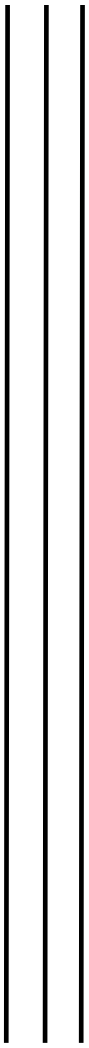
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO



FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

“Frameworks de Persistencia: Análisis comparativo entre Hibernate e iBATIS”

TESIS

que para obtener el título de:

Ingeniero en Computación

presentan:

David Ruiz Coronel

Vanessa García Reyes

Asesor: M. en C. Jesús Hernández Cabrera

2010

AGRADECIMIENTOS.

El presente trabajo es el resultado de un esfuerzo y dedicación constante. No hubiera sido posible sin la ayuda ni el apoyo de todos aquellos que siempre estuvieron ahí de forma incondicional.

Agradezco a Dios por concederme la vida, y por brindarme una infinita cantidad de elementos que han hecho posible mi existencia y mi desarrollo como ser humano.

Agradezco a David Ruiz Cruz - mi papá - y a Elia Coronel Villegas - mi mamá - que siempre me han apoyado en las diversas etapas de mi vida. Sin importar la situación siempre han estado ahí, dispuestos a ayudar y ofrecer un buen consejo.

A mi hermana y hermanos: Janis, Christian y Erick, porque con ellos he crecido y han formado parte importante en mi desarrollo personal y profesional. Los quiero mucho.

A mi novia, amiga y colaboradora Vanessa, con quien he compartido momentos de felicidad y alegría. Sin su amor, apoyo y comprensión hubiera sido imposible llegar hasta aquí. Gracias por todo hermosa, confío en que nuestra relación siga prosperando.

A mis suegros, que siempre han apoyado a Vanessa y a mí, tanto en nuestra relación como en nuestra carrera.

A nuestro asesor de Tesis, el M. en C. Jesús Hernández Cabrera, por la confianza y tiempo que dedicó para la elaboración de este trabajo.

A todos mis amigos, por el simple hecho de brindarme su amistad.

Gracias a todos ustedes, por ayudarme a alcanzar esta meta.

David Ruiz Coronel.

AGRADECIMIENTOS.

Quiero comenzar agradeciendo a dios por brindarme la vida y llenarla de bendiciones día con día. Muestra de ello es mi hermosa familia encabezada por mis padres: Mario García Zaragoza e Irene Margarita Reyes Choreño a quienes agradezco sus enseñanzas, sus cuidados, su cariño e incondicional apoyo. Ustedes son parte fundamental de este trabajo ya que siempre me han alentado a lograr mis metas y me han enseñado a vencer los obstáculos que se interpongan, por esto, éste es también su logro. Los quiero mucho.

De igual forma, quiero agradecer a mis hermanos Mario Guillermo y Edgar pues siempre me han demostrado que puedo contar con ellos. He aprendido mucho de ustedes y quiero que sepan que son un gran ejemplo a seguir.

Otra persona muy importante en la realización de esta tesis es mi novio y compañero David Ruiz Coronel, quien ha sido un apoyo incondicional no solo en éste trabajo sino en muchos otros momentos de mi vida. Gracias por tu compañía, comprensión y amor.

Por otra parte, les agradezco a todos los profesores que colaboraron en mi formación profesional, muy especialmente a nuestro asesor de tesis, el profesor Jesús Hernández Cabrera, quien nos brindo su tiempo, dedicación y experiencia para el desarrollo de este trabajo.

Así mismo expreso un profundo agradecimiento al Ingeniero Rodolfo Zaragoza Buchain por su apoyo otorgado durante este proceso.

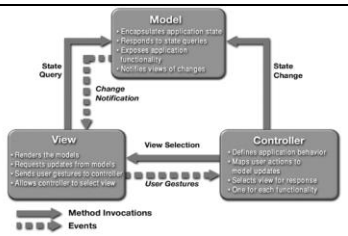
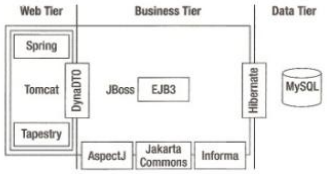
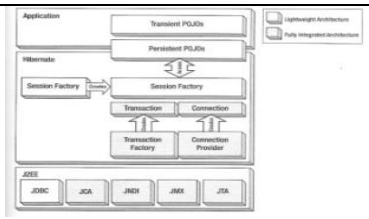
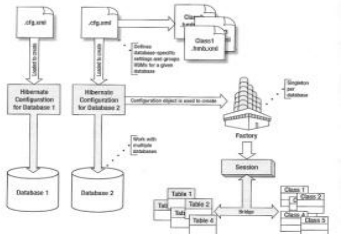
Vanessa García Reyes.

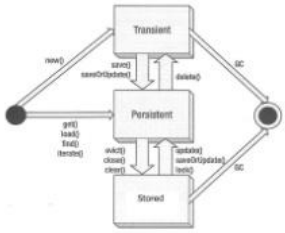
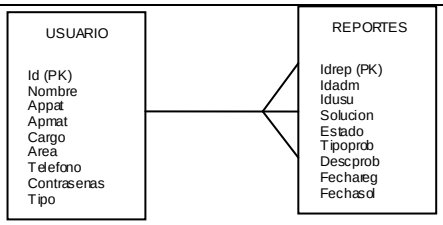
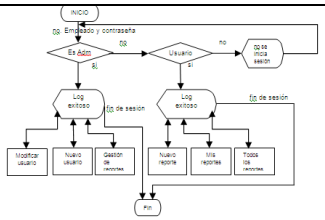
ÍNDICE

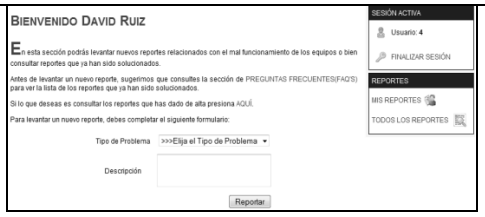
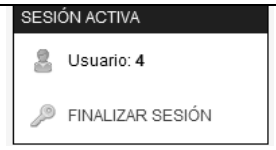
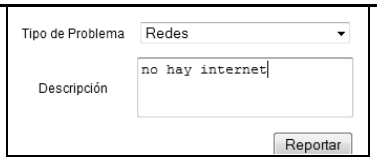
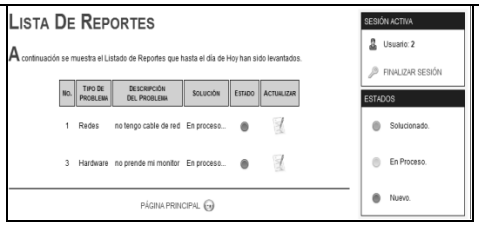
	Página
Índice de Imágenes	
Introducción	
Capítulo 1 Introducción a Frameworks de persistencia.	1
1.1 Método de integración	1
1.1.1 ¿Qué es una aplicación java?	2
1.1.2 Bases de datos	3
1.1.3 Arquitecturas de Software	4
1.1.4 Capas de la aplicación	9
1.2 Capa de persistencia	10
1.3 Herramientas de persistencia	12
1.3.1 Hibernate	12
1.3.2 iBATIS	12
1.3.3 Torque	13
1.3.4 jPersist	13
1.3.5 OJB	13
1.4 Framewoks de estudio	13
Capítulo 2 iBATIS, una herramienta flexible	17
2.1 ¿Qué es iBATIS?	17
2.2 Estructura de iBATIS	17
2.2.1 Capa de abstracción	18
2.2.2 Capa de Framework de persistencia	18
2.2.3 Capa de driver	19
2.3 DAO	20
2.4 Trabajo directo con SQL	25
2.4.1 Archivo SQLMapConfig	26
2.4.2 Archivo SQLMap	28
Capítulo 3 Hibernate, una herramienta ORM	35
3.1 ¿Qué es Hibernate?	35
3.2 Arquitectura (ORM)	35
3.3 Funcionamiento	37

3.4 Archivo de Propiedades 40
3.5 Dialectos SQL 40
3.6 Mapeo de objetos de Hibernate (Hibernate Object Mappings) 45
3.7 HQL 49
Capítulo 4 iBATIS vs Hibernate: Características y ventajas 61
4.1 Características 61
Hibernate 61
iBatis 62
4.2 ¿Para qué sirve? 62
Hibernate 62
iBatis 63
4.3 ¿Cuándo se utiliza? 63
Hibernate 63
iBatis 64
4.4 Ventajas y desventajas 65
Hibernate 65
iBatis 67
4.5 Aspectos generales 69
Hibernate 69
iBatis 69
Capítulo 5 Desarrollo de una aplicación 73
5.1 Nuestra aplicación 74
5.1.1¿Qué es un Help Desk? 74
5.1.2 Diseño de la aplicación 74
5.2 Integrando con iBATIS 80
5.2.1 Instalación y configuración de iBATIS 80
5.3 Integrando con Hibernate 86
5.3.1 Instalación y configuración de Hibernate 86
Conclusiones 97
Bibliografía 99
Referencias de Internet 100

ÍNDICE DE IMÁGENES

Nombre	Imagen	Página
FIGURA 1.1 Arquitectura de Tres Niveles		5
FIGURA 1.2 Esquema general de una aplicación basada en el diseño Modelo-Vista-Controlador		7
FIGURA 1.3 Frameworks y Herramientas en la Arquitectura de Tres Capas para el Desarrollo de una Aplicación.		9
FIGURA 1.4 Funcionalidad de la Capa de Persistencia.		11
FIGURA 3.1 Arquitectura General de Hibernate y su tarea en la capa de persistencia.		36
FIGURA 3.2 Funcionamiento de Hibernate.		37

Nombre	Imagen	Página
FIGURA 3.3 Dialectos soportados por Hibernate En la columna izquierda se muestran los sistemas gestores de bases de datos y en la derecha el dialecto correspondiente a cada gestor.		41
FIGURA 3.4 Ciclo de Persistencia de Hibernate		48
FIGURA 5.1 Diagrama Entidad-Relación		75
FIGURA 5.2 Diagrama de navegación		76
FIGURA 5.3 Página Principal		77
FIGURA 5.4 Administrador.jsp		78

Nombre	Imagen	Página
FIGURA 5.5 Usuario.jsp		79
FIGURA 5.6 Opción “finalizar sesión”		80
FIGURA 5.7 Formulario para dar de alta un reporte.		81 y 91
FIGURA 5.8 Enlaces para consultar los reportes.		82 y 92
FIGURA 5.9 Pantalla que muestra la lista de los reportes a solucionar.		83 y 93
FIGURA 5.10 Formulario para modificar el reporte seleccionado.		84 y 93
FIGURA 5.11 Enlaces dentro de la sesión de un administrador		85 y 94

INTRODUCCIÓN

A lo largo del tiempo, las empresas han tenido la necesidad de automatizar procesos para satisfacer diversos requerimientos utilizando la tecnología de su época, la cual ha ido sufriendo grandes cambios. Debido a esto, se busca una mayor precisión en los procesos, es decir, que sean más efectivos, que se realicen en un menor tiempo y a un menor costo.

Como programadores, nos vemos en la necesidad de trabajar bases de datos en conjunto con la programación orientada a objetos, lo que ocasiona que la mayor parte del tiempo se invierta en la búsqueda de un modelo eficiente entre ambos esquemas.

El objetivo de este estudio es analizar el funcionamiento de dos Frameworks enfocados a la parte de la persistencia de una aplicación para poder determinar, según las circunstancias, cuál es el ideal para cada proyecto.

El primer capítulo nos da a conocer el desarrollo de aplicaciones JAVA que nos permitan manipular información usando bases de datos y la programación orientada a objetos. Aquí se expondrán los diversos paquetes que han sido desarrollados para reducir el arduo proceso de desarrollo de sistemas que requiere el acceso a bases de datos.

Principalmente nuestro estudio estará enfocado a dos de los Frameworks más utilizados para ello: Hibernate e iBATIS.

Cada uno de estos Frameworks tendrá su propio capítulo, donde se explicará más a detalle su funcionamiento y configuración.

El primer Framework que estudiaremos será iBATIS, en el capítulo dos conoceremos más acerca de como asocia objetos de modelado con sentencias SQL o procedimientos almacenados mediante ficheros descriptores XML, simplificando la utilización de bases de datos.

En el tercer capítulo estudiaremos el Framework Hibernate, que como todas las herramientas de su tipo, busca solucionar el problema entre los modelos de datos coexistentes dentro de una aplicación: el modelo orientado a objetos y el modelo relacional.

Al término del estudio por separado de cada uno de los Frameworks, compararemos en el capítulo cuatro, algunos de sus aspectos como su flexibilidad, su curva de aprendizaje y su capacidad para facilitar la creación de aplicaciones java empleando bases de datos,

por medio de la exposición de las principales características y ventajas que ofrece cada uno, así como sus desventajas.

En nuestro quinto y último capítulo se mostrará el uso de las herramientas estudiadas con un caso práctico que consistirá en el desarrollo de un HelpDesk. Con esto se pretende demostrar la funcionalidad de cada herramienta y observar cómo cada una emplea metodologías diferentes con un fin común: facilitar la interacción entre el modelo orientado a objetos y la persistencia de los mismos.

Conocer éstas herramientas nos brinda más posibilidades de crear aplicaciones basadas en los esquemas antes mencionados, simplificando su codificación, ahorrando tiempo y dinero tal y como se aplica dentro del campo profesional, pues cada vez son más las empresas que empiezan a adoptar estas metodologías de desarrollo en sus esquemas de trabajo.

CAPÍTULO 1
INTRODUCCIÓN A FRAMEWORKS DE PERSISTENCIA

CAPÍTULO 1.
INTRODUCCIÓN A FRAMEWORKS DE PERSISTENCIA

En los últimos años las aplicaciones web han sufrido un gran auge gracias en gran parte a Internet. Su facilidad de administración centralizada las hace ideales tanto para su despliegue en internet como en intranets corporativas, además el auge de multitud de *Frameworks* open source hace que su desarrollo sea más sencillo.
Estos Frameworks utilizan el mapeo relacional de objetos (ORM), técnica de programación utilizada para combinar lenguajes de programación orientados a objetos y las bases de datos relacionales.

1.1 Método de integración

En la programación orientada a objetos, las tareas de manejo de datos son implementadas por la manipulación de objetos, los cuales son casi siempre valores no escalares. En cambio, muchos productos populares de base de datos, como los productos SQL DBMS, solamente pueden almacenar y manipular valores escalares como enteros y cadenas, organizados en tablas.

Esto se traduce en que el programador debe convertir los valores de los objetos en grupos de valores simples para almacenarlos en la base de datos (y volverlos a convertir luego de recuperarlos de la base de datos), o solo usar valores escalares simples en el programa. Todo este proceso se realiza por medio del mapeo relacional de objetos.

El problema reside en traducir estos objetos a formas en las cuales puedan ser almacenadas en la base de datos, y los cuáles puedan ser recuperados más tarde fácilmente, mientras se preserven las propiedades de los objetos y sus relaciones; estos objetos se dice entonces que son persistentes.
Los tipos de bases de datos usados mayormente son las bases de datos SQL, cuya aparición precedió al crecimiento de la programación orientada a objetos en los 1990s.

Las bases de datos SQL usan una serie de tablas para organizar datos. Los datos en distintas tablas están asociados a través del uso de restricciones declarativas en lugar de punteros o enlaces explícitos. Los mismos datos que pueden almacenarse en un solo objeto podrían requerir ser almacenados a través de varias tablas.
Una implementación del mapeo relacional de objetos podría necesitar elegir de manera sistemática y predictiva qué tablas usar y generar las sentencias SQL necesarias.
Muchos paquetes han sido desarrollados para reducir el tedioso proceso de desarrollo de sistemas de mapeo relacional de objetos proveyendo librerías de clases que son capaces de realizar mapeos automáticamente. Dada una lista de tablas en la base de datos, y objetos en el programa, ellos pueden automáticamente mapear solicitudes de un sentido a otro.

Desde el punto de vista de un programador, el sistema debe lucir como un almacén de objetos persistentes. Uno puede crear objetos y trabajar normalmente con ellos, los cambios que sufran terminarán siendo reflejados en la base de datos.
Sin embargo, en la práctica no es tan simple. Todos los sistemas ORM tienden a hacerse visibles en varias formas, reduciendo en cierto grado la capacidad de ignorar la base de datos. Peor aún, la capa de traducción puede ser lenta e ineficiente (comparada en términos de las sentencias SQL que escribe), provocando que el programa sea más lento y utilice más memoria que el código "escrito a mano".
Un buen número de sistemas de mapeo objeto-relacional se han desarrollado a lo largo de los años, pero su efectividad en el mercado ha sido diversa.

1.1.1 ¿Qué es una aplicación Java?

Este término se refiere a un programa que puede ser utilizado en todo tipo de sistemas operativos y procesadores ya que esta escrito en el lenguaje de programación Java. Este lenguaje orientado a objetos desarrollado por Sun Microsystems se utiliza para desarrollar aplicaciones multiplataforma, aplicaciones cliente/servidor, aplicaciones distribuidas en redes locales y en Internet.
Una aplicación Java es un programa que puede ser de cualquier naturaleza, ejecutarse sin problemas en computadoras con y sin conexión a Internet y realizar todo tipo de

tareas, desde cálculo científico hasta juegos, pasando por aplicaciones de negocios y oficina.

Hay aplicaciones Java web, llamadas así porque su interfaz de usuario esta alojada en un navegador web. A continuación mencionaremos algunos:

- Un Applet es una aplicación especial que se ejecuta dentro de un navegador al cargar una página HTML desde un servidor web. Un Applet se descarga desde el servidor y no requiere instalación en la computadora donde se encuentra el navegador.
- Un Servlet es una aplicación sin interface gráfica que se ejecuta en un servidor de internet.
- Un JSP (Java Server Pages) es una aplicación que combina HTML con fragmentos de java para producir páginas web dinámicas y es una extensión de los Servlets Java.

Todo este tipo de aplicaciones son consideradas fiables ya que puede controlarse su seguridad frente al acceso a recursos del sistema y es capaz de gestionar permisos y criptografía. También, según Sun, la seguridad frente a virus a través de redes locales e Internet está garantizada. Aunque al igual que ha ocurrido con otras tecnologías y aplicaciones, se han descubierto, y posteriormente subsanado, “agujeros” en la seguridad de Java.

1.1.2 Bases de datos

Hoy en día, muchas de las aplicaciones que se realizan en el lenguaje Java así como en otros lenguajes de programación, dependen de una base de datos.

Las bases de datos son colecciones de información. Hay diferentes tipos, de acuerdo a la variabilidad de los datos que almacenan se dividen en estáticas y dinámicas, las primeras son de solo lectura mientras las segundas se pueden modificar con el tiempo. También se pueden clasificar de acuerdo al manejo que le dan a la información en: Jerárquicas, de red, multidimensionales, orientadas a objetos, lógicas o deductivas y relacionales. Estas

últimas son las más usadas gracias a su fácil comprensión y forma de almacenamiento. En una base de datos relacional los datos se almacenan en tablas, cada tabla esta formada por filas y columnas. Las filas representan registros, conjuntos de datos acerca de individuos o elementos separados; y las columnas representan campos, atributos particulares de un registro. Las tablas deben tener un campo común, es decir, un campo que almacena, en cada una de ellas, la misma información para cada registro y que va a ser el que permita establecer la relación al realizar las consultas. Se utilizan identificadores únicos para cada registro y los datos no deben repetirse.

Existen unos programas denominados sistemas gestores de bases de datos, SGBD, que permiten almacenar y posteriormente acceder a los datos de forma rápida y estructurada. El lenguaje más habitual para construir las consultas a bases de datos relacionales es SQL, Structured Query Language o Lenguaje Estructurado de Consultas, un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales.

1.1.3 Arquitecturas de Software

En los inicios de la informática, la programación representaba esquemas complejos y por ello requería de mucha dedicación, pero con el tiempo se han ido descubriendo y desarrollando formas y guías generales, en base a las cuales se puedan resolver los problemas. A dichas formas se les ha denominado Arquitectura de Software porque indican la estructura, funcionamiento e interacción entre las partes del software.

