PONTIFICIA UNIVERSIDAD CATOLICA DEL ECUADOR

FACULTAD DE INGENIERIA

ESCUELA DE SISTEMAS



DISERTACION PREVIA A LA OBTENCION DEL TÍTULO DE: INGENIERO DE SISTEMAS Y COMPUTACIÓN

PROPUESTA DE ARQUITECTURA DE SOFTWARE PARA EL DESARROLLO DE APLICACIONES EMPRESARIALES BASADAS EN JEE

NOMBRE:

DIEGO ANDRÉS HINOJOSA TINOCO

DIRECTOR: ING. FABIÁN DE LA CRUZ

QUITO, 2013

DEDICATORIA

El presente trabajo se lo dedico con mucho cariño a mis queridos padres Martha y Eloy, quienes me han enseñado que los éxitos en la vida son fruto del trabajo duro, del esfuerzo y sacrificio. Y es justamente gracias a este y muchos más consejos que he podido llegar hasta este punto en mi carrera.

También quiero dedicar esta disertación a una persona que es luz de Dios en mi familia, mi hermanita Karina.

Finalmente, cómo olvidar los consejos de aquellas personas que han formado parte de mi vida y que he conocido en mí caminar, este esfuerzo va dedicado también a Sebastián Vallejo, amigo fiel como pocos en este mundo y también a Verónica Cargua, novia, amiga, compañera, gran consejera.

AGRADECIMIENTOS

Agradezco por sobre todas las cosas a Dios por darme el don de la vida, por haberme permitido nacer en este mundo, por darme una familia tan maravillosa.

A mis padres por su abnegado esfuerzo y sacrificio en pos de entregarme la mejor educación y de formar a su hijo con los mejores valores y principios morales, no soy más que el fruto de que lo ustedes han sembrado. Gracias por sus consejos y por su apoyo mis queridos padres.

A mis compañeros de la universidad, el caminar en esta vida se hace más sencillo si lo recorres con personas valiosas, cada uno formó parte de mi vida y a su manera fueron importantes para mí, gracias por su compañerismo y su calidad de personas.

A mi gran amigo de toda la vida Sebastián Vallejo, por sus grandes consejos, por siempre estar conmigo, en las buenas y en las malas, cuando fallo, cuando triunfo, tu éxito es el mío también querido amigo, gracias por ser siempre sincero y fiel.

A mi novia Verónica Cargua, por apoyarme en todos mis ideales, mis proyectos, mis decisiones, gracias por compartir tu caminar junto a mí y por estar en los momentos felices y también en los difíciles de forma incondicional, mi ángel en el camino.

A mis profesores de la facultad, gracias por sus grandes enseñanzas, por su esfuerzo continuo al impartir y compartir sus conocimientos. Y en especial al Ing. Fabián de la Cruz, director de esta disertación, por el cual tengo una gran admiración y respeto, gracias por su apoyo y consejo.

Y en general a todas las personas que confiaron en mí y que caminan junto conmigo en la vida, muchas gracias.

TABLA DE CONTENIDOS

Pág

Introducción General i

1. Introducción 1
   1.1. Aplicaciones empresariales 1
   1.2. Plataforma JEE 3
      1.2.1. Definición 3
      1.2.2. Tecnologías JEE 3
      1.2.3. Clientes JEE 5
      1.2.4. Componentes Web 6
      1.2.5. Componentes de Negocio 7
      1.2.6. Contenedores JEE 8
      1.2.7. Empaquetado 9
      1.2.8. Roles de la plataforma JEE 10
      1.2.9. La arquitectura distribuida en JEE 11
      1.2.10. Arquitectura Java Naming Directory Interface (JNDI) 12
      1.2.11. Lectura de clases DTO 13
   1.3. Arquitectura de aplicaciones 13
   1.4. Escenarios 13
   1.5. Patrones arquitectónicos 14
      1.5.1. Descripción de un patrón 14
      1.5.2. Tipos de patrones 16
      1.5.3. Cualidades del software que propician los patrones arquitectónicos 17
   1.6. Patrones de diseño 18
      1.6.1. Patrones creacionales 18
      1.6.2. Patrones estructurales 20
   1.7. Patrones JEE 21
      1.7.1. Catálogo de patrones JEE 21

2. Diseño e implementación de la capa de persistencia 24
   2.1. Herramientas de mapeo objeto-relacional 25
   2.2. Introducción JPA (Java Persistence API) 26

2.3. Propuesta de la capa de persistencia 26
2.4. Hibernate 29
2.4.1. Arquitectura de Hibernate 30
2.4.2. Archivo de configuración de Hibernate 33
2.4.3. Configuración de Base de Datos 34
2.4.4. Mapeo de POJO's con Anotaciones JPA 39
2.4.4.1. Identidad 41
2.4.4.2. Configuración por defecto 42
2.4.4.3. Lectura temprana y lectura demorada 42
2.4.4.4. Tipos enumerados 43
2.4.4.5. Transient 44
2.4.4.6. Colecciones básicas 45
2.4.4.7. Tipos insertables 46
2.4.4.8. Tipos de acceso 46
2.4.4.9. Asociaciones 48
2.4.4.10. Asociaciones unidireccionales 49
2.4.4.11. Asociaciones bidireccionales 51
2.4.4.12. Lectura temprana y lectura demorada de asociaciones 52
2.4.4.13. Ordenación de asociaciones 52
2.4.4.14. Herencia 53
2.4.4.15. Mapped superclasses, clases abstractas y no-entidades 56
2.5. Integrando Data Access Objects (DAO's) y Hibernate con Spring 58
2.5.1. Spring 59
2.5.2. Implementando los Data Access Objects (DAO's) con Spring 65
2.6. Resumen de patrones aplicados 70
2.6.1. Frameworks utilizados 70

3. Diseño e implementación de la capa de negocio 72
3.1. Propuesta de la capa de negocio 73
3.2. Desarrollo y definición de objetos de negocio 73
3.3. Inyección de dependencias 74
3.4. Demarcación de transacciones 76
3.4.1. ¿Qué es un aspecto? 78

3.4.2. ¿Qué es un proxy dinámico? 78
3.5. Implementando las clases de servicio con Spring 79
3.6. Resumen de patrones aplicados 82
3.6.1. Frameworks utilizados 82

4. Diseño e implementación de la capa de presentación 83
4.1. Introducción al framework Java Server Faces (JSF) 83
4.1.1. Modelo Vista Controlador en JSF 85
4.1.2. Accediendo a un Bean desde una página JSF 89
4.1.3. Navegación entre páginas JSF 90
4.1.4. Etiquetas básicas 92
4.2. PrimeFaces 92
4.2.1. Configurando PrimeFaces 93
4.2.2. Etiquetas básicas 95
4.3. jQuery 96
4.4. Propuesta de la capa de presentación 99
4.5. Resumen de patrones aplicados 103
4.5.1. Frameworks utilizados 104

5. Integración de las capas propuestas 106
5.1. Elección del Servidor de Aplicaciones 122
5.2. Resumen de patrones aplicados 123

6. Conclusiones y Recomendaciones 125
Bibliografía 129

INDICE DE FIGURAS

1. Introducción Pág
1.1. Componentes de una aplicación JEE 4
1.2. Comunicación entre componentes Web 6
1.3. Comunicación entre componentes de negocio 7
1.4. Contenedores de aplicaciones cliente 9
1.5. Archivos que conforman una aplicación JEE 10
1.6. Arquitectura distribuida de las aplicaciones JEE 11
1.7. Utilización de JNDI en las aplicaciones JEE 12
1.8. Patrón de diseño MVC 19
2. Diseño e implementación de la capa de persistencia
2.1. Sentencias SQL directas desde el código 27
2.2. Arquitectura DAO 27
2.3. Capa de persistencia Hibernate 28
2.4. Propuesta de Capa de persistencia 29
2.5. Integración de Hibernate y las aplicaciones Java 30
2.6. Arquitectura de Hibernate 31
2.7. Módulos que conforman Spring 61
3. Diseño e implementación de la capa de negocio
3.1. Propuesta de Capa de persistencia 73
3.2. Inyección de Dependencias 75
3.3. Aspecto Transaccional 78
4. Diseño e implementación de la capa de presentación
4.1. Estructura de archivos de una aplicación JSF 85
4.2. Patrón Modelo Vista Controlador de JSF 85
4.3. Ejemplo del componente Calendar de PrimeFaces 94
4.4. Panel de ThemeRoller 97
4.5. Temas disponibles para PrimeFaces 98
4.6. Propuesta de capa de presentación 99
5. Integración de las capas propuestas
5.1. Propuesta de la arquitectura completa 107
5.2. Diagrama conceptual de la tabla Contacto 108
5.3. Diagrama de clase entidad Contacto 108

5.4. Diagrama de clases general 109
5.5. Diagrama de secuencia guardar contacto 109
5.6. Configuración de pool de conexiones GlassFish 118
5.7. Configuración de nombre JNDI con conjunto de conexiones 119
5.8. Estructura de archivos de la aplicación 121
5.9. Pantalla de inicio sin registros listContactos.xhtml 121
5.10. Pantalla de ingreso de contacto crearContacto.xhtml 122
5.11. Pantalla de inicio después de guardar contacto listContacto.xhtml 122

Introducción general

La arquitectura de software es una pieza central en el desarrollo de productos de software modernos. El objetivo de la arquitectura consiste en desarrollar sistemas grandes de forma eficiente, estructurada y con capacidad de reúso. La arquitectura forma parte del proceso de diseño de software el cual también forma parte del proceso de desarrollo de software que comprende, requerimientos, diseño, implementación, prueba y mantenimiento. Debido a la complejidad de las aplicaciones empresariales, la fase de diseño, donde se define la arquitectura de estos sistemas, es fundamental para garantizar que cumpla con los principios de cohesión y acoplamiento.

Las aplicaciones Web empresariales pueden desarrollarse utilizando cualquier arquitectura posible. Es por tal razón que existe una gran variedad de patrones de diseño y construcción de software.

Es aquí donde las propuestas arquitectónicas se confunden en un lazo al igual que la enorme cantidad de herramientas y frameworks que al final terminan confundiendo al arquitecto de software sobre qué camino tomar para diseñar su sistema, como por ejemplo cuántas capas se ha de utilizar, la relación entre éstas, si se trabajará o no a base de frameworks, etc.

Sería desastroso pensar que, una vez teniendo construido la arquitectura del sistema empresarial, en el momento de la fase de desarrollo el arquitecto y los programadores se den cuenta que no es posible relacionar los componentes de las diversas capas y peor aún, que los frameworks que se escogieron no son compatibles entre sí y en fin, un sin número más de complicaciones que terminará llevando al arquitecto a rediseñar la estructura del sistema.

Para combatir esta problemática, el presente trabajo sugiere una arquitectura genérica basada en JEE como plataforma de desarrollo empresarial, con la integración de frameworks, como alternativa ya sea de base para, a partir de ésta, generar arquitecturas más complejas de acuerdo con las características del sistema o como una buena estructura para cualquier sistema empresarial.

Visión del resto del documento

El primer capítulo nos introduce al mundo de la arquitectura de software, sobre todo a aquella que se utiliza para cimentar una aplicación empresarial.

El capítulo 2 describirá la propuesta para la capa de persistencia y el uso de las herramientas y patrones elegidos dentro de ella.

El capítulo 3 describirá la propuesta para la capa de negocio y el uso de su respectivo patrón con servicios.

El capítulo 4 describirá la propuesta para la capa de presentación donde veremos en acción algunas tecnologías trabajando en equipo para crear una interface de usuario rica tanto visual y funcional.

El capítulo 5 mostrará la integración de las propuestas de las capas de los capítulos 2, 3 y 4 para ponerlas en funcionamiento y demostrar que la arquitectura resultante es consistente y funcional.

Finalmente en el capítulo 6 se recogerán las conclusiones y recomendaciones producidas por este trabajo.

CAPITULO I

INTRODUCCIÓN

Como su nombre lo indica, este capítulo nos introduce al mundo de la arquitectura de software, sobre todo a aquella que se utiliza para cimentar una aplicación empresarial. Se tratará sobre cuál es el panorama actual de dichas aplicaciones y cuáles son las plataformas de desarrollo más importantes que están a elección de los arquitectos de software.

Este trabajo se enfoca en la plataforma JEE como pilar fundamental para este tipo de aplicaciones, de la cual se verá una breve introducción a sus características y su arquitectura.

Daremos un repaso a lo que es la base de las distintas arquitecturas a través de patrones, de los cuales destacamos tres principales: de arquitectura, de diseño y de JEE.

1.1 Aplicaciones Empresariales

Una posible definición abstracta de una arquitectura empresarial sería: El estudio de sistemas empresariales complejos desde el punto de vista de su estructura. Un arquitecto empresarial debe ser capaz de estudiar un problema en concreto y de escoger una serie de componentes con los que pueda modelar la arquitectura más adecuada para el problema en cuestión. Dichos componentes pueden ser servidores de aplicaciones, contenedores web, servidores de mensajería, frameworks, etc. El arquitecto también debe ser capaz de establecer la manera en la que van a interactuar dichos componentes, las herramientas utilizadas y las relaciones existentes entre los mismos.

La labor más complicada y con mayor responsabilidad para un arquitecto empresarial es la elección de la plataforma empresarial sobre la que cual se cimentará la arquitectura de una empresa. Una elección errónea puede tener resultados catastróficos tanto para la empresa como para el responsable de dicha elección.

Una plataforma de desarrollo empresarial debe ofrecer una serie de servicios a los arquitectos y desarrolladores encaminados a facilitar el desarrollo de aplicaciones empresariales, al

tiempo que ofrece la mayor cantidad posible de funcionalidades a los usuarios. Normalmente una plataforma de desarrollo empresarial tiene los siguientes requisitos:

- Escalabilidad: Ofrecer una buena escalabilidad tanto horizontal como vertical de modo que si aumenta la carga del sistema podamos añadir servidores o ampliar los existentes sin que sea necesario realizar modificaciones.
- Mantenibilidad: Permitir añadir o modificar los componentes existentes sin modificar el comportamiento del sistema.
- Disponibilidad: Tener el soporte de arquitecturas a prueba de fallos, sistemas de redundancia, etc., que nos aseguren que nuestro sistema estará siempre disponible.
- Extensibilidad: Será posible añadir nuevos componentes y capacidades al sistema sin que se vean afectados el resto de componentes.
- Manejabilidad: Nuestros sistemas deben ser fácilmente manejables y configurables.
- Seguridad: Tener buenos sistemas de seguridad tanto a nivel de autenticación, como de autorización y como de transporte.
- Rendimiento: Ofrecer automáticamente soporte de clustering, balanceo de carga, pools de objetos, pools de conexiones, cachés, y en general mecanismos que permitan aumentar el rendimiento de manera transparente al usuario.

La importancia de una plataforma empresarial es que todos estos componentes se nos ofrecen de manera automática de modo que los desarrolladores son mucho más productivos. La diferencia entre utilizar una plataforma de desarrollo empresarial y no utilizarla radica en que en el segundo caso nuestros desarrolladores perderán mucho tiempo realizando sistemas de bajo nivel, no pudiéndose centrar en el desarrollo de aplicaciones y por consiguiente disminuyendo considerablemente la productividad de los mismos.

En la actualidad las plataformas más importantes son aquellas que disponen del apoyo de gran cantidad de empresas, entidades o asociaciones y que disponen de grupos de estandarización que aseguren el futuro de las mismas. En la actualidad dos son las plataformas de desarrollo empresarial más importantes:

- .NET
- JEE

1.2 Plataforma JEE

1.2.1 Definición

Java Enterprise Edition (JEE) es una arquitectura multicapa para implementar aplicaciones de tipo empresarial y aplicaciones basadas en la Web. Esta tecnología soporta una gran variedad de tipos de aplicaciones desde aplicaciones Web de gran escala a pequeñas aplicaciones cliente-servidor. El objetivo principal de la tecnología JEE es crear un simple modelo de desarrollo para aplicaciones empresariales utilizando componentes basados en el modelo de aplicación. En este modelo dichos componentes utilizan servicios proporcionados por el contenedor, que de otro modo tendrían que estar incorporados en el código de la aplicación. Podría no ser lo ideal para todos los escenarios: por ejemplo, una pequeña aplicación se cubriría mejor utilizando una solución de la tecnología Java de peso ligero (por ejemplo Servlets, JSPs, etc.). Actualmente se encuentra en la versión JEE 6 lanzada el 10 de Diciembre del 2009.

1.2.2 Tecnologías JEE

Las aplicaciones JEE están compuestas de diferentes componentes. Un componente JEE es una unidad de software funcional autocontenido que se ensambla dentro de una aplicación JEE con sus clases de ayuda y archivos y que se comunica con otros componentes de la aplicación. La especificación JEE define los siguientes componentes JEE:

- Las Aplicaciones Clientes y los Applets son componentes que se ejecutan en el lado del cliente.
- Los componentes Java Servlet, la tecnología JavaServer Pages son componentes Web que se ejecutan en el lado del servidor.
- Los Enterprise JavaBeans (EJB) son componentes de negocio que se ejecutan en el servidor de aplicación.

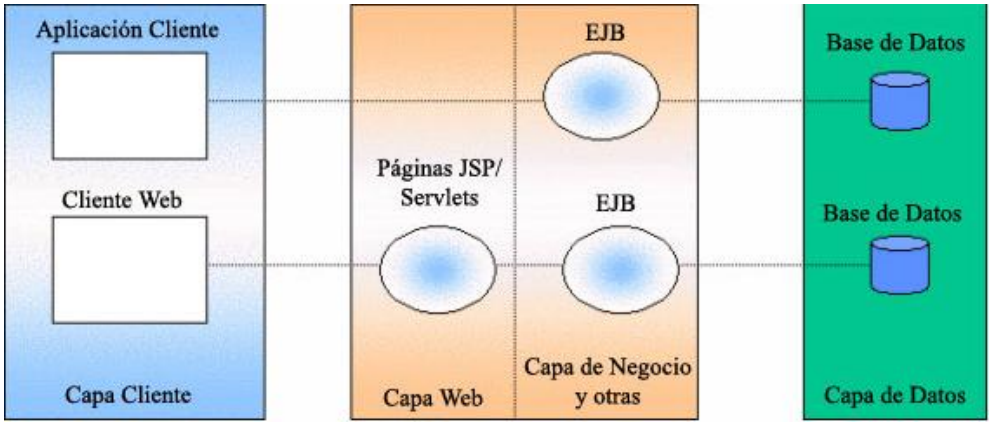


Figura 1.1 Componentes de una aplicación JEE.

Fuente: http://www.globaxon.com/j2ee_technology_expertise.htm

Todos esos componentes se ensamblan en una aplicación JEE, se verifica que están bien formados y que cumplen la especificación JEE, y se despliegan en el entorno de producción, donde se ejecutan y son controlados por el servidor de aplicaciones JEE.

Además de estos componentes principales, JEE incluye servicios estándar y tecnologías de soporte como:

- **Java Database Connectivity API (JDBC):** Tecnología que proporciona acceso a sistemas de bases de datos relacionales para la ejecución de sentencias SQL desde métodos de código JAVA.
- **Java Transaction API (JTA) o Java Transaction Service (JTS):** Es el API para manejo de transacciones a través de sistemas heterogéneos. Trabaja en la capa Web y en la capa de Negocios.
- **Java Message Service API (JMS):** Es el API para comunicación asíncrona entre componentes JEE. Trabaja en la capa de Cliente, Web y de Negocios.
- **Java Naming and Directory Interface (JNDI):** Proporcionan accesos a nombres y directorios.
- **JavaMail API:** API para el envío y recepción de correo electrónico. Trabaja en la capa Web y en la capa de Negocios.
- **Java API for XML Processing JAXP:** Para el tratamiento de documentos XML.
- **Java Persistence API (JPA):** Para la relación entre entidades Java y tablas de la Base de Datos. Trabaja en la capa del medio.

- **Java Authentication and Autorization Service (JAAS):** Permite la autenticación y autorización a usuarios o grupo de usuarios.

1.2.3 Clientes JEE

Normalmente hay dos tipos de clientes JEE: clientes Web y aplicaciones cliente como vimos en la figura anterior.

Un cliente Web consta de dos partes, páginas Web dinámicas que contienen distintos tipos de lenguajes de marcas (HTML, XML, y otros), que son generados por los componentes Web que se ejecutan en la capa Web, y un navegador Web, que dibuja las páginas recibidas desde el servidor. Otra categoría de clientes Web son los conocidos como clientes thin (delgados). Este tipo de clientes delgados normalmente no hacen cosas como consultas a bases de datos o ejecutar complejas reglas de negocio. Cuando se utilizan clientes delgados, las operaciones de peso pesado las manejan los Enterprise Bean que se ejecutan en el servidor JEE donde pueden tratar con la seguridad, los servicios y el rendimiento de las tecnologías del lado del servidor JEE.

Una página Web recibida desde la capa del cliente puede incluir un applet embebido. Un applet es una pequeña aplicación cliente escrita en Java que se ejecuta en la máquina virtual Java instalada en el navegador Web. Sin embargo, los sistemas cliente necesitarán el Plug-in Java y posiblemente un archivo de política de seguridad para poder ejecutar con éxito los applets en el navegador Web.

Normalmente los componentes Web son el API preferido para crear programas clientes Web porque no necesitan plugins ni archivos de política de seguridad en los sistemas clientes. Además esto permite un diseño más claro y modular de la aplicación porque proporciona un significado a la separación de la lógica de la aplicación del diseño de la página Web.

