedor EJB pueda crear una reserva (*pool*) de instancias, todas ellas del mismo bean de sesión sin estado y asignar cualquier instancia a cualquier cliente. Incluso un único bean puede estar asignado a múltiples clientes, ya que la asignación sólo dura el tiempo de invocación del método solicitado por el cliente.

Una de las ventajas del uso de beans de sesión, frente al uso de clases Java u objetos RMI es que no es necesario escribir los métodos de los beans de sesión de una forma segura para threads (*thread-safe*), ya que el contenedor EJB se va a encargar de que nunca haya más de un thread accediendo al objeto. Para ello usa múltiples instancias del bean para responder a peticiones de los clientes.

Cuando un cliente invoca un método de un bean de sesión sin estado, el contenedor EJB obtiene una instancia de la reserva. Cualquier instancia servirá, ya que el bean no puede guardar ninguna información referida al cliente. Tan pronto como el método termina su ejecución, la instancia del bean está disponible para otros clientes. Esta propiedad hace que los beans de sesión sin estado sean muy escalables para un gran número de clientes. El contenedor EJB no tiene que mover sesiones de la memoria a un almacenamiento secundario para liberar recursos, simplemente puede obtener recursos y memoria destruyendo las instancias.

Los beans de sesión sin estado se usan en general para encapsular procesos de negocio, más que datos de negocio (tarea de los entity beans). Estos beans suelen recibir nombres como `ServicioBroker` o `GestorContratos` para dejar claro que proporcionan un conjunto de procesos relacionados con un dominio específico del negocio.

Es apropiado usar beans de sesión sin estado cuando una tarea no está ligada a un cliente específico. Por ejemplo, se podría usar un bean sin estado para enviar un e-mail que confirme un pedido on-line o calcular unas cuotas de un préstamo.

También puede usarse un bean de sesión sin estado como un puente de acceso a una base de datos o a un bean de entidad. En una arquitectura cliente-servidor, el bean de sesión podría proporcionar al interfaz de usuario del cliente los datos necesarios, así como modificar objetos de negocio (base de datos o bean de entidad) a petición de la interfaz. Este uso de los beans de sesión sin estado es muy frecuente y constituye el denominado patrón de diseño *session facade*.

Algunos ejemplos de bean de sesión sin estado podrían ser:

Un componente que comprueba si un símbolo de compañía está disponible en el mercado de valores y devuelve la última cotización registrada.

Un componente que calcula la cuota del seguro de un cliente, basándose en los datos que se le pasa del cliente.

Beans de sesión con estado

En un bean de sesión con estado, las *variables de instancia* del bean almacenan datos específicos obtenidos durante la conexión con el cliente. Cada bean de sesión con estado, por tanto, almacena el estado conversacional de un cliente que interactúa con el bean. Este estado conversacional se modifica conforme el cliente va realizando llamadas a los métodos de negocio del bean. El estado conversacional no se guarda cuando el cliente termina la sesión.

La interacción del cliente con el bean se divide en un conjunto de pasos. En cada paso se añade nueva información al estado del bean. Cada paso de interacción suele denominarse con nombres como `setNombre` o `setDireccion`, siendo `nombre` y `direccion` dos variables de instancia del bean.

Algunos ejemplos de beans de sesión con estado podrían ser:

Un ejemplo típico es un carrito de la compra, en donde el cliente va guardando uno a uno los ítem que va comprando.

Un enterprise bean que reserva un vuelo y alquila un coche en un sitio Web de una agencia de viajes.

El estado del bean persiste mientras que existe el bean. A diferencia de los beans de entidad, no existe ningún recurso exterior al contenedor EJB en el que se almacene este estado.

Debido a que el bean guarda el estado conversacional con un cliente determinado, no le es posible al contenedor crear un almacén de beans y compartirlos entre muchos clientes. Por ello, el manejo de beans de sesión con estado es más pesado que el de beans de sesión sin estado.

En general, se debería usar un bean de sesión con estado si se cumplen las siguientes circunstancias:

El estado del bean representa la interacción entre el bean y un cliente específico.

El bean necesita mantener información del cliente a lo largo de un conjunto de invocaciones de métodos.

El bean hace de intermediario entre el cliente y otros componentes de la aplicación, presentando una vista simplificada al cliente.

Beans de entidad

Los beans de entidad modelan conceptos o datos de negocio que puede expresarse como nombres. Esto es una regla sencilla más que un requisito formal, pero ayuda a determinar cuándo un concepto de negocio puede ser implementado como un bean de entidad. Los beans de entidad representan “cosas”: objetos del mundo real como hoteles, habitaciones, expedientes, estudiantes, y demás. Un bean de entidad puede representar incluso cosas abstractas como una reserva. Los beans de entidad describen tanto el estado como la conducta de objetos del mundo real y permiten a los desarrolladores encapsular las reglas de datos y de negocio asociadas con un concepto específico. Por ejemplo un bean de entidad `Estudiante` encapsula los datos y reglas de negocio asociadas a un estudiante. Esto hace posible manejar de forma consistente y segura los datos asociados a un concepto.

Los beans de entidad se corresponden con datos en un almacenamiento persistente (base de datos, sistema de ficheros, etc.). Las variables de instancia del bean representan los datos en las columnas de la base de datos. El contenedor debe sincronizar las variables de instancia del bean con la base de datos. Los beans de entidad se diferencian de los beans de sesión en que las variables de instancia se almacenan de forma persistente.

Aunque entraremos en detalle más adelante, es interesante adelantar que el uso de los beans de entidad desde un cliente conlleva los siguientes pasos:

Primero el cliente debe obtener una referencia a la instancia concreta del bean de entidad que se está buscando (el estudiante "Francisco López") mediante un método *finder*. Estos métodos finder se encuentran definidos en la interfaz home e implementados en la clase bean. Los métodos finder pueden devolver uno o varios beans de entidad.

El cliente interactúa con la instancia del bean usando sus métodos *get* y *set*. El estado del bean se carga de la base de datos antes de procesar las llamadas a los métodos. Esto se encarga de hacerlo el contenedor de forma automática o el propio bean en la función `ejbLoad()`.

Por último, cuando el cliente termina la interacción con la instancia del bean sus contenidos se vuelcan en el almacen persistente. O bien lo hace de forma automática el contenedor o bien éste llama al método `ejbStore()`.