La Arquitectura de Software establece los fundamentos para que analistas, diseñadores, programadores, trabajen en conjunto y bajo un esquema que les permita alcanzar los objetivos de un sistema. Define, de manera abstracta, los componentes que llevan a cabo alguna tarea, sus interfaces y la comunicación entre ellos.

Entre las arquitecturas más comunes destacan:

- La Arquitectura de Tres Niveles o de Tres Capas.
- La Arquitectura Modelo-Vista-Controlador.

- La Arquitectura Orientada a Servicios.

Arquitectura de Tres Niveles o de Tres Capas.

Es una arquitectura cuyo objetivo primordial es la separación de la lógica de negocios de la lógica de presentación y de la capa de persistencia.

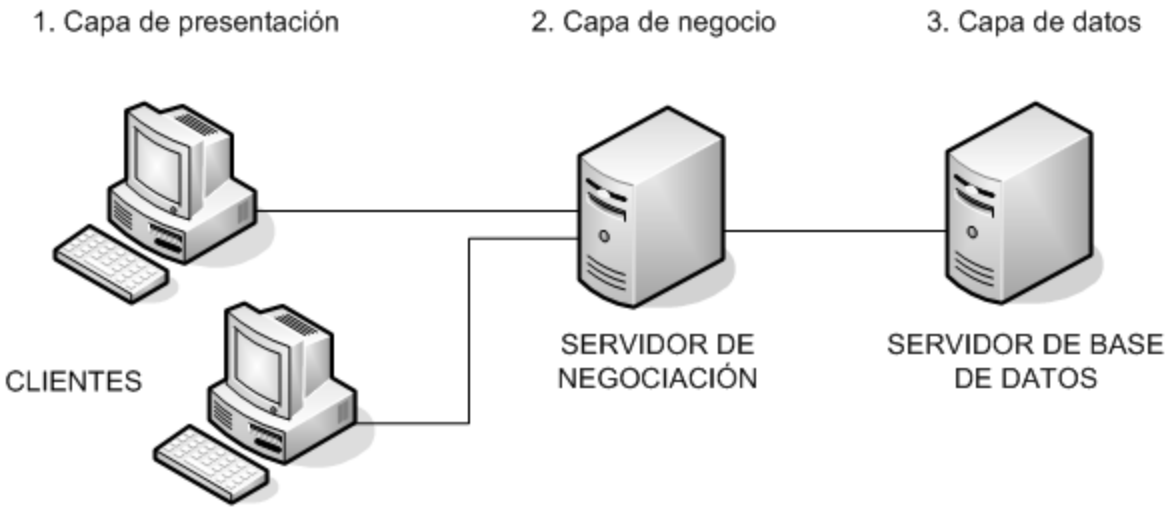


FIGURA 1.1. Arquitectura de Tres Niveles, también conocida como Arquitectura de Tres Capas.

Está conformada por:

1.- Capa de presentación: Es la que ve el usuario y la función de dicha capa es presentar el sistema al usuario. Para esto, transmite información al usuario y captura la información proporcionada por éste para después procesarla. Esta capa se comunica únicamente con la capa de negocio. Básicamente es una interfaz gráfica y debe tener la característica de ser entendible y fácil de usar.

2.- Capa de negocio: Es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Se denomina capa de negocio porque es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los

resultados, y con la capa de datos, para solicitar al gestor de base de datos para almacenar o recuperar datos de él.

3.- Capa de datos: Es donde residen los datos y es la encargada de acceder a los mismos. Está formada por uno o más gestores de bases de datos que realizan todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.

Ventajas.

La ventaja principal de este estilo es que el desarrollo se lleva a cabo en niveles y, en caso de que se necesite de algún cambio, sólo se modificará la aplicación en el nivel requerido sin tener que modificar todo el código ni lidiar contra código entre mezclado. Un buen ejemplo de este método de programación sería el patrón de diseño Modelo-Vista-Controlador.

Arquitectura Modelo/Vista/Controlador

La arquitectura MVC (Modelo/Vista/Controlador) fue diseñada para reducir el esfuerzo de programación necesario en la implementación de sistemas múltiples y sincronizados de los mismos datos. Sus características principales son que el Modelo, las Vistas y los Controladores se tratan como entidades separadas; esto hace que cualquier cambio producido en el Modelo se refleje automáticamente en cada una de las Vistas.

En la figura siguiente (fig. 1.2), vemos la arquitectura MVC en su forma más general.

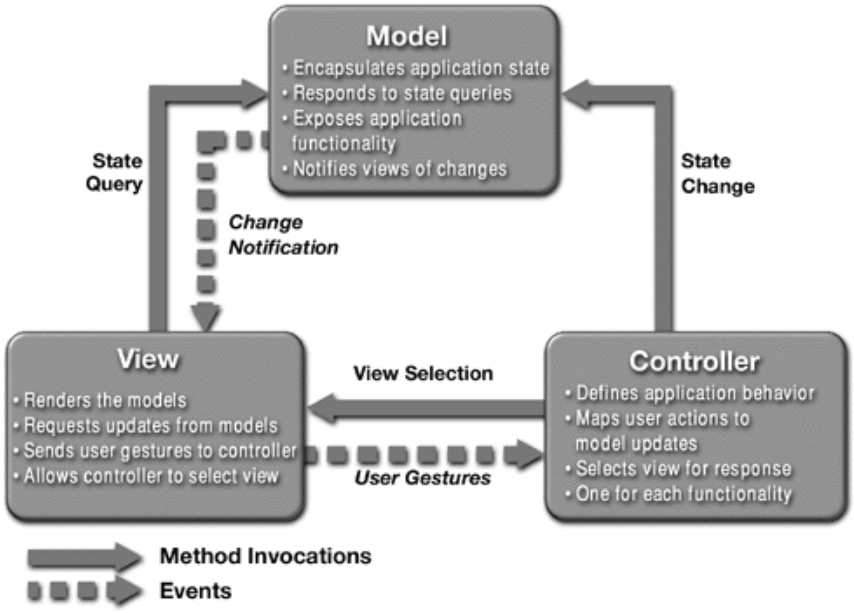


FIGURA 1.2. Esquema general de una aplicación basada en el diseño Modelo-Vista-Controlador

Existen diversos Frameworks que implementan este tipo de arquitectura como Struts, Spring y JavaServerFaces. También hay los que se enfocan a una sola parte, como los Frameworks de persistencia, que se encargan sólo de la parte del Modelo.

Definición de las partes

1.- El Modelo: Es el objeto que representa los datos del programa. Define las reglas de negocio (la funcionalidad del sistema). Maneja los datos y controla todas sus transformaciones. El Modelo no tiene conocimiento específico de los Controladores o de las Vistas, ni siquiera contiene referencias a ellos. Es el propio sistema el que tiene encomendada la responsabilidad de mantener enlaces entre el Modelo y sus Vistas, y notificar a las Vistas cuando cambia el Modelo.

2.- La Vista: Es el objeto que maneja la presentación visual de los datos representados por el Modelo. Es decir, recibe los datos del modelo y los muestra al usuario. Tiene un registro de su controlador asociado (normalmente porque además lo instancia).

Las vistas pueden tener un método de "Actualización()" que sea invocado por el controlador o por el modelo para poder realizar cambios en los datos.

3.- El Controlador: Es el objeto que proporciona significado a las órdenes del usuario (eventos de entrada como un clic, un cambio en un campo de texto, etc.), actuando sobre los datos representados por el Modelo. Cuando se realiza algún cambio, entra en acción, bien sea por cambios en la información del Modelo o por alteraciones de la Vista. Interactúa con el Modelo a través de una referencia al propio Modelo.

Ventajas:

Hay una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado.

Hay un API muy bien definido; cualquiera que use el API, podrá reemplazar el Modelo, la Vista o el Controlador, sin aparente dificultad.

La conexión entre el Modelo y sus Vistas es dinámica; se produce en tiempo de ejecución, no en tiempo de compilación.

Al incorporar el modelo de arquitectura MVC a un diseño, las piezas de un programa se pueden construir por separado y luego unirlas en tiempo de ejecución. Si uno de los Componentes, posteriormente, se observa que funciona mal, puede reemplazarse sin que las otras piezas se vean afectadas.

Arquitectura Orientada a Servicios (SOA).

La Arquitectura Orientada a Servicios (Service Oriented Architecture), es un concepto de arquitectura de software que define la utilización de servicios para dar soporte a las necesidades y requisitos de un negocio.

Permite la creación de sistemas altamente escalables que reflejan el negocio de la organización, a su vez brinda una forma estándar de exposición e invocación de servicios, lo cual facilita la interacción entre diferentes sistemas.

SOA define las siguientes capas de software:

- Aplicaciones básicas. Sistemas desarrollados bajo cualquier arquitectura o tecnología.

- De exposición de funcionalidades. Donde las funcionalidades de la aplicación son expuestas en forma de servicios web.
- De integración de servicios. Facilitan el intercambio de datos entre elementos de la aplicación orientada a procesos empresariales.
- De composición de procesos. Que define el proceso en términos del negocio y sus necesidades, y que varía en función del mismo.
- De entrega. Donde los servicios son desplegados a los usuarios.

La Arquitectura Orientada a Servicios proporciona una metodología y un marco de trabajo para documentar las capacidades de negocio y puede dar soporte a las actividades de integración y consolidación.

1.1.4 Capas de Aplicación.

Como se mencionó anteriormente, una aplicación puede adoptar una arquitectura para facilitar su implementación y diseño, lo cual supone una gran ventaja y ahorro de tiempo. En la siguiente figura (fig. 1.3) y, en particular empleando la arquitectura de tres capas, se muestra un conjunto de Frameworks diseñados para facilitar la implementación de dicha arquitectura.

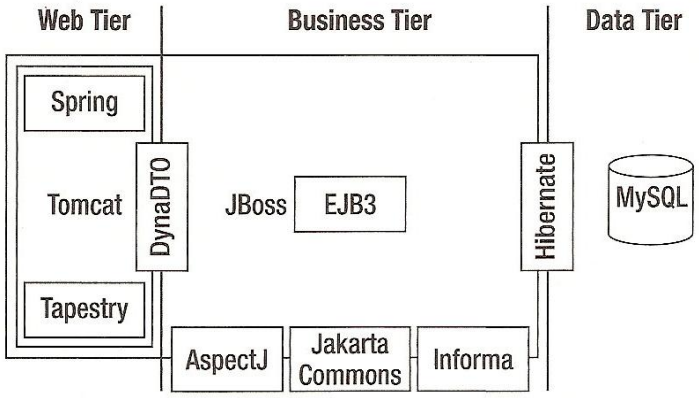


FIGURA 1.3. Frameworks y Herramientas en la Arquitectura de Tres Capas para el Desarrollo de una Aplicación.

Podemos apreciar que en cada capa de la aplicación se muestran algunos de los muchos Frameworks existentes en el mercado. Esto no significa que sean los mejores de su categoría sino simplemente son algunos de los más conocidos.

También, dentro de la misma figura, se puede apreciar uno de los Frameworks objeto de este estudio: Hibernate; y que al igual que iBATIS, está diseñado para trabajar en la capa de persistencia, que se encuentra entre la capa de la lógica del negocio y la capa de datos.

1.2 Capa de persistencia.

En la programación orientada a objetos se entiende por persistencia la capacidad que tienen los objetos de conservar su estado e identidad entre distintas ejecuciones del programa que los creó o de otros programas que accedan a ellos.

De este modo los objetos pueden clasificarse en:

- Transitorios: cuyo tiempo de vida depende directamente del ámbito del proceso que los creó.
- Persistentes: cuyo estado es almacenado en un medio secundario para su posterior reconstrucción y utilización, por lo que su tiempo de vida es independiente del proceso que los creó.

La persistencia permite al programador almacenar, transferir y recuperar el estado de los objetos. Generalmente se utiliza el patrón DAO (Data Access Objects), el cual además oculta la forma de acceder a los datos.

La filosofía de imponer un único formato de datos a toda la aplicación sufre un gran número de inconvenientes. En el caso de que toda la aplicación siga el modelo relacional, se perderían las ventajas del modelo orientado a objetos. En el caso de que toda la aplicación siga el modelo orientado a objetos, implica que las bases de datos pierdan las ventajas del modelo relacional. Por otro lado, si la aplicación sigue la lógica orientada a objetos y la base de datos es relacional, lo que en un principio parece la opción más viable, plantea el problema de cómo conseguir que dos componentes con formatos de datos muy diferentes puedan comunicarse y trabajar conjuntamente.

La solución se basa en encontrar un elemento intermedio que permita comunicar aplicación y base de datos, a este elemento se le conoce como capa de persistencia.

La capa de persistencia traduce entre los dos formatos de datos: de registros a objetos y de objetos a registros. La situación se ejemplifica en la figura 1.4. Cuando el programa quiere guardar un objeto llama a la capa de persistencia, que convierte el objeto a registros y, a su vez, la capa de persistencia llama a la base de datos para que guarde estos registros. De la misma manera, cuando el programa quiere recuperar un objeto, la base de datos recupera los registros correspondientes, los cuales son transformados en objetos por la capa de persistencia y enviados al programa.

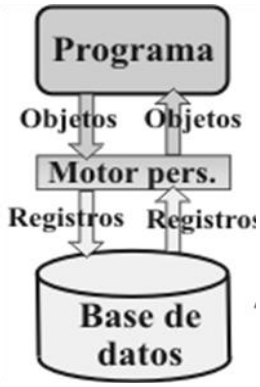


FIGURA 1.4. Funcionalidad de la Capa de Persistencia. El color gris claro representa al modelo relacional mientras que el color gris oscuro representa al modelo Orientado a Objetos.

De esta forma, se consigue que el programa manipule objetos y que la base de datos maneje registros. Es decir, cada componente de la aplicación trabaja con el modelo para el cual ha sido diseñado: el programa es orientado a objetos y la base de datos es relacional.

Entonces, podemos decir que la capa de persistencia es la que actúa como traductor entre ambos modelos, permitiendo que los componentes se comuniquen y trabajen conjuntamente.

1.3 Herramientas de persistencia.

En la actualidad existen diversas herramientas que trabajan en la capa de persistencia de una aplicación. Dichas herramientas buscan solucionar el problema existente entre los dos modelos usados hoy en día para organizar y manipular datos: el modelo orientado a objetos y el modelo relacional.

Ejemplos de estas herramientas son:

- Hibernate.
- iBATIS.
- Torque.
- jPersist.
- OJB.

1.3.1 Hibernate.

Es una herramienta de Mapeo Objeto-Relacional para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional y el modelo orientado a objetos de una aplicación, mediante archivos declarativos que permiten establecer estas relaciones. Permite a la aplicación manipular los datos de la base operando sobre objetos, con todas las características de la Programación Orientada a Objetos. Convierte los datos entre los tipos utilizados por Java y los definidos por SQL. Genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

1.3.2 iBATIS.

Es un Framework de código abierto basado en capas, y que se ocupa de la capa de Persistencia. Puede ser implementado en Java, y .Net. iBATIS asocia objetos de modelo con sentencias SQL o procedimientos almacenados mediante ficheros descriptores XML, simplificando la utilización de bases de datos. Su simplicidad es la mayor ventaja que tiene sobre las herramientas de Mapeo Objeto-Relacional. El uso de iBATIS se basa en objetos propios, SQL y XML.

1.3.3 Torque.

Es una herramienta de Mapeo Objeto-Relacional para java. Permite el acceso y manipulación de datos usando objetos Java. Genera las clases necesarias desde un archivo XML describiendo la estructura de la base de datos, con opción a generar y ejecutar sentencias SQL desde dicho archivo.

1.3.4 jPersist.

Es una poderosa herramienta de Mapeo Objeto-Relacional que evita la necesidad de configurar la aplicación, pues el mapeo es automático. Usa Java Data Base Connectivity (JDBC). Usa la información obtenida de la base de datos para realizar el mapeo entre los registros y los objetos Java.

1.3.5 OJB.

Herramienta de Mapeo Objeto-Relacional que permite la interacción entre las bases de datos relacionales y los objetos Java. Está diseñada para ser utilizada desde aplicaciones de cliente hasta arquitecturas multicapa basadas en J2EE, por lo tanto, puede ser empleada en JSP's y Servlets.

1.4 Frameworks de estudio.

Como bien se ha mencionado es necesario trabajar con programación orientada a objetos y utilizar bases de datos relacionales. Resulta obvio que se trata de dos modelos diferentes. El modelo relacional trata con relaciones, tuplas y conjuntos. El modelo orientado a objetos, sin embargo, trata con objetos, sus atributos y relaciones entre ellos.

A pesar de la gran cantidad de herramientas existentes para implementar un sistema que involucre ambos modelos, sólo nos enfocaremos al estudio de dos de las herramientas más utilizadas que ya han sido mencionadas con anterioridad: Hibernate e iBATIS. Dos Frameworks que en un principio pueden mostrar semejanza entre sí pero que, como veremos, son totalmente diferentes.

Hibernate porque es una herramienta de Mapeo Objeto-Relacional de libre distribución. Es de las herramientas más maduras y completas. Actualmente su uso está muy extendido y está siendo desarrollada de forma muy activa. Tiene su propio lenguaje de consultas denominado HQL (Hibernate Query Language) y es a través de éste donde las bases de datos se relacionan con el modelo orientado a objetos.

Por su parte iBATIS, porque es una herramienta de libre distribución que basa su funcionamiento en el mapeo de sentencias SQL lo cual significa que, al contrario de Hibernate, no posee un lenguaje propio de consultas, lo que representa facilidad para entender lo que se hace dentro de la base de datos sin necesidad de aprender un nuevo lenguaje, claro, siempre y cuando se tengan conocimientos previos de qué es y cómo se usa el SQL.

En los siguientes capítulos explicaremos más a fondo los dos Frameworks iniciando con iBATIS.

CAPÍTULO 2
IBATIS,
UNA HERRAMIENTA
FLEXIBLE

CAPÍTULO 2
IBATIS, UNA HERRAMIENTA FLEXIBLE

El Framework iBATIS es de código abierto y está basado en capas. Fue desarrollado por Apache Software Foundation y publicado en el 2001.
Al inicio fue diseñado para Java, sin embargo con el paso del tiempo se ha extendido para soportar otras plataformas incluyendo .Net y Ruby.

2.1 ¿Qué es iBATIS?

iBATIS es un Framework que se ocupa de la capa de persistencia y está constituido por dos Frameworks independientes que generalmente se usan juntos: DAO y SQL Maps. El primero simplifica la implementación, el patrón de diseño Data Access Objects (DAO) oculta la persistencia de objetos en la aplicación y proporciona una API (Application Programming Interface – interfaz de programación de aplicaciones) de acceso a datos al resto de la aplicación. El segundo simplifica la persistencia de objetos en bases de datos relacionales ya que proporciona una forma sencilla de interacción de datos entre los objetos Java y bases de datos relacionales.

iBATIS asocia objetos de modelo (JavaBeans) con sentencias SQL o procedimientos almacenados mediante ficheros descriptores XML (Extensible Markup Language – lenguaje extensible para el análisis de documentos), simplificando la utilización de bases de datos.
iBATIS, no sigue en el sentido estricto los modelos de mapeo relacional de objetos ORM, por lo que puede utilizar modelos de datos existentes o poco normalizados. Tampoco es completamente transparente, es decir, se debe programar el SQL.

2.2 Estructura de iBATIS

Como iBATIS es un Framework que se enfoca en la capa de persistencia, regularmente la divide en 3 capas: La capa de abstracción, la capa de Framework de persistencia y la capa de driver.

2.2.1 Capa de abstracción

La capa de abstracción será la interfaz con la capa de la lógica de negocio, haciendo las veces de "fachada" entre la aplicación y la persistencia. Se implementa de forma general mediante el patrón Data Access Object (DAO), y particularmente en iBATIS se implementa utilizando su Framework DAO (ibatis-dao.jar). Esta capa se configura mediante el archivo dao.xml.

2.2.2 Capa de Framework de persistencia

La capa de Framework de persistencia será la interfaz con el gestor de base de datos ocupándose de la gestión de los datos mediante una API. Normalmente en Java se utiliza JDBC (Java DataBase Connectivity); iBATIS utiliza su Framework SQL Maps (ibatis-sqlmap.jar). Esta capa se configura mediante un archivo XML, sql-map-config.xml. Además cada objeto de modelo, que representa al objeto en la aplicación, se relaciona con un archivo del tipo sqlMap.xml, que contiene sus sentencias SQL. Por ejemplo, un objeto Java Usuario con un objeto XML usuario.xml, es decir, a cada clase le corresponde un archivo XML donde sus atributos serán mapeados hacia los campos de la base de datos.