Una aplicación cliente se ejecuta sobre una máquina cliente y proporciona una forma para que los usuarios puedan manejar tareas que requieren un interface de usuario más rico que el que puede proporcionar un lenguaje de marcas. Normalmente tienen una interface gráfica de usuario (GUI) creada con los APIs Swing o Abstract Window Toolkit (AWT).

Las aplicaciones cliente acceden directamente a los Enterprise Beans que se ejecutan en la capa de negocio. Pero si se necesita un cliente Web pueden abrir una conexión HTTP para establecer comunicación con un servlet que se ejecute en la capa Web.

1.2.4 Componentes Web

Los componentes Web de JEE pueden ser servlets o páginas JSP. Los servlets son clases Java que procesan dinámicamente las peticiones y construyen las respuestas. Las páginas JSP son documentos basados en texto que se ejecutan como servlets pero permiten una aproximación más natural para crear contenido estático. Las páginas HTML y los applets se juntan con los componentes Web durante el ensamble de la aplicación, pero la especificación JEE no los considera como componentes JEE. De forma similar, las clases de utilidades del lado del servidor también se unen a los componentes Web como páginas HTML, pero tampoco se consideran como componentes JEE. En la Figura 1.2 podemos ver la comunicación entre componentes Web:

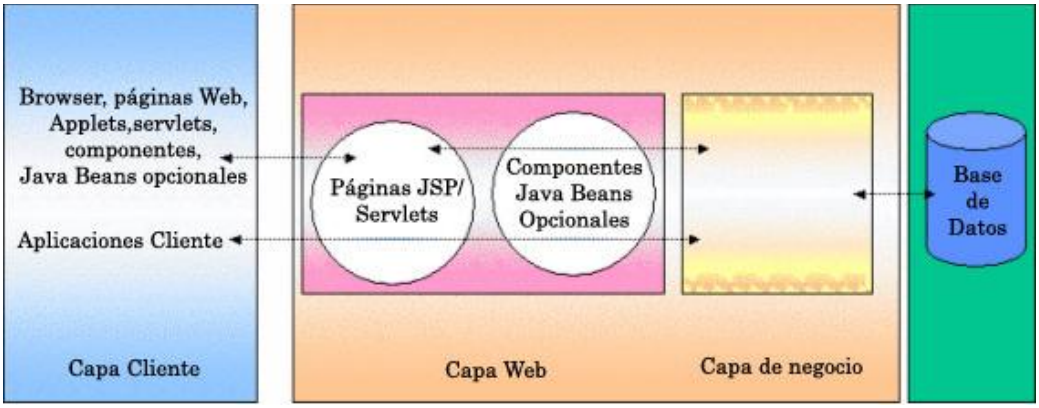


Figura 1.2: Comunicación entre componentes Web.

Fuente:http://www.programacion.com/articulo/construir_aplicaciones_ejb_con_jboss-_lomboz_y_eclipse_267/3

La capa Web podría incluir componentes Java Beans para manejar la entrada del usuario y enviar esta entrada a los Enterprise Beans que se ejecutan en la capa de negocio para su procesamiento como se observa en la figura anterior.

1.2.5 Componentes de Negocio

El código de negocio, que es la lógica que resuelve o cumple las necesidades de un negocio particular, como la banca, la venta, o la financiación, se maneja mediante Enterprise Beans que se ejecutan en la capa de negocio.

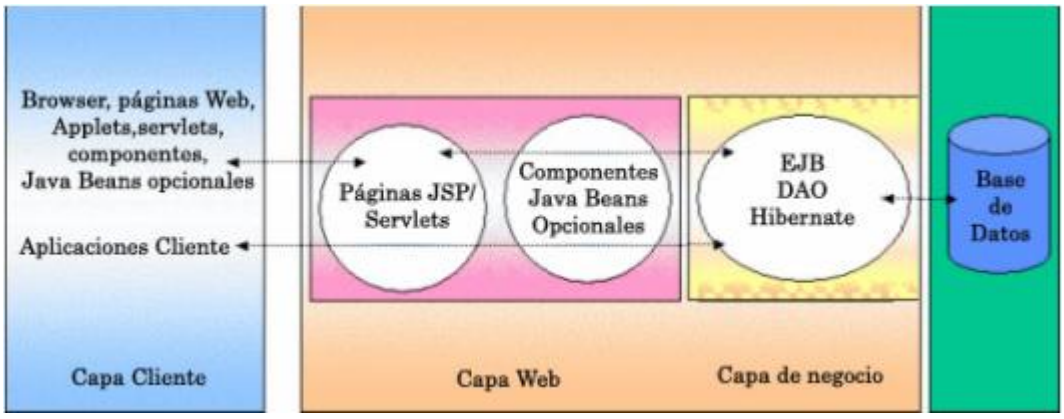


Figura 1.3 Comunicación entre componentes de negocio.

Fuente:http://www.programacion.com/articulo/construir_aplicaciones_ejb_con_jboss-_lomboz_y_eclipse_267/3

La Figura 1.3 muestra la comunicación entre los componentes de negocio, donde un Enterprise Bean recibe datos de los programas clientes, los procesa (si es necesario), y los envía a la capa del sistema de información empresarial para su almacenamiento. Un Enterprise Bean también recupera datos desde el almacenamiento, los procesa (si es necesario), y los envía de vuelta al programa cliente.

Hay tres tipos de Enterprise Beans: Beans de sesión (con o sin estado), Beans de entidad (manejados por el Bean o por el contenedor) y Beans dirigidos a mensaje. Un Bean de sesión representa una conversación temporal con un cliente. Cuando el cliente finaliza su ejecución, el Bean de sesión y sus datos desaparecen. Por el contrario, un Bean de entidad representa datos persistentes almacenados en una fila de una tabla/relación de una base de datos.

Si el cliente se termina o si se apaga el servidor, los servicios subyacentes se aseguran de grabar el Bean. Un Bean dirigido-a-mensaje combina las características de un Bean de

sesión y de un oyente de Java Message Service (JMS), permitiendo que un componente de negocio reciba asincrónicamente mensajes JMS.

La especificación JEE no considera como componentes JEE a los Java Beans ya que son diferentes de los Enterprise Beans. La arquitectura de componentes Java Beans se pueden utilizar tanto en la capa de cliente como de servidor para manejar la comunicación entre una aplicación cliente o un applet y los componentes que se ejecutan en el servidor JEE o entre los componentes del servidor y una base de datos, mientras que los componentes Enterprise Java Beans sólo se utilizan en la capa de negocio como parte de una capa de servidor. Los Java Beans tienen variables de ejemplar y métodos accesores y mutadores para acceder a las propiedades del Bean o digamos, acceso a los datos en las variables de ejemplar lo que simplifica el diseño y la implementación de los componentes Java Beans.

1.2.6 Contenedores JEE

Los contenedores JEE proporcionan acceso a los servicios subyacentes del entorno del Servidor JEE mediante contenedores para diferentes tipos de componentes. Tradicionalmente, los desarrolladores de aplicaciones tenían que escribir código para el manejo de transacciones, manejo del estado, multi-threads, almacenamiento de recursos, etc. Ahora el contenedor JEE proporciona estos servicios permitiendo que te puedas concentrar en resolver los problemas de negocio.

Los contenedores son la interface entre un componente y la funcionalidad de bajo nivel específica de la plataforma que soporta el componente. Por ejemplo, antes de poder ejecutar un componente Web, un Enterprise Bean o un componente de una aplicación cliente, debe ensamblarse dentro de una aplicación JEE y desplegarse dentro de su contenedor.

El proceso de ensamble implica especificar las configuraciones del servidor para cada componente de la aplicación JEE y para la propia aplicación JEE. Estas configuraciones personalizan el soporte subyacente proporcionado por el servidor JEE, que incluye servicios como JNI, JNDI, seguridad, control de transacciones, etc.

El servidor JEE proporciona contenedores para Enterprise Java Beans (EJB) y para componentes Web. El contenedor EJB maneja la ejecución de los Enterprise Beans de las aplicaciones JEE, mientras que el contenedor Web maneja la ejecución de las páginas JSP y los componentes servlets de la aplicación JEE. Otros contenedores distintos a estos son el contenedor de aplicaciones clientes y el contenedor de applets, que no son parte del servidor JEE porque residen en la máquina del cliente, como se muestra en la Figura 1.4.

Un contenedor de aplicaciones cliente maneja la ejecución de los componentes de la aplicación cliente mientras que un contenedor de Applets maneja la ejecución de los applets. Normalmente están en el JRE (Java Runtime Environment) y el navegador Web compatible con Java, respectivamente.

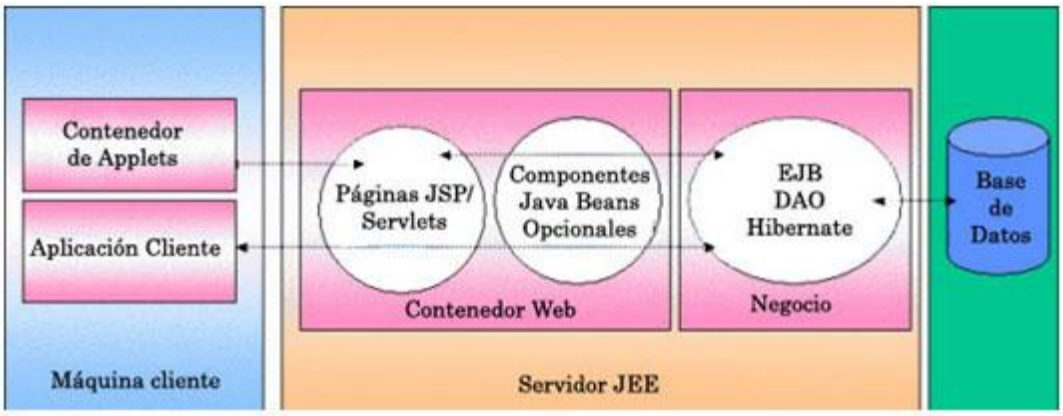


Figura 1.4 Contenedores de aplicaciones cliente.

Fuente:http://www.programacion.com/articulo/construir_aplicaciones_ejb_con_jboss-_lomboz_y_eclipse_267/3

1.2.7 Empaquetado

Para poder desplegar una aplicación JEE, después de desarrollar los diferentes componentes, se empaqueta en un contenedor de archivos especiales que contienen los archivos de las clases relevantes y los descriptores de despliegue XML. Estos descriptores de despliegue contienen información específica de cada componente empaquetado y son un mecanismo para configurar el comportamiento de la aplicación en el momento del ensamble o del despliegue. Estos se empaquetan en diferentes tipos de archivos según los distintos componentes.

Los componentes Web se empaquetan en un archivo Web (.war) que contiene los servlets, las páginas JSP y los componentes estáticos como las páginas HTML y las imágenes. El empaquetado .war contiene clases y archivos utilizados en la capa Web junto con un descriptor de despliegue de componentes Web.

Los componentes de negocio se empaquetan en un archivo Java (.jar) que contiene los descriptores de despliegue EJB, los archivos de interface remoto y de objeto junto con archivos de ayuda requeridos por el componente EJB.

Los archivos de clases del lado del cliente y los descriptores de despliegue se empaquetan en un archivo Java (.jar) que crea la aplicación cliente. Una aplicación JEE se empaqueta en un archivo enterprise (.ear) que contiene toda la aplicación junto con el descriptor de despliegue que proporciona información sobre la aplicación y sus componentes, como se puede apreciar en la Figura 1.5.

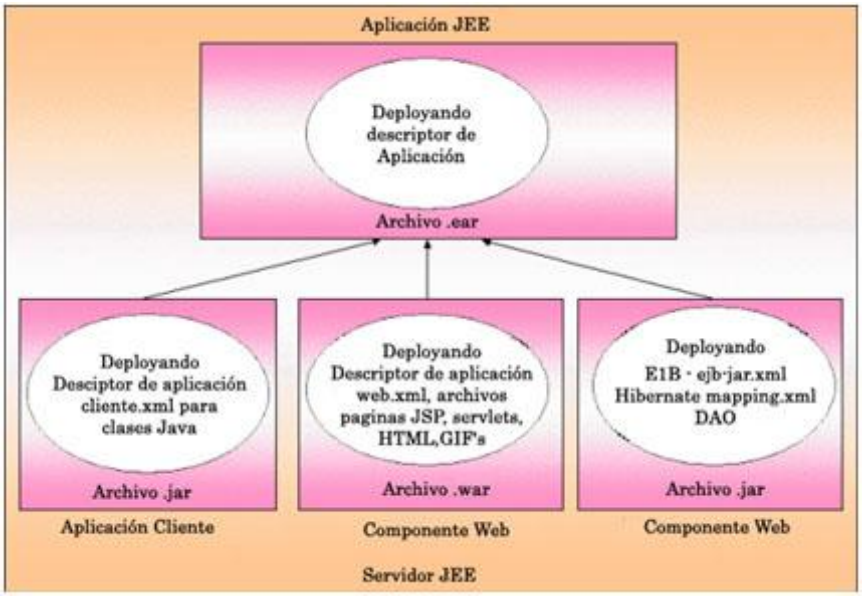


Figura 1.5: Archivos que conforman una aplicación JEE.

Fuente:http://www.programacion.com/articulo/construir_aplicaciones_ejb_con_jboss-_lomboz_y_eclipse_267/3

1.2.8 Roles de la Plataforma JEE

La construcción de los diferentes componentes de una aplicación JEE implica a varios roles en el desarrollo, despliegue y control de una aplicación empresarial:

- El proveedor de componentes de aplicación desarrolla componentes JEE reutilizables, que pueden ser componentes Web, Enterprise Beans, applets, o aplicaciones clientes para utilizar en aplicaciones JEE.
- El ensamblador de aplicaciones toma todos los bloques de los diferentes proveedores de componentes y los combina en aplicaciones JEE.
- El desarrollador es el responsable de la instalación/despliegue de los componentes en un entorno o servidor JEE.
- El administrador del sistema es el responsable de configurar y administrar los sistemas informáticos en una empresa.
- El proveedor de herramientas es un vendedor utilizado para desplegar, empaquetar y desplegar aplicaciones JEE.

1.2.9 La Arquitectura Distribuida en JEE

Todas las aplicaciones JEE implementan una arquitectura distribuida. En ésta, un objeto está asociado con un nombre, donde los nombres los proporciona un servicio de nombres, notificando a distintos componentes y resolviendo las referencias de clientes para estos componentes de servicio como se muestra en la siguiente figura:

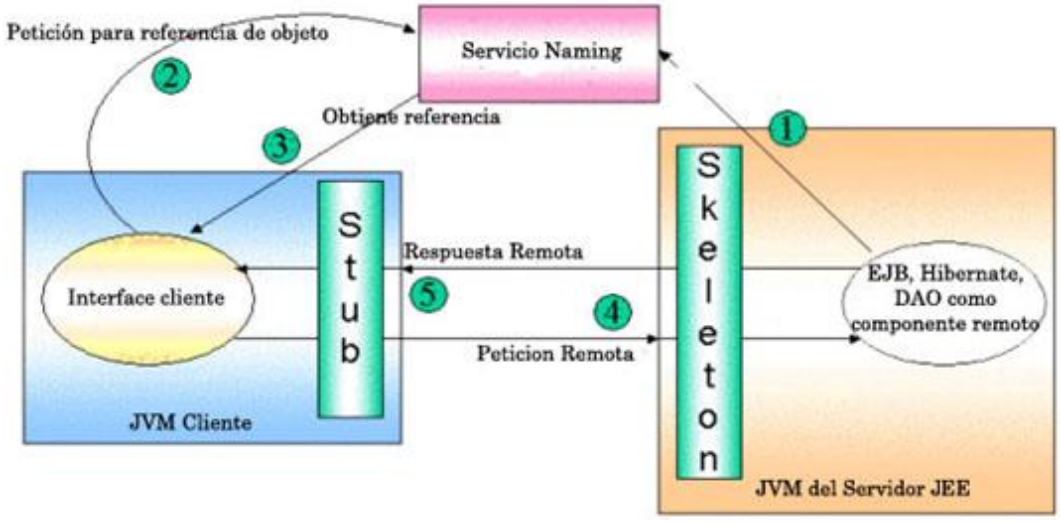


Figura 1.6: Arquitectura distribuida de las aplicaciones JEE.

Fuente:http://www.programacion.com/articulo/construir_aplicaciones_ejb_con_jboss-_lomboz_y_eclipse_267/3

Como resultado de ésto, las referencias de objetos se obtienen buscando un objeto por su nombre notificado, una vez encontrado, se obtiene la referencia, y se llevan a cabo las operaciones necesarias sobre ese objeto utilizando los servicios del host. Un objeto remoto notifica su disponibilidad en el servicio de nombres utilizando un nombre lógico y el servicio de nombres lo traduce a la localización física del objeto en el entorno JEE. Una vez que la petición del cliente obtiene una referencia a un componente remoto, puede enviarle peticiones. El sistema de ejecución maneja la comunicación distribuida entre objetos remotos, lo que incluye la serialización y deserialización de parámetros.

Algunos de los sistemas de nombres utilizados en los sistemas distribuidos son RMI (sólo para implementaciones Java), CORBA, LDAP, DNS, NIS. El servidor de aplicaciones JBOSS utiliza RMI como su servicio de nombres.

1.2.10 La Arquitectura Java Naming Directory Interface (JNDI)

JEE utiliza el API JNDI para acceder genéricamente a servicios de nombrado y directorio utilizando la tecnología Java. El API JNDI reside entre la aplicación y el servicio de nombres y hace que el servicio de nombres subyacente sea transparente para los componentes de la aplicación, como se puede ver en la figura siguiente.

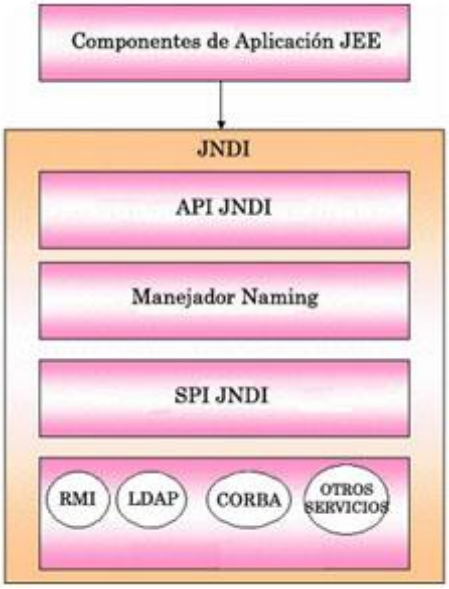


Figura 1.7: Utilización de JNDI en las aplicaciones JEE.

Fuente:http://www.programacion.com/articulo/construir_aplicaciones_ejb_con_jboss-_lomboz_y_eclipse_267/3

Un cliente puede buscar referencias a componentes EJB u otros recursos en un servicio de nombres como el mencionado arriba. El código del cliente no se modifica, sin importar el servicio de nombres que se esté utilizando o en qué tecnología esté basada, y esto no crea ninguna diferencia en el modo en que los clientes localizan los objetos remotos mediante el API JNDI.

1.2.11 Lectura de clases DTO

Un objeto DTO es un POJO, que va a servir para el transporte de datos entre las capas de una aplicación, de allí sus iniciales DTO (Data Transfer Object).

1.3 Arquitectura de aplicaciones

Este es un término usado al diseñar aplicaciones, particularmente del tipo cliente-servidor. Esta arquitectura se refiere a la manera en la que es diseñada tanto física como lógicamente una aplicación.

En el diseño físico se especifica exactamente donde se encontrarán las piezas de la aplicación como discos, ejecutables, cable de red y computadoras.

En el diseño lógico o conceptual se especifica la estructura de la aplicación y sus componentes sin tomar en cuenta dónde se localizará el software, hardware e infraestructura. Tales conceptos incluyen el orden de procesamiento, mantenimiento y seguimiento comunes en sistemas organizacionales.

Muchas veces se toma demasiado en cuenta el diseño físico de una aplicación. Por añadidura los desarrolladores generalmente asumen, indebidamente, que el diseño lógico corresponde punto a punto con el diseño físico. Contrario a ésto, un diseño adecuado debería permitir su implantación en varias plataformas y configuraciones. Como se puede ver, esta característica de portabilidad es un punto deseable para permitir que una aplicación sea flexible y escalable.

1.4 Escenarios

Un escenario es una breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema con éste. Por ejemplo, un usuario hará la descripción en términos de la ejecución de una tarea mientras que un encargado de mantenimiento hará referencia a

cambios que deban realizarse sobre el sistema y finalmente un desarrollador se enfocará en el uso de la arquitectura para efectos de su construcción o predicción de su desempeño.

Un escenario consta de tres partes: el estímulo, el contexto y la respuesta. El estímulo es la parte del escenario que explica o describe lo que el involucrado en el desarrollo hace para iniciar la interacción con el sistema. Puede incluir ejecución de tareas, cambios en el sistema, ejecución de pruebas, configuración, etc. El contexto es el que describe qué sucede en el sistema al momento del estímulo. La respuesta describe, a través de la arquitectura, cómo debería responder el sistema ante el estímulo. Este último elemento es el que permite establecer cuál es el atributo de calidad asociado.

Los escenarios proveen un vehículo que permite concretar y entender atributos de calidad. Su uso es importante para la evaluación de arquitecturas de software.
Entre las ventajas de su uso están:

- Son simples de crear y entender.
- Son poco costosos y no requieren mucho entrenamiento.
- Son efectivos.