Son muchas las ventajas de usar beans de entidad en lugar de acceder a la base de datos directamente. El uso de beans de entidad nos da una perspectiva orientada a objetos de los datos y proporciona a los programadores un mecanismo más simple para acceder y modificar los datos. Es mucho más fácil, por ejemplo, cambiar el nombre de un estudiante llamando a `student.setName()` que ejecutando un comando SQL contra la base de datos. Además, el uso de objetos favorece la reutilización del software. Una vez que un bean de entidad se ha definido, su definición puede usarse a lo largo de todo el sistema de forma consistente. Un bean `Estudiante` proporciona un forma completa de acceder a la información del estudiante y eso asegura que el acceso a la información es consistente y simple.

La representación de los datos como beans de entidad puede hacer que el desarrollo sea más sencillo y menos costoso.

Diferencias con los beans de sesión

Los beans de entidad se diferencian de los beans de sesión, principalmente, en que son persistentes, permiten el acceso compartido, tienen clave primaria y pueden participar en relaciones con otros beans de entidad:

Persistencia

Debido a que un bean de entidad se guarda en un mecanismo de almacenamiento se dice que es persistente. Persistente significa que el estado del bean de entidad existe más tiempo que la duración de la aplicación o del proceso del servidor J2EE. Un ejemplo de datos persistentes son los datos que se almacenan en una base de datos.

Los beans de entidad tienen dos tipos de persistencia: **Persistencia Gestionada por el Bean** (BMP, *Bean-Managed Persistence*) y **Persistencia Gestionada por el Contenedor** (CMP, *Container-Managed Persistence*). En el primer caso (BMP) el bean de entidad contiene el código que accede a la base de datos. En el segundo caso (CMP) la relación entre las columnas de la base de datos y el bean se describe en el fichero de propiedades del bean, y el contenedor EJB se ocupa de la implementación.

Acceso compartido

Los clientes pueden compartir beans de entidad, con lo que el contenedor EJB debe gestionar el acceso concurrente a los mismos y por ello debe usar transacciones. La forma de hacerlo dependerá de la política que se especifique en los descriptores del bean.

Clave primaria

Cada bean de entidad tiene un identificador único. Un bean de entidad alumno, por ejemplo, puede identificarse por su número de expediente. Este identificador único, o *clave primaria*, permite al cliente localizar a un bean de entidad particular.

Relaciones

De la misma forma que una tabla en una base de datos relacional, un bean de entidad puede estar relacionado con otros EJB. Por ejemplo, en una aplicación de gestión administrativa de una universidad, el bean `alumnoEjb` y el bean `actaEjb` estarían relacionados porque un alumno aparece en un acta con una calificación determinada.

Las relaciones se implementan de forma distinta según se esté usando la persistencia manejada por el bean o por el contenedor. En el primer caso, al igual que la persistencia, el desarrollador debe programar y gestionar las relaciones. En el segundo caso es el contenedor el que se hace cargo de la gestión de las relaciones. Por ello, estas últimas se denominan a veces relaciones gestionadas por el contenedor.

Beans dirigidos por mensajes

Son el tercer tipo de beans propuestos por la última especificación de EJB. Estos beans permiten que las aplicaciones J2EE reciban mensajes JMS de forma asíncrona. Así, el hilo de ejecución de un cliente no se bloquea cuando está esperando que se complete algún método de negocio de otro enterprise bean. Los mensajes pueden enviarse desde cualquier componente J2EE (una aplicación cliente, otro enterprise bean, o un componente Web) o por una aplicación o sistema JMS que no use la tecnología J2EE.

Diferencias con los beans de sesión y de entidad

La diferencia más visible es que los clientes no acceden a los beans dirigidos por mensajes mediante interfaces (explicaremos esto con más detalle más adelante), sino que un bean dirigido por mensajes sólo tienen una clase bean.

En muchos aspectos, un bean dirigido por mensajes es parecido a un bean de sesión sin estado.

Las instancias de un bean dirigido por mensajes no almacenan ningún estado conversacional ni datos de clientes.

Todas las instancias de los beans dirigidos por mensajes son equivalentes, lo que permite al contenedor EJB asignar un mensaje a cualquier instancia. El contenedor puede almacenar estas instancias para permitir que los streams de mensajes sean procesados de forma concurrente.

Un único bean dirigido por mensajes puede procesar mensajes de múltiples clientes.

Las variables de instancia de estos beans pueden contener algún estado referido al manejo de los mensajes de los clientes. Por ejemplo, pueden contener una conexión JMS, una conexión de base de datos o una referencia a un objeto enterprise bean.

Cuando llega un mensaje, el contenedor llama al método `onMessage` del bean. El método `onMessage` suele realizar un casting del mensaje a uno de los cinco tipos de mensajes de JMS y manejarlo de forma acorde con la lógica de negocio de la aplicación. El método onMessage puede llamar a métodos auxiliares, o puede invocar a un bean de sesión o de entidad para procesar la información del mensaje o para almacenarlo en una base de datos.

Un mensaje puede enviarse a un bean dirigido por mensajes dentro de un contexto de transacción, por lo que todas las operaciones dentro del método onMessage son parten de un única transacción.

Desarrollo de beans

El desarrollo y programación de los beans suele ser un proceso bastante similar sea cual sea el tipo de bean. Consta de los siguientes 5 pasos:

Escribe y compila la clase *bean* que contiene a todos los métodos de negocio.

Escribe y compila las dos interfaces del bean: *home* y *componente*.

Crea un descriptor XML del despliegue en el que se describa qué es el bean y cómo debe manejarse. Este fichero debe llamarse `ejb-jar.xml`.

Pon la clase bean, los interfaces y el descriptor XML del despliegue en un fichero EJB JAR . Podría haber más de un bean el mismo fichero EJB JAR, pero nunca habrá más de un descriptor de despliegue.

Despliega el bean en el servidor usando las herramientas proporcionadas por el servidor de aplicaciones.

Vamos a ver con algo más de detalle estos cinco pasos, usando un ejemplo sencillo de un bean de sesión sin estado que implementa el método de negocio `saluda()` que devuelve un string con un saludo.

Clase bean

En la clase bean se encuentran los denominados métodos de negocio. Son los métodos finales a los que el cliente quiere acceder y los que debes programar. Son también los métodos definidos en la interfaz componente.