En el siguiente código se muestra la clase Usuario la cual será empleada en ejemplos posteriores para explicar el funcionamiento del Framework.

```
 1 package fes.model;
 2
 3 public class Usuario {
 4
 5     private int idUsuario;
 6     private String nombre;
 7     private String apellidos;
 8     private String password;
 9
10     public int getIdUsuario() {
11         return idUsuario;
```

```
12      }
13
14      public void setIdUsuario (int id) {
15          this.idUsuario = id;
16      }
17
18      public String getNombre() {
19          return nombre;
20      }
21
22      public void setNombre(String nombre) {
23          this.nombre = nombre;
24      }
25
26      public String getApellidos() {
27          return apellidos;
28      }
29
30      public void setApellidos(String apellidos) {
31          this.apellidos = apellidos;
32      }
33
34      public String getPassword() {
35          return password;
36      }
37
38      public void setPassword(String pass) {
39          this.password = pass;
40      }
41  }
```

2.2.3 Capa de driver

La capa de driver se ocupa de la comunicación con la propia base de datos utilizando un driver específico para la misma.

A continuación se explicarán los Frameworks que constituyen iBATIS: DAO y SQL Maps.

2.3 DAO

iBATIS implementa el patrón DAO, el cual ayuda a centralizar la responsabilidad del acceso a datos en un único punto, y oculta al resto de la aplicación la implementación específica. Se trata de que el software cliente se centre en los datos que necesita y se olvide de cómo se realiza el acceso a los datos o de cual es la fuente de almacenamiento.

El patrón DAO nos ayuda a organizar las tareas de persistencia (guardado, búsqueda, recuperación, borrado, etc.) de los objetos y nos permite definir múltiples implementaciones para un mismo objeto mediante la definición de una interfaz. En el ejemplo que mostraremos, la interfaz UsuarioDao tiene los métodos insertUsuario (Usuario usuario), deleteUsuario (String idUsuario), etc. y se implementará para que se pueda utilizar en SQL Maps.

Si tuviéramos 2 implementaciones, por ejemplo: UsuarioMySqlDao y UsuarioLdapDao donde se implementaran las operaciones para MySql y Ldap respectivamente podríamos configurar cuándo usar una implementación u otra implementación sin necesidad de modificar código mediante iBATIS DAO.

La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (el cual contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

En iBATIS DAO, se extienden clases donde se implementa la interfaz y el comportamiento específico. A continuación se mencionan algunos pasos a seguir:

1. Se crea un archivo *DAO.xml:*

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE daoConfig PUBLIC "-//iBATIS.com//DTD DAO Configuration 2.0//EN"
3 "http://www.ibatis.com/dtd/dao-2.dtd">
```

```
 4         <daoConfig>
 5            <context>
 6               <transactionManager type="SQLMAP">
 7                  <property name="SqlMapConfigResource"
 8                  value="fes/dao/sqlMap/sqlMapConfig.xml" />
 9               </transactionManager>
10               <dao interface="fes.dao.UsuarioDao"
11               implementation="fes.dao.sqlMap.UsuarioSqlMapDao" />
12            </context>
13         </daoConfig>
```

Explicando un poco el código, podemos observar que en la línea 6 se específica el tipo de administrador de transacciones, que en este caso es SQLMAP, el cual se encarga de la gestión de los objetos relacionados con el mapeo.

2. Después se crea una clase de configuración: *DaoConfig*

```
 1 public class DaoConfig {
 2
 3     private static final String DAO_XML = "fes/dao/dao.xml";
 4     private static final DaoManager daoManager;
 5
 6     static {
 7         try {
 8             daoManager = newDaoManager();
 9         } catch (Exception e) {
10             throw new RuntimeException("Descripcion. Causa: " + e, e);
11         }
12     }
13
14     public static DaoManager getDaoManager() {
15         return daoManager;
16     }
17
18     public static DaoManager newDaoManager() {
19         try {
20             Reader reader = Resources.getResourceAsReader(DAO_XML);
21             return DaoManagerBuilder.buildDaoManager(reader, null);
```

```
22          } catch (Exception e) {
23              throw new RuntimeException(
24                      "No se pudo iniciar DaoConfig. Causa" + e, e);
25          }
26      }
27  }
```

En este código se ocupa la clase DaoManager, la cual es responsable de la configuración del Framework DAO (via dao.xml) instanciando las implementaciones de DAO y actuando como una fachada para el resto de la API.

En las líneas 20 y 21 se lee el archivo de configuración. El archivo dao.xml es leído por el método buildDaoManager() de la clase DaoManagerBuilder el cual toma como parámetro al objeto reader de la clase Reader el cual apunta al archivo dao.xml.

En las líneas 10 y 23 podemos ver la sentencia RuntimeException que es el único tipo de excepción que lanza la API de DAO.

3. Crear la interfaz del Dao para Usuarios, UsuariosDao:

En esta clase se declararán los métodos que se implementarán más adelante.

```
 1 public interface UsuarioDao extends Dao {
 2
 3     public int updateUsuario(Usuario usuario);
 4
 5     public int insertUsuario(Usuario usuario);
 6
 7     public int deleteUsuario(String idUsuario);
 8
 9     public Usuario getUsuario(String idUsuario);
10
11     public Usuario getUsuarioValidado(String idUsuario, String password);
12
13     public List getUsuarios(Usuario usuario);
14 }
```

4. Crear su implementación:

```
 1 public class UsuarioSqlMapDao extends SqlMapDaoTemplate implements
 2         fes.dao.UsuarioDao {
 3
 4     public UsuarioSqlMapDao(DaoManager arg0) {
 5         super(arg0);
 6     }
 7
 8     public int updateUsuario(Usuario usuario) {
 9         try {
10             return getSqlMapExecutor().update("updateUsuario", usuario);
11         } catch (SQLException e) {
12             throw new DaoException("Error actualizando usuario. Cause: "+e,e);
13         }
14     }
15
16     public int insertUsuario(Usuario usuario) {
17         try {
18             getSqlMapExecutor().insert("insertUsuario", usuario);
19         } catch (SQLException e) {
20             throw new DaoException("Error insertando usuario. Cause: "+e,e);
21         }
22         return 1;
23     }
24
25     public int deleteUsuario(String idUsuario) {
26         try {
27             return getSqlMapExecutor().delete("deleteUsuario", idUsuario);
28         } catch (SQLException e) {
29             throw new DaoException("Error actualizando usuario. Cause: "+e,e);
30         }
31     }
32
33     public Usuario getUsuario(String idUsuario) {
34         try {
35             Usuario usuario = (Usuario) getSqlMapExecutor().queryForObject(
36                     "getUsuarioById", idUsuario);
```

```
37              return usuario;
38          } catch (SQLException e) {
39              throw new DaoException("Error recuperando usuario. Cause: "+e,e);
40          }
41      }
42
43      public List getUsuariosByExample(Usuario usuario) {
44          try {
45              Usuario usuarioExample = new Usuario();
46              if (usuario.getApellidos() != null) {
47                  usuarioExample.setApellidos("%"
48                  + usuario.getApellidos().toLowerCase() + "%");
49              }
50              if (usuario.getNombre() != null) {
51                  usuarioExample.setNombre("%"
52                  + usuario.getNombre().toLowerCase() + "%");
53              }
54              usuarioExample.setIdUsuario(usuario.getIdUsuario());
55              List usuarios = (List) getSqlMapExecutor().queryForList(
56                  "getUsuariosByExample", usuarioExample);
57              return usuarios;
58          } catch (SQLException e) {
59              throw new DaoException("Error recuperando usuarios. Cause: "+e,e);
60          }
61      }
62
63      public Usuario getUsuarioValidado(String idUsuario, String password) {
64          Usuario usuario = getUsuario(idUsuario);
65          return usuario.getPassword().equals(password) ? usuario : null;
66      }
67  }
```

En la línea 1 de nuestra clase se hereda de SqlMapDaoTemplate la cual es una plantilla que empata con el administrador de transacciones SQLMAP, para que se pueda tener un fácil acceso a los objetos de la aplicación. El método que provee SqlMapDaoTemplate es: getSqlMapExecutor() el cual se utiliza para recuperar los objetos apropiados que se necesitan para interactuar con el Framework de persistencia.

La sentencia de la línea 10 también se puede escribir como:
SqlMapExecutor ejecuta = getSqlMapExecutor(null);
return ejecuta.update(“updateUsuario”, usuario);

En la línea 35 el método de iBATIS queryForObject() devuelve un solo objeto Usuario.
Recibe dos parámetros, el primero es el nombre que le dimos a la consulta SELECT, en este caso *"getUsuarioById"*, en el archivo *Usuario.xml*.
El otro parámetro es la clave o identificador del usuario en la base de datos. iBATIS se encarga de poner este valor en #value# en el archivo *Usuario.xml*.
Después de esto el Framework se encarga de hacer la consulta, hacer un objeto resultado de la clase Usuario y de rellenar todos sus parámetros llamando a los distintos métodos set() de la clase Usuario (el nombre de los métodos set() debe coincidir con los nombres que le pusimos en la consulta por medio del operador AS, esto en el archivo *Usuario.xml*).

Siguiendo con la explicación de nuestro código, en la línea 55 se utiliza el método queryForList() el cual es parecido al explicado antes, pero con la diferencia de que éste se utiliza para generar una lista de objetos resultado. El parámetro *“getUsuariosByExample”* es el nombre de la consulta SELECT. El otro parámetro es usado para suplir los datos de una clausula WHERE en la sentencia SELECT, en este caso es *usuarioExample*.

2.4 Trabajo directo con SQL

Cuando hacemos un análisis y diseño orientado a objetos obtenemos nuestro modelo de clases y también diseñamos el modelo de datos donde se almacena la información. Y siempre nos queda la tarea de conectarnos con una base de datos, crear sentencias (statements), selecciones (selects), actualizaciones (updates), recorrer resultados (resultsets) y establecer atributos de objetos, etcétera para guardar, buscar, recuperar, etcétera, los valores de los atributos de un objeto. iBATIS SQL Maps simplifica esta tarea resumiéndola a la configuración de archivos XML, con SQL ANSI(el estándar del lenguaje SQL) o propietario y funciona con prácticamente cualquier base de datos con driver JDBC.

iBATIS SQL Maps no te limita a la hora de establecer relaciones del tipo tabla-por-clase o multiples-tablas-por-clase o múltiples-clases-por-tabla. Existen muy pocas restricciones dado que dispones de toda la potencia que ofrece el SQL.

Para utilizar iBatis dentro de una aplicación es necesario colocar las librerías ibatis-x.x.x.jar dentro del classpath. Además, es necesario incluir, al menos, tres archivos de configuración:

1. El archivo de configuración de los parámetros de base de datos. El código siguiente es un claro ejemplo:

```
1 driver=com.mysql.jdbc.Driver
2 url=jdbc:mysql://localhost/usuariosbd
3 username=usufes
4 password=icofes
```

2. El archivo SqlMapConfig.xml que contiene la localización de los archivos donde se declaran los Sql Map.
3. Los archivos Sql Maps. Son uno o más archivos donde se declara la configuración de cada Sql Map y el mapeo de una entidad dentro de la aplicación, a menudo representada por una única clase. Desde esas clases se establecen las referencias con los métodos creados en el SqlMap de la entidad.

2.4.1 Archivo SQLMapConfig

Una vez que tengamos las clases y las tablas con las que vamos a trabajar, la mejor manera de empezar con SQL Maps es creando el archivo de configuración. Éste actuará como el archivo maestro para la configuración de nuestra implementación basada en SQL Map.

Este archivo es un XML y dentro de él configuraremos ciertas propiedades, el DataSource JDBC y los mapeos SQL que utilizaremos en nuestra aplicación. Este archivo ofrece un lugar central donde configurar tu DataSource. El Framework puede manejar varias

implementaciones de DataSources, dentro de los que se incluyen iBATIS están SimpleDataSource, Jakarta DBCP (Commons), y cualquier DataSource que se pueda obtener a través de un contexto JNDI (por ejemplo, un DataSource configurado dentro de un servidor de aplicaciones).

A continuación se muestra el código ejemplo *sqlMapConfig.xml* donde se crea la configuración:

```
 1 <?xml version="1.0" encoding="UTF-8" ?>
 2 <!DOCTYPE sqlMapConfig PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
 3         "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
 4
 5 <sqlMapConfig>
 6    <properties resource="properties/SqlMapConfig.properties" />
 7    <settings cacheModelsEnabled="true" enhancementEnabled="true"
 8    lazyLoadingEnabled="true" maxRequests="32" maxSessions="10"
 9    maxTransactions="5" useStatementNamespaces="false" />
10    <transactionManager type="JDBC">
11       <dataSource type="SIMPLE">
12          <property name="JDBC.Driver" value="${driver}" />
13          <property name="JDBC.ConnectionURL" value="${url}" />
14          <property name="JDBC.Username" value="${username}" />
15          <property name="JDBC.Password" value="${password}" />
16       </dataSource>
17    </transactionManager>
18    <sqlMap resource="fes/dao/sqlMap/Usuario.xml" />
19 </sqlMapConfig>
```

En la línea 6 se especifica el archivo de configuración de los parámetros de base de datos, su contenido será usado posteriormente en el código (de la línea 10 a la 17).

En la línea 7 se definen las características que controlan los detalles de configuración de SqlMap, principalmente las de gestión de transacciones.

La línea 18 indica el archivo de mapeo XML usado en SQLMAPS para cargar los mapeos de SQL.

2.4.2 Archivo SQLMap

Ahora que tenemos configurado un DataSource y listo el archivo de configuración central, necesitaremos proporcionar al archivo de SQL Map nuestro código SQL y los mapeos para cada uno de los objetos parámetro y de los objetos resultado (entradas y salidas respectivamente).

Continuando con nuestro ejemplo de arriba, vamos a construir un archivo SQL Map al cual nombraremos: *Usuario.xml*; donde configuraremos las operaciones y sus sentencias SQL:

```
 1 <?xml version="1.0" encoding="UTF-8" ?>
 2 <!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
 3         "http://www.ibatis.com/dtd/sql-map-2.dtd">
 4
 5 <sqlMap namespace="Usuario">
 6
 7 <select id="getUsuarioById" resultClass="fes.model.Usuario">
 8    SELECT rtrim(ID_USUARIO) as idUsuario, rtrim(PASSWORD) as
 9    password, rtrim(NOMBRE) as nombre, rtrim(APELLIDOS) as apellidos
10    FROM usuario WHERE id_usuario = #value#
11 </select>
12
13 <select id="getUsuarios" parameterClass="fes.model.Usuario"
14 resultClass="fes.model.Usuario">
15    SELECT rtrim(ID_USUARIO) as idUsuario, rtrim(PASSWORD) as
16    password, rtrim(NOMBRE) as nombre, rtrim(APELLIDOS) as apellidos
17    FROM usuario
18    <dynamic prepend="WHERE">
19       <isNotNull prepend="AND" property="nombre">
20          lower(NOMBRE) like #nombre#
21       </isNotNull>
22       <isNotNull prepend="AND" property="apellidos">
23          lower(APELLIDOS) like #apellidos#
24       </isNotNull>
25       <isNotNull prepend="AND" property="idUsuario">
```

```
26          ID_USUARIO = #idUsuario#
27       </isNotNull>
28    </dynamic>
29 </select>
30
31 <insert id="insertUsuario" parameterClass="fes.model.Usuario">
32    INSERT INTO usuario (ID_USUARIO, NOMBRE,
33    APELLIDOS, PASSWORD) VALUES (#idUsuario#, #nombre#,
34    #apellidos#, #password#)
35 </insert>
36
37 <update id="updateUsuario" parameterClass="fes.model.Usuario">
38    UPDATE usuario SET PASSWORD = #password#, NOMBRE =
39    #nombre#, APELLIDOS = #apellidos# WHERE ID_USUARIO = #idUsuario#
40 </update>
41
42 <delete id="deleteUsuario" parameterClass="string">
43    DELETE FROM usuario WHERE ID_USUARIO = #value#
44 </delete>
45
46 </sqlMap>
```

En el código anterior se usa una característica del Framework SQL Maps que permite mapear de forma automática columnas de un ResultSet a propiedades de un JavaBean como lo es nuestra clase *Usuario*; esto, haciendo coincidir los nombres de los campos del ResultSet con los de los atributos de nuestra clase.

Los diferentes tags <select>, <update>, <insert> y <delete> especifican las diferentes consultas SQL que queremos hacer.
Dentro de estas sentencias se definen ciertas “variables” entre almohadillas (# #), que se utilizan para la transferencia de parámetros. Estas “variables” casi siempre son las propiedades de nuestra clase *Usuario.*
El símbolo *#value#* es un parámetro de entrada que se sustituye por un valor de tipo primitivo cuando se realiza el mapeo y la consulta.

Desde la clase *UsuarioSqlMapDao* accedemos a estas consultas mediante métodos proporcionados por iBATIS (como *queryForObject()*). Estos métodos admiten como primer parámetro un *String* que es el nombre que identifica a la sentencia SQL en el archivo *Usuario.xml*. Este identificador es el atributo *id="nombre"* obligatorio en todos los tags.
Como segundo parámetro admiten un *Object*. El atributo *parameterClass* en los tags es el nombre generalmente de una clase que pasaremos como parámetro *Object* en el método iBATIS del archivo *UsuarioSqlMapDao*. De este parámetro *Object* iBATIS tratará de obtener los valores para reemplazarlos en las variables entre #.

El atributo *parameterClass* puede tomar diferentes valores:

- Si *parameterClass*="int" o un tipo simple, en *UsuarioSqlMapDao* deberemos pasar un *Integer* o un tipo simple, e iBATIS pondrá directamente ese *Integer* reemplazando a #id#.
- Si *parameterClass*="unaClaseJavaBean", en *UsuarioSqlMapDao* pasaremos esa clase como parámetro e *Ibatis* tratará de obtener el id llamando al método getId().
- Si *parameterClass*="unHashTable" en *UsuarioSqlMapDao* pasaremos un Hashtable como parámetro e *Ibatis* tratará de obtener el id llamando a get("id").

El atributo *resultClass* indica qué clase devolverán los métodos iBATIS de la clase *UsuarioSqlMapDao* en el return y es obligatorio cuando la sentencia SQL tiene que devolver algo. iBATIS, igual que con *parameterClass*, instanciará una o más clases del tipo indicado en *resultClass* y tratará de rellenar dentro los datos. Por ejemplo en la línea 14 de nuestro anterior código tenemos *resultClass*="fes.model.Usuario", iBATIS hará un objeto de la clase Usuario y tratará de rellenar los valores llamando a todos sus métodos *set().*

Con todos estos archivos terminamos con la configuración. Ahora mostraremos ejemplos de diferentes consultas.

INSERT

```
 1 public class Prueba {
 2
 3     public static void main(String[] args) {
 4         Usuario usu = new Usuario();
 5         usu.setIdUsuario(5);
 6         usu.setNombre("laura");
 7         usu.setApellidos("avila mendez");
 8         usu.setPassword("lauAvimen");
 9         UsuarioSqlMapDao ejemplo = new UsuarioSqlMapDao();
10         ejemplo.insertUsuario(usu);
11         }
12 }
```

UPDATE

```
 1 public class Prueba {
 2
 3     public static void main(String[] args) {
 4         Integer clave = new Integer(5);
 5         UsuarioSqlMapDao ejemplo = new UsuarioSqlMapDao();
 6         Usuario usu= (Usuario)ejemplo. getUsuario(clave);
 7         usu.setPassword("lauavimen");
 8         ejemplo.updateUsuario(usu);
 9     }
10 }
```

DELETE

```
 1 public class Prueba {
 2
 3     public static void main(String[] args) {
 4         Integer clave = new Integer(5);
 5         UsuarioSqlMapDao ejemplo = new UsuarioSqlMapDao();
```