1.5 Patrones arquitectónicos

Son patrones de software los cuales se encargan de definir la estructura de un sistema, estos a su vez se componen de subsistemas con sus responsabilidades, también tienen una serie de directivas para organizar los componentes del mismo sistema, con el objetivo de facilitar la tarea del diseño de tal sistema. Un patrón arquitectónico se enfoca a dar solución a un problema en específico, de un atributo de calidad, y abarca solo parte de la arquitectura.

Aun cuando un patrón arquitectónico transporta una imagen de un sistema, no es parte de la arquitectura como tal. Un patrón arquitectónico es un concepto que captura elementos esenciales de una arquitectura de software. Diversas arquitecturas pueden poner el mismo patrón en ejecución y de tal modo compartir las mismas características.

1.5.1 Descripción de un patrón

Para describir un patrón las notaciones gráficas no son suficientes. Para que el patrón pueda ser reutilizado se necesita representar las decisiones, alternativas, ventajas e

inconvenientes que son su razón de ser. Normalmente, un patrón está documentado en forma de una plantilla. Es una práctica común documentar los patrones en un formato de plantilla, pero no significa que sea la única forma de hacerlo. Además, existen muchos formatos de plantillas propuestos por muchos autores, lo cual permite la creatividad en la documentación de patrones. A continuación se presentan como referencia los elementos de descripción de patrones propuesto por Gamma [GA95]:

- Nombre del patrón: Debe ser un nombre corto y significativo, generalmente una o dos palabras. El nombre pasará a formar parte de nuestro vocabulario de diseño.
- Clasificación: Para el tipo del patrón no hay una clasificación formal pero suelen agruparse en patrones de creación, estructura, comportamiento, concurrencia, etc.
- Propósito: Representado por una frase breve que describe el problema concreto de diseño y que hace el patrón para resolverlo.
- Motivación: Describe el escenario que ilustra el problema de diseño y como las estructuras de clases y objetos del patrón resuelven el problema.
- Aplicabilidad: Describe en que situaciones se puede aplicar el patrón, ejemplos y formas de reconocer tales situaciones.
- Estructura: Se hace mediante una representación gráfica del patrón que muestren sus elementos y relaciones constitutivas.
- Participantes: Especificación de las clases y objetos que consta el patrón incluyendo las respectivas responsabilidades.
- Colaboraciones: Representación de cómo colaboran los participantes para cumplir con sus responsabilidades.
- Consecuencias: Se especifican las ventajas e inconvenientes a los que conlleva usar el patrón.
- Implementación: Descripción de las dificultades, trucos o técnicas que deberíamos tener presentes al momento de aplicar el patrón.
- Código de ejemplo: Especificación de código que ejemplifique la forma como debemos implementar el patrón.
- Usos conocidos: Descripción de ejemplos donde haya sido utilizado el patrón.
- Patrones relacionados: Especificaciones de otros patrones con los cuales esté relacionado, las principales diferencias y los patrones con los que se debería usar.

1.5.2 Tipos de patrones

La clasificación de los patrones no está estandarizada, pero la mayoría de autores suele referirse a los siguientes tipos:

- Fundamentales: Los patrones de esta categoría son los más fundamentales e importantes patrones de diseño conocidos. Estos patrones son utilizados extensivamente en otros patrones de diseño.
- De Creación (Creational): Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades.
- De partición: En la etapa de análisis, se examina el problema para identificar los actores, casos de uso, requerimientos y las relaciones que constituyen el problema. Los patrones de esta categoría proveen la guía sobre cómo dividir actores complejos y casos de uso en múltiples clases.
- Estructura (Structural): Describen la forma como se pueden relacionar, diferentes tipos de objetos, para trabajar unos con otros y formar estructuras de mayor tamaño.
- Conducta (Behavioral): Describen la forma cómo organizar, administrar, y combinar, conductas y responsabilidades de objetos.
- Concurrencia (Concurrency patterns): Describen como coordinar operaciones concurrentes para compartir recursos o secuenciar dichas operaciones.
- Orientaciones de aplicación: Por la naturaleza de la idea de los patrones, éstos solucionan problemas que existen en muchos niveles de abstracción.

Los patrones inicialmente fueron aplicados en la fase de diseño de los sistemas de información, por la necesidad respectiva que se tenía en este ámbito. Sin embargo existen otros ámbitos de la ingeniería del software donde se puede aplicar el concepto genérico de patrón.

Hay patrones que describen soluciones para todo, desde el análisis hasta el diseño y desde la arquitectura hasta la implementación. Además, los patrones existen en diversas áreas de interés y tecnologías. Por ejemplo se muestran algunos:

- Patrones organizativos: Describen la estructura y prácticas de las organizaciones humanas, especialmente las productoras de software.
- Patrones de análisis: Describen un conjunto de prácticas destinadas a elaborar modelos de los conceptos principales de la aplicación que se va a construir. La intención principal de estos patrones es ayudar a las personas que realizan trabajo de modelado, pues no siempre tienen experiencia al respecto y, en la mayoría de los casos, construyen sus modelos sin referencia alguna.
- Patrones de arquitectura: Expresan un paradigma fundamental para estructurar u organizar un sistema software. Proporcionan un conjunto de subsistemas o módulos predefinidos, con reglas y guías para organizar las relaciones entre ellos. Ejemplo:
    - Capas (Layers)
    - Aplicaciones: JVM, API, Windows NI
    - Pipes and Filters
    - Aplicaciones: UNIX
    - Pizarrón (Blackboard)
    - Aplicaciones: Hearsay, Inteligencia Artificial
- Patrones de diseño: Proporciona un esquema para refinar los subsistemas o componentes de un sistema software y las relaciones entre ellos. Describe estructuras recurrentes de comunicar componentes que resuelven un problema de diseño en un contexto particular. Son patrones de un nivel de abstracción menor que los patrones de arquitectura. Están por lo tanto más próximos a lo que sería el código fuente final. Su uso no se refleja en la estructura global del sistema.
- Patrones de programación (Idioms patterns): Un idioma es un patrón de bajo nivel, específico a un lenguaje de programación determinado. Describe como implementar aspectos particulares de los componentes de un patrón de diseño usando las características y potencialidades de un lenguaje de programación concreto.

1.5.3 Cualidades del software que propician los patrones arquitectónicos

Uno de los aspectos más importantes de patrones arquitectónicos es que incorporan diversas cualidades de la calidad. Por ejemplo, algunos patrones representan soluciones a los problemas de funcionamiento y otros se pueden utilizar con éxito en sistemas de alta

disponibilidad. En la fase de diseño temprana, un arquitecto del software hace una opción de la cual los patrones arquitectónicos proporcionen lo mejor posible las cualidades de calidad deseadas del sistema.

1.6 Patrones de Diseño

Un patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo siquiera dos veces de la misma forma.

Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software. En otras palabras, brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Debemos tener presente los siguientes elementos de un patrón: su nombre, el problema (cuando aplicar un patrón), la solución (descripción abstracta del problema) y las consecuencias (costos y beneficios). Los patrones de diseño se clasifican como se muestra a continuación:

- Patrones Creacionales: Inicialización y configuración de objetos.
- Patrones Estructurales: Separan la interfaz de la implementación. Se ocupan de cómo las clases y objetos se agrupan, para formar estructuras más grandes.
- Patrones de Comportamiento: Más que describir objetos o clases, describen la comunicación entre ellos.

Veamos un poco en qué consisten los distintos tipos de patrones, cuáles son sus fines y qué beneficios nos aportan.

1.6.1 Patrones Creacionales

Fábrica Abstracta (Abstract Factory)

El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).

Método de Fabricación (Factory Method)

Parte del principio de que las subclases determinan la clase a implementar, como se muestra en el código siguiente:

```
public class ConcreteCreator extends Creator {
    protected Product FactoryMethod(){
      return new ConcreteProduct();
    }
}

public interface Product{
}

public class ConcreteProduct implements Product{
}

public class Client{
    public static void main(String args[]){
            Creator UnCreator;
            UnCreator = new ConcreteCreator();
            UnCreator.AnOperations();
    }
}
```

Prototipado (Prototype)

Se basa en la clonación de ejemplares copiándolos de un prototipo.

Singleton

Restringe la instanciación de una clase o valor de un tipo a un solo objeto.

MVC (Model View Controller)

Este patrón plantea la separación del problema en tres capas: la capa modelo, que representa la realidad; la capa controlador**,** que conoce los métodos y atributos del modelo, recibe y realiza lo que el usuario quiere hacer; y la capa vista, que muestra un aspecto del modelo y es utilizada por la capa anterior para interaccionar con el usuario.

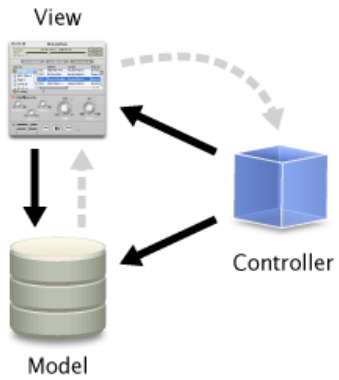


Figura 1.8: Patrón de Diseño MVC

Fuente: http://blogdeaitor.wordpress.com/2008/10/20/model-view-controller/

Autor: Aitor Rigada

1.6.2 Patrones Estructurales

Adaptador (Adapter): Convierte una interfaz en otra.

Puente (Bridge): Desacopla una abstracción de su implementación permitiendo modificarlas independientemente.

Objeto Compuesto (Composite): Utilizado para construir objetos complejos a partir de otros más simples, utilizando para ello la composición recursiva y una estructura de árbol.

Envoltorio (Decorator): Permite añadir dinámicamente funcionalidad a una clase existente, evitando heredar sucesivas clases para incorporar la nueva funcionalidad.

Fachada (Facade): Permite simplificar la interfaz para un subsistema.

Peso Ligero (Flyweight): Elimina la redundancia o la reduce cuando tenemos gran cantidad de objetos con información idéntica.

Apoderado (Proxy): Un objeto se aproxima a otro.

1.6.3 Patrones de Comportamiento

Cadena de responsabilidad (Chain of responsibility): La base es permitir que más de un objeto tenga la posibilidad de atender una petición.

Orden (Command): Encapsula una petición como un objeto dando la posibilidad de "deshacer" la petición.

Intérprete (Interpreter): Intérprete de lenguaje para una gramática simple y sencilla.

Iterador (Iterator): Define una interfaz que declara los métodos necesarios para acceder secuencialmente a una colección de objetos sin exponer su estructura interna.

Mediador (Mediator): Coordina las relaciones entre sus asociados. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones.

Recuerdo (Memento): Almacena el estado de un objeto y lo restaura posteriormente.

Observador (Observer): Notificaciones de cambios de estado de un objeto.

Estado (Server): Se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo.

Estrategia (Strategy): Utilizado para manejar la selección de un algoritmo.

Método plantilla (Template Method): Algoritmo con varios pasos suministrados por una clase derivada.

Visitante (Visitor): Operaciones aplicadas a elementos de una estructura de objetos heterogénea.

1.7 Patrones JEE

Con la aparición del JEE, todo un nuevo catálogo de patrones de diseño apareció. Desde que JEE es una arquitectura por sí misma que involucra otras arquitecturas, incluyendo servlets, JavaServer Pages, Enterprise JavaBeans, y más, merece su propio conjunto de patrones específicos para diferentes aplicaciones empresariales. De acuerdo al libro "J2EE PATTERNS Best Practices and Design Strategies", existen 5 capas en la arquitectura JEE:

- Cliente
- Presentación
- Negocios
- Integración
- Recurso

El libro explica 15 patrones JEE que están divididos en 3 de las capas: presentación, negocios e integración.

1.6.1 Catálogo de patrones JEE

Capa de Presentación

Decorating Filter / Intercepting Filter	Un objeto que está entre el cliente y los componentes Web. Este procesa las peticiones y las respuestas.
Front Controller/ Front Component	Un objeto que acepta todos los requerimientos de un cliente y los direcciona a manejadores apropiados. El patrón Front Controller podría dividir la funcionalidad en 2 diferentes objetos: el Front Controller y el Dispatcher. En ese caso, El Front Controller acepta todos los requerimientos de un cliente y realiza la autenticación, y el Dispatcher direcciona los requerimientos a manejadores apropiados.
View Helper	Un objeto helper que encapsula la lógica de acceso a datos en beneficio de los componentes de la presentación. Por ejemplo, los JavaBeans pueden ser usados como patrón View Helper para las páginas JSP.
Composite view	Un objeto vista que está compuesto de otros objetos vista. Por

	ejemplo, una página JSP que incluye otras páginas JSP y HTML usando la directiva include o el action include es un patrón Composite View.
Service To Worker	Es como el patrón de diseño MVC con el Controlador actuando como Front Controller pero con una cosa importante: aquí el Dispatcher (el cual es parte del Front Controller) usa View Helpers a gran escala y ayuda en el manejo de la vista.
Dispatcher View	Es como el patrón de diseño MVC con el controlador actuando como Front Controller pero con un asunto importante: aquí el Dispatcher (el cual es parte del Front Controller) no usa View Helpers y realiza muy poco trabajo en el manejo de la vista. El manejo de la vista es manejado por los mismos componentes de la Vista.

Tabla 1.1: Patrones capa de presentación

Capa de Negocio

Business Delegate	Un objeto que reside en la capa de presentación y en beneficio de los otros componentes de la capa de presentación llama a métodos remotos en los objetos de la capa de negocios.
Value Object/ Data Transfer Object/ Replicate Object	Un objeto serializable para la transferencia de datos sobre la red.
Session Façade/ Session Entity Façade/ Distributed Façade	El uso de un Bean de sesión como una fachada (facade) para encapsular la complejidad de las interacciones entre los objetos de negocio y participantes en un flujo de trabajo. El Session Facade maneja los objetos de negocio y proporciona un servicio de acceso uniforme a los clientes.
Aggregate Entity	Un Bean entidad que es construido o es agregado a otros Beans de entidad.
Value Object	Un objeto que reside en la capa de negocios y crea Value Objets

Assembler	cuando es requerido.
Value List Handler/ Page-by-Page Iterator/ Paged List	Es un objeto que maneja la ejecución de consultas SQL, caché y procesamiento del resultado. Usualmente implementado como beans de sesión.
Service Locator	Consiste en utilizar un objeto Service Locutor para abstraer toda la utilización JNDI y para ocultar las complejidades de la creación del contexto inicial, de búsqueda de objetos home EJB y recreación de objetos EJB. Varios clientes pueden reutilizar el objeto Service Locutor para reducir la complejidad del código, proporcionando un punto de control.

Tabla 1.2: Patrones capa de negocio

Capa de Integración

Data Access Object Service Activator	Consiste en utilizar un objeto de acceso a datos para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos.
Service Activator	Se utiliza para recibir peticiones y mensajes asíncronos de los clientes. Cuando se recibe un mensaje, el Service Activator localiza e invoca a los métodos de los componentes de negocio necesarios para cumplir la petición de forma asíncrona.

Tabla 1.3: Patrones capa de integración

CAPITULO II

DISEÑO E IMPLEMENTACIÓN DE LA CAPA DE PERSISTENCIA

Trabajar con datos relacionales dentro de una aplicación Java es una tarea que consume mucho tiempo. La estrategia a seguir depende en parte de los datos. Algunos de los factores más importantes a considerar son:

- Paralelismo entre el modelo relacional y la aplicación y sus modelos de objetos
- Si el modelo relacional existe antes de la concepción de la aplicación
- Si el modelo relacional es anticuado y utiliza un gran número de claves primarias naturales y compuestas
- Grado de normalización del modelo relacional

La traducción entre datos relacionales y objetos de aplicación es más sencilla en aquellos casos en que el arquitecto de software tiene cierto control en el diseño de datos relacionales. El diseño coordinado del modelo de objetos y el modelo de datos establece compromisos por ambas partes, arquitectos empresariales y de base de datos.

Como resultado, los objetos de la aplicación reflejan con más exactitud el dominio del negocio, y a la vez se garantiza que el modelo de datos relacional es eficiente, robusto y reusable entre distintas aplicaciones.

No siempre es necesario diseñar un modelo de objetos que se corresponda con el modelo de datos. Esta es una posible solución al problema. Por ejemplo en el caso de aplicaciones sencillas, o bien aplicaciones que manejan datos en representación tabular, una librería de estilo procedural de acceso a la base de datos como JDBC es suficiente.

No obstante, en aplicaciones de mediana y elevada complejidad lógica, el código basado en un modelo de entidades orientado a objetos facilita el desarrollo de la aplicación y produce código de más calidad. A menudo se precisan ambas estrategias en una misma aplicación.

En los puntos de este capítulo se describe la propuesta para la capa de persistencia, cómo trabajar con Hibernate, mapear entidades utilizando JPA, su integración con Spring, y como desarrollar clases de acceso de datos DAO.

2.1 Herramientas de mapeo objeto-relacional.

Cuando se trabaja con programación orientada a objetos y bases de datos relacionales, podemos observar que estos son dos paradigmas diferentes. El modelo relacional trata con relaciones y conjuntos, lo cual es algo matemático por naturaleza. Mientras tanto el paradigma orientado a objetos trata con objetos, sus atributos, métodos y asociaciones de unos con otros. Tan pronto como se desea persistir los objetos utilizando una base de datos relacional se puede observar que hay una desavenencia entre estos dos paradigmas, la también llamada diferencia objeto-relacional.

Un mapeador objeto-relacional (ORM) ayuda a evitar esta diferencia, entre los objetos y una base de datos relacional. ¿Pero cómo se manifiesta esta diferencia? Si se tiene objetos en una aplicación y algunas veces se alcanza el punto donde se necesita que sean persistentes, normalmente se abre una conexión JDBC, se crea una sentencia SQL y se copia todos los valores de las propiedades sobre la selección. Esto podría ser fácil para un objeto pequeño, pero si se considera esto para un objeto con muchas propiedades, ésta solución no sería eficiente.

Este no es el único problema que se presenta, pues otro se da cuando se obtiene una lista de objetos que representan una lista de registros y sus asociaciones se tiene que recorrer la lista de registros obtenidos para copiar los valores de cada una de las propiedades para obtener una lista de objetos.

Estos mismos criterios se aplican para la carga, supongamos que deseamos cargar un objeto Libro desde la base de datos y que tiene una colección de autores, se necesitará cargar los autores también y este proceso se deberá realizar en otra consulta más tarde y si consideramos que cargamos los autores debemos considerar que cada objeto autor tiene una asociación con otros objetos, los cuales los necesitamos presentar. Para casos como el descrito es mejor cargar todo el árbol de objetos. Este proceso de carga se lo puede realizar explícitamente más tarde si se quiere acceder a ellos.

Como podemos ver la diferencia objeto-relacional se amplía muy rápidamente si tenemos grandes modelos de objetos. Y además hay otras cosas que debemos considerar como la carga lenta al realizar el traslado, los datos del registro a los objetos, otros problemas son las referencias circulares, el caché, etc. De hecho, se han realizado estudios que demuestran que el 35% del código de una aplicación se produce para mapear datos entre la aplicación y la base de datos.

Entonces un ORM básicamente intenta quitarte la mayoría de esa carga de trabajo. Con un buen ORM, sólo tenemos que definir una vez la forma en que las clases se mapean a tablas, que propiedad se mapea a qué columna, que clase se mapea a qué tabla, etc.

2.2 Introducción JPA (Java Persistence API).

JPA (Java Persisence API) es un estándar de Java (JSR-220) que permite a los desarrolladores trabajar de forma sencilla con bases de datos relacionales. JPA permite la abstracción de la fuente de datos, permitiendo que el código sea portable entre distintos sistemas gestores de bases de datos: Oracle, DB2, MySQL, PostgresSQL, etc.

Gracias a JPA el desarrollador se puede centrar en la lógica de negocio, olvidando los detalles de implementación la capa de acceso a datos. Esto permite realizar desarrollos más rápidos y más seguros.

2.3 Propuesta de la capa de persistencia.

El concepto de la capa de persistencia aplicado a una aplicación empresarial basada en JEE es de suma importancia, ya que es la forma como accedemos a una base de datos. Determinar la forma como se va a implementar es un punto crítico en el desarrollo de una aplicación.

La forma tradicional y más conocida de acceder a una base de datos es vía JDBC, conectados a la base de datos mediante la ejecución de sentencias SQL.

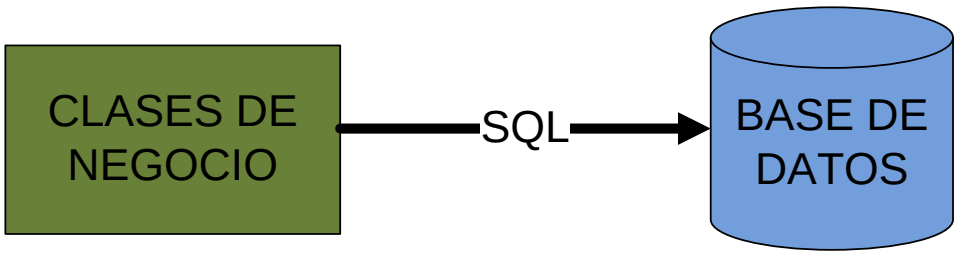


Figura 2.1: Sentencias SQL directas desde el código.

Autor: Diego Hinojosa.

Esta forma de generación de la capa de persistencia resulta difícil de dar mantenimiento ya que si se da un cambio en el modelo de datos relacional resulta necesario reestructurar el código, es decir un mínimo cambio implica una revisión minuciosa de todo el código, así como la compilación e instalación de la aplicación.