Lo primero que debes hacer es decidir qué tipo de bean necesitas implementar: un bean de sesión, de entidad o uno dirigido por mensajes. Estos tres tipos se definen con tres interfaces distintas: `SessionBean`, `EntityBean` y `MessageBean`. La clase bean que vas a escribir debe implementar una de ellas. En nuestro caso, vamos a definir un bean de sesión sin estado, por lo que la clase `SaludoBean` implementará la interfaz `SessionBean`.

```
package especialista;
import javax.ejb.*;

public class SaludoBean implements SessionBean {
   private String[] saludos = {"Hola", "Que tal?", "Como estas?",
   "Cuanto tiempo sin verte!", "Que te cuentas?", "Que hay de
nuevo?"};

   // Los cuatro metodos siguientes son los de la interfaz
   // SessionBean
   public void ejbActivate() {
      System.out.println("ejb activate");
   }

   public void ejbPassivate() {
      System.out.println("ejb pasivate");
   }

   public void ejbRemove() {
      System.out.println("ejb remove");
   }

   public void setSessionContext(SessionContext cntx) {
      System.out.println("set session context");
   }

   // El siguiente es el metodo de negocio, al que
   // van a poder llamar los clientes del bean
   public String saluda() {
      System.out.println("estoy en saluda");
      int random = (int) (Math.random() * saludos.length);
      return saludos[random];
   }

   // Por ultimo, el metodo ejbCreate que no es de
   // la interfaz sessionBean sino que corresponde al
   // metodo de creacion de beans de la interfaz Home del EJB
   public void ejbCreate() {
      System.out.println("ejb create");
   }
}
```

Pregunta:

¿Por qué los métodos de la clase bean, a diferencia de los métodos de las interfaces *componente* y *home*, no definen la excepción `RemoteException`?

Interfaces *componente* y *home*

Una vez implementado el fichero `SaludoBean.java`, en el que se define el enterprise bean, debes pasar a definir las interfaces *componente* y *home* del bean. Vamos a llamar a estas interfaces `Saludo` y `SaludoHome`. Estas interfaces son las que deberá usar el cliente para comunicarse con el bean.

La interfaz *componente* hereda de la interfaz `EJBObject` y en ella se definen los métodos de negocio del bean, los que va a poder llamar el cliente para pedir al bean que realice sus funciones:

```
package especialista;

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

public interface Saludo extends EJBObject {
   public String saluda() throws RemoteException;
}
```

Todos los métodos definidos en esta interfaz se corresponden con los métodos de negocio del bean y todos van a ser métodos remotos (¿recuerdas RMI?), ya que van a implementarse en una máquina virtual Java distinta de la máquina. Por ello, todos estos métodos deben declarar la excepción `RemoteException`.

La interfaz *home* hereda de la interfaz `EJBHome`. El cliente usa los métodos de esta interfaz para obtener una referencia a la interfaz componente. Puedes pensar en el home como en una especie de fábrica que construye referencias a los beans y las distribuye entre los clientes.

```
package especialista;

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

public interface SaludoHome extends EJBHome {
   public Saludo create() throws CreateException, RemoteException;
}
```

El método `create()` se corresponde con el método `ejbCreate()` definido en la clase `SaludoBean`, y debe devolver el tipo `Saludo` de la interfaz *componente*. La interfaz también va a ser una interfaz remota y, por tanto, debe declarar la excepción `RemoteException`. Además, el método `create` debe declarar la excepción `CreateException`.

Cuando se despliega un bean en el contenedor EJB, éste crea dos objetos que llamaremos EJBObject y EJBHome que implementarán estas interfaces. Estos objetos separan el bean del cliente, de forma que el cliente nunca accede directamente al bean. Así el contenedor puede incorporar sus servicios a los métodos de negocio.

Preguntas:

¿Dónde se deberán instalar los ficheros `.class` resultantes de las compilaciones de estas interfaces: en el servidor, en el cliente o en ambos?

¿Qué sucede si definimos algún método en la clase bean que después no lo definimos en la interfaz *componente*?

Descriptor del despliegue

Los tres ficheros Java anteriores son todo lo que tienes que escribir en Java. Recuerda: una clase (`SaludoBean`) y dos interfaces (`SaludoHome` y `Saludo`). Ya queda poco para terminar. El cuarto y último elemento es tan importante como los anteriores. Se trata del descriptor de despliegue (*deployment descriptor*, DD) del bean. El descriptor de despliegue es un fichero XML en el que se detalla todo lo que el servidor necesita saber para gestionar el bean. El nombre de este fichero siempre debe ser `ejb-jar.xml`.

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

<!DOCTYPE ejb-jar PUBLIC
'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <display-name>Ejb1</display-name>
  <enterprise-beans>

    <session>
      <display-name>SaludoBean</display-name>
      <ejb-name>SaludoBean</ejb-name>
      <home>especialista.SaludoHome</home>
      <remote>especialista.Saludo</remote>
      <ejb-class>especialista.SaludoBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>

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

En este caso, dado que el ejemplo es muy sencillo, tenemos que definir pocos elementos. Le estamos diciendo al servidor cuáles son las clases bean y las interfaces home y componente (en el fichero XML la llaman *remote*) del bean. Debes saber que no existe ninguna norma en la arquitectura EJB sobre los nombres de las distintas clases Java que conforman el bean. Por eso es necesario indicarle al servidor cuáles son éstas mediantes los correspondientes elementos en el DD. También le decimos qué tipo de bean queremos utilizar (un bean se sesión sin estado).

Existen muchos más elementos que podemos definir en el DD. Los iremos viendo poco a poco a lo largo de las siguientes sesiones.

Fichero ejb-jar

Una vez escritas las clases e interfaces y el descriptor del despliegue, debemos compactar todos los ficheros resultantes (los ficheros .class y el fichero XML) en un único fichero JAR.

La estructura de este fichero JAR es:

```
/META-INF/ejb-jar.xml

/especialista/Saludo.class
```

```
/especialista/SaludoHome.class
/especialista/SaludoBean.class
```

En el directorio META-INF se incluye el DD. El resto del fichero JAR corresponde al directorio definido por el package `especialista` y a los tres ficheros `.class` que definen el bean.

La mayoría de servidores de aplicaciones proporcionan herramientas gráficas que crean el fichero de descripción y empaquetan el bean de forma automática.

Este fichero es el que se desplegará en el servidor de aplicaciones. Puedes nombrar a este fichero con el nombre que quieras. Una costumbre bastante usada es llamarlo <nombre>-ejb.jar, siendo <nombre> el nombre del bean o de la aplicación. En nuestro caso, podríamos llamarlo `saludo-ejb.jar`.