```
6         ejemplo.deleteUsuario(clave);
7     }
8 }
```

SELECT

```
1 public class Prueba {
2
3     public static void main(String[] args) {
4         Usuario usu = new Usuario();
5         UsuarioSqlMapDao ejemplo = new UsuarioSqlMapDao();
6         usu.setNombre("laura");
7         ejemplo.getUsuarios(usu);
8 }
```

Ya que conocemos más acerca de iBATIS, podemos dar paso al otro Framework de estudio, Hibernate.

En el siguiente capítulo explicaremos sus configuraciones, funcionamiento y además las funciones básicas de SQL sobre una base de datos tal y como lo hicimos con iBATIS.

CAPÍTULO 3
HIBERNATE, UNA HERRAMIENTA ORM

CAPÍTULO 3
HIBERNATE, UNA HERRAMIENTA ORM

3.1 ¿Qué es Hibernate?

Hibernate es una Framework de Mapeo Objeto Relacional (ORM) de código abierto cuyo objetivo es permitir a una aplicación manipular una base de datos operando sobre objetos, con todas las características de la Programación Orientada a Objetos. Convierte los datos entre los tipos utilizados por Java y los definidos por SQL. Además genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.

A diferencia de iBatis, este Framework ofrece un lenguaje de consulta de datos llamado HQL (*Hibernate Query Language*). Al final del capítulo se expondrá a mayor detalle este lenguaje.

3.2 Arquitectura (ORM)

Como se mencionó con anterioridad, Hibernate es una herramienta ORM.

ORM o Mapeo Objeto Relacional es el nombre dado a las tecnologías, herramientas y técnicas empleadas para, de cierta forma, minimizar las diferencias entre objetos y bases de datos relacionales. Las herramientas de Mapeo Objeto Relacional permiten a los programadores trabajar con información de una base de datos relacional en forma de objetos (POO). Además, éstas herramientas generan todo el SQL necesario para interactuar con una base de datos y, de esta forma, los conceptos de consultas y transacciones trabajan a un nivel de objeto en lugar de trabajar a un nivel de base de datos.

Internamente Hibernate utiliza JDBC, con lo cual proporciona una capa de abstracción hacia la base de datos cuando éste emplea JTA y JNDI para integrarse con otras aplicaciones.

Hibernate está conformado por gran variedad de interfaces, pero las principales son la interfaz de Sesión (Session) y la Interfaz de Transacción (Transaction), que junto con la interfaz de Consultas (Query), se encuentran dentro de la capa de persistencia. Las clases definidas en la Capa de Negocio de la aplicación interactúan de forma independiente con la Capa de Persistencia de Hibernate la cual, a su vez, se comunica con la Capa de Datos usando ciertos API s de JDBC. Adicionalmente Hibernate usa interfaces de configuración y también hace uso de otras interfaces para extender su funcionalidad con respecto al mapeo. En la siguiente figura se expone de manera general la arquitectura de Hibernate:

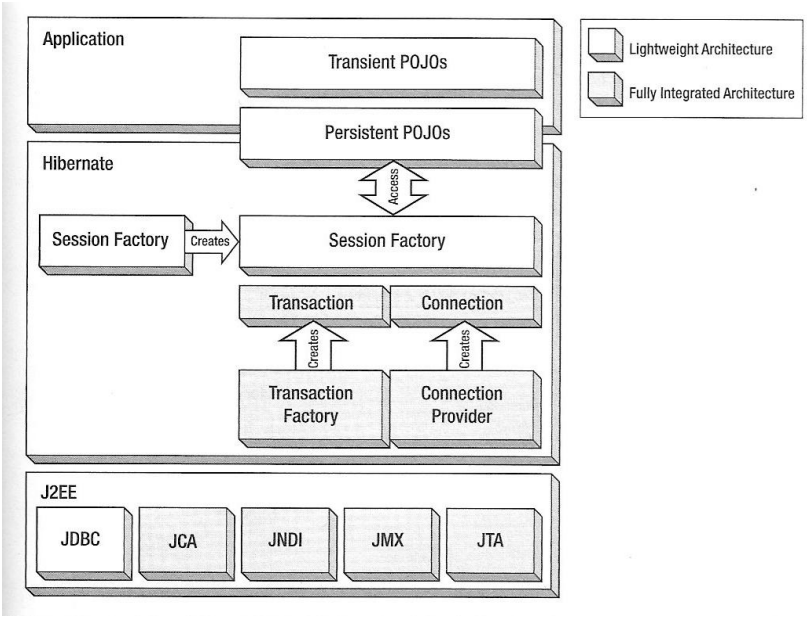


FIGURA 3.1 Arquitectura General de Hibernate y su tarea en la capa de persistencia

3.3 Funcionamiento

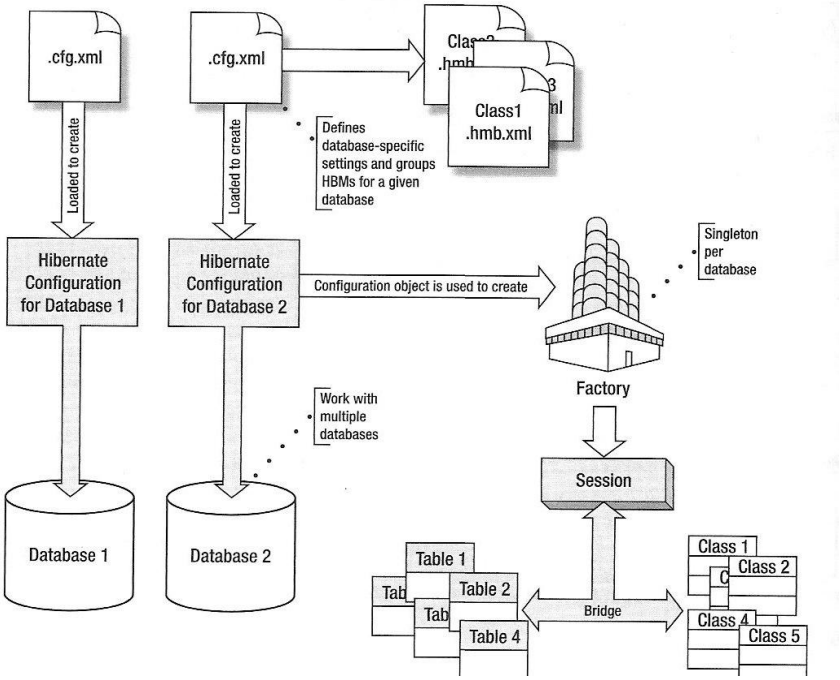


FIGURA 3.2 Funcionamiento de Hibernate.

La Sesión Hibernate (Hibernate Session).

La sesión Hibernate incorpora el concepto de persistencia (servicio de persistencia o administrador de persistencia) que puede emplearse para realizar una consulta y efectuar las operaciones de insertar (insert), actualizar (update) y borrar (delete) sobre instancias de una clase mapeada por Hibernate. Dentro de una herramienta ORM se realizan estas acciones empleando el modelo orientado a objetos, es decir, ya no se hace referencia a tablas y columnas de una base de datos, en cambio, se usan Clases Java y propiedades de objetos. Como su nombre indica, la Sesión (Session) es un objeto usado como conexión entre una aplicación y una base de datos.

La Fábrica de Sesión (Session Factory)

Hibernate requiere que se le proporcione la información necesaria para conectarse a la base de datos que está siendo empleada por una aplicación, así como también la información relacionada con las clases mapeadas a una base de datos dada.

La información correspondiente a cada base de datos, junto con las asociaciones de clases-tablas, es compilada y almacenada por SessionFactory, para después ser recuperada por las Sesiones de Hibernate (Hibernate Sessions).

La Fábrica de Sesión (Session Factory) es un objeto que idealmente debe crearse una sola vez dentro de una aplicación, pues es un objeto que requiere de bastante tiempo de ejecución; y debe estar disponible para el código que necesite efectuar operaciones de persistencia.

Cada Fábrica de Sesión (Session Factory) se configura para trabajar con ciertas plataformas o distribuciones de bases de datos (MySQL, PostgreSQL, etcétera) usando uno de los dialectos proporcionados por Hibernate. La elección del dialecto es un aspecto importante cuando se desean emplear ciertas características específicas como esquemas de generación de llaves primarias (Primary Keys - PK) o cierre de Sesión (Session locking). Una lista de los dialectos soportados por Hibernate se muestra en el siguiente tema.

Para mostrar el funcionamiento de Hibernate tomaremos el ejemplo expuesto en el capítulo de IBatis para así mostrar las diferencias entre ellos.

Nuestra clase será la siguiente:

```
1 package usuario.beans;
2
3 public class Usuario {
4
5     private int id;
6     private String nombre;
7     private String apellido;
```

```
 8     private String password;
 9
10     public int getId() {
11         return id;
12     }
13     public void setId(int id) {
14         this.id = id;
15     }
16     public String getPassword() {
17         return password;
18     }
19     public void setPassword(String password) {
20         this.password = password;
21     }
22
23     public String getNombre() {
24         return nombre;
25     }
26     public void setNombre(String nombre) {
27         this.nombre = nombre;
28     }
29     public String getApellido() {
30         return apellido;
31     }
32     public void setApellido(String apellido) {
33         this.apellido = apellido;
34     }
35
36     public String toString(){
37
38         return "Nombre: " + getNombre() +
39         "\nApellido: " + getApellido() +
40         "\nPassword: " + getPassword();
41     }
42 }
43
44
```

Una vez creada la clase, continuaremos con la configuraciones correspondientes.

3.4 Archivo de propiedades

Configuración de Hibernate.

Antes de crear una Fábrica de Sesión (Session Factory), se debe indicar a Hibernate dónde encontrar los archivos de mapeo que relacionan una clase Java con su respectiva tabla en la base de datos. Hibernate también emplea un conjunto de ajustes de configuración que normalmente se encuentra dentro de un archivo de propiedades Java llamado hibernate.properties o dentro de un archivo XML denominado hibernate.cfg.xml.

Cualquiera de los archivos mencionados anteriormente puede ser usado para configurar Hibernate, sin embargo, el más recomendado es el archivo XML, pues también ofrece soporte para crear los archivos de mapeo de manera rápida y sencilla.

Los archivos de configuración deben encontrarse en el directorio raíz de la aplicación (classpath).

3.5 Dialectos SQL.

JDBC simplifica muchos de los detalles de conexión para cada base de datos relacional, pero las distribuciones de bases de datos (SQL Server, My SQL, etcétera) se diferencian entre sí, pues cada una tiene sus características propias y el SQL puede ser ligeramente diferente entre ellas.

Hibernate encapsula todas esas diferencias en los denominados Dialectos SQL y así, a cada distribución de base de datos soportada por Hibernate le corresponde un Dialecto SQL. De esta manera, cuando Hibernate requiere hacer una consulta, usa el dialecto correspondiente a la base de datos empleada. Todos los dialectos de Hibernate se encuentran dentro del paquete org.hibernate.dialect.

Hibernate soporta actualmente los siguientes dialectos:

DB2/390	DB2390Dialect
DB2/400	DB2400Dialect
DB2	DB2Dialect
Derby	DerbyDialect
Firebird	FirebirdDialect
FrontBase	FrontBaseDialect
HSQLDB	HSQLDialect
Informix	InformixDialect
Ingres	IngresDialect
Interbase	InterbaseDialect
Mckoi	MckoiDialect
MySQL	MySQLDialect
MySQL with InnoDB tables	MySQLInnoDBDialect
MySQL with MyISAM tables	MySQLMyISAMDialect
Oracle 9	Oracle9Dialect
Oracle	OracleDialect
PointBase	PointbaseDialect
PostgreSQL	PostgreSQLDialect
Progress	ProgressDialect
SAP DB	SAPDBDialect
SQL Server	SQLServerDialect
Sybase 11	Sybase11Dialect
Sybase Anywhere	SybaseAnywhereDialect
Sybase	SybaseDialect

Figura 3.3 Dialectos soportados por Hibernate. En la columna izquierda se muestran los sistemas gestores de bases de datos y en la derecha el dialecto correspondiente a cada gestor.

Hibernate traduce el HQL en una sentencia SQL propia de una distribución usando el dialecto SQL correspondiente a la misma. Es decir, si la distribución de la base de datos es, por ejemplo, MySQL se hará uso del dialecto de MySQL para que de esta forma, el HQL sea transformado en SQL nativo de MySQL. Como se mencionó con anterioridad, cada distribución de bases de datos es ligeramente diferente de otras y el lenguaje SQL nativo puede variar entre ellas, es por eso que en Hibernate se usan los Dialectos SQL.

Propiedades de Hibernate.

Normalmente, la configuración de Hibernate se especifica dentro de un archivo de propiedades, ya sea hibernate.properties o hibernate.cfg.xml.

Dentro del archivo de propiedades se configura la conexión JDBC a ser usada por la aplicación, o bien, se especifica el nombre del contenedor JNDI proporcionado por una

conexión JDBC. También, dentro de éste archivo, se especifica el Dialecto SQL correspondiente que Hibernate debe usar para generar el código SQL.

Hibernate posee una extensa lista de propiedades de configuración, sin embargo, sólo se hará uso de las propiedades necesarias para nuestra aplicación y qué describiremos más adelante.

Configuración XML.

Hibernate ofrece la posibilidad de manejar un archivo de propiedades XML. La ventaja de usar éste archivo es que nos brinda la posibilidad de configurar los archivos de mapeo de Hibernate. La estructuración y sintaxis del archivo XML necesita apegarse al formato descrito por Hibernate, que hasta el momento de redacción corresponde a la Configuración DTD de Hibernate 3.

El siguiente fragmento de código es un archivo de propiedades XML para configurar nuestro ejemplo:

```
 1 <?xml version='1.0' encoding='utf-8'?>
 2 <!DOCTYPE hibernate-configuration PUBLIC
 3         "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 4         "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
 5
 6 <hibernate-configuration>
 7
 8  <session-factory>
 9
10  <!-- Configuración para la conexión a la base de datos -->
11  <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
12  <property name="connection.url">jdbc:mysql://localhost/datos</property>
13  <property name="connection.username">root</property>
14  <property name="connection.password">icofesa</property>
15
16  <!-- Dialecto SQL -->
17  <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
18
19  <!-- Propiedad para mostrar en la consola el SQL usado por Hibernate -->
```

```
20  <property name="show_sql">true</property>
21
22  <!-- Propiedad que indica el archivo de mapeo a utilizar-->
23  <mapping resource="usuario/beans/Usuario.hbm.xml"/>
24
25  </session-factory>
26 </hibernate-configuration>
```

En el archivo de configuración existen diferentes etiquetas que nos permitirán configurar nuestra aplicación.

En la línea 11 se muestra la etiqueta encargada de especificarle a Hibernate el manejador correspondiente a la base de datos que se emplea:

```
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
```

El atributo `name` indica la propiedad a configurar y lo que se encuentra entre las etiquetas `<property>` es el nombre del manejador de la base de datos, que en nuestro caso corresponde a MySQL.

En la etiqueta de la línea 12 se especifica la dirección URL donde se encuentra la tabla de la base de datos a usar:

```
<property name="connection.url">jdbc:mysql://localhost/datos</property>
```

En esta etiqueta se especifica el driver, el nombre del servidor, que en nuestro caso es `localhost`; y la base de datos que vamos a usar (`datos`).

En la etiqueta de la línea 13 se declara el nombre de usuario para ingresar a la base de datos:

```
<property name="connection.username">root</property>
```

En nuestro caso, el nombre de usuario es `root`, sin embargo, el nombre puede corresponder a otro usuario que posea privilegios de administrador dentro de la base de datos. ROOT es el usuario default con tales privilegios para MySQL.

En la etiqueta de la línea 14 se especifica la contraseña para ingresar a la base de datos:

```
<property name="connection.password">drako</property>
```

Aquí se debe declarar la contraseña correspondiente al usuario que está ingresando a la base de datos.

En la línea 17 se encuentra la propiedad donde se declara el dialecto de Hibernate a usar:

```
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
```

Como el sistema gestor que estamos empleando es MySQL, entonces dentro de esta propiedad declaramos el dialecto correspondiente a MySQL.

La propiedad declarada en la línea 20 es opcional, a diferencia de las demás que son requeridas por Hibernate:

```
<property name="show_sql">true</property>
```

La función de esta etiqueta es indicarle a Hibernate que muestre en consola el SQL empleado para ingresar a la base de datos. En este caso le indicamos que lo muestre con el valor `true`.

Por último, en la línea 23 se indica el archivo de mapeo que usará nuestra aplicación:

```
<mapping resource="usuario/beans/Usuario.hbm.xml"/>
```

Como bien se puede apreciar, el archivo de mapeo a usar se especifica en el atributo `resource` de la etiqueta `<mapping>`. Este archivo de mapeo se describe en la siguiente sección.

3.6 Mapeo de objetos de Hibernate (Hibernate Object Mappings)

Hibernate determina la forma de cómo cada objeto es recuperado y almacenado en la base de datos a través de un archivo de configuración XML. Los mapeos de Hibernate se cargan al inicio y son guardados en la Fábrica de Sesión (Session Factory). Cada mapeo especifica la cantidad de parámetros relacionados con el ciclo de persistencia de instancias de clase, tales como:

- Mapeo de Llave Primaria y Esquemas de generación.
- Mapeo Objeto-Campo-a-Tabla-Columna.
- Asociaciones / Colecciones.
- Configuración y / o ajustes de Almacenamiento.
- SQL personalizado como Procedimientos Almacenados (Store Procedures), Filtros, etcétera.

Básicamente la función de un archivo de mapeo es determinar la relación entre una clase Java y una Tabla, también determina la relación entre los atributos de una clase y los campos de dicha tabla. El siguiente fragmento de código muestra el archivo de mapeo usado por nuestro ejemplo:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!DOCTYPE hibernate-mapping PUBLIC
 3 "-//Hibernate/Hibernate Mapping DTD//EN"
 4 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 5
 6 <hibernate-mapping>
 7     <class name="usuario.beans.Usuario" table="Usuario">
 8         <id name="id" type="integer" column="ID">
 9             <generator class="increment"/>
10         </id>
```

```
11
12          <property name="nombre" column="nombre"
13          type="string"
14          not-null="true"/>
15
16          <property name="apellido" column="apellido"
17          type="string"
18          not-null="true" />
19
20          <property name="password" column="password"
21          type="string"
22          not-null="true" />
23
24      </class>
25
26 </hibernate-mapping>
```

El elemento `<hibernate-mapping>` (línea 6) permite especificar mediante atributos, diversas características y relaciones entre nuestra aplicación y la base de datos.

```
<hibernate-mapping>
.
.
.
</hibernate-mapping>
```

En lo que al elemento `<class>` (línea 7) respecta, es aquí donde declaramos la relación existente entre nuestra clase y la tabla de la base de datos:

```
<class name="usuario.beans.Usuario" table="Usuario">
.
.
.
</class>
```

En el atributo `name` se escribe el nombre de la clase y en el atributo `table` la tabla con la cual está relacionada. Por lo tanto, estamos relacionando nuestra clase `usuario.beans.Usuario` con la tabla `Usuario`.

En la línea 8, se declara el elemento que será considerado como llave primaria:

```
<id name="id" type="integer" column="ID">
     <generator class="increment"/>
</id>
```

En el atributo `name` se declara el nombre de nuestra variable de Java, en el atributo `type` se indica el tipo de dato que es y en el atributo `column` se indica el nombre del campo con el cual está relacionada nuestra variable declarada en `name`. En nuestro caso estamos relacionando nuestra variable `id` con el campo `ID`. No necesariamente los nombres tanto de las variables como de los campos deben coincidir, pues nuestra variable puede llamarse `identificador` y nuestro campo de la base de datos `id` y no causaría inconveniente alguno.

Dentro del elemento `<id>` se encuentra a su vez el subelemento `<generator>` (línea 9). Este elemento es el encargado de definir la forma de generar valores para el id. Existen diferentes tipos de `<generator>` los cuales los cuales se especifican mediante el atributo `class`. A continuación se describen brevemente 3 de los más conocidos:

- Nativo (`native`). Asigna valores en función de las características del gestor de la base de datos.
- Asignado (`assigned`). Con esto, Hibernate permite que nuestra aplicación sea la encargada de asignar un id a un objeto antes de guardarlo.
- Incremento (`increment`). Este método genera identificadores de tipo `short`, `long` e `int` que son únicos solo cuando no hay otro proceso que éste insertando datos en la misma tabla.

Como bien se aprecia en nuestro ejemplo, en la línea 9 en el atributo `class` declaramos `increment`, para que de esta forma sea Hibernate quien asigne los id a los elementos de nuestra aplicación.