Aunque no podemos desechar su utilidad. El acceso a través de sentencias SQL directas puede ser utilizado de manera puntual para realizar operaciones a través del lenguaje SQL lo cual sería mucho más efectivo que la carga de gran cantidad de objetos en memoria. Un buen motor de persistencia debería implementar mecanismos para ejecutar estas operaciones masivas sin necesidad de acceder a este nivel.

Una aproximación más avanzada es la creación de clases de acceso a datos DAO (Data Acces Object) las cuales guardan toda la lógica del manejo de transacciones hacia la base de datos. De esta manera nuestra capa de negocio interactuaría con la capa DAO y ésta sería la encargada de realizar las operaciones sobre la base de datos.

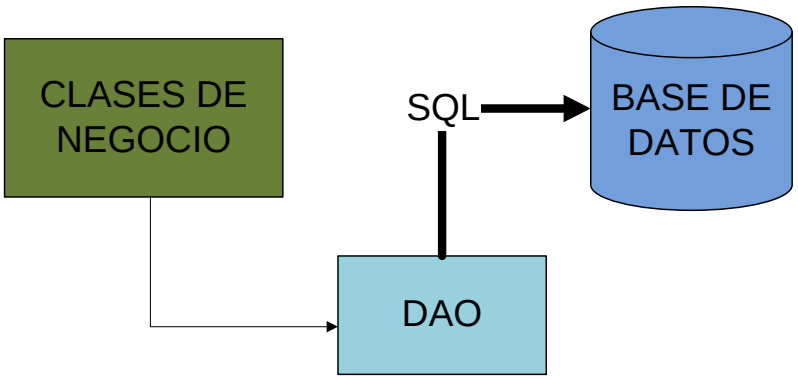


Figura 2.2: Arquitectura DAO.

Autor: Diego Hinojosa.

La gran ventaja que se obtiene es que ahora el código de transacciones se encuentra encapsulado en las clases. Lo que se busca en la capa de persistencia es que los desarrolladores no necesiten conocer nada acerca del esquema utilizado en la Base de Datos. Tan solo conocerán la interface proporcionada por el motor de persistencia implementado. De esta manera se consigue separar de manera clara y definida, la lógica de negocios de la aplicación con el diseño de la Base de Datos.

Esta arquitectura lleva un proceso de desarrollo más costoso, pero una vez implementada las ventajas que trae consigo merecerán la pena. Es en este punto donde entra en juego Hibernate, el cual es una herramienta que permite realizar el mapeo objeto-relacional de una forma cómoda pero potente, sin tener que implementarlo directamente mediante JDBC. Hibernate actúa como un puente entre la capa de negocio y la base de datos, sus funciones van desde la ejecución de sentencias SQL a través de JDBC, hasta la creación, modificación y eliminación de objetos persistentes.

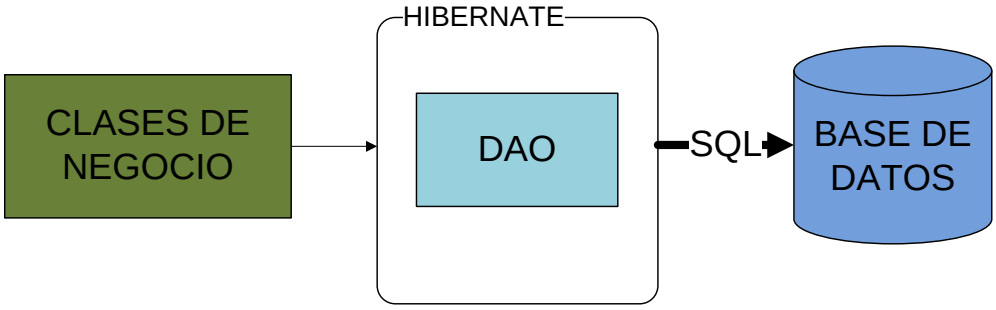


Figura 2.3: Capa de persistencia Hibernate.

Autor: Diego Hinojosa

Hibernate provee una capa de persistencia desarrollada la cual solo tenemos que adaptarla a nuestra arquitectura, además de que proporciona capacidades para la obtención y almacenamiento de datos de la base de datos que reducen el tiempo de desarrollo y también soporta una de las mayores comunidades de Open Source.

Lo ideal es combinar la capa de persistencia Hibernate y esconder toda la lógica de sentencias y transacciones contra la base en los ya mencionados DAO's. De esta manera aislamos completamente esta capa de la siguiente que es la capa de negocio.

A continuación se detalla de forma gráfica el diseño para esta capa:

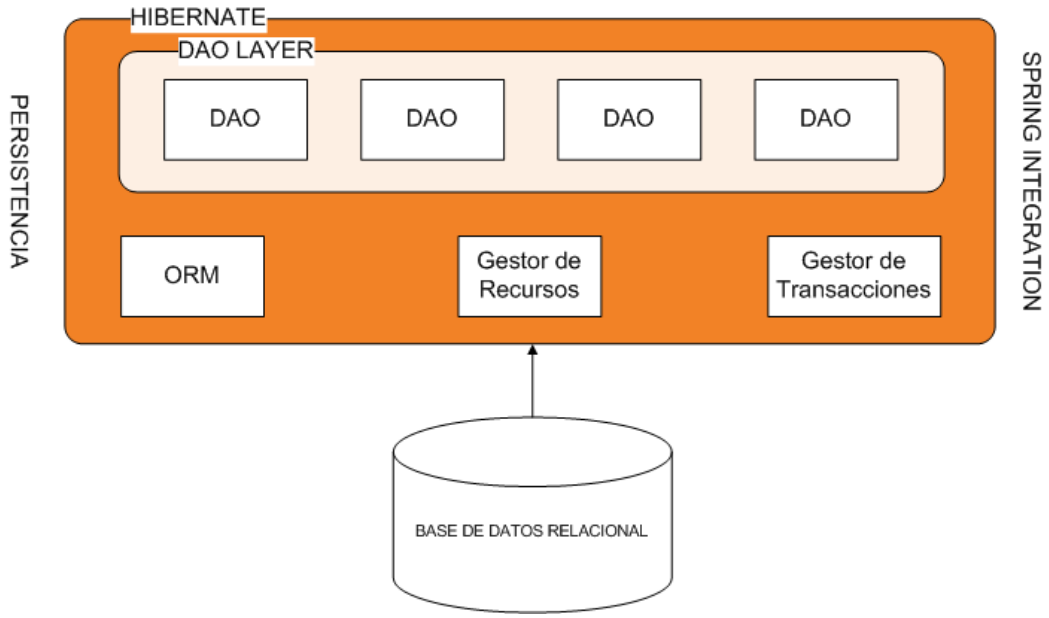


Figura 2.4 Propuesta de Capa de persistencia

Autor: Diego Hinojosa

2.4 Hibernate

Hibernate es un motor de persistencia de código abierto. Permite diseñar objetos persistentes que podrán incluir polimorfismo, relaciones, colecciones, y un gran número de tipos de datos. De una manera muy rápida y optimizada podremos generar la Base de Datos en cualquiera de los entornos soportados: Oracle, DB2, MySql, etc.

Cuenta con una amplia documentación, tanto a nivel de libros publicados como disponibles gratuitamente en su Web. A nivel comercial está respaldado por JBoss, que proporciona servicios de soporte, consultoría y formación en el mismo.

Actualmente es el rey indiscutible de la persistencia. Desde su versión 1.0, el motor no ha parado de evolucionar, incorporando todas las nuevas ideas que se iban incorporando en este campo. Hoy en día se encuentra en su versión 4.1.3 lanzada el 3 de Mayo del 2012 y tiene soporte para EJB 3.

Existen dos posibles procesos de desarrollo: El primero consiste en, una vez tengamos el

diseño de datos realizado, mapear este a archivos XML siguiendo la DTD de mapeo de Hibernate, el segundo consiste en mapear directamente utilizando anotaciones JPA soportadas por Hibernate.

Hibernate se integra en cualquier tipo de aplicación justo por encima del contenedor de datos. Una posible configuración básica de Hibernate es la siguiente:

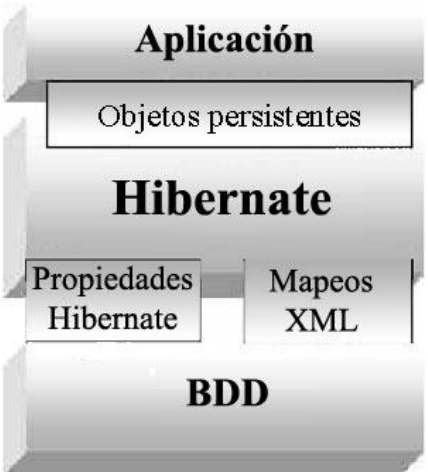


Figura 2.5: Integración de Hibernate y las aplicaciones Java.

Fuente: http://docs.jboss.org/hibernate/orm/3.6/reference/es-ES/html/architecture.html

Podemos observar como Hibernate utiliza la Base de Datos y la configuración de los datos para proporcionar servicios y objetos persistentes a la aplicación que se encuentre justo por arriba de él.

2.4.1 Arquitectura de Hibernate

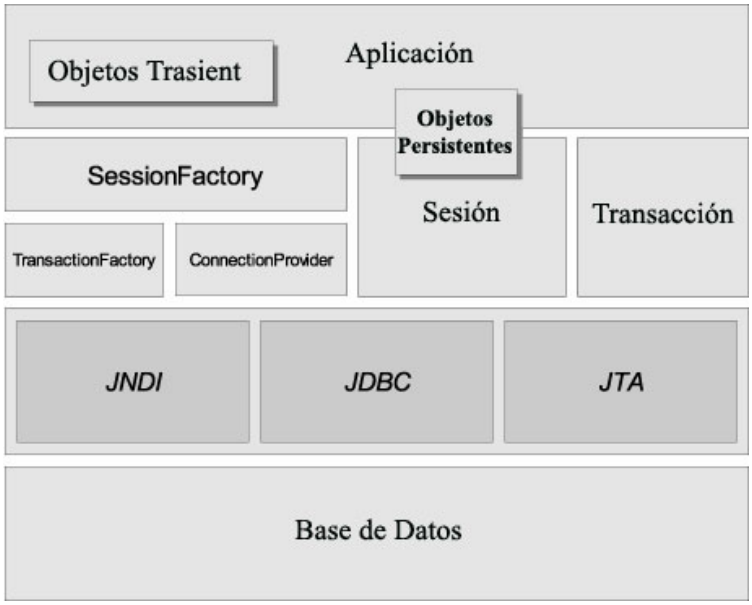


Figura 2.6 Arquitectura de Hibernate.

Fuente: http://docs.jboss.org/hibernate/orm/3.6/reference/es-ES/html/architecture.html

SessionFactory (org.hibernate.SessionFactory)

Es thread-safe, posee caché inmutable de mapeos compilados para una sola base de datos. Es una fábrica para las instancias de los objetos org.hibernate.Session. Es un cliente de org.hibernate.connection.ConnectionProvider. Opcionalmente mantiene un segundo nivel de cache de datos que es reutilizable entre las operaciones en un proceso o nivel de clúster.

Session (org.hibernate.Session)

Es un objeto de vida corta que representa una conversación entre la aplicación y el almacenamiento persistente. Envuelve un JDBC java.sql.Connection y es una fábrica para org.hibernate.Transaction. Mantiene un primer nivel de cache de objetos persistentes de la aplicación y colecciones, este caché se utiliza cuando se navega el gráfico de objetos o cuando se busca objetos por su identificador.

Objetos y colecciones persistentes

Objetos de vida corta que contienen estados persistentes y funciones del negocio. Estos pueden ser JavaBeans o POJO's ordinarios. Están asociados exactamente con una org.hibernate.Session. Una vez que el org.hibernate.Session está cerrado, serán liberados

para usarse en cualquier capa de la aplicación (por ejemplo, directamente como objetos de transferencia de datos hacia y desde la presentación).

Transacción (org.hibernate.Transaction)

(Opcional) Objeto de duración corta utilizado por la aplicación para especificar unidades atómicas de trabajo. En algunos casos un org.hibernate.Session puede abarcar varios org.hibernate.Transaction.

ConnectionProvider (org.hibernate.connection.ConnectionProvider)

(Opcional) Es una fábrica y un pool de conexiones. No se expone a la aplicación, sino que se puede extender y / o implementar por el desarrollador.

TransactionFactory (org.hibernate.TransactionFactory)

(Opcional) Es una fábrica para instancias de org.hibernate.Transaction. No se expone a la aplicación, sino que se puede extender y / o implementar por el desarrollador.

Esta arquitectura puede variar dependiendo de la configuración de Hibernate que se aplique a una aplicación, pero en general puede ser una configuración básica en la cual podemos observar que la aplicación se comunica mediante objetos persistentes con una sesión, una transacción o un objeto Factory de Hibernate el cual puede proveer conexiones y transacciones, cualquiera de estas formas puede utilizar JNDI, JDBC, o JTA para conectarse a una Base de Datos. Vemos además, que Hibernate se integra dentro de los servicios de una plataforma JEE siendo capaz de obtener conexiones a través de objetos DataSource vía JNDI, ejecutar sus transacciones dentro de un entorno JTA, etc.

Hibernate funciona asociando a cada tabla de la base de datos un Plain Old Java Object (POJO, a veces llamado Plain Ordinary Java Object). Un POJO es similar a un Java Bean, con propiedades accesibles mediante métodos setters y getters, como por ejemplo:

```
package estudiantes

public class Estudiantes {
    private String cedula;
    private String nombre;
    private Float promedio;

    public Estudiantes () {
    }

    public String getCedula() {
        return cedula;
    }

    private void setCedula(String cedula) {
        this.cedula = cedula;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public Float getPromedio() {
        return promedio;
    }

    public void setPromedio(Float promedio) {
        this.promedio = promedio;
    }
}
```

Para comenzar a utilizar Hibernate, necesitamos nuestros objetos persistentes para mapearlos hacia una base de datos, para lo cual vamos a utilizar anotaciones JPA lo cual se verá en la sección 2.4.4 y también se debe crear el archivo de configuración de Hibernate el cual es fundamental para comenzar a utilizar esta herramienta.

En el siguiente punto se detalla el contenido de este archivo de configuración.

2.4.2 Archivo de configuración de Hibernate

Para configurar Hibernate debemos crear un archivo de configuración XML. El archivo de configuración que se llamará **hibernate.cfg.xml**, ya que aquí estará la configuración general de la capa de persistencia implementada en Hibernate. Entre estas configuraciones esta el Data Source es decir el origen de datos y parámetros generales de

Hibernate para el momento de ejecución. De entre estos parámetros tenemos los siguientes:

- Dialecto.
- Parámetros de depuración como sentencias HQL y SQL.
- Datos de conexión a la Base de datos.
- Especificación de las Clases Entidad.
- Manejo de Transaccionalidad.

2.4.3 Configuración de Base de Datos (hibernate.cfg.xml)

Para la configuración de la Base de Datos utilizamos el archivo **hibernate.cfg.xml,** no es necesario que este archivo se llame así pero para plantear un estándar de desarrollo es conveniente mantener este nombre para el archivo de configuración que va a contener los datos de conexión a la Base de Datos.

A continuación se presenta la estructura del archivo de configuración de Hibernate

```
<Encabezado XML>
<Declaración de la DTD>
<hibernate-configuration>
        <session-factory>
            <property name=”nombre de propiedad”> valor de la
             propiedad</property>
             <mapping dirección de la clase mapeo/>
          </session-factory>
</hibernate-configuration>
```

A continuación se detalla las características, parámetros y definición de las etiquetas arriba utilizadas a la hora de conectarse a la Base de Datos:

1. **Declaración de la DTD**: El documento DTD que usaremos en nuestros archivos XML se encuentra en cada distribución de Hibernate en el propio .jar o en el directorio src.
2. **Elemento Raíz <hibernate-configuration>**: Dentro de él se declaran las propiedades de conexión a la base de datos y otros más, aquí es posible declarar más de un elemento <property>.

3. **SessionFactory <session-factory>**: Es una fábrica global responsable de una base de datos en particular. Si se dispone de varias bases de datos, se debería usar varias configuraciones <session-factory> en varios archivos de configuración para simplificar el arranque.
4. **Elemento <property>**: Propiedades de la configuración Hibernate de la aplicación. De entre las principales propiedades están las siguientes:
   - **hibernate.cache.provider_class.-** Driver de conexión, clase Java que contiene la conexión de base de datos la cual debe estar en el archivo .jar que proporciona el proveedor de la base de datos.
   - **hibernate.connection.url.-** URL de la base de datos que depende del proveedor**.**
   - **hibernate.connection.username.-** Nombre del usuario de la base de datos.
   - **hibernate.connection.password.-** Contraseña del usuario de la base de datos.
   - **dialect.-** Dialecto depende de la base de datos, es una clase Hibernate que se encarga de traducir el lenguaje HQL a SQL del proveedor de la base de datos, depende del proveedor de la base de datos ya que existe variación en el SQL de cada una de estas.

     En este parámetro se indica el nombre de la clase que se encargará de comunicarse con la base de datos en el SQL que entienda la base de datos. Este parámetro ha de ser siempre especificado. El valor debe ser una subclase que herede de org.hibernate.dialect.Dialect

     Los dialectos proporcionados por Hibernate se resumen en la tabla siguiente:

RDBMS	**Dialect**
DB2	org.hibernate.dialect.DB2Dialect
MySQL	org.hibernate.dialect.MySQLDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9	org.hibernate.dialect.Oracle9Dialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Progress	org.hibernate.dialect.ProgressDialect

Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
Ingres	org.hibernate.dialect.IngresDialect
Informix	org.hibernate.dialect.InformixDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect

Tabla 2.1 Dialectos proporcionados por Hibernate.

Aquí observamos la gran importancia del archivo de configuración, pues es aquí donde se especifica qué base de datos usamos, por lo que si cambiáramos de base de datos bastaría con cambiar este archivo de configuración, manteniendo nuestra aplicación intacta.

- **show_sql.-** Presentación de las sentencias HQL y SQL generadas para depuración y verificación del código de la aplicación.
- **transaction.factory_class.-** Clase Hibernate que controla el manejo de la transaccionalidad de la aplicación.
- **hibernate.cache.provider_class.-** Clase para proveer cache a la aplicación.
- **hibernate.hbm2ddl.auto.-** Esta propiedad le dice a Hibernate que ajuste automáticamente las tablas en la base de datos de acuerdo a nuestros mapeos.

4. **Elemento <mapping>:** Referencia las propiedades de mapeo.
    - **package.-** Se define el paquete que contiene las clases persistentes.