Despliegue del bean

Una vez construido el fichero EJB JAR es necesario desplegarlo en el servidor de aplicaciones. El despliegue conlleva dos acciones: la primera es darle al bean un nombre externo (un nombre JNDI, hablando más técnicamente) para que los clientes y otros beans puedan referirse a él. En nuestro caso, le daremos como nombre "SaludoBean". La segunda acción es la de llevar al servidor de aplicaciones el fichero EJB JAR.

Existen dos escenarios diferenciados para la realización del despliegue, dependiendo de si has desarrollado el bean en el mismo host que se encuentra el servidor de aplicaciones o en un host distinto. El primer caso suele suceder cuando estás trabajando en modo de prueba y estás depurando el desarrollo. El segundo caso suele suceder cuando ya has depurado el bean y quieres desplegarlo en modo de producción: ¡es recomendable no desarrollar y depurar en el mismo host en el que se encuentra el servidor de aplicaciones en producción!.

El proceso de despliegue no está definido en la especificación J2EE y cada servidor de aplicaciones tiene unas características propias. En general, la mayoría de servidores de aplicaciones proporcionan un interfaz gráfico de administración para gestionar el despliegue. También la mayoría de servidores proporcionan una tarea de ant para poder realizar el despliegue usando esta herramienta desde la línea de comando.

En el la sesión práctica veremos un ejemplo de cada tipo con el servidor de aplicaciones weblogic de BEA.

Por último, hay un elemento previo al despliegue que, por simplificar, no hemos comentado. Se trata del proceso de ensamblaje de la aplicación (Application Assembly). En este proceso se construye, a partir de uno o más ficheros EJB JAR con beans, un único fichero de aplicación EAR en el que además se pueden asignar valores a ciertas constantes que serán utilizadas por los beans. Todo ello se hace sin necesidad de recompilar las clases e interfaces de cada uno de los beans.

Clientes de los beans

Una vez que el bean está desplegado en el contenedor, ya podemos usarlo desde un cliente. El cliente puede ser una clase Java cualquiera, ya sea un cliente aislado o un servlet que se está ejecutando en el contenedor web del servidor de aplicaciones. El código que deben ejecutar los clientes del bean es básicamente el mismo en cualquier caso.

Puedes ver a continuación el código de un cliente que usa el bean.

```
 1. import java.io.*;
 2. import java.text.*;
 3. import java.util.*;
 4. import javax.servlet.*;
 5. import javax.servlet.http.*;
 6. import javax.naming.*;
 7. import javax.rmi.*;
 8. import especialista.*;
 9.
10. public class SaludoClient {
11.
12.   public static void main(String [] args) {
13.      try {
14.         Context jndiContext = getInitialContext();
15.         Object ref = jndiContext.lookup("SaludoBean");
16.         SaludoHome home = (SaludoHome)
17.            PortableRemoteObject.narrow(ref, SaludoHome.class);
18.         Saludo sal = (Saludo)
19.            PortableRemoteObjet.narrow(home.create(),
Saludo.class);
20.         System.out.println("Voy a llamar al bean");
21.         System.out.println(sal.saluda());
22.         System.out.println("Ya he llamado al bean");
23.      } catch (Exception e) {e.printStackTrace();}
24.   }
25.
26.   public static Context getInitialContext()
27.     throws javax.naming.NamingException {
28.        Properties p = new Properties();
29.        p.put(Context.INITIAL_CONTEXT_FACTORY,
30.                "weblogic.jndi.WLInitialContextFactory");
31.        p.put(Context.PROVIDER_URL, "t3://localhost:7001");
32.        return new javax.naming.InitialContext(p);
33.   }
34. }
```

Básicamente, el cliente debe realizar siempre las siguientes tareas:

Acceder al servicio JNDI (línea 14 y líneas 26-34), obteniendo el contexto JNDI inicial. Para ello se llama a la función `javax.naming.InitialContext()`, pasándole como argumento unas propiedades dependientes del servidor que implementa el JNDI. En este caso estamos asumiendo que el servicio JNDI lo proporciona un servidor de aplicaciones BEA weblogic que está ejecutándose en el localhost.

Localizar el bean proporcionando a JNDI su nombre lógico (línea 15). En este caso, el nombre JNDI del bean es `SaludoBean`.

Hacer un casting del objeto que devuelve JNDI para convertirlo en un objeto de la clase `SaludoHome` (líneas 16 y 17). La forma de hacer el casting es especial, ya que antes de hacer el casting hay que obtener un objeto Java llamando al método `PotableRemoteObject.narrow()` porque estamos recibiendo de JNDI un objeto que ha sido serializado usando el protocolo IIOP.

Llamar al método `create()` del objeto home para crear un objeto de tipo `Saludo` (línea 19). Lo que se obtiene es un stub (ya hablaremos más de ello en la siguiente sesión) y hay que llamar otra vez a narrow para asegurarse de que el objeto devuelto satisface

Llamar a los métodos de negocio del bean (línea 21).

Roles EJB

La arquitectura EJB define seis papeles principales. Brevemente, son:

Desarrollador de beans: desarrolla los componentes enterprise beans.

Ensamblador de aplicaciones: compone los enterprise beans y las aplicaciones cliente para conformar una aplicación completa

Desplegador: despliega la aplicación en un entorno operacional particular (servidor de aplicaciones)

Administrador del sistema: configura y administra la infraestructura de computación y de red del negocio

Proporcionador del Contenedor EJB y Proporcionador del Servidor EJB: un fabricante (o fabricantes) especializado en manejo de transacciones y de aplicaciones y otros servicios de bajo nivel. Desarrollan el servidor de aplicaciones.

Ventajas de la tecnología EJB

La arquitectura EJB proporciona beneficios a todos los papeles que hemos mencionado previamente (desarrolladores, ensambladores de aplicaciones, administradores, desplegadores, fabricantes de servidores). Vamos en enumerar las ventajas que obtendrán los desarrolladores de aplicaciones y los clientes finales.

Las ventajas que ofrece la arquitectura Enterprise JavaBeans a un desarrollador de aplicaciones se listan a continuación.