En la línea 12, a través de la etiqueta `<property>` se declara otro de los elementos de nuestra aplicación y sus respectivas relaciones entre ellos:

```
<property name="nombre" column="nombre" type="string" not-null="true"/>
```

En el atributo `name` se declara el nombre de nuestra variable Java, en el atributo `column` se define el campo con el que estará relacionada nuestra variable, en el atributo `type` se especifica el tipo de dato que se manipula entre estos elementos y el atributo `not-null` indica si el campo de la base de datos debe permitir elementos de tipo nulo. En este caso se relaciona la variable Java `nombre` (`name="nombre"`) con el campo de la base de datos `nombre` (`column="nombre"`), el tipo de dato que manejan es `string` (`type="string"`) y se indica que no debe aceptar valores nulos con `true` (`not-null="true"`).

Lo anterior es equivalente para las etiquetas de las líneas 16 y 20 para la relación de los elementos `apellido` y `password` respectivamente.

Ciclo de Persistencia.

Existen tres posibles estados en los que un objeto de mapeo puede encontrarse. Entendiendo estos estados y las acciones que causan transiciones entre ellos es un aspecto sumamente importante cuando se tratan problemas complejos con Hibernate.

En la siguiente figura se muestran los posibles estados de un objeto y las acciones que sobre él actúan:

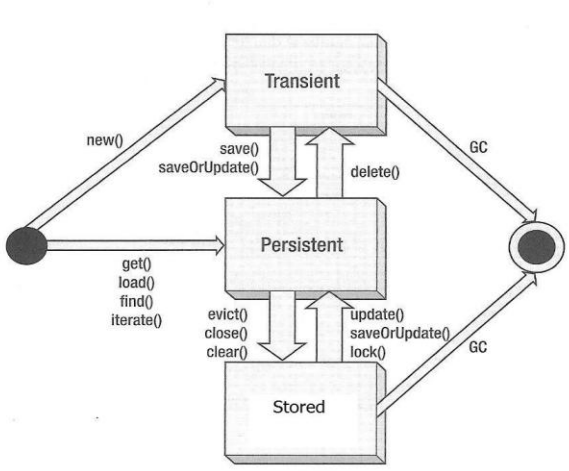


FIGURA 3.4 Ciclo de Persistencia de Hibernate

En el estado transitorio (Transient) el objeto no está relacionado con alguna base de datos. Esto quiere decir que el objeto no ha sido almacenado, y por lo tanto, no tiene asociado un identificador. Los objetos de este tipo son no transaccionales, lo cual significa que no son considerados en los procesos de la Fábrica de Sesión (Session Factory). Después de una invocación exitosa de métodos como guardar (save()) o guardar y/o actualizar (saveOrUpdate()) un objeto deja de ser transitorio y se convierte en persistente. El método borrar (delete()) produce el efecto contrario, es decir, ocasiona que un objeto deje de ser persistente y se convierta en transitorio.

Los objetos persistentes (Persistent) son objetos con un identificador de base de datos; si a los objetos se les ha asignado un identificador pero aún no han sido guardados se dice que se encuentran en el estado de 'nuevo'. Los objetos persistentes son transaccionales, lo cual significa que pueden participar en procesos relacionados con una Sesión (Session).

Un objeto persistente que ya no se encuentra dentro de una Sesión (Session), se convierte en un objeto almacenado (Stored). Esto ocurre cuando, después de una operación exitosa, la Sesión (Session) se cierra, limpia o finaliza. Los objetos almacenados pueden convertirse en objetos de transferencia.

3.7 HQL.

HQL es un lenguaje de consultas orientado a objetos, semejante a SQL en cuanto a sintaxis, solo que en vez de estar trabajando con tablas y columnas, HQL trabaja con objetos persistentes y sus propiedades. HQL es efectivo al estar trabajando con características relacionadas a la POO tales como herencia y asociaciones.

Las consultas en HQL son insensibles a mayúsculas y minúsculas a excepción de nombres de clases Java y sus propiedades. Las consultas en HQL regresan resultados como objetos que pueden ser manipulados por un programador. HQL es un poderoso lenguaje diseñado para usarse con Hibernate y además es fácil de usar.

A continuación se muestra un pequeño ejemplo para mostrar las diferencias y semejanzas existentes entre SQL y HQL.

En SQL una sentencia para mostrar a todos los usuarios cuyo apellido sea ‘Pérez’ luciría como:

```
SELECT nombre, apellido FROM usuario WHERE apellido = “Pérez”;
```

Como bien se puede apreciar, estamos haciendo referencia a las tablas y columnas de nuestra base de datos. La sentencia se leería como: “MUESTRA el nombre y el apellido DE la tabla usuario DONDE el campo apellido sea igual a ‘Pérez’”.

En HQL el equivalente al código anterior es:

```
FROM persona.beans.Persona as Personas WHERE Personas.apellido = “Pérez”
```

En este caso, estamos haciendo referencia a los objetos de nuestra clase Java, y el código se leería como: “DE la clase persona.beans.Persona muestra como Personas a los objetos DONDE la propiedad apellido de Personas tenga como valor ‘Pérez’”.

Para mayor detalle de HQL se puede consultar la documentación de Hibernate.

Acciones SQL básicas.

Una vez que se han realizado las configuraciones necesarias para hacer uso de Hibernate dentro de una aplicación, se pueden realizar las operaciones que normalmente se harían utilizando SQL. En nuestro caso, solo se describirán las operaciones básicas que son:

- Insertar o Guardar registros (INSERT).
- Mostrar o Seleccionar registros (SELECT).
- Actualizar o Modificar registros (UPDATE).
- Borrar o Eliminar registros (DELETE).

Para ello, crearemos una clase, la cual, será la encargada de inicializar los parámetros necesarios para conectarse a una base de datos y se expone a continuación:

```
 1 package persona.util;
 2
 3 import org.hibernate.SessionFactory;
 4 import org.hibernate.cfg.Configuration;
 5
 6 public class HibernateUtil {
 7
 8     private static final SessionFactory sessionFactory;
 9
10     static {
11         try{
12             sessionFactory = new
Configuration().configure().buildSessionFactory();
13         }
14         catch(Throwable ex){
15             System.err.println("Error al crear Session Factory: "+ex);
16             throw new ExceptionInInitializerError(ex);
17         }
18     }
19
20     public static SessionFactory getSessionFactory() {
21         return sessionFactory;
22     }
23
24 }
```

La función del código anterior es crear una instancia de Fábrica de Sesión (Session Factory), la cual, como se describió anteriormente, es necesaria para realizar las transacciones necesarias entre nuestra aplicación y la base de datos.

En la línea 8 se declara el nombre de la variable a la cual asignaremos dicha instancia. Dentro del bloque try – catch que abarca desde la línea 11 hasta la 18, en la línea 12 se realiza la asignación correspondiente a nuestra variable y en la línea 14 se maneja una excepción producto de algún error ocurrido durante la asignación del valor a nuestra variable. Por último, la funcionalidad del método getSessionFactory() de la línea 20 no es más que devolver la variable de la línea 8.

Una vez completado el paso anterior, expondremos los elementos necesarios para efectuar cada una de las operaciones SQL básicas antes mencionadas.

El siguiente código muestra una clase denominada AccionesSQL, la cual contiene los métodos necesarios para realizar dichas operaciones:

```
 1 package persona.acciones;
 2 import java.util.List;
 3 import org.hibernate.Session;
 4 import org.hibernate.Transaction;
 5 import persona.beans.Persona;
 6 import persona.util.HibernateUtil;
 7
 8 public class AccionesSQL {
 9
10     public void insertar(Persona persona){
11    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
12         session.beginTransaction();
13         session.save(persona);
14         session.getTransaction().commit();
15     }
16
17     public void mostrar(){
18    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
19         session.beginTransaction();
20         String consulta = "FROM persona.beans.Persona as Personas";
21   List<Persona> list = (List<Persona>)session.createQuery(consulta).list();
22         List<Persona> personas = list;
23         for(int i=0;i<=(personas.size()-1);i++){
24            Persona persona = (Persona)personas.get(i);
25            System.out.println(persona.getId() + " " + persona.getNombre()+
26            " " + persona.getApellido() + " " + persona.getTelefono()+ " " +
27            persona.getEdad());
28         }
29         session.close();
30     }
31
32     public void eliminar(Persona persona){
33         Transaction tx = null;
```

```
34          persona.toString();
35      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
36          try{
37              tx = session.beginTransaction();
38              session.delete(persona);
39              tx.commit();
40          }
41          catch(Exception e){
42              System.out.println("Error al intentar borrar registros");
43              e.printStackTrace();
44          }
45      }
46
47       public void actualizar(Persona persona){
48          Transaction tx = null;
49          persona.toString();
50      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
51          try{
52              tx = session.beginTransaction();
53              session.update(persona);
54              tx.commit();
55          }
56          catch(Exception e){
57              System.out.println("Error al intentar actualizar el registro");
58              e.printStackTrace();
59          }
60      }
61  }
```

Y procederemos a detallar cada uno de los métodos de la clase anterior.

Guardar o Insertar – INSERT

El método encargado de realizar el INSERT es el método insertar ():

```
13      public void insertar(Persona persona){
14
15          Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
```

```
16          session.beginTransaction();
17          session.save(persona);
18          session.getTransaction().commit();
19          //session.close();
20      }
```

Este método recibe como parámetro un objeto del tipo Persona. Para poder guardar el objeto que se recibe es necesario crear una sesión (Session). Luego, invocamos el método beginTransaction () para indicar que inicia la interacción entre la base de datos y nuestra aplicación, y con el método save () es con el que se guarda el objeto como un registro de la base de datos. El método save () es el equivalente a realizar la consulta SQL "INSERT INTO nombre Tabla (campo, campo1) VALUES (valor, valor1)".

Mostrar o Seleccionar – SELECT.

El método correspondiente a esta operación es el siguiente:

```
23      public void mostrar(){
24          Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
25          session.beginTransaction();
26          String consulta = "FROM persona.beans.Persona as Personas";
27          List<Persona> list =
(List<Persona>)session.createQuery(consulta).list();
28          List<Persona> personas = list;
29          for(int i=0;i<=(personas.size()-1);i++){
30              Persona persona = (Persona)personas.get(i);
31              System.out.println(persona.getId() + " " +
32                      persona.getNombre() + " " +
33                      persona.getApellido() + " " +
34                      persona.getTelefono()+ " " +
35                      persona.getEdad());
36          }
37          session.close();
38      }
```

En este método, no se recibe parámetro alguno y al igual que el método mostrar (), se crea una sesión para poder interactuar con la base de datos (línea 24). Luego, se crea

una variable denominada 'consulta' que contiene una cadena HQL (línea 26). Se crea una Lista que contendrá objetos del tipo Persona, cuyos valores se obtienen al invocar el método createQuery () con la variable 'consulta' como argumento y aplicando después el método list () (línea 27). En el ciclo FOR (de la línea 29 a la 36) se recorre la lista y se va mostrando cada uno de los elementos que contiene dicha lista (línea 31). Por último, finalizamos la sesión que se mantuvo activa durante las transacciones (línea 37).

Actualizar o Modificar – UPDATE.

El método que corresponde a esta acción es actualizar () y se muestra a continuación:

```
56     public void actualizar(Persona persona){
57         Transaction tx = null;
58         persona.toString();
59         Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
60         try
61         {
62             tx = session.beginTransaction();
63             session.update(persona);
64             tx.commit();
65         }
66         catch(Exception e){
67             System.out.println("Error al intentar actualizar el registro");
68             e.printStackTrace();
69         }
70
71     }
```

Este método recibe un objeto de tipo Persona. Luego se crea una variable de tipo Transaction (línea 57). Luego, con la sentencia de la línea 58 solo se muestra el objeto que se está recibiendo. Nuevamente se crea una sesión (línea 59) y dentro del bloque try – catch (de la línea 60 a la 69) se inicia la transacción (línea 62), y es con el método update () con el que indicamos que el registro se actualizará (línea 63).
El método update () es el equivalente a la sentencia SQL "UPDATE tabla SET campo = valor WHERE campo1 = valor 1".

Borrar o Eliminar – DELETE.

El método funcional para eliminar registros de la base de datos es el método eliminar ():

```
40     public void eliminar(Persona persona){
41         Transaction tx = null;
42         persona.toString();
43         Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
44         try{
45             tx = session.beginTransaction();
46             session.delete(persona);
47             tx.commit();
48         }
49         catch(Exception e){
50             System.out.println("Error al intentar borrar registros");
51             e.printStackTrace();
52         }
53
54     }
```

La lógica y sintaxis es idéntica a la del método actualizar (), salvo en la línea 46, donde se hace uso del método delete () cuya función es la de eliminar un registro de la base de datos, al igual que la sentencia SQL "DELETE FROM tabla WHERE campo = valor".

Funcionamiento de la aplicación.

Por último, necesitamos verificar que la aplicación funcione correctamente, y para ello, creamos una clase de 'pruebas' dentro de la cual estará contenido el 'main' y en particular, es denominada 'PruebaEjecucion' y se muestra a continuación:

```
 1 package persona.pruebas;
 2
 3 import persona.acciones.AccionesSQL;
 4 import persona.beans.Persona;
 5 import persona.util.HibernateUtil;
 6
```

```
 7 public class PruebaEjecucion {
 8
 9     public static void main(String[] args) {
10         AccionesSQL accion = new AccionesSQL();
11
12         Persona persona = new Persona();
13         System.out.println("INICIO");
14         // Creando una nueva persona
15         persona.setNombre("Rotan");
16         persona.setApellido("Rebih");
17         persona.setPassword("Hibernate1234");
18         // Mostrando datos de la bd antes de guardar a la nueva persona
19         accion.mostrar();
20         // Mostrando los datos de la persona
21         System.out.println("DATOS PROPORCIONADOS");
22         System.out.println(persona.toString());
23         // Guardar en la bd a la persona
24         accion.insertar(persona);
25         // Mostrar datos de la bd
26         accion.mostrar();
27         // Actualizar a persona
28         persona.setPassword("Hib1234");
29         accion.actualizar(persona);
30         // Mostrar datos actualizados
31         accion.mostrar();
32         // Eliminar a la persona de la bd
33         accion.eliminar(persona);
34         // Mostrar nuevamente los datos
35         accion.mostrar();
36         HibernateUtil.getSessionFactory().close();
37         System.out.println("FIN");
38     }
39
40 }
```

Básicamente, la intención de esta clase es la de mostrar el funcionamiento de la aplicación en su totalidad. Se hace uso de las operaciones básicas y conforme se va realizando cada operación se van mostrando los registros de la base de datos para visualizar de mejor forma cada cambio que sufre dicha base de datos.

Para ello se crea un objeto del tipo AccionesSQL, con el fin de emplear los métodos de la clase (línea 10). Luego, creamos un objeto del tipo Persona (línea 12) para asignarle valores a sus atributos (línea 15, 16 y 17) y de esta forma poder interactuar con la base de datos haciendo uso de los métodos declarados en AccionesSQL. Cuando nuestra aplicación funcione correctamente, es decir, sin errores de compilación y que realice las operaciones indicadas, habremos creado exitosamente una aplicación básica con Hibernate.

CAPÍTULO 4

IBATIS VS HIBERNATE: CARACTERÍSTICAS Y VENTAJAS

CAPÍTULO 4.
IBATIS VS HIBERNATE: CARACTERÍSTICAS Y VENTAJAS

4.1 Características

Hibernate.

Entre las principales características técnicas que aporta Hibernate están las siguientes:

- Hibernate es una herramienta de persistencia mapeo objeto/relacional que permite diseñar objetos persistentes que podrán incluir entre sus características elementos tales como: polimorfismo, relaciones, colecciones, y un gran número de tipos de datos.
- Hibernate en su funcionamiento genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todas las bases de datos con un ligero incremento en el tiempo de ejecución.
- Hibernate distingue entre objetos transitorios y persistentes: los primeros son objetos que sólo existen en memoria y no en un almacén de datos, en algunos casos no serán almacenados jamás en la base de datos y en otros es un estado en el que se encuentran hasta ser almacenados en ella. Los segundos se caracterizan por haber sido ya almacenados y ser por tanto objetos persistentes. Dicho de otra manera los objetos transitorios han sido instanciados por el desarrollador sin haberlos almacenado mediante una sesión, los objetos persistentes han sido creados y almacenados en una sesión o bien devueltos en una consulta realizada con la sesión.
- En Hibernate, al igual que con las conexiones JDBC, se deben crear y cerrar sesiones, aunque no hay una relación 1:1 entre sesiones y conexiones, es decir, no se tiene que abrir y cerrar simultáneamente sesiones y conexiones JDBC, la política a seguir dependerá del contexto del proceso de negocio de cada situación, permitiendo así a Hibernate brindar amplias posibilidades para la implementación de nuestras políticas como: conexiones JDBC gestionadas por la aplicación, por

Hibernate, por un posible servidor de aplicaciones, etcétera; siendo solamente necesario en la práctica crear y cerrar explícitamente las sesiones de Hibernate.

iBATIS.

iBATIS asocia objetos de modelo (JavaBeans) con sentencias SQL o procedimientos almacenados mediante ficheros descriptores XML, simplificando la utilización de bases de datos.

Este Framework basa su funcionamiento en el mapeo de sentencias SQL incluyéndolas en archivos XML.

Permite la optimización de las consultas SQL, con el lenguaje propietario del motor de base de datos utilizado.

Con iBATIS, siempre se sabe lo que se está ejecutando en la base de datos, además tiene herramientas para generar consultas dinámicas muy potentes.

Requiere conocimiento de SQL por parte del programador.

Subdivide la capa de la persistencia en tres capas: abstracción, de framework y de driver.

iBATIS implementa el patrón DAO. Este patrón ayuda a centralizar la responsabilidad del acceso a datos en un único punto, y oculta al resto de la aplicación la implementación específica. Gracias a este patrón, una posible migración del Sistema Gestor de Base de Datos sólo afectaría a la capa de persistencia, resultando transparente para el resto de capas.

4.2 ¿Para qué sirve?

Hibernate

Hibernate es un Framework que simplifica el acceso a base de datos, para ello permite establecer una correspondencia entre el modelo de la Base de Datos Relacional y una

serie de clases que modelan los objetos de la aplicación. Con este Framework disponemos de un sistema de acceso a bases de datos relacionales, y para ello únicamente se deben escribir sentencias java, y de manera independiente, el código escrito con Hibernate funcionará en cualquier motor de datos al que se dé soporte, permitiendo de esta manera manipular objetos de aplicaciones Java en forma de registros de tablas de sistemas de bases de datos relacionales usando metadatos que describen la relación entre los objetos y la base de datos.

iBATIS.

El patrón DAO que se emplea con iBATIS nos ayuda a organizar las tareas de persistencia (guardado, búsqueda, recuperación, borrado, etc.) de los objetos y nos permite definir múltiples implementaciones para un mismo objeto mediante la definición de una interfaz. Con iBATIS DAO podremos configurar cuándo usar una u otra implementación sin necesidad de modificar código.

Cuando hacemos un análisis y diseño orientado a objetos obtenemos nuestro modelo de clases y también diseñamos el modelo de datos donde se almacena la información. Luego nos queda la tarea de conectarnos con una base de datos, crear consultas, actualizaciones, recorrer colecciones, asignar valores de atributos de objetos para guardar, buscar, recuperar, etc. los valores de los atributos de un objeto. iBATIS SQLMap simplifica esta tarea resumiéndola a la configuración de ficheros XML, con SQL ANSI o propietario y funciona con prácticamente cualquier base de datos con driver JDBC.

4.3 ¿Cuándo se utiliza?

Hibernate.

Hibernate es la herramienta ORM más destacada. Proporciona una solución bastante completa al relacionar bases de datos con clases Java y cuenta con la posibilidad de crear consultas SQL propias. Hibernate es la solución ideal para aquellas situaciones donde se tiene completo control sobre toda la aplicación, es decir, control tanto sobre las clases como sobre las tablas de la base de datos y, de esta forma, adaptar nuestra

aplicación para que encaje a la perfección, lo cual significa que podemos realizar adecuaciones en ambas partes de nuestra aplicación. Hibernate es recomendado para crear aplicaciones donde sea necesario integrar el modelo orientado a objetos con el modelo relacional de las bases de datos. Hibernate también puede ser una opción para aquellas personas que no deseen escribir código SQL o que no estén tan familiarizados con el mismo, pues Hibernate posee su propio lenguaje de consultas llamado HQL y además tiene la posibilidad de generar SQL de forma automática, sin que el programador escriba sentencia SQL alguna.

iBATIS

Cuando el modelo de datos es muy cambiante o es preexistente al desarrollo de la aplicación (y compartido con otras), iBATIS es un claro caso de uso. También lo es cuando las relaciones entre las entidades del modelo son muy complicadas, porque con algo de trabajo se puede conseguir que el número de consultas que se pasan a la base de datos no sea excesivo, sobre todo en los listados descriptivos.