A continuación se presenta un ejemplo de un archivo de configuración Hibernate **hibernate.cfg.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/ejemplo
    </property>
    <property name="hibernate.connection.username">
      root
    </property>
    <property name="hibernate.current_session_context_class">
     thread
    </property>
    <property name="hibernate.hbm2ddl.auto">
     Create
    </property>
    <mapping package="entities"/>
  </session-factory>
</hibernate-configuration>
```

Una vez que se ha terminado de configurar este archivo se habrá terminado de crear la SessionFactory de Hibernate con las respectivas entidades (clases) de mapeo.

Además, Hibernate nos proporciona un lenguaje con el que se puede realizar consultas a la base de datos. Este lenguaje es similar a SQL y es utilizado para obtener objetos de la base de datos según las condiciones especificadas en el HQL.

El uso de HQL nos permite usar un lenguaje intermedio que según la base de datos que usemos y el dialecto que especifiquemos será traducido al SQL dependiente de cada base de datos de forma automática y transparente. Para realizar consultas sobre la base de datos necesitamos declarar un objeto de tipo Session y abrir la Sesión de la SessionFactory lo cual se lo realiza de la siguiente manera:

```
Session session = sessionFactory.openSession();
```

Así una forma de recuperar datos de la base de datos con Hibernate sería:

HIBERNATE

```
Session session = sessionFactory.openSession();
List cats = null;
try {
     categories = session.find("from Cat");
     Iterator i = categories.iterator();
     while (i.hasNext() == true) {
       Cat cat = (Cat)i.next();
       ...
     }
 } finally {
    session.close();
 }
```

JDBC

```
Driver d = (Driver)
Class.forName("com.mysql.jdbc.Driver").newInstance();
DriverManager.registerDriver(d);
try {
   Connection con =
DriverManager.getConnection("jdbc:mysql://yamcha/test",
   ”cesar","");
   Statement stmt = con.createStatement();
   String select =  "SELECT * from cat";
   ResultSet res = stmt.executeQuery(select);
   while (res.next() == true) {
      String catID = res.getString("id");
      String catName = res.getString("name");
      Cat cat = new Cat(catID,catName);
      (.......)
     list.add(cat);
  }
  stmt.close();
  con.commit();
  con.close();
  } catch (Throwable ex) {}
```

De esta forma:

HQL	SQL
from Cat	select * from cat

Tabla 2.2 Equivalencia de sentencias HQL y SQL.

A continuación se presenta varios ejemplos de conexiones a algunas Bases de Datos:

Mysql

```
hibernate.dialect org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jbc.Driver
hibernate.connection.url jdbc:mysql:///test
hibernate.connection.username root
hibernate.connection.password mysql
```

Oracle

```
hibernate.dialect org.hibernate.dialect.Oracle9Dialect
hibernate.dialect org.hibernate.dialect.OracleDialect
hibernate.connection.driver_class oracle.jdbc.driver.OracleDriver
hibernate.connection.username ora
hibernate.connection.password ora
hibernate.connection.url jdbc:oracle:thin:@localhost:1521:test
```

Una vez escrito el archivo de configuración veremos cómo realizar el mapeo de clases persistentes (POJO's) a través de anotaciones JPA.

2.4.4 Mapeo de POJO's con Anotaciones JPA

JPA es el actual estándar parte de Java EE 5 para persistencia. JPA incluye un lenguaje de definición de la correspondencia entre objetos y datos así como un API para la gestión y acceso a objetos persistentes.

Hibernate proporciona su propio lenguaje y API, pero a la vez también incluye una implementación del estándar JPA, y por lo tanto las dos opciones están disponibles.

Nuestra elección es siempre a favor del estándar, que en teoría garantiza una mayor independencia, aunque por otro lado es altamente improbable que en el futuro se decida sustituir Hibernate por una herramienta distinta, debido sobre todo a la enorme inversión en educación por parte del equipo de desarrollo.

De acuerdo con JPA, la definición de la correspondencia entre datos y objetos se puede configurar bien utilizando un lenguaje XML, o bien a través de un nuevo elemento del lenguaje Java incorporado en la versión J2SE 5: anotaciones.

¿Qué son anotaciones?

Una anotación es una primitiva del lenguaje Java que permite definir metadata asociado a un interfaz, una clase, a sus campos o métodos.

Una anotación se implementa con un tipo de clase especial y es ejecutada bien en tiempo de compilación o de ejecución.

¿Qué es metadata?

El término inglés metadata significa en general información adicional que caracteriza a los datos. En el caso de elementos de programación, metadata se entiende cómo información de configuración, infraestructura, o de programación declarativa que se aplica a los objetos de la aplicación y se procesa fuera del flujo de la lógica de la aplicación, a lo que también a veces se refiera como "ortogonal" a la lógica de la aplicación.

Desde su aparición en J2SE 5, la técnica de anotaciones ha sido adoptada por un gran número de productos y herramientas Java EE, tanto comerciales como de código abierto.

Es posible argumentar a favor o en contra de tener la configuración junto al código al que se aplica o aparte en archivos XML. Por lo general en el caso de utilizar anotaciones, el proceso de inicialización del framework necesitará un paso de escaneado de las clases Java compiladas para obtener la información de configuración contenida en las anotaciones.

En el caso de la información de persistencia, utilizando anotaciones la información ORM se incluye junto en los mismos archivos Java junto con los objetos a los que se aplica. Esto en mi experiencia simplifica el mantenimiento de las definiciones ORM, y por tanto del modelo de objetos, puesto que facilita su comprensión por los programadores.

Las anotaciones son de la forma "@nombre-anotacion" y definen la correspondencia entre la clase y la tabla relacional, los campos de la clase y las columnas de la tabla, y las relaciones entre clases. Por ejemplo, algunas de las anotaciones más comunes que se utilizarán en los ejemplos de esta sección son:

- @Entity: se aplica a nivel de clase Java e indica que la clase es persistente.
- @Table: declara el nombre de la tabla en la Base de Datos.
- @Id: indica qué campo se corresponde con la clave primaria
- @GeneratedValue: declara la estrategia de generación de la clave primaria.
- @Column: establece la correspondencia entre campo y columna
- @OneToMany: establece el lado "1" de una relación 1 a N
- @ManyToOne: establece el lado "N" de una relación 1 a N

- @Temporal: indica que un campo es de tipo fecha
- @Enumerated: indica que un campo es del tipo enum

Este tipo de configuración permite tener en un mismo lugar el código Java y sus instrucciones de comportamiento. Empezaremos entendiendo el concepto de una entidad. Este es el ejemplo más simple de entidad:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;

@Entity
public class Pelicula implements Serializable{
    @Id
    @GeneratedValue
    private Long id;
    private String titulo;
    private int duracion;
         // Getters y Setters
}
```

Las entidades suelen ser POJO's. La clase Pelicula se ha anotado con @Entity, lo cual informa a Hibernate que cada instancia de esta clase es una entidad. Para ser válida, toda entidad debe:

- Tener un constructor por defecto.
- Ser una clase de primer nivel (no interna).
- No ser final.
- Implementar la interface java.io.Serializabe si es accedida remotamente.

2.4.4.1 *Identidad*

Todas las entidades tienen que poseer una identidad que las diferencie del resto, por lo que deben contener una propiedad marcada con la anotación @Id (es aconsejable que dicha propiedad sea de un tipo que admita valores null, como Integer en lugar de int).

De manera adicional, la identidad de una entidad va a ser gestionada por el proveedor de persistencia, en nuestro caso Hibernate será quien le asigne un valor la primera vez que almacene la entidad en la base de datos. Para tal efecto, le añadimos a la propiedad de identidad la anotación @GeneratedValue.

2.4.4.2 *Configuración por defecto*

JPA aplica a las entidades que maneja una configuración por defecto, de manera que una entidad es funcional con una mínima cantidad de información (@Entity e @Id). Con esta configuración por defecto, todas las entidades del tipo Pelicula serán mapeadas a una tabla de la base de datos llamada PELICULA, y cada una de sus propiedades será mapeada a una columna con el mismo nombre (la propiedad id será mapeada en la columna ID, la propiedad titulo será mapeada en la columna TITULO, etc). Sin embargo, no siempre es posible ajustarse a los valores de la configuración por defecto: imaginemos que tenemos que trabajar con una base de datos heredada, con nombres de tablas y filas ya definidos. En ese caso, podemos configurar el mapeo de manera que se ajuste a dichas tablas y filas:

```
@Entity
@Table(name = "TABLA_PELICULAS")
public class Pelicula {
    @Id
    @GeneratedValue
    @Column(name = "ID_PELICULA")
    private Long id;
    ...
}
```

La anotación @Table nos permite configurar el nombre de la tabla donde queremos almacenar la entidad mediante el atributo name. Así mismo, mediante la anotación @Column podemos configurar el nombre de la columna donde se almacenará una propiedad, si dicha propiedad puede contener un valor null, etc.

2.4.4.3 *Lectura temprana y lectura demorada*

Cuando leemos una entidad desde la base de datos, ciertas propiedades pueden no ser necesarias en el momento de la creación del objeto. JPA nos permite leer una propiedad desde la base de datos la primera vez que un cliente intenta leer su valor (lectura demorada), en lugar de leerla cuando la entidad que la contiene es creada (lectura temprana). De esta manera, si la propiedad nunca es accedida, nos evitamos el costo de crearla. Esto puede ser útil si la propiedad contiene un objeto de gran tamaño:

```
@Basic(fetch = FetchType.LAZY)
private Imagen portada;
```

La propiedad portada es un objeto que representa una imagen de la portada de una película (un objeto de gran tamaño). Puesto que el costo de crear este objeto al leer la entidad Pelicula es muy alto, lo hemos marcado como una propiedad de lectura demorada mediante la anotación @Basic(fetch = FetchType.LAZY). El comportamiento por defecto de la mayoría de tipos Java es el contrario (lectura temprana). Este comportamiento, a pesar de ser implícito, puede ser declarado explícitamente de la siguiente manera:

```
@Basic(fetch = FetchType.EAGER)
private Imagen portada;
```

Solo los objetos de gran tamaño y ciertos tipos de asociación deben ser leídos de forma demorada. Si, por ejemplo, marcamos todas las propiedades de tipo int, String o Date de una entidad con lectura demorada, cada vez que accedamos por primera vez a cada una de ellas, Hibernate tendrá que hacer una llamada a la base de datos para leerla. Esto, evidentemente, va a provocar que se efectúen múltiples llamadas a la base de datos cuando con solo una (al crear la entidad en memoria) podría haberse leído todas con menor costo. Por tanto, es importante tener presente que la manera de configurar el tipo de lectura de una propiedad puede afectar enormemente al rendimiento de nuestra aplicación.

Por otro lado, la anotación @Basic solo puede ser aplicada a tipos primitivos, sus correspondientes clases wrapper, BigDecimal, BigInteger, Date, arrays, algunos tipos del paquete java.sql, enums, y cualquier tipo que implemente la interface Serializable. Además del atributo fetch, la anotación @Basic admite otro atributo, optional, el cual permite definir si la propiedad sobre la que se está aplicando la anotación puede contener el valor null (esto es debido a que en bases de datos relacionales, algunas columnas pueden definir una constricción de tipo non null, la cual impide que se inserte un valor null; por tanto, con @Basic(optional=false) nos ajustaríamos a la citada restricción).

2.4.4.4 *Tipos enumerados*

JPA puede mapear los tipos enumerados (enum) mediante la anotación Enumerated:

```
@Enumerated
private Genero genero;
```

La configuración por defecto de JPA mapeará cada valor ordinal de un tipo enumerado a una columna de tipo numérico en la base de datos. Por ejemplo, siguiendo el ejemplo anterior, podemos crear un tipo enum que describa el género de una película:

```
public enum Genero {
    TERROR,
    DRAMA,
    COMEDIA,
    ACCION,
}
```

Si la propiedad género del primer ejemplo tiene el valor Genero.COMEDIA, en la columna correspondiente de la base de datos de insertará el valor 2 (que es el valor ordinal de Genero.COMEDIA). Sin embargo, si en el futuro introducimos un nuevo tipo de género en una posición intermedia, o reordenamos las posiciones de los géneros, nuestra base de datos contendrá valores erróneos que no se corresponderán con los nuevos valores ordinales del tipo enumerado. Para evitar este problema potencial, podemos forzar a la base de datos a utilizar una columna de texto en lugar de una columna numérica: de esta manera, el valor almacenado será el nombre del valor enum, y no su valor ordinal:

```
@Enumerated(EnumType.STRING)
private Genero genero;
```

2.4.4.5 *Transient*

Ciertas propiedades de una entidad pueden no representar su estado. Por ejemplo, imaginemos que tenemos una entidad que representa a una persona:

```
@Entity
public class Persona {
    @Id
    @GeneratedValue
    private Long id;
    private String nombre;
    private String apellidos
    private Date fechaNacimiento;
    private int edad;

    // getters y setters
}
```

Podemos considerar que la propiedad edad no representa el estado de Persona, ya que si no se actualiza en cada cumpleaños, terminará conteniendo un valor equivocado. Ya que su valor puede ser calculado gracias a la propiedad fechaNacimiento, no vamos a almacenarlo en la base de datos, si no a calcular su valor en tiempo de ejecución cada vez que lo necesitemos. Para indicar a Hibernate que ignore una propiedad cuando realice el mapeo, usamos la anotación @Transient:

```
@Transient
private int edad;
```

Ahora, para obtener el valor de edad utilizamos su método getter:

```
public int getEdad() {
    // calcular la edad y devolverla
}
```

2.4.4.6 *Colecciones básicas*

Una entidad puede tener propiedades de tipo java.util.Collection y/o java.util.Map que contengan tipos básicos (todos los tipos wrapper, String, BigDecimal, BigInteger, tipos enumerados y tipos insertables). Los elementos de estas colecciones serán almacenados en una tabla diferente a la que contiene la entidad donde están declarados. El tipo de colección tiene que ser concreto (como ArrayList o HashMap) y nunca una interface.

```
private ArrayList comentarios
```

El código de arriba es mapeado por defecto con unos valores predeterminados por JPA, como vimos en la sección 2.4.4.2. Y por supuesto, podemos cambiar dicha configuración por defecto mediante diversas anotaciones, entre las que se encuentran:

```
@ElementCollection(fetch = FetchType.LAZY)
@CollectionTable(name = "TABLA_COMENTARIOS")
private ArrayList comentarios;
```

@ElementCollection permite definir el tipo de lectura (temprana o demorada), así como la clase de objetos que contiene la colección (obligatorio en caso de que la colección no sea de tipo genérico). Por otro lado, @CollectionTable nos permite definir el nombre de

la tabla donde queremos almacenar los elementos de la colección. Si nuestra colección es de tipo Map, podemos añadir la anotación @MapKeyColumn(name = "NOMBRE_COLUMNA"), la cual nos permite definir el nombre de la columna donde se almacenarán las claves del Map.

2.4.4.7 *Tipos insertables*

Los tipos insertables (embeddables) son objetos que no tienen identidad, por lo que para ser persistidos deben ser primero insertados dentro de una entidad. Cada propiedad del tipo insertable será mapeada a la tabla de la entidad que lo contenga, como si fuera una propiedad declarada en la propia entidad. Definimos un tipo insertable con la anotación @Embeddable:

```
@Embeddable
public class Direccion {
    private String calle;
    private int codigoPostal;
    ...
}
```

Y lo insertamos en una entidad (u otro tipo insertable) con la anotación @Embedded:

```
@Entity
public class Persona {
    ...
    @Embedded
    private Direccion direccion;
}
```

Ahora, la tabla que contenga las entidades de tipo Persona contendrá sus propias columnas, más las definidas en el tipo insertable Direccion.

2.4.4.8 *Tipo de acceso*

Vamos a ver que son los tipos de acceso, y como aplicarlos correctamente. JPA permite definir dos tipos de acceso:

- Acceso a variable (Field access)
- Acceso a propiedad (Property access)

El tipo de acceso que usará una entidad está definido por el lugar donde situemos sus anotaciones de mapeo. Si las anotaciones están en las variables que conforman la clase

(como hemos hecho hasta ahora), estaremos indicando a JPA que debe realizar acceso a variable. Si, por el contrario situamos las anotaciones de mapeo en los métodos getter, estaremos indicando un acceso a propiedad. Para fines prácticos, no existe diferencia alguna entre ambas opciones (más allá de gustos personales y de organización de código). Sin embargo, en determinadas ocasiones debemos ser consecuentes con el tipo de acceso que elijamos, evitando mezclar tipos de acceso (lo cual puede inducir a JPA a actuar de forma errónea). Un ejemplo típico es el uso de clases insertables: salvo que se especifique lo contrario, las clases insertables heredan el tipo de acceso de la entidad donde son insertadas, ignorando cualquier anotación que se haga hecho sobre ellas previamente. Veamos un ejemplo donde se muestra este problema:

```
@Embeddable
public class Insertable {
    private int variable;

    @Column(name = "VALOR_DOBLE")
    public int getVariable() {
        return variable * 2;
    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
}

@Entity
public class Entidad
    @Id
    @GeneratedValue
    private Long id;
    @Embedded
    private Insertable insertable;
}
```

En el ejemplo anterior, la clase Insertable define un tipo de acceso a propiedad (ya que las anotaciones se han definido en los métodos getter), y queremos que se acceda a través de estos métodos al valor de las variables (tal vez necesitemos realizar cierto procesamiento sobre la variable, como multiplicarla por dos antes de devolverla). Sin embargo, la clase Entidad define un tipo de acceso a variable (ya que las anotaciones se han definido las variables). Puesto que el tipo insertable va a heredar el tipo de acceso de la entidad donde se encuentra definido, cuando accedamos al valor de Entidad.insertable.variable obtendremos un valor no esperado (el valor de variable, en lugar de su valor multiplicado

por dos que devuelve getVariable()). Para evitar estos problemas debemos indicar explícitamente el tipo de acceso de Insertable mediante la anotación @Access:

```
@Embeddable
@Access(AccessType.PROPERTY)
public class Insertable { ... }
```

Anotando la clase Insertable con @Access(AccessType.PROPERTY), cuando accedamos a Entidad.insertable.variable lo haremos a través de su método getter, a pesar de que en Entidad se está usando un tipo de acceso a variable. Un efecto de definir explícitamente el tipo de acceso, es que las anotaciones que no correspondan con ese tipo de acceso serán ignoradas (a no ser que vengan acompañadas a su vez con la anotación @Access):

```
@Embeddable
@Access(AccessType.FIELD)
public class Insertable {

    private int variable;
    @Column(name = "VALOR_DOBLE")

    public int getVariable() {
        return variable * 2;
    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
}
```

En el ejemplo anterior, se ha definido un acceso a variable de forma explícito (con @Access(AccessType.FIELD)), por lo que la anotación @Column será ignorada (no accederemos a la variable a través del método getter, a pesar de estar anotado). Toda la discusión sobre tipos de acceso está directamente relacionada con la capacidad de JPA para acceder a todas las variables y métodos de una clase, independientemente de su nivel de acceso (public, protected, package-default, o private).

2.4.4.9 *Asociaciones*

Cuando queremos mapear colecciones de entidades, debemos usar asociaciones. Estas asociaciones pueden ser de dos tipos:

- Asociaciones unidireccionales
- Asociaciones bidireccionales

Las asociaciones unidireccionales reflejan un objeto que tiene una referencia a otro objeto (la información puede viajar en una dirección). Por el contrario, las asociaciones bidireccionales reflejan dos objetos que mantienen referencias al objeto contrario (la información puede viajar en dos direcciones). Además del concepto de dirección, existe otro concepto llamado cardinalidad, que determina cuantos objetos puede haber en cada extremo de la asociación.

2.4.4.10 *Asociaciones unidireccionales*

Para entender el concepto de unidireccionalidad, veamos un ejemplo de asociación unidireccional:

```
@Entity
public class Cliente {

    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    private Direccion direccion;
    // Getters y setters
}

@Entity
public class Direccion {
    @Id
    @GeneratedValue
    private Long id;
    private String calle;
    private String ciudad;
    private String pais;
    private Integer codigoPostal;

    // Getters y setters
}
```

Como se puede ver en el ejemplo anterior, cada entidad Cliente mantiene una referencia a una entidad Direccion. Esta relación es de tipo uno a uno (one-to-one) unidireccional, puesto que una entidad Cliente contiene una referencia a una entidad Direccion, relación que se ha declarado mediante la anotación @OneToOne. Cliente es el dueño de la relación de manera implícita, y por tanto cada entidad de este tipo contendrá por defecto

una columna adicional en su tabla correspondiente de la base de datos (mediante la que podremos acceder al objeto Direccion). Esta columna es una clave foránea (foreign key), un concepto muy importante en bases de datos relaciones.

Cuando Hibernate realice el mapeo de esta relación, cada entidad será almacenada en su propia tabla, añadiendo a la tabla donde se almacenan los clientes (la dueña de la relación) una columna con las claves foráneas necesarias para asociar cada fila con la fila correspondiente en la tabla donde se almacenan las direcciones. Hay que recordar que JPA utiliza configuración por defecto para realizar el mapeo, pero podemos customizar este proceso definiendo el nombre de la columna que contendrá la clave foránea mediante la anotación @JoinColumn:

```
@OneToOne
@JoinColumn(name = "DIRECCION_FK")
private Direccion direccion;
```

Otro tipo de asociación muy común es la de tipo uno a muchos (one-to-many) unidireccional. Veamos un ejemplo:

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;

    @OneToMany
    private List direcciones;

    // Getters y setters
}
```

En el ejemplo anterior, cada entidad de tipo Cliente mantiene una lista de direcciones a través de la propiedad direcciones. Puesto que un objeto List puede guardar múltiples objetos en su interior, debemos anotarlo con @OneToMany. En este caso, en vez de utilizar una clave foránea en Cliente, Hibernate utilizará por defecto una tabla de unión (join table). Cuando ocurre esto, las tablas donde se almacenan ambas entidades contienen una clave foránea a una tercera tabla con dos columnas; esta tercera tabla es llamada tabla de unión, y es donde se refleja la asociación entre las entidades

relacionadas. Como siempre, podemos configurar el mapeo mediante metadatos (anotaciones/XML) para que se ajuste a nuestras necesidades:

```
@OneToMany
@JoinTable(name = ...,
        joinColumn = @JoinColumn(name = ...),
        inverseJoinColumn = @JoinColumn(name = ...))
private List direcciones;
```

2.4.4.11 *Asociaciones bidireccionales*

En las asociaciones bidireccionales, ambos extremos de la relación mantienen una referencia al extremo contrario. En este caso, el dueño de la relación debe ser especificado explícitamente, de manera que Hibernate pueda realizar el mapeo correctamente. Veamos un ejemplo de bidireccionalidad en una relación de tipo uno a uno:

```
@Entity
public class Mujer {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    private Marido marido;
    // Getters y setters
}

@Entity
public class Marido {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne(mappedBy = "marido")
    private Mujer mujer;
}
```

En el ejemplo anterior, Mujer es la dueña de la relación, ya que la relación es bidireccional, ambos lados de la relación deben estar anotados con @OneToOne, pero ahora uno de ellos debe indicar de manera explícita que la parte contraria es dueña de la relación. Esto lo hacemos añadiendo el atributo mappedBy en la anotación de asociación de la parte no dueña. El valor de este atributo es el nombre de la propiedad asociada en la entidad que es dueña de la relación. El atributo mappedBy puede ser usado en relaciones

de tipo @OneToOne, @OneToMany y @ManyToMany, únicamente el cuarto tipo de relación, @ManyToOne, no permite el uso de este atributo.

2.4.4.12 *Lectura temprana y lectura demorada de asociaciones*

En el punto 2.4.4.3 se vio lo que significaban los conceptos de lectura temprana y lectura demorada. Las asociaciones son parte del mapeo relacional, y por tanto también son afectadas por este concepto. El tipo de lectura por defecto para las relaciones uno a uno y muchos a uno es temprana (eager). Por el contrario, el tipo de lectura para los dos tipos de relaciones restantes (uno a muchos y muchos a muchos), es demorada (lazy). Por supuesto, ambos comportamientos pueden ser modificados:

```
@OneToMany(fetch = FetchType.EAGER)
private List pedidos;
```

Al igual que se indicó en el punto 2.4.4.3, se debe ser consciente del impacto en el rendimiento de la aplicación que puede causar una configuración errónea en el tipo de lectura. En el caso de las asociaciones, donde se pueden ver involucrados muchos objetos, leerlos de manera temprana puede ser, además de innecesario, inadecuado. En otros casos, una lectura temprana puede ser necesaria si la entidad que es dueña de la relación se desconecta de Hibernate sin haber inicializado aún una asociación con lectura demorada, al intentar acceder a dicha asociación se producirá una excepción de tipo LazyInitializationException, que al no apuntar a ninguna instancia en memoria produce un error si es usado.

2.4.4.13 *Ordenación de asociaciones*

Se puede ordenar los resultados devueltos por una asociación mediante la anotación @OrderBy:

```
@OneToMany
@OrderBy("nombrePropiedad asc")
private List pedidos;
```

El atributo de tipo String que hemos proporcionado a @OrderBy se compone de dos partes: el nombre de la propiedad sobre la que queremos que se realice la ordenación, y, opcionalmente, el sentido en que se realizará puede ser:

- Ascendente (añadiendo asc al final del atributo)
- Descendente (añadiendo desc al final del atributo)

Así mismo, podemos mantener ordenada la colección a la que hace referencia una asociación en la propia tabla de la base de datos; para ello, debemos usar la anotación @OrderColumn. Sin embargo, el impacto en el rendimiento de la base de datos que puede producir este comportamiento es algo que se debe tener muy presente, ya que las tablas afectadas tendrán que reordenarse cada vez que se hayan cambios en ellas (en tablas de cierto tamaño, o en aquellas donde se inserten o modifiquen registros con cierta frecuencia, es totalmente no aconsejable forzar una ordenación automática).

2.4.4.14 *Herencia*

En las aplicaciones Java, el concepto de herencia es usado de forma intensiva. Por supuesto, Hibernate nos permite gestionar la forma en que nuestros objetos son mapeados cuando en ellos interviene el concepto de herencia. Esto puede hacerse de maneras distintas:

- Una tabla por familia (comportamiento por defecto)
- Unión de subclases
- Una tabla por clase concreta

El mapeo por defecto es una tabla por familia. Una familia no es, ni más ni menos, que todas las subclases que están relacionadas por herencia con una clase madre. Todas las clases que forman parte de una misma familia son almacenadas en una única tabla. En esta tabla existe una columna por cada atributo de cada clase y subclase de la familia, además de una columna adicional donde se almacena el tipo de clase al que hace referencia cada fila. Imaginemos el ejemplo siguiente:

```
@Entity
public class SuperClase {
    @Id
    @GeneratedValue
    private Long id;

    private int propiedadUno;
    private String propiedadDos;
     // Getters y Setters
}
```

```
@Entity
public class SubClase extends SuperClase {
    @Id
    @GeneratedValue
    private Long id;

    private float propiedadTres;
    private float propiedadCuatro;
         // Getters y setters
}
```

En el ejemplo anterior, tanto las instancias de las entidades SuperClase y SubClase serán almacenadas en una única tabla que tendrá el nombre por defecto de la clase raíz (SuperClase). Dentro de esta tabla habrá seis columnas que se corresponderán con:

- Una columna para la propiedad id (válida para ambas entidades, pues en ambas se mapearía a una columna con el mismo nombre).
- Cuatro columnas para las propiedades propiedadUno, propiedadDos, propiedadTres y propiedadCuatro.

Una última columna discriminatoria, donde se almacenará el tipo de clase al que hace referencia cada fila.

La columna discriminatoria suele contener el nombre de la clase. Esta columna es necesaria para que al recuperar un objeto desde la base de datos, Hibernate sepa que clase concreta debe instanciar, y que propiedades leer. Las propiedades que son propias de cada clase no deben ser configuradas como not null, ya que al intentar persistir un objeto de la misma familia pero de otra clase (y que tal vez no dispone de las mismas propiedades) obtendríamos un error. Aunque una tabla por familia es el comportamiento por defecto cuando mapeamos entidades en las que interviene la herencia entre clases, podemos declararlo explícitamente mediante el atributo SINGLE_TABLE (tabla única) de la anotación @Inheritance:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class SuperClass { ... }
```

Hay que tener presente que los valores definidos en @Inheritance son heredados por todas las subclases de la entidad anotada, aunque estas pueden sobrescribir dichos valores si desean adoptar una política de mapeo diferente. En lo que se refiere al nombre de la

columna discriminatoria y a su tipo, estos son por defecto DTYPE y String (este último es un tipo SQL). Podemos cambiar ambos mediante las anotaciones @DiscriminatorColumn y @DiscriminatorValue:

```
@Entity
@Inheritance
@DiscriminatorColumn(name = "...", discriminatorType = CHAR)
@DiscriminatorValue("C")
public class SuperClase { ... }
```

La anotación @DiscriminatorColumn solo debe ser usada en la clase raíz de la herencia (a no ser que una subclase desee cambiar los parámetros del mapeo, en cuyo caso ella se convertiría en clase raíz por derecho). El atributo discriminatorType de la citada anotación nos permite cambiar el tipo de valor que almacenará la columna discriminatoria. Los tipos soportados son STRING (por defecto), CHAR e INTEGER. Si cambiamos en la clase raíz el tipo de valor que almacenará la columna discriminatoria a otro distinto de STRING, cada subclase tendrá que indicar de forma explícita el valor que lo representará en la columna discriminatoria. Esto lo hacemos mediante la anotación DiscrimitadorValue:

```
@Entity
@DiscriminatorValue("S")
public class SubClase extends SuperClase { ... }
```

En el ejemplo anterior, la columna discriminatoria (que es de tipo CHAR) contendrá un valor C si la instancia correspondiente es de tipo SuperClase, y una S si es de tipo SubClase. Por supuesto, no estaría permitido usar @DiscriminatorValue("C") si se ha definido una columna discriminatoria de tipo INTEGER, etc.

El segundo tipo de mapeo cuando existe herencia es unión de subclases, en el que cada clase y subclase (sea abstracta o concreta) será almacenada en su propia tabla:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class SuperClase { ... }
```

La tabla raíz contiene una columna con una clave primaria usada por todas las tablas, así como la columna discriminatoria. Cada subclase almacenará en su propia tabla

únicamente sus atributos propios (nunca los heredados), así como una clave foránea que hace referencia a la clave primaria de la tabla raíz. La mayor ventaja de este sistema es que es intuitivo. Por contra, su mayor inconveniente es que, para construir un objeto de una subclase, hay que hacer una (o varias) operaciones JOIN en la base de datos, de manera que se puedan unir los atributos de la subclase con los de sus superclases. Por tanto, una subclase que esté varios niveles por debajo de la superclase en la herencia, necesitará realizar múltiples operaciones JOIN (una por nivel), lo cual puede producir un impacto en el rendimiento que tal vez no deseemos.

El tercer y último tipo de mapeo cuando existe herencia es una tabla por clase concreta. Mediante este comportamiento, cada entidad será mapeada a su propia tabla (incluyendo todos los atributos propios y heredados). Con este sistema no hay tablas compartidas, columnas compartidas, ni columna discriminatoria.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class SuperClase { ... }
```

El único requerimiento al utilizar una tabla por clase concreta es que todas las tablas dentro de una misma familia compartan el valor de la clave primaria. De esta manera, cuando almacenemos una subclase, tanto su tabla como las de sus superclases contendrán los mismos valores para las propiedades comunes, gracias a que todas ellas comparten la misma ID. Este sistema puede provocar en determinadas circunstancias problemas de rendimiento, ya que al igual que con la unión de subclases, la base de datos tendrá que realizar múltiples operaciones JOIN ante determinadas solicitudes.

2.4.4.15 *Mapped superclasses, clases abstractas y no-entidades*

Veamos de manera muy superficial la forma en la que gestiona JPA tres tipos de clases no vistas hasta ahora: Mapped Superclases, clases abstractas, y no entidades.

Mapped Superclasses son clases que no son manejadas por Hibernate, pero que comparten sus propiedades con cualquier entidad que extienda de ellas:

```
@MappedSuperclass
@Inheritance(strategy = InheritanceType.XXX)
```

```
public class MapedSuperClase {
    private String propiedadUno;
    private int propiedadDos;

    // Getters y setters
}


@Entity
public class SuperClase extends MappedSuperClase { ... }
```

En el ejemplo anterior, únicamente SuperClase será manejada Hibernate. Sin embargo, durante el mapeo se incluirán todas las propiedades heredadas de MappedSuperClase.

Por otro lado, todas las clases abstractas son tratadas exactamente igual que si fueran clases concretas, y por tanto son entidades 100% usables siempre que sean declaradas como tal (@Entity, etc). Precisamente por esto último, toda clase que no sea declarada como entidad, será ignorada a la hora de realizar el mapeo relacional. A este tipo de clases se las conoce como no entidades.

A continuación podemos ver un ejemplo de un mapeo sencillo de dos clases (POJO's), una llamada Customer (Cliente) y otra llamada Order (Orden), donde se aplican conceptos básicos sobre el mapeo con Hibernate y las anotaciones vistas en los puntos anteriores:

```
@Entity
public class Customer {
   @Id
   @GeneratedValue
   private Integer id;

   public Integer getId() {
     return id;
   }
   public void setId(Integer id) {
     this.id = id;
   }

   @OneToMany(mappedBy="customer")
   @OrderBy("number")
   private List<Order> orders;

   public List<Order> getOrders() {
     return orders;
   }
```

```
   public void setOrders(List<Order> orders) {
     this.orders = orders;
   }
}

@Entity
public class Order {
   @Id
   @GeneratedValue
   private Integer id;

   public Integer getId() {
     return id;
   }

   public void setId(Integer id) {
     this.id = id;
   }

   public String getNumber() {
     return number;
   }
   public void setNumber(String number) {
     this.number = number;
   }

   private String number;

   @ManyToOne
   private Customer number;

   public Customer getCustomer() {
      return customer;
   }

   public void setCustomer(Customer customer) {
      this.customer = customer;
   }
} // End of Order Entity

-- Table schema
|-----------------|    |---------------|
| Order           |    | Customer      |
|-----------------|    |---------------|
| id              |    | id            |
| number          |    |---------------|
| customer_id     |
|-----------------|
```

2.5 Integrando Data Access Objects (DAO's) y Hibernate con Spring

Una vez realizado el mapeo de las clases POJO's necesitamos seguir con el desarrollo de los Data Access Objects (DAO's) para lo cual necesitamos una manera eficiente de gestionar las Sesiones de Hibernate, es aquí donde Spring entra en acción.

Spring es un conjunto de librerías a la carta, entre las que podemos escoger aquellas que faciliten el desarrollo de nuestra aplicación. Entre sus posibilidades más potentes está su contenedor de Inversión de Control (también llamado Inyección de Dependencias), es una técnica alternativa a las clásicas búsquedas de recursos vía JNDI. Permite configurar las clases en un archivo XML y definir en él las dependencias, de esta forma la aplicación se vuelve muy modular. También integra la introducción de aspectos, plantillas de utilidades para Hibernate, iBatis y JDBC así como la integración con JSF.

2.5.1 Spring

Spring es un framework de aplicaciones Java/JEE desarrollado usando licencia de OpenSource. Se basa en una configuración a base de JavaBeans bastante simple. Es potente en cuanto a la gestión del ciclo de vida de los componentes y fácilmente ampliable. Su última versión (3.1.0) fue lanzada el 13 de Diciembre del 2011.

Es interesante el uso de programación orientada a aspectos (IoC). Tiene plantillas que permiten un uso más fácil de Hibernate, iBatis, JDBC. Se integra de fábrica con Quartz, Velocity, Freemarker, Struts, Webwork2 y tiene un plugin para Eclipse.

Ofrece un ligero contenedor de Beans para los objetos de la capa de negocio, DAO’s y repositorio de Datasources JDBC y sesiones Hibernate. Mediante un archivo xml definimos el contexto de la aplicación siendo una potente herramienta para manejar objetos Singleton o “fábricas” que necesitan su propia configuración.

El objetivo de Spring es no ser intrusivo, aquellas aplicaciones configuradas para usar Beans mediante Spring no necesitan depender de interfaces o clases de Spring, pero obtienen su configuración a través de las propiedades de sus Beans. Este concepto puede ser aplicado a cualquier entorno, desde una aplicación JEE a un applet. Como ejemplo podemos pensar en conexiones a base de datos o de persistencia de datos, como Hibernate, la gestión de transacciones genérica de Spring para DAO’s es muy interesante.

La meta a conseguir es separar los accesos a datos y los aspectos relacionados con las transacciones, para permitir objetos de la capa de negocio reutilizables que no dependan de ninguna estrategia de acceso a datos o transacciones.

¿Que nos proporciona Spring?

Spring nos proporciona las siguientes características:

- Una potente gestión de configuración basada en JavaBeans, aplicando los principios de Inversión de Control (IoC). Esto hace que la configuración de aplicaciones sea rápida y sencilla. Ya no es necesario tener singletons ni archivos de configuración, propone una aproximación consistente y elegante. Estas definiciones de Beans se realizan en lo que se llama el contexto de aplicación.
- Una capa genérica de abstracción para la gestión de transacciones, permitiendo gestores de transacción añadibles (pluggables), y haciendo sencilla la demarcación de transacciones sin tratarlas a bajo nivel. Se incluyen estrategias genéricas para JTA y un único JDBC DataSource. En contraste con el JTA simple o EJB CMT, el soporte de transacciones de Spring no está atado a entornos JEE.
- Una capa de abstracción JDBC que ofrece una significativa jerarquía de excepciones (evitando la necesidad de obtener de SQLException los códigos que cada gestor de base de datos asigna a los errores), simplifica el manejo de errores, y reduce considerablemente la cantidad de código necesario.
- Integración con Hibernate, JDO e iBatis SQL Maps en términos de soporte a implementaciones DAO y estrategias con transacciones. Especial soporte a Hibernate añadiendo convenientes características de IoC, y solucionando muchos de los comunes problemas de integración de Hibernate. Todo ello cumpliendo con las transacciones genéricas de Spring y la jerarquía de excepciones DAO.

Toda esta funcionalidad puede usarse en cualquier servidor JEE, y la mayoría de ella ni siquiera requiere su uso. El objetivo central de Spring es permitir que objetos de negocio y de acceso a datos sean reutilizables, no atados a servicios JEE específicos. Estos objetos pueden ser reutilizados tanto en entornos JEE (Web o EJB), aplicaciones “standalone”, entornos de pruebas, etc., sin ningún problema.

La arquitectura en capas de Spring ofrece mucha de flexibilidad. Toda la funcionalidad está construida sobre los niveles inferiores. Por ejemplo se puede utilizar la gestión de configuración basada en JavaBeans sin utilizar el framework MVC (Modelo Vista Controlador) o el soporte AOP (Programación Orientada a Aspectos).

Arquitectura de Spring

Spring prácticamente está formado por siete módulos los cuales se presentan claramente en el siguiente gráfico:

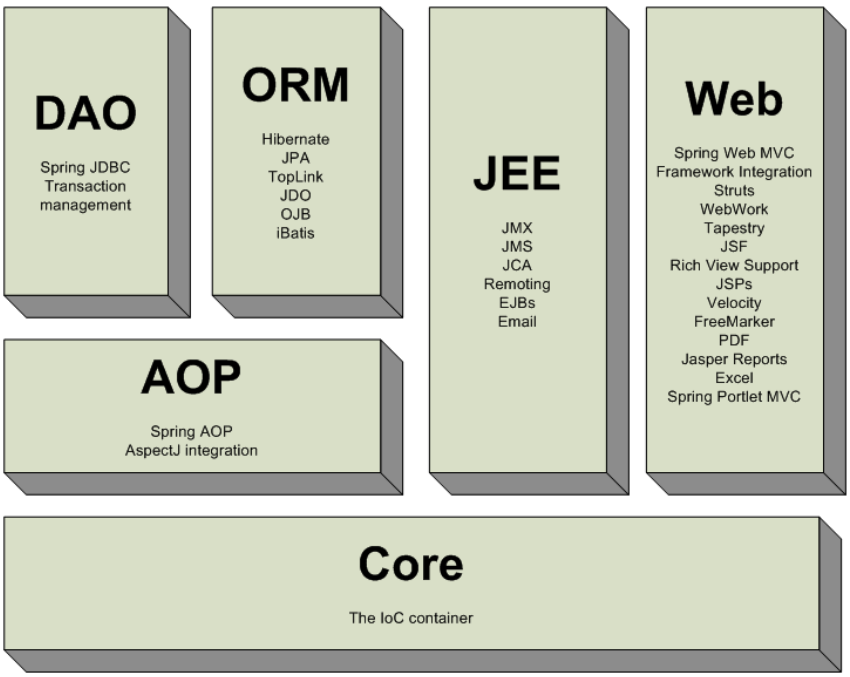


Figura 2.7 Módulos que conforman Spring.

Fuente: http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/overview.html

El paquete Core es el más importante y fundamental para Spring, y provee independencia de cada una de las capas anteriores, su concepto fundamental se basa en un BeanFactory el cual provee una implementación sofisticada del Patrón de diseño Factory, ya explicitado en la sección patrones de diseño, el cual evita los pequeños problemas programáticos y nos permite desacoplar la configuración de las aplicaciones y especificación de dependencias desde una aplicación lógica a otra.

Uno de los paquetes de Spring es el paquete Web en el cual podemos claramente distinguir la utilización y la integración de varios framework de desarrollo Web, como JSF (Java Server Faces), Struts, Spring Web MVC, entre otros de mucha importancia, además de esto Spring provee un soporte de internacionalización que es la capacidad de una aplicación de soportar varios idiomas mediante la utilización de archivos de recursos y otras técnicas para la capa de presentación. En la imagen podemos ver cada unos de los

frameworks o formas de presentación de la capa Web que utiliza Spring.

El paquete DAO proporciona una capa de abstracción JDBC de tipo Spring que evita la necesidad de hacer JDBC tedioso y complicado, codificando y analizando bases de datos con una lista de códigos de errores específicos y claros para la depuración de las aplicaciones. También, el paquete de JDBC proporciona una manera de manejar la transaccionalidad de las aplicaciones desde interfaces y clases POJO's.

El paquete ORM mantiene las capas de las aplicaciones integradas mediante la integración de API's de manejo de objetos correlativos, entre estos están JPA, JDO, Hibernate, e iBatis. Con la utilización del paquete ORM podemos incluir en nuestras aplicaciones integración y combinación de capas y manejo de la transaccionalidad mediante la utilización de archivos mapeo XML.

El paquete AOP de Spring proporciona un AOP Alliance-compliant aspect-oriented donde está implementado métodos que permiten al desarrollador definir métodos de intersección, puntos de la aplicación donde se duplica el código, este nos permite separar la funcionalidad llevando la estructura de nuestra aplicación a un nivel más alto de abstracción tanto a nivel de clases como de atributos.

El paquete MVC de Spring proporciona un Modelo-Vista-Controlador (MVC) a las aplicaciones Web. El framework MVC de Spring no es sólo cualquier aplicación vieja; proporciona una separación limpia entre el dominio, el código y los formularios de las páginas JSP.

Hibernate en el contexto de Spring

Para el desarrollo de las capas de aplicación de proyectos JEE, se hace muy necesaria la utilización de Spring como framework de integración de las capas de persistencia, gestión y servicios, además del manejo de transaccionalidad con las distintas bases de datos que la soporten.

Los desarrolladores de Spring en su afán de hacer la vida más fácil a los programadores proporcionaban desde Hibernate 2 unas clases de apoyo para integrar Hibernate en una

filosofía IoC, actualmente dichas clases están deprecadas y se recomienda no utilizarlas. Por otro lado se puede utilizar el concepto de Inyección de Dependencias (DI) para inyectar directamente el SessionFactory de Hibernate en las clases DAO, para lo cual la debemos definir como un Bean en la configuración de Spring al igual que el DataSource, conceptos que veremos enseguida.

Para configurar Spring necesitamos un archivo xml al cual llamaremos **applicationContext.xml.** A continuación veamos la estructura de este archivo en un ejemplo vinculado con Hibernate:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
http://www.springframework.org/dtd/springbeans.dtd">

<context:component-scan base-package="dao"/>
<context:annotation-config/>

    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/ejemplo"/>
      <property name="username" value="root"/>
      <property name="password" value=""/>
    </bean>

    <bean id="SessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFa
ctoryBean">
        <property name="configLocation">
            <value>/WEB-INF/hibernate.cfg.xml</value>
        </property>
         <property name="dataSource">
            <ref local="dataSource"/>
        </property>
        <property name="packagesToScan" value="entities"/>
    </bean>

    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="SessionFactory"/>
    </bean>
</beans>
```

Como podemos ver, se puede vincular Hibernate con Spring para tener un mayor control de la aplicación. Spring tiene clases que nos ayudan a vincular mediante su archivo xml nuestras clases Hibernate y darles manejo transaccional más claro, rápido y efectivo.

Para la vinculación entre Hibernate y Spring, a continuación se presenta el Bean de Hibernate SessionFactory creado en el archivo de configuración de Spring **applicationContext.xml:**

```
<bean id="SessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFa
ctoryBean">
        <property name="configLocation">
            <value>/WEB-INF/hibernate.cfg.xml</value>
        </property>
         <property name="dataSource">
            <ref local="dataSource"/>
        </property>
        <property name="packagesToScan" value="entities"/>
    </bean>
```

El Bean Hibernate SessionFactory que es creado mediante el tag bean, al cual se le llama SessionFactory asignado mediante el atributo id y para crearlo se utiliza el atributo class donde se asigna el paquete y la clase de la cual se desea sea el bean org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean. A este Bean se le agregan varias propiedades que son las siguientes:

- **configLocation.** Esta propiedad se crea mediante el atributo name e indica mediante el tag value la ubicación del archivo de configuración de Hibernate (**hibernate.cfg.xml**).
- **dataSource.** Esta propiedad se crea mediante el atributo name y hace referencia a un bean configurado en el mismo archivo XML. Para determinar ésto, se utiliza el atributo ref que quiere decir que el tipo de esta propiedad es de un Bean del mismo archivo llamado dataSource.
- **packagesToScan.** Esta propiedad se crea mediante el atributo name y mediante el atributo value se hace referencia al paquete que contiene las clases persistentes.

En el siguiente código está definido el Bean dataSource, con la configuración de acceso a la base de datos:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/ejemplo"/>
      <property name="username" value="root"/>
      <property name="password" value=""/>
</bean>
```

Tomemos en cuenta que en el Bean sessionFactory ya estamos explicitando el paquete que contiene las entidades a ser mapeadas y en el Bean dataSource ya estamos especificando la conexión a la base de datos, por lo tanto estas características que también se encontraban en el archivo de configuración de Hibernate (**hibernate.cfg.xml**) estarían sobrando. El archivo quedaría como se ve a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE    hibernate-configuration    PUBLIC    "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
       org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.current_session_context_class">
       thread
    </property>
    <property name="hibernate.hbm2ddl.auto">
       update
    </property>
  </session-factory>
</hibernate-configuration>
```

Finalmente delegamos el SessionFactory de Hibernate a Spring para que se encargue de gestionarlo a través de su clase HibernateTransactionManager en un Bean llamado transactionManager:

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="SessionFactory"/>
</bean>
```

2.5.2 Implementando los Data Access Objects (DAO's) con Spring

Se debe considerar en que forma se habrá de codificar la lógica de persistencia, es decir el acceso, y manipulación de los objetos. En general, mezclar código de acceso a la base de datos con el de aplicación no es buena idea porque va en contra del principio de separación de código perteneciente a distintos ámbitos (infraestructura, aplicación, datos, presentación, etc.). Con el objetivo de mantener dicha separación y proporcionar una forma consistente de programar la lógica de acceso a datos, se utilizará el patrón de diseño conocido como DAO o Data Access Object.