Simplicidad. Debido a que el contenedor de aplicaciones libera al programador de realizar las tareas del nivel del sistema, la escritura de un enterprise bean es casi tan sencilla como la escritura de una clase Java. El desarrollador no tiene que preocuparse de temas de nivel de sistema como la seguridad, transacciones, multi-threading o la programación distribuida. Como resultado, el desarrollador de aplicaciones se concentra en la lógica de negocio y en el dominio específico de la aplicación.

Portabilidad de la aplicación. Una aplicación EJB puede ser desplegada en cualquier servidor de aplicaciones que soporte J2EE.

Reusabilidad de componentes. Una aplicación EJB está formada por componentes enterprise beans. Cada enterprise bean es un bloque de construcción reusable. Hay dos formas esenciales de reusar un enterprise bean a nivel de desarrollo y a nivel de aplicación cliente. Un bean desarrollado puede desplegarse en distintas aplicaciones, adaptando sus características a las necesidades de las mismas. También un bean desplegado puede ser usado por múltiples aplicaciones cliente.

Posibilidad de construcción de aplicaciones complejas. La arquitectura EJB simplifica la construcción de aplicaciones complejas. Al estar basada en componentes y en un conjunto claro y bien establecido de interfaces, se facilita el desarrollo en equipo de la aplicación.

Separación de la lógica de presentación de la lógica de negocio. Un enterprise bean encapsula típicamente un proceso o una entidad de negocio. (un objeto que representa datos del negocio), haciéndolo independiente de la lógica de presentación. El programador de gestión no necesita de preocuparse de cómo formatear la salida; será el programador que desarrolle la página Web el que se ocupe de ello usando los datos de salida que proporcionará el bean. Esta separación hace posible desarrollar distintas lógicas de presentación para la misma lógica de negocio o cambiar la lógica de presentación sin modificar el código que implementa el proceso de negocio.

Despliegue en muchos entornos operativos. Entendemos por entornos operativos el conjunto de aplicaciones y sistemas (bases de datos, sistemas operativos, aplicaciones ya en

marcha, etc.) que están instaladas en una empresa. Al detallarse claramente todas las posibilidades de despliegue de las aplicaciones, se facilita el desarrollo de herramientas que asistan y automaticen este proceso. La arquitectura permite que los beans de entidad se conecten a distintos tipos de sistemas de bases de datos.

Despliegue distribuido. La arquitectura EJB hace posible que las aplicaciones se desplieguen de forma distribuida entre distintos servidores de una red. El desarrollador de beans no necesita considerar la topología del despliegue. Escribe el mismo código independientemente de si el bean se va a desplegar en una máquina o en otra (cuidado: con la especificación 2.0 esto se modifica ligeramente, al introducirse la posibilidad de los interfaces locales).

Interoperabilidad entre aplicaciones. La arquitectura EJB hace más fácil la integración de múltiples aplicaciones de diferentes vendedores. El interfaz del enterprise bean con el cliente sirve como un punto bien definido de integración entre aplicaciones.

Integración con sistemas no-Java. Las APIs relacionadas, como las especificaciones Connector y Java Message Service (JMS), así como los beans manejados por mensajes, hacen posible la integración de los enterprise beans con sistemas no Java, como sistemas ERP o aplicaciones mainframes.

Recursos educativos y herramientas de desarrollo. El hecho de que la especificación EJB sea un estándar hace que exista una creciente oferta de herramientas y formación que facilita el trabajo del desarrollador de aplicaciones EJB.

Entre las ventajas que aporta esta arquitectura al cliente final, destacamos la posibilidad de elección del servidor, la mejora en la gestión de las aplicaciones, la integración con las aplicaciones y datos ya existentes y la seguridad.

Elección del servidor. Debido a que las aplicaciones EJB pueden ser ejecutadas en cualquier servidor J2EE, el cliente no queda ligado a un vendedor de servidores. Antes de que existiera la arquitectura EJB era muy difícil que una aplicación desarrollada para una determinada capa intermedia (Tuxedo, por ejemplo) pudiera portarse a otro servidor. Con la arquitectura EJB, sin embargo, el cliente deja de estar atado a un vendedor y puede cambiar de servidor cuando sus necesidades de escalabilidad, integración, precio, seguridad, etc.lo requieran.
Existen en el mercado algunos servidores de aplicaciones gratuitos (JBOSS, el servidor de aplicaciones de Sun, etc.) con los que sería posible hacer unas primeras pruebas del sistema, para después pasar a un servidor de aplicaciones con más funcionalidades.

Gestión de las aplicaciones. Las aplicaciones son mucho más sencillas de manejar (arrancar, parar, configurar, etc.) debido a que existen herramientas de control más elaboradas.

Integración con aplicaciones y datos ya existentes. La arquitectura EJB y otras APIs de J2EE simplifican y estandarizan la integración de aplicaciones EJB con aplicaciones no Java y sistemas en el entorno operativo del cliente. Por ejemplo, un cliente no tiene que cambiar un esquema de base de datos para encajar en una aplicación. En lugar de ello, se puede construir una aplicación EJB que encaje en el esquema cuando sea desplegada.

Seguridad. La arquitectura EJB traslada la mayor parte de la responsabilidad de la seguridad de una aplicación de el desarrollador de aplicaciones al vendedor del servidor, el Administrador de Sistemas y al Desplegador (papeles de la especificación EJB) La gente que ejecuta esos papeles están más cualificados que el desarrollador de aplicaciones para hacer segura la aplicación. Esto lleva a una mejor seguridad en las aplicaciones operacionales.

Tema 6: El Framework Spring

Introducción.

Con este documento se pretende hacer una breve introducción al “Framework Spring”. No se pretende hacer un documento que profundice en todos los aspectos de “Spring”, sólo se desarrollarán aquellos detalles suficientes para comprender la forma de utilizar Spring y poder posteriormente profundizar en detalles más concretos usando la documentación existente en su sitio web.
Toda la documentación de Spring la podemos encontrar en:
http://www.Springframework.org

¿Qué es Spring?

Spring es un framework de aplicaciones Java/J2EE desarrollado usando licencia de OpenSource.
Se basa en una configuración a base de javabeans bastante simple. Es potente en cuanto a la gestión del ciclo de vida de los componentes y fácilmente ampliable. Es interesante el uso de programación orientada a aspectos (IoC). Tiene plantillas que permiten un más fácil uso de Hibernate, iBatis, JDBC..., se integra "de fábrica" con Quartz, Velocity, Freemarker, Struts, Webwork2 y tienen un plugin para eclipse.
Ofrece un ligero contenedor de bean para los objetos de la capa de negocio, DAOs y repositorio de Datasources JDBC y sesiones Hibernate. Mediante un xml definimos el contexto de la aplicación siendo una potente herramienta para manejar objetos Songleton o “factorias” que necesitan su propia configuración.