Además, iBATIS posee un conjunto de características que hacen de él una opción bastante viable:

- Tiene una curva de aprendizaje rápida y se necesitan pocos conocimientos previos, y no requiere aprender un lenguaje de consultas como en Hibernate, por lo tanto se puede obtener un alto rendimiento en poco tiempo, es decir optimización del proyecto.
- Se necesita manipular el SQL (para utilizar SQL propietario, optimizarlo, etc.) o se necesita llamar a procedimientos almacenados. Por la tanto, requiere de conocimientos sobre SQL.

iBATIS puede emplearse cuando el modelo de datos existe previamente y no está muy normalizado (aunque lógicamente se puede utilizar con modelos nuevos y normalizados), también es una solución ideal cuando se requiere un completo control sobre SQL.

Para el caso de iBATIS DAO, éste es válido cuándo:

- Se sabe que el motor de la base de datos puede cambiar en el futuro
- La implementación del acceso a datos puede requerir cambios sustanciales en el futuro
- La implementación del acceso a datos puede variar entre un entorno de producción y otro (software comercial en distintos clientes)

4.4. Ventajas y Desventajas.

Hibernate

Hay un conjunto de factores que hacen de Hibernate una potente herramienta:

Productividad: Usado conjuntamente con otras herramientas reducirán significativamente el tiempo de desarrollo.

Utilizar un Framework de ORM simplifica enormemente la programación de lógica de persistencia. En aplicaciones donde la lógica de negocios trabaja contra un modelo de dominio completamente orientado a objetos la generación de código se reduce en entre un 30% y un 40%.

Mantenimiento: Al tener menos líneas de código hace que el sistema sea más comprensible y nos centramos en la lógica de negocio en lugar de en la conexión. Un sistema con menos código es más fácil de depurar.

Rendimiento: La mayoría de las optimizaciones son más fáciles disponiendo de un ORM así algunos aspectos como consultas son más fáciles de optimizar que con código manejado vía SQL/JDBC

Independencia: Si la herramienta soporta un número distinto de base de datos, esto proporciona un cierto nivel de portabilidad. Hibernate tiene soporte para la mayoría de los gestores de bases de datos.

Flexibilidad: Hibernate puede integrarse en una arquitectura J2EE. Su flexibilidad permite configurarlo no sólo con JDBC, sino también con JNDI, JTA y JMX.

Es tan flexible su configuración que es válida para su uso como aplicación independiente o usada dentro de un servidor de aplicaciones.

Versatilidad en el acceso a BD relacionales: Hibernate, sobre cualquier otro sistema de acceso a base de datos relacionales, ofrece distintas formas de hacer la misma cosa. Por ejemplo, las consultas se pueden hacer de cuatro formas:

- HQL. Hibernate Query Language. Es un lenguaje de acceso a base de datos en donde las consultas se hacen a los objetos, no a las tablas. No es complicado de aprender y ofrece una gran potencia y simplifica los accesos. Este lenguaje es exclusivo de Hibernate.
- QBE. Query by Example. Simplemente, crea un objeto, rellena un campo con datos, y se crea una sentencia SQL con ese criterio.
- QBC. Query by Criteria. Parecido al anterior, pero para criterios no sólo de igualdad. Con código se crean comparaciones, sentencias tipo LIKE. Ideal para buscadores. La cantidad de tiempo y código que ahorra es digna de mención.
- SQL. En Hibernate es posible ejecutar sentencias SQL, por si alguna vez se tuviera la imperiosa necesidad de hacerlo. En este sentido, podríamos trabajar con Hibernate como si fuera iBATIS.

Lenguaje de Consultas. En el caso de decidirse por HQL, o SQL, para acceder a los datos, no es necesario escribir las sentencias en el código, podemos materializarlas en los ficheros de configuración, darles un identificador y referirnos a ellas por este nombre que le acabamos de otorgar. También proporciona herramientas para ejecutar sentencias HQL, y ver su traducción a SQL. El Hibernate Query Language nos sirve para encapsular todas las sentencias y facilitar su posterior uso.

Orientado a Objetos. Hibernate nos permite desarrollar objetos persistentes siguiendo el lenguaje común de Java: incluyendo asociación, herencia, polimorfismo, composición y las Colecciones de Java.

Si bien Hibernate reúne un conjunto de características que lo hacen ser muy llamativo, no es una solución óptima en el caso de proyectos de migración de datos, ni tampoco cuando los criterios con que se creo la base de datos no tienen un mínimo de calidad.

Además la curva de aprendizaje en Hibernate es un tanto lenta y al principio se complica si no se tienen los conocimientos suficientes.

Por otro lado la definición de los ficheros de Hibernate es compleja y requiere el conocimiento de qué es lo que se va a necesitar, cómo y cuando. Una mala configuración puede acarrear problemas de rendimiento o un cambio en la configuración puede hacer que fallen otros desarrollos.

iBATIS.

Existe un gran conjunto de características que hacen de iBATIS un Framework bastante útil para trabajar con ciertas bases de datos. A continuación se exponen las ventajas y desventajas de este Framework.

Ventajas:

- Provee los beneficios de trabajar con SQL.
- Estabilidad y Facilidad para encontrar dónde está un problema dado. Las transacciones funcionan sin dar dolores de cabeza.
- Maneja Objetos de Acceso a Datos (DAO) que son un Patrón de Diseño Core J2EE y considerados una buena práctica. La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (el cual contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

- iBATIS DAO facilita el manejo de la capa de persistencia mediante la utilización de interfaces.
- iBATIS SQL MAPS facilita la asignación de datos entre el modelo de datos de nuestra base de datos y el modelo de clases de nuestra aplicación.
- Simplicidad:
    - Es de fácil aprendizaje.
    - Es de fácil manutención.
    - Programación declarativa (configurando ficheros XML)
    - Separación entre SQL y el lenguaje de programación de la aplicación (Java/.NET).
- Portabilidad: entre Java y .NET; y entre diferentes proveedores de bases de datos.
- Es de código abierto (open source).

Desventajas:

- No auto-genera código SQL
- No tiene un lenguaje propietario de consultas, utiliza el SQL común.
- No utiliza objetos de entidad (Entity) ni objetos de persistencia.
- No construye objetos en caché.
- Las bases de datos que gestiona iBATIS deben ser exclusivamente relacionales.
- No es totalmente transparente (hay que programar SQL).
- Pierde funcionalidad si casi todas las sentencias SQL son construidas dinámicamente a menos de que se utilice JDBC puro.

iBatis no debe ser utilizado cuando se requiere una automatización total y transparente de la persistencia. Tampoco es recomendable su uso cuando se requiere soporte para diversos motores de bases de datos de forma transparente.

4.5 Aspectos generales.

Hibernate.

Para poder entender y, por consecuente, usar Hibernate, es necesario que los programadores tengan conocimientos previos sobre JDBC y bases de datos relacionales. Hasta cierto punto, Hibernate funciona de forma semejante a JDBC, con la diferencia que es mucho más sencillo de codificar.

Hibernate tiene una curva de aprendizaje alta pues, en promedio, lleva de 15 a 30 días crear una aplicación sencilla y sin toda la funcionalidad que nos ofrece. Sin embargo, su complejidad y su capacidad para trabajar en la capa de Persistencia es lo que ha hecho de Hibernate una de las ORM preferidas por los programadores.

Hibernate es recomendable cuando se trata de proyectos grandes, tales como los empresariales, pues la cantidad de información que se maneja dentro de una empresa es bastante grande (empleados, departamentos, proveedores, etc.). Al trabajar con Hibernate en este tipo de aplicaciones se reduce el código de la aplicación, y por lo tanto, el mantenimiento de dichas aplicaciones resulta más sencillo, ahorrando de esta forma bastante tiempo.

Esto no quiere decir que Hibernate es exclusivo de proyectos grandes, sin embargo, para el caso de proyectos pequeños es recomendable ahorrarse toda la codificación y configuración de Hibernate y emplear Frameworks mucho más sencillos como es el caso de iBATIS.

iBATIS.

iBATIS no requiere de gran experiencia por parte del programador, pues es muy intuitivo y simple, lo cual hace que su curva de aprendizaje sea muy baja, en promedio de 7 a 15 días.

iBATIS es la solución ideal cuando se trata de proyectos pequeños. No requiere de mucha configuración, solo basta relacionar nuestros objetos con las bases de datos.

Para proyectos grandes, iBATIS no es recomendado, pues la cantidad de procesos que se llevan a cabo requieren de una automatización y posiblemente de creación dinámica de consultas y es, en este punto, donde iBATIS deja de ser flexible. Para este tipo de proyectos lo más recomendable sería usar Hibernate que ofrece la posibilidad de automatizar los procesos y crear el SQL por nosotros, únicamente indicándole las acciones a ejercer sobre nuestros objetos.

Resumen

En la siguiente tabla se resumen los aspectos generales tanto de iBATIS como de Hibernate.

Características	iBATIS	Hibernate
Curva de Aprendizaje	Baja (7-15 días)	Alta(15 o más días)
Experiencia	Poca, solo requiere conocimientos de SQL por parte del programador.	Media, requiere conocimientos sobre JDBC.
Tipo de Proyecto	Pequeño / Mediano, por su simplicidad sería fácil de implementar	Mediano / Grande, por su capacidad de automatizar procesos y crear SQL de forma interna.

CAPÍTULO 5.
DESARROLLO DE UNA APLICACIÓN

CAPÍTULO 5.
DESARROLLO DE UNA APLICACIÓN

En capítulos anteriores se explicó la configuración y uso básico de los Frameworks de estudio, ahora, tomando como referencia esos ejemplos, explicaremos a lo largo de este capítulo el desarrollo de una aplicación tanto con Hibernate como con iBATIS con el fin de exponer de una forma más detallada su funcionamiento dentro de una aplicación. Con esto no sólo se expondrán los beneficios que ofrece cada una de las herramientas, sino también, podremos realizar la comparación entre cada una de ellas, es decir, entre sus características y su forma de operación para que en un futuro dado podamos elegir uno de los Frameworks para un proyecto en particular.

Dicha aplicación será un Help Desk, el cual desarrollaremos empleando las siguientes herramientas:

1. Un IDE (Integrated Development Environment).
2. Un Sistema Gestor de Base de Datos (SGBD).
3. Un contenedor JAVA.
4. Un Framework de persistencia.

Cabe aclarar que también se pueden emplear otros recursos para facilitar la creación del Help Desk, por lo que, en nuestro particular caso, usaremos un elemento adicional: El Framework Struts.

De igual forma y en particular, usaremos el IDE Eclipse, mientras que el SGBD será MySQL, y en lo que al contenedor de aplicaciones respecta emplearemos Tomcat. Es de vital importancia estar familiarizado con la instalación, configuración y uso de estos elementos, ya que estos conocimientos no se encuentran dentro de los objetivos de este trabajo, por lo que, en caso de existir dudas relacionadas, se puede consultar la información proporcionada en el Sitio Oficial de cada una de ellas.

El alcance de este trabajo se limita única y exclusivamente a comparar y valorar el desempeño de los Frameworks de Persistencia.

Una vez definido lo anterior, daremos paso al desarrollo y descripción del proyecto.

5.1 Help Desk, nuestra aplicación.

5.1.1 ¿Qué es un Help Desk?

Esta aplicación maneja sus tareas usando un sistema de solicitud por formularios. Cuando los usuarios tienen algún problema con sus PCs, llenan un formulario, en línea o bien pueden proporcionar sus datos vía telefónica. En este sistema de solicitudes se catalogan las peticiones de ayuda de varias maneras. Una de ellas puede ser el tipo de problema para el cual se necesita una solución; otra, el departamento en el cual trabaja el usuario final.

Además de responder a las solicitudes, los técnicos de soporte del Help Desk llevan a cabo las revisiones de inventario y realizan diversas rutinas de mantenimiento y actualización de las PCs y redes dentro de la organización. Otra función importante del Help Desk es la recolección y uso de datos. Todas las peticiones se registran en una base de datos. Estas solicitudes proporcionan información valiosa que la organización puede usar a su conveniencia para tomar decisiones acerca del mejoramiento del soporte técnico, comprar nuevas PCs y software, sistemas de actualización y determinar la necesidad de implementar más programas de capacitación.

5.1.2 Diseño de la aplicación

Nuestro Help Desk será un pequeño sitio web donde los usuarios al iniciar sesión podrán realizar diferentes actividades como: Levantar reportes de una manera fácil y sencilla, darle seguimiento a los mismos, consultar todos los reportes solucionados a manera de auto-ayuda.

Los administradores por su parte podrán gestionar tanto las cuentas de usuarios y administradores así como también los reportes que se estarán guardando en la base de datos.

Para esta aplicación crearemos 2 tablas en la base de datos las cuales se detallan a continuación:

Usuario

Descripción del Campo	Tipo	Nombre del Campo BD
Número de Empleado (id)	Integer	Id
Nombre	Varchar 20	Nombre
Apellido Paterno	Varchar 20	Appat
Apellido Materno	Varchar 20	Apmat
Cargo	Varchar 20	Cargo
Área	Varchar 20	Area
Teléfono	Varchar 10	Telefono
Contraseña	Varchar 15	Contrasena
Tipo	Varchar 3	Tipo

Reportes

Descripción del Campo	Tipo	Nombre del Campo BD
Número de reporte	Integer	Idrep
Administrador Responsable	Integer	Idadm
Reportador (id-empleado)	Integer	Idusu
Solución	Varchar 200	Solucion
Estado	Varchar 10	Estado
Tipo de problema	Varchar 10	tipoProb
Descripción	Varchar 200	Descprob
Fecha de Registro	Varchar 10	Fechareg
Fecha de Solución	Varchar 10	Fechasol

A continuación presentaremos su diagrama Entidad-Relación:

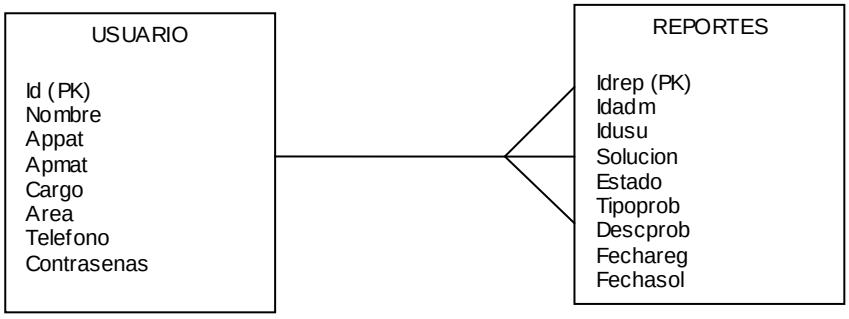


FIGURA 5.1 Diagrama Entidad-Relación.

Comencemos por describir el flujo de navegación de nuestro Help Desk.

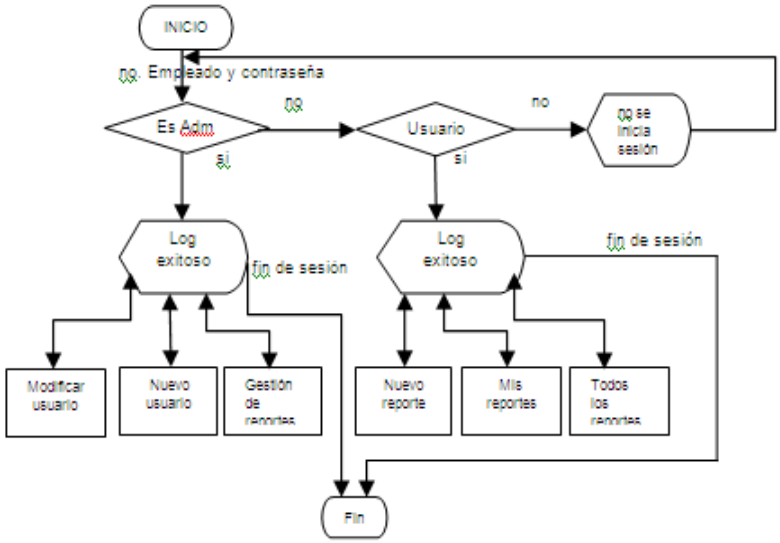


FIGURA 5.2 Diagrama de navegación

Nuestra aplicación se creará con base a la arquitectura MVC. La parte de la vista estará compuesta por JSP’s los cuales recibirán y mostrarán información, es decir serán el punto de conexión entre el usuario y la aplicación.

La parte del controlador será manipulada por Struts mediante los servlets que heredarán de la clase Action y que para identificarlos llevarán al final de su nombre la palabra *Action*. Estas dos partes se relacionarán entre sí por el archivo *struts-config.xml*.

La capa del modelo estará conformada por las reglas de negocio así como por el acceso a la base de datos, por lo que los Frameworks de estudio estarán involucrados en esta capa. Aquí crearemos la interfaz *HelpDeskModel*, donde se declararán los métodos a utilizar por cada uno de los beans, la cual quedaría de la siguiente forma:

```
 5    public interface HelpDeskModel<B>  {
 6       public List<B> consultarTodo();
 7       public B consultarPorId(Integer id);
 8       public void actualizar (B bean);
 9       public void borrar (B bean);
10       public void insertar (B bean);
11  }
```

Esta interfaz será implementada por las clases: *HelpDeskUsuImpl* y *HelpDeskRepImpl*, las cuales corresponderán a los beans *Usuario.java* y *Reporte.java* respectivamente.

Cabe aclarar que para el desarrollo del proyecto se han creado diversas consultas con base a la funcionalidad del mismo, pero con el fin de exponer de una manera sencilla el funcionamiento de los Frameworks, nos enfocaremos en las operaciones básicas que se pueden realizar en una base de datos y que además son empleadas dentro del proyecto. Por lo tanto, en apartados posteriores nos limitaremos única y exclusivamente a detallar los métodos antes descritos en la Interfaz.

Para tener una mejor percepción sobre el funcionamiento del proyecto, daremos una breve introducción sobre su comportamiento en general:

Index es la página principal desde donde se puede iniciar sesión, esta compuesta por un formulario donde se pide el número de usuario y contraseña.

FIGURA 5.3 Página Principal

Estos datos se validarán por medio del archivo *validation.xml* el cual es empleado no solo por este JSP sino por varios para la validación de datos. Su funcionamiento esta basado en algoritmos propios de Struts.

Este JSP se debe ligar con una clase a la cual llamaremos *LoginAction.java*, esta relación se definirá a través del archivo *struts-config.xml*, en el cual se establecerá ésta y muchas más relaciones.
Una vez dentro del sistema se valida si el usuario tiene permisos de administrador o no y dependiendo de esto se realiza el proceso correspondiente.

Si el usuario es un administrador, se mostrará la página de *Administrador.jsp* desde donde se pueden realizar diferentes acciones:

- Solucionar reportes
- Modificar usuarios
- Agregar usuarios

FIGURA 5.4 Administrador.jsp

La primer acción estará ligada con *ConsultarReportesAction*, este se encargará de seleccionar los reportes que no han sido solucionados, los cuales se mostrarán en *ListaReportes.jsp* desde aquí el administrador puede elegir un reporte para modificarlo. Al seleccionar un reporte se llama al action *DetallarReporteAction* el cual mostrará el detalle de dicho reporte en: *DetallesReporte.jsp* que mandará llamar a *ActualizarReporteAction* para modificar los datos del reporte elegido, al finalizar se regresará a *ListaReportes.jsp* que desplegará los cambios.

La segunda acción nos envía a *AgregarUsuario.jsp*, los datos correspondientes al formulario de esta página se procesarán en *AgregarUsuarioAction* después de lo cual se enviará a *Administrador.jsp*.