El patrón DAO tiene su origen en las directivas de diseño de Sun Microsystems para la plataforma Java EE. El DAO define un interfaz para las operaciones básicas de persistencia: creación, lectura, modificación y borrado (definidas a menudo en la literatura con el acrónimo inglés operaciones CRUD), más métodos de búsqueda, y es relativo a una entidad específica del modelo de datos.

Una vez vista la manera de integrar Spring con Hibernate en la capa de persistencia ya podemos implementar las clases DAO. Nuevamente usaremos el concepto IoC para guardar estas clases como Beans de Spring. Las siguientes líneas deben añadirse al archivo **applicationContext.xml:**

```
<context:component-scan base-package="dao"/>
<context:annotation-config/>
```

Revisemos las características de las líneas anteriores:

- **<context:component-scan base-package="dao"/>:** Se especifica el paquete que contiene las clases DAO**.** Spring escanea automáticamente el paquete especificado y registra las clases como Beans automáticamente.
- **<context:annotation-config/>:** Habilita las anotaciones Spring y la inyección de dependencias.

Recordemos que DAO es un objeto de Acceso a los datos, es decir, será la clase donde resida la lógica de manejo de Hibernate. De esta forma conseguimos que nuestra lógica de negocio no sepa nada de Hibernate, y siempre que quiera acceder a los datos lo hará usando esta clase. Veamos un ejemplo genérico sencillo: Primero definimos una interface, así podemos intercambiar la implementación fácilmente:

```
public interface Dao {
   public void save(Object entity);     //Guarda una entidad
   public void save(Object[] entities); //Guarda las entidades
                                        //contenidas en un arreglo

   public <T> T find(Long id);  //Busca un registro por su campo
                                //Identificador (Id)
}
```

Para el ejemplo sólo hemos definido algunas operaciones simples. Ahora veamos la implementación de esta interface usando las facilidades que nos proporciona Spring y Hibernate:

```
@Repository
public class SpringHibernateDao implements Dao{

   @Autowired
   private SessionFactory sessionFactory;

   public void save(Object entity) {
        Transaction trans = this.getCurrentSession().beginTransaction();
        this.getCurrentSession().save(entity);
        trans.commit();
   }

   public void save(Object[] entities) {
       Transaction trans = this.getCurrentSession().beginTransaction();
       for (int i = 0; i < entities.length; i++) {
              this.getCurrentSession().save(entities[i]);
              trans.commit();
        }
   }

   public <T> T find(Long id) {
   Transaction trans = this.getCurrentSession().beginTransaction();
   final T entity = (T) this.getCurrentSession().load(entityClass, id);
         return entity;
   }

   protected final Session getCurrentSession() {
             return this.sessionFactory.getCurrentSession();
  }
}
```

Vamos a hacer especial hincapié en las nuevas anotaciones que se presentan en la clase de ejemplo SpringHibernateDao:

- En la línea 1 nos encontramos con **@Repository**. Esta es una anotación de Spring. Estamos indicando que ésta es una clase relacionada con la capa de persistencia, y que debe ser un Singleton.
- En la línea 4 nos encontramos con **@Autowired**. Esta es una anotación de Spring. Sirve para indicarle a Spring que cuando vaya a crear la instancia de SpringHibernateDao debe "inyectarle" (pasarle) en el constructor una referencia al SessionFactory (el SessionFactory, como vimos anteriormente, se configura en el Bean de Spring).

Veamos otro ejemplo concreto de una interfaz y una implementación para una clase Contact (Contacto), esta vez haciendo hincapié en la implementación de los métodos. Para mantener un estándar de desarrollo llamaremos a la interface con el nombre de su entidad seguido de la palabra DAO (EntityDao):

```
public interface ContactDAO {
    public void addContact(Contact contact);
    public List<Contact> listContact();
    public void removeContact(Integer id);
}
```

Para la clase que implementa la interface la llamaremos de manera idéntica pero añadiendo la palabra Impl (EntityDAOImpl):

```
@Repository
public class ContactDAOImpl implements ContactDAO {

    @Autowired
    private SessionFactory sessionFactory;

    public void addContact(Contact contact) {
        Transaction trans = this.getCurrentSession().beginTransaction();
        sessionFactory.getCurrentSession().save(contact);
        trans.commit();
    }

    public List<Contact> listContact() {
     Transaction trans = this.getCurrentSession().beginTransaction();
     return this.getCurrentSession().createQuery("from Contact").list();
    }

    public void removeContact(Integer id) {
      Transaction trans = this.getCurrentSession().beginTransaction();
      Contact contact = (Contact)
this.getCurrentSession().load(Contact.class, id);
        if (null != contact) {
            this.getCurrentSession().delete(contact);
            trans.commit();
        }
    }

    protected final Session getCurrentSession() {
              return this.sessionFactory.getCurrentSession();
    }
}
```

Revisemos los métodos de la clase ContactDAOImpl:

- Primero, el método propio de la clase implementación **getCurrentSession()** nos devuelve una instancia de un objeto de tipo Session con el cual podemos realizar

transacciones hacia la base de datos. Cabe destacar que tiene el modificador final, el cual no permite que este método sea sobrescrito por herencia.

- Segundo, el método **beginTransaction()** indica la apertura hacia una transacción con un objeto del tipo Transaction y el método **commit()** actúa sobre este para hacer un commit a la base y confirmar la transacción.
- Tercero, el método **addContact(Contact contact)** recibe como parámetro una entidad Persistente de tipo Contact el cual es referenciado hacia el método **save(contact)**. El método **save(Object obj)** pertenece al objeto Session el cual permite persistir dicha entidad enviada como parámetro. En resumen este método añade un registro en la base de datos.
- Cuarto, el método **List<Contact> listContact()** retorna una lista de objetos del tipo Contact. El método **createQuery(String str)** pertenece al objeto Session el cual crea una nueva instancia de un Query para la sentencia HQL pasada como parámetro a través de un String. Adicionalmente el método **.list()** devuelve una lista genérica de las Entidades consultadas. Como se vio en el punto 2.4.3 la sentencia "from Contact" se encuentra en Hibernate Query Language (HQL) y su equivalente en SQL es "Select * from Contact" la cual devuelve desde la base todos los registros de la tabla Contact. A través del método **createQuery(String str)** podemos ejecutar cualquier sentencia hacia la base de datos a través del lenguaje HQL.
- Por último, el método **removeContact(Integer id)** elimina un registro de la base de datos. Recibe como parámetro un identificador de tipo Integer.
  Analicemos la siguiente línea:

```
Contact contact = (Contact)
this.getCurrentSession().load(Contact.class, id);
```

El método **load(Class,id)** pertenece al objeto Session y retorna una instancia persistente de tipo Object, la cual se busca de acuerdo a los dos parámetros dados, el primero es la clase Entidad en la cual se realizará la búsqueda y el Id es el identificador de la Entidad. Para este caso se guarda la Entidad buscada en una variable de tipo Contact para lo cual es necesario hacer el correspondiente Casting.

El ejemplo anterior es una muestra de una implementación sencilla para clases DAO en la capa de persistencia. Estas clases serán utilizadas en la siguiente capa de esta arquitectura (capa de negocio) en la que se volverá a utilizar las bondades de Spring para hacer posible la conexión entre ellas, todo esto se verá en el siguiente capítulo.

2.6 Resumen de Patrones Aplicados

En resumen, la propuesta para esta capa se apoya en la arquitectura de persistencia de Hibernate, encapsulando toda su lógica transaccional en una subcapa donde se alojan los ya mencionados Data Access Objects (DAO's).

Se utiliza Spring para integrarlo con Hibernate a través de Beans en su archivo de configuración, para que, a través de su contenedor del patrón tipo Factory, se haga cargo de gestionar la SessionFactory y por medio del IoC sea fácilmente referenciado en las clases del patrón DAO que también son guardados como Beans.

2.6.1 Frameworks Utilizados

Hibernate

Motor de persistencia de código abierto que permite mapear un modelo de clases a un modelo relacional sin imponer ningún tipo de restricción en ambos diseños.

Otras alternativas descartadas:

- **EJB 3** dispone de un tipo de EJB, los de Entidad, que permiten gestionar la persistencia de la información. Precisamente estos EJB de Entidad se basan en JPA para realizar su labor. Esto quiere decir que cualquier desarrollo con JPA y sin usar EJB sería fácilmente portable a un entorno de EJB si fuera necesario. Y la ventaja de trabajar directamente con JPA en lugar de con EJB es, como ya se ha mencionado anteriormente, que los EJB son muy pesados sobre todo en tiempo de desarrollo, alargando los períodos de construcción del sistema.
- **iBatis** no es exactamente un mapeador objeto-relacional, o al menos en el sentido puro del término. Es un sistema para mapear objetos contra una base de datos relacional mediante DAO's en donde todo el código en SQL debe ser escrito (no dispone de los mapeos que ofrecen a Hibernate la potencia que le ha hecho famoso). Es más sencillo que Hibernate y parece un proyecto muy interesante cuando hay que trabajar contra un modelo de base de datos ya existente siempre y

cuando ésta no sea muy compleja o no tenga muchas peculiaridades o defectos de diseño.

- **Torque** otra de las opciones de mapeo por medio de DAO's. No se encontraron muchos proyectos, fuera de la esfera del propio Torque, que lo utilizasen ni dispone de la potencia de Hibernate (a nivel de mapeos y utilidades como su doble caché, carga perezosa, etc). Además su comunidad es, comparada con la de Hibernate, extremadamente reducida.

Spring

Conjunto de librerías entre las que podemos escoger aquellas que faciliten el desarrollo de nuestra aplicación. Entre sus posibilidades más potentes está su contenedor de Inversión de Control, la programación orientada a aspectos, plantillas de utilidades para Hibernate, iBatis y JDBC así como la integración con JSF. En esta propuesta, dentro de la capa de persistencia y a lo largo de este trabajo, lo utilizaremos como contenedor de Beans a través de su IoC.

Otras alternativas descartadas:

- **Avalon** fue una de las primeras opciones consideradas y es uno de los contenedores IoC más antiguos. Sin embargo, lejos de ser una ventaja, esto se nota mucho en su diseño, mucho menos flexible y elegante que Spring. Su comunidad es realmente pequeña y no se han encontrado muchas referencias a usos en aplicaciones reales.
- **Picocontainer y Nanocontainer** pequeños contenedores de IoC que no disponen con muchísima diferencia de la funcionalidad y conjunto de librerías de Spring. Parecen útiles para ser empleados en applets y en aplicaciones J2ME, pero no en una aplicación de tipo Enterprise.
- **HiveMind** una de las opciones más interesantes después de Spring. De la mano de Apache es un completo contenedor IoC. Se han visto sin embargo tres grandes áreas en donde Spring es superior: conjunto de librerías, apoyo de la comunidad a todos los niveles y elegancia de uso.

CAPITULO III

DISEÑO E IMPLEMENTACIÓN DE LA CAPA DE NEGOCIO

Uno de los objetivos de la arquitectura de software es proporcionar una plataforma común de objetos que implementen procesos del negocio. Los componentes de un sistema, tanto aplicaciones de Web como programas independientes, precisan invocar estos procesos para ejecutar funcionalidad demandada por los usuarios, o bien realizar tareas de servidor, que en muchos casos son los mismos.

En general, los procesos del negocio manejan información privada como dinero, valores, etc. Los fallos en un sistema que maneje este tipo de datos tienen repercusiones de gran importancia para los clientes y las empresas.

En un sistema de este tipo, una estrategia bien definida de clases de servicios comunes no es simplemente una forma de garantizar la calidad del código, sino que es fundamental para garantizar el correcto funcionamiento del sistema. La lógica del negocio es demasiado complicada para estar desorganizada.

A la hora de elegir una tecnología de componentes de negocio en la plataforma Java EE, lo primero que se debe analizar es cuales son las necesidades de este tipo de objetos, es decir qué requerimientos de infraestructura. Una vez que se comprende esto, se puede determinar qué paradigma de diseño de componentes y qué tipo de software de soporte se precisa.

Los objetos de negocio precisan los siguientes servicios de infraestructura:

- **Transacciones:** Los objetos de negocio van a precisar el acceso a transacciones que pueden en ciertos casos involucrar distintas fuentes de datos, transaccionales (base de datos, mainframe) o no (sistemas externos).
- **Seguridad:** Por otra parte ha de ser posible definir reglas de autorización para la ejecución de lógica que así lo precise.

En los puntos de este capítulo se describe la propuesta para la capa de negocio, así como las técnicas utilizadas para definir y ligar un objeto de negocio al contenedor Spring, hacerlo accesible a código cliente, y obtener los servicios necesarios.

3.1 Propuesta de la capa de negocio

Los requisitos técnicos de los objetos de negocio expuestos en los párrafos anteriores pueden ser satisfechos utilizando un contenedor EJB, pero puesto que Spring proporciona estos servicios fundamentales y cómo hemos visto supera en muchos aspectos la oferta de los servidores de aplicación, con más sencillez y sin atarnos a un producto comercial, la elección es implementar nuestros objetos de servicio mediante objetos POJO que se ejecutan en el contenedor Spring.

A continuación se detalla de forma gráfica el diseño para esta capa:

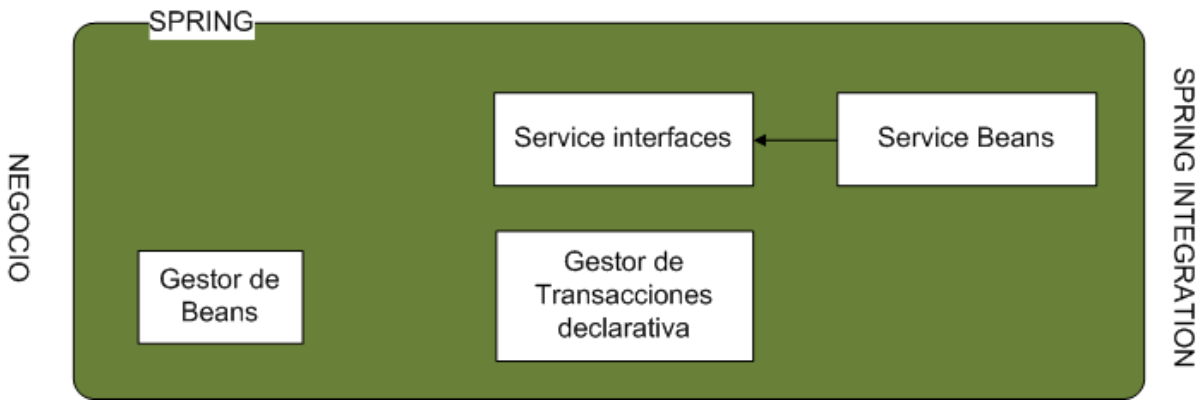


Figura 3.1: Propuesta de capa de negocio

Autor: Diego Hinojosa

3.2 Desarrollo y definición de objetos de negocio

Un objeto de negocio se corresponde con un área funcional y proporciona una serie de operaciones relativas a dicha área. Estas operaciones se definen mediante un interfaz Java. La clase de implementación extiende y realiza este interfaz de negocio.

En terminología Spring los objetos que constituyen la capa de negocio y se ejecutan dentro del contenedor Spring se denominan Beans. Un Bean es simplemente un objeto que se instancia, se enlaza y vive dentro del contenedor. Cada Bean y su conjunto de dependencias

deben ser declarados mediante información de configuración que Spring necesita durante su inicialización.

Como se vio en el capítulo 2, la definición de Beans se realiza mediante XML en el archivo de configuración de Spring el cual tiene el siguiente formato:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xsi:schemaLocation="http://www.springframework.org/schema/beans/spring
beans-2.5.xsd ">

 <!-- Definición de un bean -->
 <bean id="…" class="…">

 <!—dependencias y demás configuración aquí -->
 </bean>
</beans>
```

3.3 Inyección de dependencias

En el capítulo 2 vimos a breves rasgos este tema, ahora lo profundizaremos porque tomará fuerza a partir de este punto.

Este paradigma de programación se conoció inicialmente por el nombre Inversión de Control o IoC. El concepto fue formulado por Martin Fowler (Fowler, 2004), en el que se planteaba la cuestión de qué aspecto del control de la aplicación se había de invertir para conseguir una aplicación desacoplada. La conclusión fue que era necesario invertir el control de dependencias entre clases. De ahí que se comenzará a utilizar el término Inyección de Dependencias normalmente referido en la literatura en inglés con el acrónimo DI.

Cualquier aplicación no trivial utiliza muchas clases y colaboraciones entre clases para implementar la lógica. Tradicionalmente, cada objeto es responsable de obtener referencias a otros objetos de los que depende. Esto convierte al código en altamente dependiente entre sí haciendo más difícil el mantenimiento. El uso de interfaces discretas es una técnica clásica para controlar el número de dependencias.

La técnica de inyección invierte la responsabilidad de obtención de dependencias de la siguiente forma: el contenedor de objetos, Spring en nuestro caso, proporciona a un objeto todas las referencias a los objetos con los que colabora en el momento de creación. En otras

palabras, el objeto sencillamente declara sus dependencias, y Spring inserta las referencias. El objeto cliente sólo conoce el interfaz público de la dependencia, no su implementación, que puede ser intercambiada, no sólo en tiempo de ensamblado, sino también en tiempo de ejecución. Spring permite incluso el enlazado en tiempo de ejecución con clases de implementación escritas en lenguajes interpretados emparentados con Java como Groovy, Ruby, y otros, abriendo las puertas a la interacción con plataformas de desarrollo dinámico que se utilizan en aplicaciones especiales como juegos.

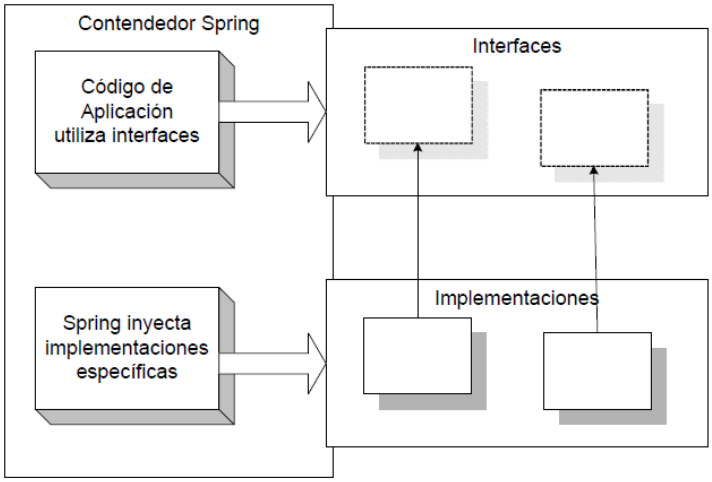


Figura 3.2: Inyección de Dependencias

Autor: Diego Hinojosa

Un Bean puede depender bien de otros Beans o de variables cuyo tipo es una primitiva del lenguaje Java y que son configurables. Las dependencias se declaran mediante XML, y pueden tomar la forma de un argumento del constructor, o una propiedad del Bean con su par de métodos get y set. Por ejemplo usando el estilo de dependencias mediante propiedades:

```
<bean id="bean1" class="BeanClass1">
 <!– primitive -->
 <property name="dependency1" value="1"/>

 <!– el atributo ref contiene la referencia al id de otro bean -->
 <property name="dependency2" ref="bean2"/>

</bean>
```

```
public class BeanClass1 implements Bean1 {

  private int dependency1;
  private Bean2 dependency2;

  public void setDependency1(int dependency1) {
    this.dependency1=dependency1;
  }

  public void setDependency2(Bean2 dependency2) {
    this.dependency2=dependency2;
  }