El objetivo de Spring es no ser intrusito, aquellas aplicaciones configuradas para usar beans mediante Spring no necesitan depender de interfaces o clases de Spring, pero obtienen su configuración a través de las propiedades de sus beans. Este concepto puede ser aplicado a cualquier entorno, desde una aplicación J2EE a un applet.

Como ejemplo podemos pensar en conexiones a base de datos o de persistencia de datos, como Hibernate, la gestión de transacciones genérica de Spring para DAOs es muy interesante.

La meta a conseguir es separar los accesos a datos y los aspectos relacionados con las transacciones, para permitir objetos de la capa de negocio reutilizables que no dependan de ninguna estrategia de acceso a datos o transacciones. Spring ofrece una manera simple de implementar DAOs basados en Hibernate sin necesidad de manejar instancias de sesion de Hibernate o participar en transacciones. No necesita bloques “try-catch”, innecesario para el chequeo de transacciones. Podríamos conseguir un método de acceso simple a Hibernate con una sola línea.

¿Que proporciona?

- Una potente gestión de configuración basada en JavaBeans, aplicando los principios de Inversión de Control (IoC). Esto hace que la configuración de aplicaciones sea rápida y sencilla. Ya no es necesario tener singletons ni ficheros de configuración, una aproximación consistente y elegante. Estas definiciones de beans se realizan en lo que se llama el contexto de aplicación.

- Una capa genérica de abstracción para la gestión de transacciones, permitiendo gestores de transacción añadibles (pluggables), y haciendo sencilla la demarcación de transacciones sin tratarlas a bajo nivel. Se incluyen estrategias genéricas para JTA y un único JDBC DataSource. En contraste con el JTA simple o EJB CMT, el soporte de transacciones de Spring no está atado a entornos J2EE.

- Una capa de abstracción JDBC que ofrece una significativa jerarquía de excepciones (evitando la necesidad de obtener de SQLException los códigos que cada gestor de base de datos asigna a los errores), simplifica el manejo de errores, y reduce considerablemente la cantidad de código necesario.

- Integración con Hibernate, JDO e iBatis SQL Maps en términos de soporte a implementaciones DAO y estrategias con transacciones. Especial soporte a Hibernate añadiendo convenientes características de IoC, y solucionando muchos de los comunes problemas de integración de Hibernate. Todo ello cumpliendo con las transacciones genéricas de Spring y la jerarquía de excepciones DAO.

- Funcionalidad AOP, totalmente integrada en la gestión de configuración de Spring. Se puede aplicar AOP a cualquier objeto gestionado por Spring, añadiendo aspectos como gestión de transacciones declarativa. Con Spring se puede tener gestión de transacciones declarativa sin EJB, incluso sin JTA, si se utiliza una única base de datos en un contenedor Web sin soporte JTA.

- Un framework MVC (Model-View-Controller), construido sobre el núcleo de Spring. Este framework es altamente configurable vía interfaces y permite el uso de múltiples tecnologías para la capa vista como pueden ser JSP, Velocity, Tiles, iText o POI. De cualquier manera una capa modelo realizada con Spring puede ser fácilmente utilizada con una capa web basada en cualquier otro framework MVC, como Struts, WebWork o Tapestry.

Toda esta funcionalidad puede usarse en cualquier servidor J2EE, y la mayoría de ella ni siquiera requiere su uso. El objetivo central de Spring es permitir que objetos de negocio y de acceso a datos sean reutilizables, no atados a servicios J2EE específicos. Estos objetos pueden ser reutilizados tanto en entornos J2EE (Web o EJB), aplicaciones “standalone”, entornos de pruebas, etc.… sin ningún problema. La arquitectura en capas de Spring ofrece mucha de flexibilidad. Toda la funcionalidad está construida sobre los niveles inferiores. Por ejemplo se puede utilizar la gestión de configuración basada en JavaBeans sin utilizar el framework MVC o el soporte AOP.

¿Qué es Ioc?

Spring se basa en IoC. IoC es lo que nosotros conocemos como El Principio de Inversión de Dependencia, Inversion of Control" (IoC) o patrón Hollywood ("No nos llames, nosotros le llamaremos") consiste en:

- Un Contenedor que maneja objetos por ti.
- El contenedor generalmente controla la creación de estos objetos. Por decirlo de alguna manera, el contenedor hace los “new” de las clases java para que no los realices tu.
- El contenedor resuelve dependencias entre los objetos que contiene.

Estos puntos son suficientes y necesarios para poder hablar de una definición básica de IoC. Spring proporciona un contenedor que maneja todo lo que se hace con los objetos del IoC. Debido a la naturaleza del IoC, el contenedor más o menos ha definido el ciclo de vida de los objetos. Y, finalmente, el contenedor resuelve las dependencias entre los servicios que él controla.

Herramientas necesarias.

Para poder realizar los siguientes ejemplos necesitaremos varias librerias. Para facilitar la ejecución recomiendo tener instalado un ide de desarrollo java como eclipse o netBeans para poder navegar por el código con soltura. En el caso concreto de este documento recomiendo usar eclipse, puesto que es el el IDE que usaré para compilar y ejecutar

Para el conjunto de los ejemplos necesitaremos las librerias:

- Spring – en http://www.springframework.org existe un fichero “Springframework-whith-dependences.zip” que contiene todas las clases necesarias para ejecutar todas las herramientas de spring.
- Log4j – En el fichero anterior encontramos el jar necesario.
- Jakarta Common-logging – Lo mismo ocurre con esta librería.
- Hibernate – Podemos encontrarlo en su pagina web.
- Struts – Podremos localizarlo en la pagina de jakarta-struts.
- JUnit – Podemos encontrar las clases en el fichero de spring-withdependences. En caso contrario las librerias las podemos encontrar en su sitio web. Si se usa eclipse como ide lo incluye.