En caso de que no se tengan permisos de administrador, la página mostrada será *Usuario.jsp* en donde se podrán realizar las siguientes acciones:

- Enviar un reporte nuevo
- ConsultarFAQ's
- Consultar mis Reportes

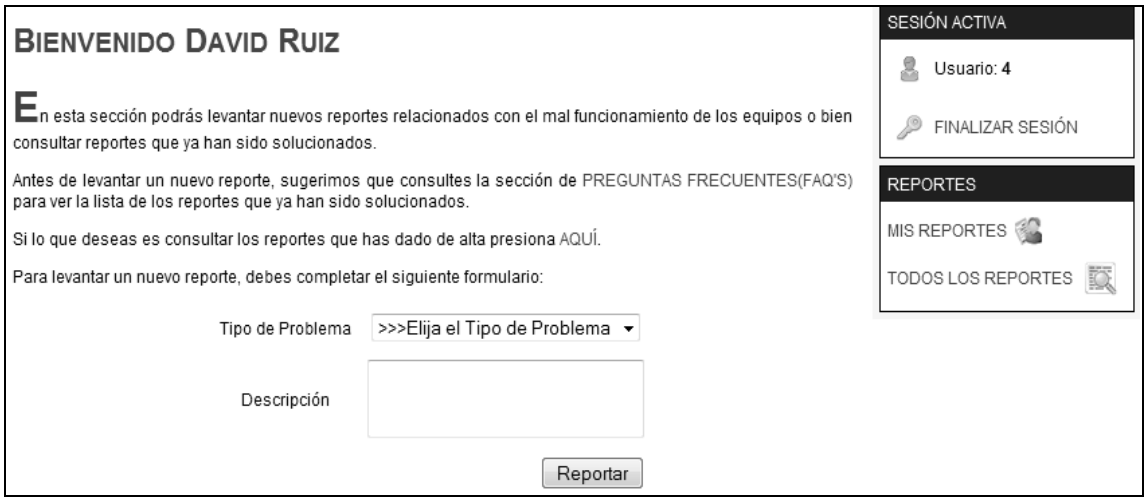


FIGURA 5.5 Usuario.jsp

En este JSP hay un formulario para dar de alta un reporte, después de llenar los datos correspondientes, éstos se procesarán en *GuardarReporteAction* la cual pasará el control a *ReporteOk.jsp* desde donde se podrá regresar a *Usuario.jsp*.

En el caso de que se dé click en 'ConsultarFAQ's', la clase que tomará el control será *ConsultarReportesAction* la cual estará ligada a su vez a *ListaReportes.jsp*.

La última opción, 'Consultar mis Reportes', estará ligada con *ConsultarReportesAction* que enviará la información correspondiente de los reportes propios de ese usuario a la página *ListarReportes.jsp*.

Cuando se está dentro de una sesión ya sea como administrador o como usuario se tiene siempre la opción de finalizar sesión:

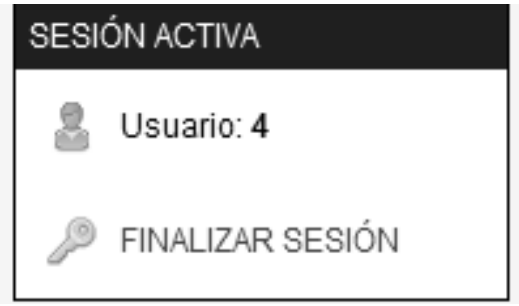


FIGURA 5.6 Opción “finalizar sesión”

La clase encargada de procesar ésta información será *LogoutAction* que enviará al JSP *Index*.

Con lo anterior concluimos la breve descripción de nuestro proyecto. Ahora, daremos paso a la integración del mismo con cada uno de los Frameworks.

5.2 Integrando con iBATIS

Haciendo uso de lo expuesto en el capítulo 2 de este trabajo explicaremos como conectar ésta herramienta con nuestra aplicación.

5.2.1 Instalación y configuración de iBATIS

Para la instalación de este Framework necesitamos descargar de la página: *http://ibatis.apache.org* una distribución para java, la que se ocupará aquí será la 2.0, que contiene los archivos:

- Ibatis-common-2.jar
- Ibatis-dao-2.jar
- Ibatis-sqlmap-2.jar

Estos 3 archivos se deben pegar en la carpeta de ‘plugins’ de eclipse y posteriormente se deben agregar al classpath de nuestra aplicación.

Para comenzar con la configuración debemos crear las clases java que representarán a cada entidad de nuestra base de datos, por lo tanto tendremos 2 clases: *Usuario.java* y *Reportes.java*, a cada una de estas clases le corresponde un archivo .xml, donde se

definirán las operaciones SQL que actuarán sobre nuestra base de datos, es decir, en estos archivos xml se ligarán los atributos de los beans con los resultados que obtengamos de las operaciones SQL.
Después se crea un archivo dao.*xml* donde se especifica cual será el administrador de transacciones que se utilizará, en este caso será SQLMAPS de iBATIS.
Este mismo archivo se mandará llamar en la clase *DaoConfig* que es la encargada de configurar el Framework DAO de iBATIS.
Tenemos que crear un archivo *sqlMapConfig.xml* donde se describirán los parámetros de la base de datos. Dicho archivo es necesario para configurar el Framework SQLMAP.

Una vez editados nuestros archivos de configuración, comenzaremos por hacer uso de las funciones del Framework, para lo cual la clase *Reporte.java* se usará como referencia.

Empecemos por explicar el proceso para guardar un Reporte en nuestra Base de Datos, lo que en nuestro particular caso corresponde a dar de alta un Nuevo Reporte.
Dentro de nuestro Help Desk, el usuario debe llenar el siguiente formulario:

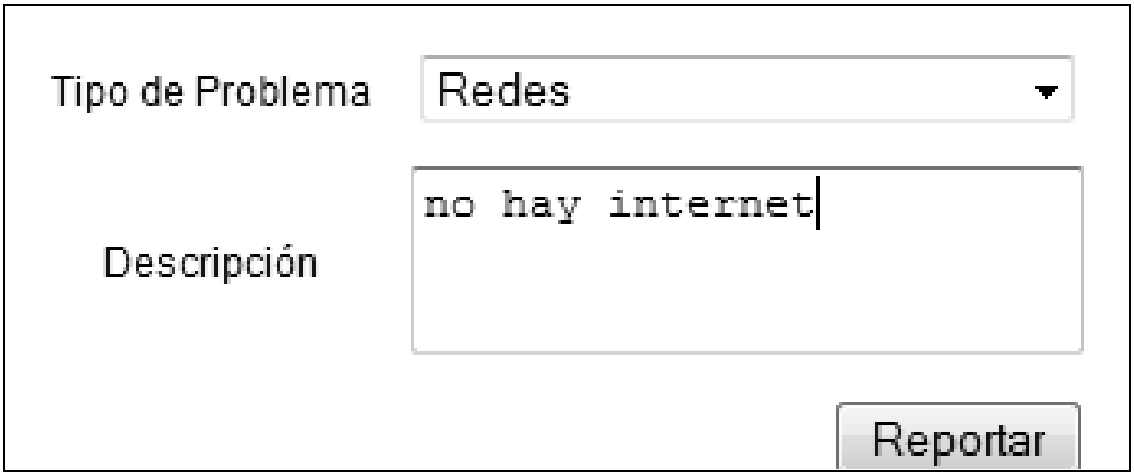


FIGURA 5.7 Formulario para dar de alta un reporte.

Al dar click en el botón reportar, los datos son procesados por el servlet *GuardarReporteAction*. Después de hacer las respectivas validaciones se llama al método de persistencia correspondiente de iBATIS : `insertar`:

```
16     public void insertar(Reporte rep) {
17         try {
```

```
18              getSqlMapExecutor().insert("insertar", rep);
19          } catch (SQLException e) {
20              throw new DaoException("Error insertando reporte. Causa:"+e,e);
21          }
22      }
```

Desde aquí se llama a la sentencia SQL correspondiente, definida en *Reporte.xml*:

```
31 <insert id="insertar" parameterClass="helpdesk.beans.Reporte">
32    INSERT INTO reportes (idrep, idadm, idusu, solucion, estado, tipoprob,
33    descprob, fechareg, fechasol) VALUES (#numeroReporte#,
34    #numeroAdmin#, #numeroUsuario#, #solucion#, #estadoReporte#,
35    #tipoProblema#, #descProblema#, #fechaRegistro#, #fechaSolucion#)
36 </insert>
```

Ahora, para ejemplificar la forma de realizar un ‘select’ dentro de nuestra aplicación, nos basaremos en la función de consultar “Mis Reportes”, la cuál está disponible una vez que un usuario o administrador ha iniciado sesión dentro del sistema.

FIGURA 5.8 Enlaces para consultar los reportes.

Esta petición es procesada por la clase *ConsultarReportesAction*, la cual manda llamar al método de persistencia: `consultarMisReportes`

```
33   public List<Reporte> consultarMisReportes(Integer numeroUsuario) {
34       try {
35          List<Reporte> reportes=
36(List<Reporte>)getSqlMapExecutor().queryForList("consultarMisReportes",
37 numeroUsuario);
38           return reportes;
39       } catch (SQLException e) {
```

```
40          throw new DaoException("Error recuperando reportes. Causa:"+e,e);
41        }
42    }
```

Y esta a su vez ejecuta la sentencia SQL:

```
 7 <select id="consultarMisReportes" resultClass="helpdesk.beans.Reporte">
 8    SELECT * FROM reportes WHERE idusu = #numeroUsuario#
 9 </select>
```

Para explicar el uso de la función ‘update’ nos enfocaremos en la opción de ”Solucionar Reportes”, la cuál se encuentra dentro de una sesión de administrador. Dentro de dicha opción se puede modificar un reporte, por lo que después de darle click a este link, nos enviará la siguiente pantalla:

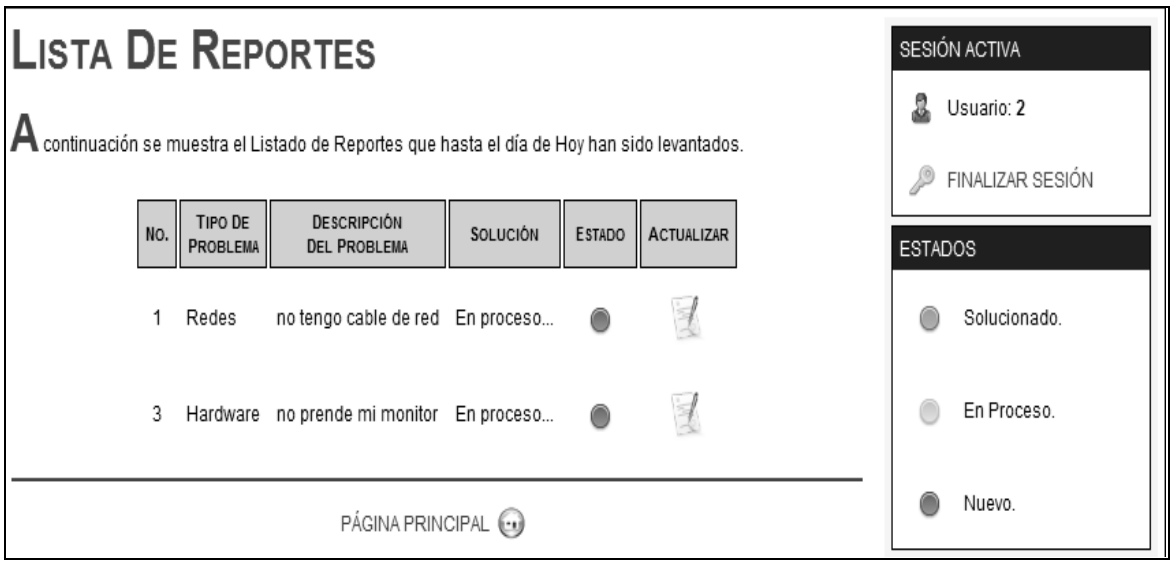


FIGURA 5.9 Pantalla que muestra la lista de los reportes a solucionar.

De esta lista debemos elegir un reporte dando click sobre su icono “actualizar”, enseguida nos mandará la pantalla:

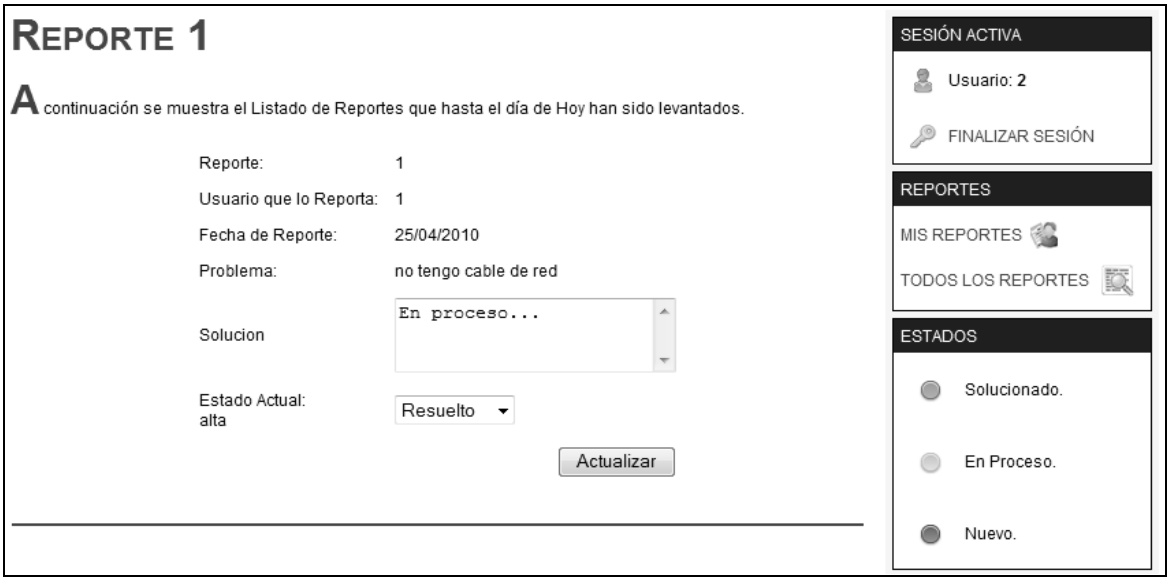


FIGURA 5.10 Formulario para modificar el reporte seleccionado.

Al terminar de llenar los datos de este formulario, damos click en el botón "Actualizar", lo cual nos guiará a *ActualizarReporteAction*, que llama a nuestro método: `actualizar`

```
 8   public void actualizar(Reporte rep) {
 9       try {
10           getSqlMapExecutor().update("actualizar", rep);
11       } catch (SQLException e) {
12           throw new DaoException("Error al actualizar reporte Causa:"+e,e);
13       }
14   }
```

Este método definido en la clase implementadora *HelpDeskRepImpl* esta ligada con la sentencia:

```
38 <update id="actualizar" parameterClass=" helpdesk.beans.Reporte ">
39   UPDATE reportes SET solucion = #solucion#, idadm = #numeroAdmin#, estado
= 40  #estadoReporte#, fechasol =#fechaSolucion# WHERE idrep = #numeroReporte#
41 </update>
```

Otra de las acciones que podemos realizar dentro de una sesión de administrador es eliminar a un usuario determinado, lo cuál se hace dando click en el enlace "Modificar el Perfil de un Usuario":

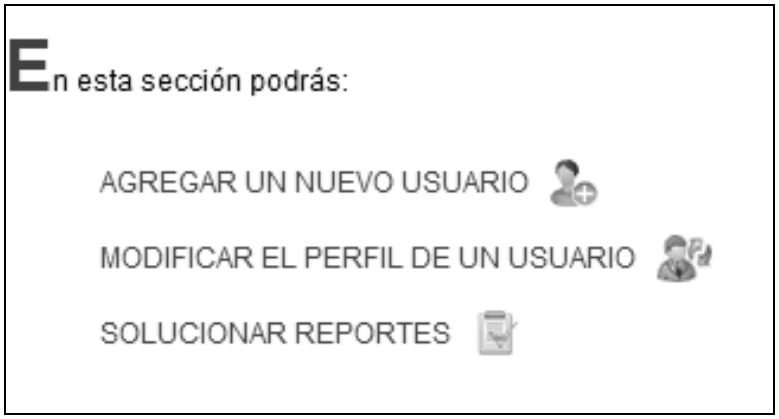


FIGURA 5.11 Enlaces dentro de la sesión de un administrador

Este link nos envia a un JSP donde se listan todos los usuarios existentes, enfrente de cada registro están los botones: “modificar” y “eliminar”. Al presionar el botón “eliminar” de un registro, se envía un mensaje de confirmación, el cual debe ser aceptado para poder llamar al método: `borrar`

```
25   public void borrar(Usuario usu) {
26       try {
27          getSqlMapExecutor().delete("eliminar", usu);
28       } catch (SQLException e) {
29           throw new DaoException("Error al eliminar usuario. Causa: "+e,e);
30       }
31   }
```

La sentencia SQL correspondiente a este método es:

```
42 <delete id="eliminar" parameterClass=" helpdesk.beans.Reporte ">
43    DELETE FROM usuario WHERE idusu = #id#
44 </delete>
```

Con esto, finalizamos la integración de iBATIS con el Help Desk. En el siguiente tema sustituiremos este Framework por Hibernate, veremos la integración del mismo con el proyecto y la forma de implementar los métodos explicados con anterioridad.

5.3 Integrando con Hibernate

Es el turno de este Framework para interactuar con nuestro Help Desk, ahora vamos a explicar los pasos a seguir tal y como se hizo con iBATIS.

5.3.1 Instalación y configuración de Hibernate

Necesitamos conseguir una distribución de este Framework, lo cual se puede hacer desde el portal de Hibernate en Internet: *http://hibernate.org.*

La versión que se utilizará para el desarrollo de este trabajo es la 3.3.1, ésta contiene el archivo: hibernate3.jar, el cuál se debe pegar en la carpeta 'plugins' de eclipse y posteriormente se debe agregar al classpath de la aplicación. Adicionalmente se requieren otros .jar's para el correcto funcionamiento de Hibernate, estos también se encuentran dentro de la distribución y son: antlr-2.7.6.jar, commons-collections-3.1.jar, dom4j-1.6.1.jar, javassist-3.4.GA.jar, jta-1.1.jar y slf4j-api-1.5.2.jar.

Todos los .jar's antes mencionados, también pueden añadirse directamente en la ruta de la aplicación, es decir, en WebContent/WEB-INF/lib.

Como se explicó anteriormente en el capítulo 3, se debe crear un archivo de configuración *hibernate.cfg.xml,* el cuál contendrá la información necesaria para conectarse a la base de datos, además de las referencias a los archivos de mapeo necesarios para la aplicación. El archivo de configuración correspondiente a nuestra aplicación es:

```
1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE hibernate-configuration PUBLIC
3        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5
6<hibernate-configuration>
7    <session-factory>
8        <!-- Configuración para la conexión a la base de datos -->
9        <property name = "connection.driver_class">com.mysql.jdbc.Driver
</property>
10     <property name = "connection.url">jdbc:mysql://localhost/helpdesk
</property>
11      <property name = "connection.username">root</property>
12      <property name = "connection.password">miPassword</property>
```

```
13
14      <!-- Dialecto SQL -->
15      <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
16
17        <!-- Enable Hibernate's automatic session context management -->
18        <property name="current_session_context_class">thread</property>
19
20        <!-- Disable the second-level cache  -->
21        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
22
23    <!-- Propiedad para mostrar en la consola el SQL usado por Hibernate -->
24        <property name="show_sql">false</property>
25
26        <!-- Propiedad que indica el(los) archivo(s) de mapeo a utilizar-->
27        <mapping resource="helpdesk/beans/Usuario.hbm.xml"/>
28
29        <mapping resource="helpdesk/beans/Reporte.hbm.xml"/>
30
31    </session-factory>
32 </hibernate-configuration>
```

Luego, se deben crear archivos hbm.xml. Estos archivos contendrán la relación existente entre los beans de nuestra aplicación y los campos de la base de datos, por lo que en nuestro caso tendremos un archivo *Usuario.hbm.xml* y un archivo *Reporte.hbm.xml* correspondientes a *Usuario.java* y *Reporte.java* respectivamente. Estos archivos deben estar dentro del mismo paquete donde se ubican nuestros beans, además de que deben ser referenciados desde el archivo *hibernate.config.xml*.