Primer ejemplo de uso

Hasta ahora solo hemos visto la teoría de que es el framework Spring. Ahora vamos a realizar un ejemplo sencillo, muy básico para simular el uso de la capa de configuración de beans, el núcleo básico de Spring, para poder posteriormente ir añadiendo funcionalidades a la aplicación.
En este ejemplo no necesitaremos base de datos, simularemos los accesos a base de datos con una clase que devuelva unos datos constantes. Después intentaremos sustituir esta clase de datos por un acceso a Hibernate, para, posteriormente, incluir transacciones.
Para el desarrollo de este ejemplo usaremos JUNIT para el proceso de test de las clases, se podría usar una clase main(), pero se ha considerado más adecuado introducir JUNIT por las posibilidades de test que ofrece.

Librerias necesarias.

En este ejemplo necesitaremos:

- JUNIT
- SPRING (con todas las librerías que necesita)
- Common-logging
- Log4j

La estructura de directorios

La estructura de nuestro primer ejemplo será bastante simple:
Siendo “src” la carpeta donde irán los fuentes.
Las librerías se cargarán configurando eclipse o el IDE que se use.

Configurando LOG4J

Para nuestro uso personal vamos a configurar log4j para que nos vaya dejando trazas. El fichero será el siguiente:

```
log4j.rootCategory=INFO, Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=(%-35c{2} %-4L) %m%n
log4j.logger.paquete=ALL
```

Con este fichero, que colocaremos en la raiz de nuestra carpeta fuente (src), nos mostrara todas nuestras trazas en la consola.

Programando nuestra clase.

Spring se basa mucho en la programación mediante interfaces, de forma que nosotros crearemos los interfaces y la implementación que los crea. Así que crearemos un interfaz y una clase de modelo de datos. Estas son:

```
/* Clase que representa al usuario */
public class Usuario {
private Integer id;
private String nombre;
/* faltan los get y st correspondientes. */
}
/* Interface de acceso a los datos */
public interface UsuarioDao {
public void saveUsuario (Usuario usuario);
public Usuario findUsuario (Integer id);
public void deleteUsuario (Integer id);
public List listAll ();
}
```

Con estas clases realizamos una primera implementación de acceso a datos. Esta clase almacena los datos en una clase interna de almacenamiento:

```
public class UsuarioDaoStatic implements UsuarioDao {
private static final Log log = LogFactory.getLog(UsuarioDaoStatic.class);
private static HashMap tabla;
public UsuarioDaoStatic ()
{
log.debug("Constructor de la implementacion DAO");
tabla = new HashMap ();
}
public void saveUsuario (Usuario usuario) {
log.debug("Guardamos el usuario "+usuario);
if (usuario != null)
tabla.put(usuario.getId(),usuario);
}
public Usuario findUsuario (Integer id) {
log.debug("Estamos buscando usuario "+id);
return (Usuario) tabla.get(id);
}
public void deleteUsuario (Integer id) {
log.debug ("Borramos el usuario "+ id);
tabla.remove(id);
}
}
```

Esta seria una forma normal de cualquier aplicación que accede a una capa de acceso a datos. Ahora configuraremos Spring para que cada vez que se solicite acceso al interfaz UsuarioDao se haga mediante la implementación que nosotros deseamos.

Configuración de *Spring*.

Para este primer ejemplo, bastante básico, debemos de configurar Spring para que al solicitar el bean UsuarioDao, en este fichero es donde le especificamos la implementación concreta. El fichero lo llamaremos applicationContext.xml y tendrá como contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/springbeans.
dtd">
<beans>
<bean id="usuarioDao" class="paquete.dao.impl1.UsuarioDaoStatic" />
</beans>
```

Clases de test

Ahora vamos a ver como se combinan en un uso normal. Para ello crearemos una clase TestUsuarioDao. El código es el siguiente:

```
public class TestUsuarioDao extends TestCase {
private ClassPathXmlApplicationContext ctx;
private UsuarioDao dao;
private Usuario usuario;
private static final Log log = LogFactory.getLog(TestUsuarioDao.class);
protected void setUp() throws Exception {
        log.debug("SETUP del test");
        String[] paths = {"applicationContext.xml"};
        ctx = new ClassPathXmlApplicationContext(paths);
        dao = (UsuarioDao) ctx.getBean("usuarioDao");
        log.debug("hemos obtenido el objeto que implementa usuarioDao");
}
protected void tearDown() throws Exception {
        usuario = null;
        dao = null;
}
public void testAddFindBorrar () throws Exception {
        usuario = dao.findUsuario(new Integer(1));
        log.debug("-----------> "+usuario);
        // Solo para verificar que hay conexión y no salta excepción
        usuario = new Usuario ();
        usuario.setId(new Integer (1));
        usuario.setNombre("Nombre usuario");
        dao.saveUsuario(usuario);
        assertTrue(usuario != null);
        Usuario usuario2 = dao.findUsuario(new Integer (1));
        log.debug("Recuperado usuario"+usuario2);
        assertTrue(usuario2 != null);
        log.debug ("Comparamos : "+usuario2 + " con : "+usuario);
        assertTrue (usuario2.equals(usuario));
        // recuperamos el mismo usuario
        dao.deleteUsuario(new Integer(1));
        usuario2 = dao.findUsuario(new Integer(1));
        assertNull("El usuario no debe de existir",usuario2);
}
public static void main (String[] args) {
        junit.textui.TestRunner.run(TestUsuarioDao.class);
}
}
```

El test lo podemos realizar usando el interfaz gráfico que proporciona eclipse o directamente desde la linea de comando. En cualquier caso, en la salida de la consola obtenemos:

```
(test.TestUsuarioDao 48) SETUP del test
(impl1.UsuarioDaoStatic 46) Constructor de la implementacion DAO
(test.TestUsuarioDao 52) hemos obtenido el objeto que implementa usuarioDao
(impl1.UsuarioDaoStatic 51) Guardamos el usuario paquete.modelo.Usuario@15212bc
(impl1.UsuarioDaoStatic 57) Estamos buscando usuario 1
(impl1.UsuarioDaoStatic 62) Borramos el usuario 1
(impl1.UsuarioDaoStatic 57) Estamos buscando usuario 1
```

Podemos comprobar como Spring, usando el fichero de configuración que hemos generado, nos carga la implementación que nosotros le hemos pedido.

Programando implementación alternativa.

Para probar el cambio de una implementación sin tener que modificar ni una sola línea de código haremos lo siguiente. Creamos una nueva clase que llamaremos UsuarioDaoOtraImpl que para ahorrar tiempo tendrá el mismo contenido que UsuarioDaoStatic , solo que lo situaremos en otro paquete, de forma que la estructura total de nuestro proyecto quede.

Si ahora modificamos el fichero applicationContext.xml cambiando la linea:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/springbeans.
dtd">
<beans>
<bean id="usuarioDao" class="paquete.dao.impl2.UsuarioDaoOtraImpl" />
</beans>
```

Y volvemos a lanzar el test, comprobamos en las trazas obtenidas que hemos cambiado de implementación cambiando un fichero de configuración. Este aspecto es muy interesante para la aplicación final porque así se centralizan los controles de implementaciones en un fichero y simplificamos el código.

NOTA: Existe un método .refresh() que nos permite recargar el fichero de configuración haciendo una llamada a este metodo. Es decir, que se podria cambiar de implementacion (por ejemplo de acceso con Hibernate a otro tipo de acceso) “en caliente”.

Segundo ejemplo.

El ejemplo anterior es sólo una pequeña muestra de cómo se puede usar Spring, las posibilidades sólo se dejan intuir con este ejemplo. Con el siguiente vamos a intentar que nuestra aplicación de ejemplo anterior, usando Spring se conecte a base de datos mediante Hibernate.

Para poder Realizar este proceso vamos a procurar realizar la menor cantidad de modificaciones en el código anterior, para así apreciar el proceso de integración de Spring, que se anuncia como no intrusivo.

Creacion de base de datos.

Hay que considerar que la creación de una base de datos usando MySql, HSQLDB, Oracle o cualquier otro método se escapa de la finalidad de este documento. Solo comentar que para mi ejemplo concreto usé MySQL.
El script de creación de la tabla es el siguiente.

```
CREATE TABLE `atril`.`USUARIO` (
`id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
`nombre` VARCHAR(45) NOT NULL,
PRIMARY KEY(`id`)
)
TYPE = InnoDB;
```

Configurando Hibernate

Para realizar la configuración de Hibernate, necesitaríamos crear dos documentos de configuración, uno para la configuración y otro para el mapeo de los datos.

Sin embargo Spring se nos anuncia como un método de centralizar la configuración, por lo que no crearemos un fichero de configuración de conexión a Hibernate.
Sí crearemos el fichero de mapeo de clase:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="paquete.modelo.Usuario" table="USUARIO">
<id name="id" column="id" type="integer" unsaved-value="0">
<generator class="assigned" />
</id>
<property name="nombre" column="nombre" type="string" not-null="true"/>
</class>
</hibernate-mapping>
```

Programando clases de negocio.

En la documentación existente de Spring se indica que Spring proporciona integración con Hibernate, JDO y iBATIS para el mantenimiento de recursos, soporte para implementación de clases DAO y estrategias de transacción. Spring esta especialmente integrado con Hibernate proporcionando una serie de características muy prácticas.
Así que seguiremos el ejemplo existente en dicha documentación para integrar Hibernate.
Asi que creamos la siguiente clase:

```
package paquete.dao.hibernate;
import java.util.List;
import org.springframework.orm.hibernate.HibernateTemplate;
import org.springframework.orm.hibernate.support.HibernateDaoSupport;
import paquete.dao.UsuarioDao;
import paquete.modelo.Usuario;
// Extiende de una clase que proporciona los métodos necesarios para acceder a Hibernate
public class UsuarioDaoHibernate extends HibernateDaoSupport implements
UsuarioDao {
public void saveUsuario (Usuario usuario) {
this.logger.debug("Intentamos guardar el usuario "+usuario);
HibernateTemplate temp = getHibernateTemplate();
if (usuario!= null)
{
List listado = temp.find("FROM "+Usuario.class.getName()+" as usuario where usuario.id
="+usuario.getId());
if (listado.isEmpty())
{
this.logger.debug("No contieneo, hacemos un save");
temp.save(usuario);
} else {
this.logger.debug("Contiene, hacemos un update");
temp.update(usuario);
}
}
}
public Usuario findUsuario (Integer id) {
this.logger.debug("Buscamos el usuario "+id);
return (Usuario) getHibernateTemplate()
.get (Usuario.class,id);
}
public void deleteUsuario (Integer id) {
this.logger.debug("Borramos el usuario "+id);
Usuario usu = (Usuario) getHibernateTemplate().load(Usuario.class,id);
getHibernateTemplate().delete(usu);
```

```
}
}
```

Esta será la implementación de acceso a Hibernate, que sustituye a las implementaciones hechas anteriormente. Como se puede comprobar extiende de HibernateDaoSupport, una clase que Spring proporciona para facilitar la integración con Hibernate.

Modificando configuración.

El fichero de configuración debemos de modificarlo para que use la implementación de la clase que hemos escrito anteriormente.
La configuración quedaría como:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.Springframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="usuarioDao" class="paquete.dao.hibernate.UsuarioDaoHibernate">
<property name="sessionFactory">
<ref local="sessionFactory" />
</property>
</bean>
<!--<bean id="usuarioDao" -->
<!-- class="paquete.dao.impl1.UsuarioDaoStatic" /> -->
<!-- class="paquete.dao.impl2.UsuarioDaoOtraImpl" /> -->
<!-- Aqui configuramos hibernate -->
<!-- Conección a base de datos -->
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName"><value>org.gjt.mm.mysql.Driver</value></property>
<property name="url"><value>jdbc:mysql://localhost/atril</value></property>
<property name="username"><value>root</value></property>
<property name="password"><value>root</value></property>
</bean>
<!-- Hibernate SessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
<property name="dataSource"><ref local="myDataSource" /></property>
<!-- Must references all OR mapping files. -->
<property name="mappingResources"><list>
<value>paquete/modelo/Usuario.hbm.xml</value>
</list></property>
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
<prop key="hibernate.connection.pool_size">1</prop>
<prop key="hibernate.show_sql">false</prop>
</props>
</property>
</bean>
</beans>
```

Como se puede apreciar, básicamente se ha añadido al bean “usuarioDao” un parámetro más, “sessionFactory”, que no tenian las implementaciones anteriores. Este nuevo parámetro toma su valor de otro bean que a su vez necesita de otro, “myDataSource”. Usando estos dos beans extras la implementación con Hibernate quedará configurada.

Ejecución

Si volvemos a ejecutar la clase de test de los ejemplos anteriores, el funcionamiento debe de ser el mismo. Podemos comprobarlo consultando la base de datos y revisando las trazas obtenidas.