El archivo *Usuario.hbm.xml* se estructura de la siguiente forma:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD//EN"
4  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5
6  <hibernate-mapping>
7      <class name="helpDesk.beans.Usuario" table="Usuario">
8            <id name="idUsuario" type="integer" column="ID">
```

```
9                   <generator class="increment"/>
10          </id>
11
12          <property name="nombre" column="nombre"
13          type="string"
14          not-null="true"/>
15
16          <property name="apPat" column="appat"
17          type="string"
18          not-null="true" />
19
20          <property name="apMat" column="apmat"
21          type="string"
22          not-null="true" />
23
24          <property name="cargo" column="cargo"
25          type="string"
26          not-null="true" />
27
28          <property name="area" column="area"
29          type="string"
30          not-null="true" />
31
32          <property name="telefono" column="telefono"
33          type="string"
34          not-null="true" />
35
36          <property name="password" column="contrasena"
37          type="string"
38          not-null="true" />
39
40          <property name="tipo" column="tipo"
41          type="string"
42          not-null="true" />
43
44    </class>
45
46 </hibernate-mapping>
```

Mientras que *Reporte.hbm.xml* se estructura de la siguiente forma:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD//EN"
4  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5
6  <hibernate-mapping>
7      <class name="fes.model.Reporte" table="Reportes">
8            <id name="numeroReporte" type="integer" column="idrep">
9                  <generator class="increment"/>
10           </id>
11
12           <property name="numeroAdmin" column="idadm"
13           type="integer" />
14
15           <property name="numeroUsuario" column="idusu"
16           type="integer"
17           not-null="true" />
18
19           <property name="solucion" column="solucion"
20           type="string"
21           not-null="true" />
22
23           <property name="estadoReporte" column="estado"
24           type="string"
25           not-null="true" />
26
27           <property name="tipoProblema" column="tipoprob"
28           type="string"
29           not-null="true" />
30
31           <property name="descProblema" column="descprob"
32           type="string"
33           not-null="true" />
34
35           <property name="fechaRegistro" column="fechareg"
36           type="string"
37           not-null="true" />
38
```

```
39            <property name="fechaSolucion" column="fechasol"
40            type="string" />
41
42      </class>
43
44 </hibernate-mapping>
```

Una vez definidos los archivos .hbm.xml el siguiente paso es crear la clase donde se encontrará la Fábrica de Sesiones (Session Factory) la cuál, como mencionamos durante el capítulo 3, debe instanciarse sola una vez dentro de la aplicación. Nuestra clase se denominará *HibernateUtil* y se muestra a continuación.

```
1  package helpdesk.hibernate.util;
2
3  import org.hibernate.SessionFactory;
4  import org.hibernate.cfg.Configuration;
5
6  public class HibernateUtil {
7
8      private static final SessionFactory sessionFactory;
9
10     static {
11           try{
12                 sessionFactory =new
Configuration().configure().buildSessionFactory();
13           }
14           catch(Throwable ex){
15                 System.err.println("Error al crear Session Factory: "+ex);
16                 throw new ExceptionInInitializerError(ex);
17           }
18     }
19
20     public static SessionFactory getSessionFactory() {
21           return sessionFactory;
22     }
23
24}
```

Esta clase será empleada para crear las 'Sesiones' de Hibernate que nos permitirán conectarnos con la Base de Datos.
Ahora, estamos listos para hacer uso del Framework. Como recordatorio, solo se expondrán las operaciones básicas que se pueden realizar dentro de una base de datos.

Método Insertar / Guardar (insert):

Empecemos por explicar el proceso para guardar un Reporte en nuestra Base de Datos tomando como ejemplo, la función correspondiente al dar de alta un Nuevo Reporte.

Dentro de nuestro Help Desk, el usuario debe llenar el siguiente formulario:

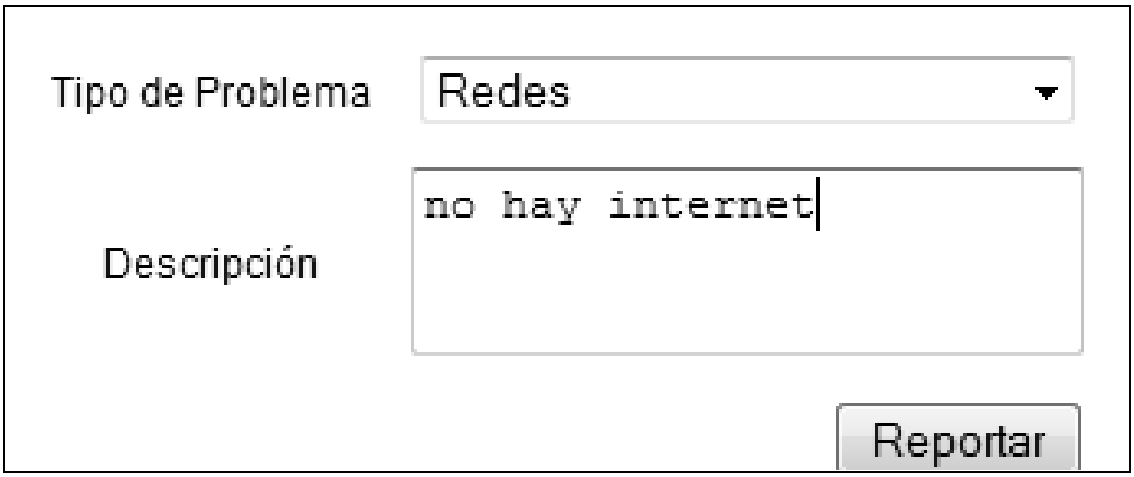


FIGURA. Formulario para dar de alta un reporte.

Al dar click en el botón reportar, los datos son procesados por el servlet correspondiente y después de hacer las respectivas validaciones se llama al método: `insertar`.

```
1  public void insertar(Reporte reporte){
2      Session session =HibernateUtil.getSessionFactory().getCurrentSession();
3             session.beginTransaction();
4             session.save(reporte);
5             session.getTransaction().commit();
6      }
```

Con lo anterior, nuestra aplicación guardará un nuevo objeto (Reporte) en la base de datos.

Método Consultar (select):

Ahora, para ejemplificar la forma de realizar una consulta dentro de nuestra aplicación, nos basaremos en la función de consultar “Mis Reportes”, la cuál está disponible una vez que un usuario o administrador ha iniciado sesión dentro del sistema.

FIGURA. Enlaces para consultar los reportes.

Esta petición es procesada por un servlet, dentro del cual después de realizar las validaciones correspondientes se invoca al método: `consultarMisReportes`.

```
1  public List<Reporte> consultarMisReportes(Integer idUsu){
2      try{
3      Session session =HibernateUtil.getSessionFactory().getCurrentSession();
4      session.beginTransaction();
5      String consulta = "FROM fes.model.Reporte as Reporte WHERE
numeroUsuario="+ idUsu;
6      List<Reporte> reportes =
(List<Reporte>)session.createQuery(consulta).list();
7      session.close();
8      return reportes;
9      }
10     catch(Exception e){
11           e.printStackTrace();
12           return null;
13     }
14 }
```

Un dato importante de este método es que dentro del mismo se usa HQL para la obtención de una lista de objetos de tipo Reporte (línea 5). La ejecución de dicha sentencia se lleva a cabo en la línea 6.

Método Actualizar (update):

Para explicar el uso de la función actualizar nos enfocaremos en la opción de "Solucionar Reportes", la cuál se encuentra dentro de una sesión de administrador. Dentro de dicha opción se puede modificar un reporte, por lo que después de darle click, nos enviará la siguiente pantalla:

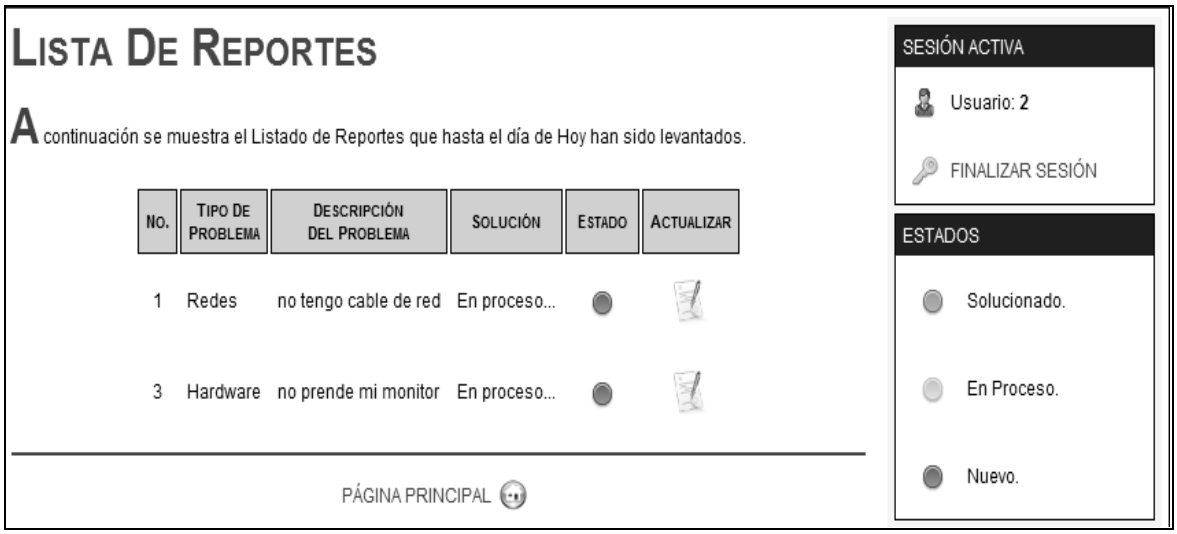


FIGURA. Pantalla que muestra la lista de los reportes a solucionar.

Para actualizar un reporte debemos elegir el reporte deseado dando click sobre su icono "actualizar", enseguida nos mandará la pantalla:

FIGURA. Formulario para modificar el reporte seleccionado.

Al terminar de llenar los datos de este formulario y dar click en el botón "Actualizar", la petición se envía al servlet correspondiente para su validación y procesamiento. Después, dentro del mismo, se invoca al método: `actualizar`.

```
1  public void actualizar(Reporte reporte){
2      Transaction tx = null;
3      Session session =HibernateUtil.getSessionFactory().getCurrentSession();
4      try{
5             tx = session.beginTransaction();
6             session.update(reporte);
7             tx.commit();
8      }catch(Exception e){
9             e.printStackTrace();
10     }
11 }
```

Y de esta forma estaremos actualizando un registro con ayuda de Hibernate.

Método Eliminar / Borrar (delete):

Otra de las acciones que podemos realizar dentro de una sesión de administrador es eliminar a un usuario determinado, lo cuál se hace dando click en el enlace "Modificar el Perfil de un Usuario":

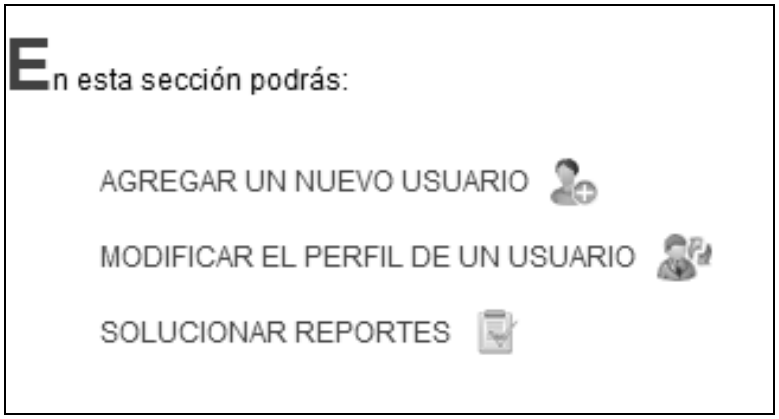


FIGURA. Enlaces dentro de la sesión de un administrador

Este link nos envía a un JSP donde se listan todos los usuarios existentes, enfrente de cada registro están los botones: "modificar" y "eliminar". Al presionar el botón "eliminar" de

un registro, se envía un mensaje de confirmación, el cual debe ser aceptado para poder llamar al método: `borrar`.

```
1  public void borrar(Reporte reporte){
2      Transaction tx = null;
3      Session session =HibernateUtil.getSessionFactory().getCurrentSession();
4      try{
5             tx = session.beginTransaction();
6             session.delete(reporte);
7             tx.commit();
8      }catch(Exception e){
9             e.printStackTrace();
10     }
11 }
```

De ésta forma eliminamos un objeto de tipo Reporte de nuestra base de datos.

Con esto, damos por terminada la integración de Hibernate con el Help Desk. Recordemos que únicamente estamos explicando las operaciones básicas sobre una base de datos, pues a lo largo del proyecto, se realizan diversas consultas, las cuáles se han omitido por tratarse de consultas compuestas que se derivan de las básicas.

Conclusiones

En el mundo real, los sistemas empresariales ya sean grandes, medianos o pequeños, la mayor parte de las veces requieren de consultas a bases de datos para obtener información específica y al mismo tiempo, requieren procesar y analizar dicha información. Estos procesos eran complejos para los programadores por lo que se crearon diferentes Frameworks de Persistencia cuyo fin es simplificar el desarrollo de los sistemas.

Nuestro trabajo estuvo dedicado a comparar y analizar dos de los Frameworks que hasta el momento son los más estables y usados: Hibernate e iBATIS.

A nuestro parecer, Hibernate es una herramienta que simplifica mucho la recuperación de datos, con ayuda de sus archivos de configuración, su fábrica de sesión y sus archivos de mapeo se puede establecer la conexión a la base de datos así como también se puede establecer la relación entre los campos de una tabla y los atributos de un objeto. Sin embargo, para proyectos pequeños en donde las bases de datos no sean muy grandes, la integración de Hibernate lo consideramos trabajo extra. Además su curva de aprendizaje es un poco elevada. Es un Framework bastante completo que ofrece muchas otras ventajas aparte de las que hemos expuesto en el trabajo. Consideramos a Hibernate una herramienta excelente para proyectos de medianos a grandes.

Por otra parte, iBATIS, en nuestra opinión, es sumamente sencillo, se integra perfectamente con aplicaciones pequeñas o medianas. Además, toda la parte de acceso a bases de datos se implementa en archivos de mapeo y además en forma de SQL puro. Esto nos deja la posibilidad de formular las consultas a nuestra manera, cosa que no sucede con Hibernate. Su curva de aprendizaje es pequeña, pues solo es necesario entender la relación entre los archivos de mapeo y nuestras clases.

Este proyecto nos brindó la oportunidad de conocer más a fondo dos de las principales herramientas que se utilizan actualmente en la persistencia de objetos ya que abarca desde sus conceptos base hasta la forma en cómo se implementan dentro de una aplicación real. Por medio del análisis de las características de cada uno comprendimos

que ambos tienen cualidades y desventajas propias, así que desde nuestra experiencia, recomendamos que éstas siempre se tomen en cuenta al momento de elegir una herramienta de persistencia para una aplicación, es decir, que empaten con las características del proyecto a desarrollar.

Los Frameworks aquí expuestos no son los únicos, pero sí son de los más utilizados actualmente, por lo que pensamos que este estudio es de gran utilidad para un profesionista, pues expone herramientas de trabajo para facilitar y mejorar su desempeño. El campo de la programación es bastante amplio y se vuelve una necesidad conocer las herramientas disponibles.

Como un beneficio personal, el desarrollo de este trabajo nos permitió integrarnos con el uso de los Frameworks y, por lo tanto, incrementar nuestra experiencia en cuanto al uso de los mismos. De esta forma, tendremos un mayor repertorio de recursos para futuros proyectos en el campo de la programación.

BIBLIOGRAFÍA

Clinton Begin, Brandon Goodin and Larry Meadors. *iBATIS in Action.* Wiley, 2007 India.

Minter, Dave y Linwood Jeff. *Beginning Hibernate from Novice to Professional.* Apress, 2006 E.U.

Baver, Christian y King, Gavin. *Java Persistence with Hibernate.* Manning Publications Co., 2007 E.U.

Doray, Arnold. *Beginning Apache Struts from Novice to Professional.* Apress, 2006 E.U.

Sam-Bodden, Brian. *Beginning POJO's from Novice to Professional.* Apress, 2006 E.U.

Ceballos, Fco. Javier. *Java 2 Interfaces gráficas y aplicaciones para Internet.* Alfaomega RA-MA, segunda edición. 2006.

REFERENCIAS DE INTERNET

Tecnologías para la Integración de Bases de Datos en la Web, http://di.uca.edu.sv/investigacion/bdweb/tecnolog.html, fecha de consulta: 25 de mayo del 2009.

González, Carlos Daniel. *Fundamentos y conceptos básicos de Java, JSP y MySQL*, http://www.usabilidadweb.com.ar/javaCorporativo.php, fecha de consulta: 25 de mayo del 2009.

Fundamentos teóricos sobre la Persistencia de Objetos. Wikipedia, la enciclopedia libre, http://es.wikipedia.org/wiki/Persistencia_de_objetos, fecha de consulta: 25 de mayo del 2009.

Sánchez González, Carlos, *ONess: Fundamentos teóricos de las aplicaciones Web creadas con Herramientas Open Source.* http://oness.sourceforge.net/proyecto/html/ch03.html, fecha de consulta: 25 de mayo del 2009.

Mapeo Objeto Relacional, herramientas, funcionamiento y conceptos. Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Mapeo_objeto-relacional, fecha de consulta: 27 de mayo del 2009.

Elizalde Vieyra, Guadalupe. *Bases de datos relacionales.* http://www.fismat.umich.mx/~elizalde/tesis/node15.html, fecha de consulta 27 de mayo del 2009.

Chávez García, Carlos. Conceptos y Fundamentos sobre Bases de Datos. http://www.maestrosdelweb.com/principiantes/¿que-son-las-bases-de-datos/, fecha de consulta: 28 de mayo del 2009.

Características Generales del Framework de Persistencia Apache iBATIS. http://groups.google.es/group/masterenfwpa/web/ibatis?version=19, fecha de consulta: 3 de junio del 2009.

Estructura de los Ficheros de Configuración de Apache iBATIS. http://www.hachisvertas.net/jcs/wiki/index.php?title=Dao.xml, fecha de consulta: 16 de junio del 2009.

Smith, Adam. *Introducción al Framework de Persistencia Apache iBATIS.* http://www.javalobby.org/articles/ibatis-introduction/, fecha de consulta: 16 de junio del 2009.

Patil, Sunil*. Patrón de Diseño iBATIS DAO. Aspectos y uso general.* http://www.onjava.com/pub/a/onjava/2005/08/10/ibatisdao.html, fecha de consulta: 16 de junio del 2009.

Descripción y características generales del Framework de Persistencia Apache iBATIS. www.juntadeandalucia.es/xwiki/bin/view/MADEJA/iBatis, fecha de consulta: 16 de junio del 2009.

Guía de Uso del Framework de Persistencia Apache iBATIS http://kickjava.com/src/com/ibatis/dao/client/template/SqlMapDaoTemplate.java.htm*, fecha de consulta: 18 de junio del 2009.*

Comentarios y Puntos de Vista generales sobre los Frameworks de Persistencia. http://www.javisjava.com/blog/?p=50 , fecha de consulta: 21 de junio del 2009.

Guía de Uso del Framework de Persistencia Apache iBATIS. http://kickjava.com/src/com/ibatis/dao/client/DaoManagerBuilder.java.htm, fecha de consulta: 7 de julio del 2009.

El Framework de Persistencia Hibernate y las Herramientas ORM: Aspectos generales. http://www.trickyweb.cl/2009/09/18/hibernate-mapeo-objetorelacional-en-java/?wscr=1152x864, fecha de consulta: 7 de julio del 2009.

*Javi Sanromán. Uso y configuración general del Framework de Persistencia Hibernate. http://*www.jsanroman.net/.../curso-j2ee-3ª-semana-hibernate/*, fecha de consulta: 15 de julio del 2009.*

Aspectos Generales de una aplicación creada con el Framework de Persistencia Hibernate. http://jansoblog.wordpress.com/tag/hibernate/ -, fecha de consulta: 15 de julio del 2009.

*Java Hispano. Guía general sobre las características y uso del Framework de Persistencia Hibernate.*http://www.javahispano.org/ManualHibernate-Asociacion*javaHispano.htm, fecha de consulta: 15 de julio del 2009.*