  // …getters etc
}
```

Cuando Spring instancia un Bean de tipo **BeanClass1**, invocará **setDependency1** con valor 1 y **setDependency2** con el objeto **bean2**.

Alternativamente también es posible proporcionar esta información utilizando el mecanismo de anotaciones directamente en la clase Java que implementa el Bean. La anotación **@Autowired** es equivalente a una declaración de dependencia como vimos en el punto 2.5.2 cuando inyectamos el Bean SessionFactory en nuestras clases DAO:

```
@Autowired
private Bean2 dependency2;
```

Es posible configurar Spring de tal forma que, durante su inicialización, obtenga la información de configuración necesaria mediante un escáner del classpath de la aplicación. Es decir, Spring lee archivos .class y busca ciertas anotaciones como **@Autowired** que proporcionan metadata. Si un archivo .class contiene alguna de estas anotaciones especiales, entonces Spring decide que dicha clase es un bean. Usar **@Autowired** es por tanto equivalente a definir un bean en el archivo XML y puede reducir considerablemente el mantenimiento de metadata de configuración. No obstante, ciertos tipos de configuración sólo están disponibles a través de XML.

3.4 Demarcación de transacciones

Como hemos visto, un objeto de negocio necesita definir transacciones a nivel de lógica de negocio. Tradicionalmente los contenedores Java EE han hecho disponibles dos tipos de transacciones: locales y globales.

Transacciones locales involucran una sola fuente de datos, como la base de datos. Por ejemplo una transacción local se utiliza cuando se precisa crear un nuevo registro en una tabla de la base de datos. El API JDBC proporciona este tipo de transacciones.

Transacciones globales involucran más de una fuente de datos y requieren de un protocolo del tipo two-phase commit. El API JTA proporciona este tipo de transacciones. JDBC es parte de Java Estándar Edition y por tanto no precisa de un servidor de aplicación Java EE. JTA por otra parte precisa de Java EE y por lo tanto su uso directamente en el código de aplicación reduce la portabilidad.

Spring resuelve este problema mediante su extenso soporte para transacciones. Por un lado Spring proporciona una abstracción común y sencilla para la gestión de transacciones, que es capaz de utilizar de forma transparente muchas de las APIs disponibles (JDBC, JTA, y otras como Hibernate o JDO) y por otro permite la gestión de transacciones tanto de forma programática (mediante el uso de un API directamente en la lógica del negocio), como declarativa (mediante el uso de metadata, fuera de la lógica del negocio).

La elección para la demarcación de transacciones es el estilo declarativo puesto que al no requerir el uso de un API determinado, el código de negocio tiene mayor portabilidad.

Spring proporciona transacciones declarativas mediante el mecanismo de programación orientada a aspectos. Una transacción declarativa se puede definir mediante XML o, mediante la anotación **@Transactional**.

Por ejemplo, utilizando la anotación **@Transactional** se puede indicar que todos los métodos de una clase han de ser transaccionales:

```
@Transactional
public class …
```

O un método determinado:

```
@Transactional
void sellActions(Transaccion t) throws ExcepcionTransaction
{..
```

Tras esta anotación se esconde un aspecto. Durante la inicialización de una clase con operaciones transaccionales, Spring construye un objeto especial del tipo dynamic proxy que es un tipo de objeto disponible en J2SE. Este proxy es el responsable de ejecutar el aspecto transaccional.

3.4.1 ¿Qué es un Aspecto?

Es un concepto de AOP o programación orientada a aspectos. Un Aspecto es un tipo de funcionalidad que se puede aplicar a objetos de forma declarativa, u ortogonal al flujo de la aplicación. Un Aspecto se implementa mediante una clase Java, y es ejecutado por el contendor AOP (como Spring). El contenedor precisa de metadata para saber sobre qué conjunto de objetos se ha de ejecutar el aspecto y cuándo, es decir en qué momento de la ejecución (instanciación, ejecución de un método, etc) se habrá de procesar el aspecto.

3.4.2 ¿Qué es un Proxy dinámico?

Un proxy dinámico es una primitiva del lenguaje Java que permite definir un tipo de objeto que es capaz de interceptar la invocación de un objeto y realizar tareas ortogonales al flujo normal de la aplicación. El proxy implementa la misma interfaz o interfaces que el objeto Java y por tanto el código cliente no sabe que está utilizando el proxy en lugar del objeto.

Cuando el código de aplicación solicita una instancia del objeto transaccional, Spring proporciona el proxy en lugar del objeto. Cuando el código invoca el método transaccional, el proxy se encarga de obtener una transacción, ejecutar el método en el objeto, y dependiendo de si el método finaliza con éxito o genera una excepción, el proxy completará con éxito (commit) o abortará (rollback) la transacción.

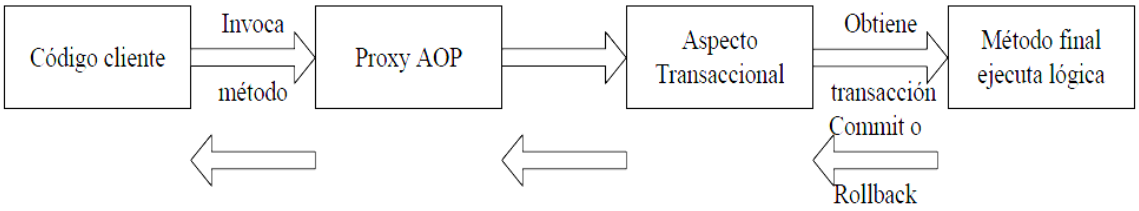


Figura 3.3: Aspecto Transaccional

Autor: Diego Hinojosa

Spring proporciona atributos adicionales para la anotación **@Transactional** que permiten hilar más fino el proceso de las transacciones y su propagación, permitiendo detallar el comportamiento del aspecto transaccional incluyendo: reglas de propagación de transacciones durante la invocación de varios métodos transaccionales, valor de timeout, valor del nivel de aislamiento o isolation level, reglas de rollback, etc.

El soporte de Spring para transacciones no acaba aquí y es mucho más amplio que lo mencionado en esta sección, y más extenso que el actual estándar EJB. Spring se puede combinar con programación de aspectos en formas muy sofisticadas pero no es nuestro objetivo explicar todo lo que es posible, sino las decisiones de infraestructura relativas a nuestra capa de negocio y las técnicas más comunes utilizadas en nuestra aplicación.

3.5 Implementando las clases de servicio con Spring

Para entender esta implementación y reunir todos los conceptos que se han visto en los párrafos anteriores de este capítulo, tomemos como referencia la interface ContactDAO y su implementación ContactDAOImpl de la capa de persistencia:

```
public interface ContactDAO {

    public void addContact(Contact contact);
    public List<Contact> listContact();
    public void removeContact(Integer id);
}
```

```
@Repository
public class ContactDAOImpl implements ContactDAO {

    @Autowired
    private SessionFactory sessionFactory;

    public void addContact(Contact contact) {
        Transaction trans = this.getCurrentSession().beginTransaction();
        sessionFactory.getCurrentSession().save(contact);
        trans.commit();
    }

    public List<Contact> listContact() {
     Transaction trans = this.getCurrentSession().beginTransaction();
     return this.getCurrentSession().createQuery("from Contact").list();
    }

    public void removeContact(Integer id) {
      Transaction trans = this.getCurrentSession().beginTransaction();
      Contact contact = (Contact)
this.getCurrentSession().load(Contact.class, id);
```

```
        if (null != contact) {
            this.getCurrentSession().delete(contact);
            trans.commit();
        }
    }

    protected final Session getCurrentSession() {
             return this.sessionFactory.getCurrentSession();
    }
}
```

Nuevamente vamos a utilizar Spring para conectar el objeto de negocio con su clase DAO correspondiente de la capa de persistencia utilizando las bondades del paradigma DI (Inyección de Dependencias) a través de la anotación @Autowired.

Al igual como se vio en el capítulo 2, necesitamos añadir el paquete de clases de servicio al contexto de Spring de la siguiente manera en el archivo **applicationContext.xml:**

```
<context:component-scan base-package="service"/>
```

Tal como se vio en la capa de persistencia, seguiremos trabajando con interfaces para mantener un bajo acoplamiento.

Manteniendo un estándar de desarrollo el nombre de la interface estará conformado de la siguiente manera: EntityService o ActionService. Las clases que implementen las interfaces correspondientes se nombrarán de la misma forma pero añadiendo la palabra Impl: EntityServiceImpl o ActionServiceImpl.

Veamos a continuación la implementación de las clases de servicio:

```
public interface ContactService {

    public void addContact(Contact contact);
    public List<Contact> listContact();
    public void removeContact(Integer id);
}
```

```
@Service
public class ContactServiceImpl implements ContactService {
```

```
    @Autowired
    private ContactDAO contactDAO;

    @Transactional
    public void addContact(Contact contact) {
        contactDAO.addContact(contact);
    }

    @Transactional
    public List<Contact> listContact() {
         return contactDAO.listContact();
    }

    @Transactional
    public void removeContact(Integer id) {
        contactDAO.removeContact(id);
    }
}
```

La implementación de los métodos es suficientemente clara y sencilla como para explicarla, sabemos que hacen referencia a los métodos del DAO.

Revisemos las anotaciones que aparecen esta vez en la clase ContactServiceImpl:

- En la línea 1 nos encontramos con **@Service**. Esta es una anotación de Spring, similar a @Repository que ya habíamos visto en el capítulo 2. Estamos indicando que esta es una clase relacionada con la capa de servicio (clases de negocio), y que debe ser un Singleton.
- En la línea 4 nos encontramos de nuevo con **@Autowired**. Como ya vimos en el concepto de DI, esta es una anotación de Spring la cual sirve para "inyectarle" una referencia al DAO correspondiente (en este caso ContactDAO).
- En las línea 7, 12 y 17 nos encontramos con la anotación **@Transactional**. Como se vio en el punto 3.4 esta es una anotación de Spring con la cual estamos indicando que el método en cuestión es transaccional. Lo que hará Spring es comprobar si ya existe una transacción abierta, si existe se unirá a ella, y si no existe, abrirá una nueva transacción (este comportamiento es configurable). De esta forma nos aseguramos que toda operación de la base de datos se realiza dentro de una transacción. Además si durante la ejecución del método se produce alguna excepción de Runtime, se hará automáticamente rollback de la transacción (este comportamiento también es configurable).

El ejemplo anterior es una muestra de una implementación sencilla para clases de servicio en la capa de negocio, conectadas con la capa de persistencia a través de Spring y DAO. Estas clases serán utilizadas en la siguiente capa de esta arquitectura (capa de presentación) en la que volveremos a utilizar las bondades de Spring para hacer posible la conexión entre ellas, todo esto se verá en el siguiente capítulo.

3.6 Resumen de Patrones Aplicados

En resumen, la propuesta para esta capa maneja objetos de negocio registrados como Beans de Spring, los cuales, toman referencia a través del paradigma de Inyección de Dependencias de los objetos de acceso de datos de la capa inferior para desarrollar la lógica del negocio como tal.

Se continúa con el uso de Spring como contenedor de Beans, pero para simplificar el mantenimiento del archivo XML de configuración se decidió utilizar la anotación @Autowired en las propias clases de servicio.

3.6.1 Frameworks Utilizados

Spring

En esta propuesta de la capa de negocio se hace uso de Spring nuevamente como contenedor de Beans (IoC) y utilizando su paradigma DI se establecen las referencias desde las clases de servicio hacia los DAO's de la capa inferior.

CAPITULO IV

DISEÑO E IMPLEMENTACIÓN DE LA CAPA DE PRESENTACIÓN

Hasta ahora hemos estructurado una arquitectura independiente, modular y totalmente intercambiable, es así que, hemos llegado a la capa de presentación en la cual podemos integrar cualquier framework para generar la vista hacia el cliente independientemente de las capas anteriores.

La capa de presentación es la encargada de lograr que la funcionalidad de la aplicación esté disponible a World Wide Web (WWW). Accede a datos y reglas de negocio alojadas en otras capas y maneja el flujo de las pantallas. Típicamente interactúa con la capa cliente usando HTTP, y puede acceder a otras capas utilizando diversos protocolos. En escenarios donde se utilizan Web Services, otras aplicaciones pueden ocupar el lugar de la capa cliente, accediendo a la aplicación por intermedio de la capa Web.

En los puntos de este capítulo se describe la propuesta para la capa de presentación basada en el framework JavaServer Faces utilizando una implementación relativamente nueva en el mercado como lo es PrimeFaces.

4.1 Introducción al framework JavaServer Faces (JSF)

JavaServer Faces es el estándar presentado por Sun para la capa de presentación Web. Forma parte de la especificación JEE 5 que deberán cumplir todos los servidores de aplicaciones y se presenta como una evolución natural de los frameworks actuales hacia un sistema de componentes.

Es un estándar sencillo que aporta los componentes básicos de las páginas web además de permitir crear componentes más complejos (menús, pestañas, árboles, etc). Ya hay disponibles diferentes implementaciones de la especificación, tanto comerciales como de código abierto, así como librerías de componentes adicionales que amplían la funcionalidad de esos componentes iniciales.

JSF ha sido acogida por la comunidad como "el framework que hacía falta". Muchos de los proyectos de código abierto y las compañías con más influencia lo han identificado como el framework de presentación web del futuro.

Características

La tecnología JavaServer Faces constituye un marco de trabajo (framework) de interfaces de usuario del lado de servidor para aplicaciones web basadas en tecnología Java y en el patrón MVC (Modelo Vista Controlador).

Incluye un API y una implementación de referencia para:

- Representar componentes de interfaz de usuario (UI-User Interface) y manejar su estado
- Manejar eventos, validar en el lado del servidor y convertir datos
- Definir la navegación entre páginas
- Soportar internacionalización y accesibilidad
- Proporcionar extensibilidad para todas estas características.

Además de una librería de etiquetas JavaServer Pages (JSP) personalizadas para dibujar componentes UI dentro de una página JSP.

Este modelo de programación bien definido y la librería de etiquetas para componentes UI facilita de forma significativa la tarea de la construcción y mantenimiento de aplicaciones web con UI's en el lado servidor. Con un mínimo esfuerzo, es posible:

- Conectar eventos generados en el cliente a código de la aplicación en el lado servidor.
- Mapear componentes UI a una página de datos en el lado servidor.
- Construir una interfaz de usuario con componentes reutilizables y extensibles.

Estructura de archivos de una aplicación JSF

A continuación se presenta la jerarquía de archivos presentes en una aplicación con JSF en su forma de distribución:

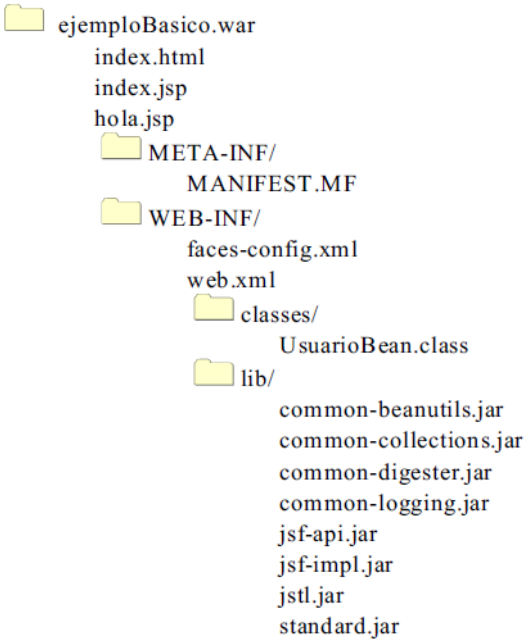


Figura 4.1: Estructura de archivos de una aplicación JSF

Autor: Diego Hinojosa

Por el momento los archivos que nos interesan de esta distribución son **faces-config.xml** y **web.xml**, de los cuales veremos su manera de configurar más adelante.

4.1.1 Modelo Vista Controlador en JSF

El patrón MVC (Modelo Vista Controlador), nos permite separar la lógica de control, la lógica de negocio y la lógica de presentación.

Utilizando este tipo de patrón es posible conseguir más calidad y un mantenimiento más fácil. Al margen de todo esto, una de las cosas más importantes que permite el uso de este patrón consiste en normalizar y estandarizar el desarrollo de Software.

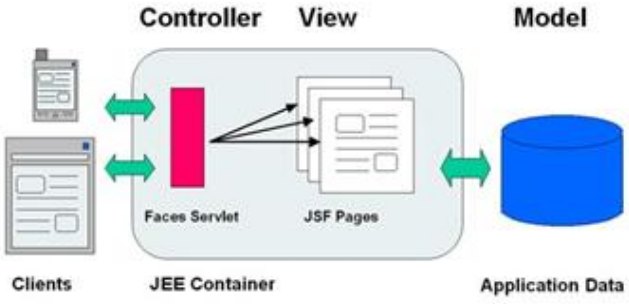


Figura 4.2: Patrón Modelo Vista Controlador de JSF

Fuente: http://aragorn.pb.bialystok.pl/~dmalyszko/PSS_Project/JavaServer%20Faces.htm

Modelo

El modelo es el objeto que representa y trabaja directamente con los datos del programa, gestiona los datos y controla todas sus transformaciones. El modelo no tiene conocimiento específico de los diferentes controladores y/o 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 cuándo deben reflejar un cambio en el modelo.

En el contexto de JSF, el modelo está representado por Beans. Como hemos visto a lo largo de este trabajo, un Bean es una clase Java que contiene atributos. Un atributo es un valor identificado por un nombre, pertenece a un tipo determinado y puede ser leído y/o escrito sólo a través de métodos a tal efecto llamados métodos getter y setter.

Un ejemplo sencillo de un Bean puede ser el siguiente:

```
public class UsuarioBean {
    private String nombre;
    private String password;

 public String getNombre() {
     return nombre;
  }

 public void setNombre(String nuevoValor) {
     nombre = nuevoValor;
  }

 public String getPassword() {
    return password;
  }

 public void setPassword(String nuevoValor) {
    password = nuevoValor;
 }
}
```

En una aplicación JSF, se deben usar Beans para todos los datos accedidos por una página. Los Beans son el puente de comunicación entre la interfaz de usuario (páginas JSF) y la arquitectura de nivel inferior (las clases de servicio en nuestro caso).

Los Beans deben ser definidos en el archivo **faces-config.xml** para que JSF pueda hacer uso de ellos de la siguiente manera:

```
</faces-config>
  <managed-bean>
    <managed-bean-name> usuario </managed-bean-name>
    <managed-bean-class> UsuarioBean </managed-bean-class>
    <managed-bean-scope> session </managed-bean-scope>
  </managed-bean>
</faces-config>
```

Revisemos las etiquetas que aparecen en la definición de un Bean:

- **<managed-bean>:** Delimita la definición de un Bean.
- **<managed-bean-name>:** Se especifica el nombre con el que se va a invocar al Bean desde la Vista.
- **<managed-bean-class>:** Se especifica el original del Bean en el paquete.
- **<managed-bean-scope>:** Se define el ámbito o el alcance del Bean.

Detallemos de mejor manera cuales son los ámbitos de un Bean:

- **Ámbito de tipo petición**: Es el de vida más corta. Empieza cuando una petición HTTP comienza a tramitarse y acaba cuando la respuesta se envía al cliente. Se define a través de la palabra **request**.
- **Ámbito de sesión:** Permanece desde que la sesión es establecida hasta que se termina. Una sesión termina si la aplicación web invoca el método invalidate en el objeto HttpSession o si su tiempo expira. Las aplicaciones Web típicamente colocan la mayor parte de sus Bean dentro de un ámbito de sesión. Por ejemplo, un Bean UsuarioBean puede contener información acerca de usuarios que son accesibles a lo largo de la sesión entera. Un Bean CarritoCompraBean puede irse llenando gradualmente durante las demandas que levantan una sesión. Se define a través de la palabra **session**.
- **Ámbito de tipo aplicación**: Persiste durante toda la aplicación web. Este ámbito es compartido entre todas las peticiones y sesiones. Se define a través de la palabra **application**.

Vista

La vista es el objeto que maneja la presentación visual de los datos gestionados por el Modelo. Genera una representación visual del modelo y muestra los datos al usuario. Interacciona con el modelo a través de una referencia al propio modelo.

La interface para el usuario está representada por las páginas JSF. Se necesita una página JSF por cada pantalla de presentación. Dependiendo de lo que se quiera hacer, las páginas típicas son **.jsp .jsf** y **.xhtml.**

Veamos un ejemplo sencillo de una página .xhtml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
     <h:head>
        <title>JSF 2.0 Hello World</title>
     </h:head>
    <h:body>
         <h3>JSF 2.0 Hello World Example - hello.xhtml</h3>
         <h:form>
            <h:inputText value="#{helloBean.name}"></h:inputText>
            <h:commandButton value="Welcome Me" action="welcome">
             </h:commandButton>
         </h:form>
    </h:body>
</html>
```

Por el momento pongamos especial hincapié en las etiquetas de declaración de importación:

- En la línea 5, se aprecia la importación de la librería de etiquetas **core**, se usan, entre otras aplicaciones, para manejo de eventos, atributos, conversión de datos, validadores, recursos y definición de la página, las cuales usan el prefijo **f**, por ejemplo **f:view**
- En la línea 6, se aprecia la importación de la librería de etiquetas **html_basic**, su utilización es básicamente para la construcción de formularios y demás elementos de interfaz de usuario que hemos usado trabajando con páginas html, usa el prefijo **h**, por ejemplo **h:form**.

Controlador

El controlador es el objeto que proporciona significado a las órdenes del usuario, actuando sobre los datos representados por el modelo. Entra en acción cuando se realiza alguna operación, ya sea un cambio en la información del modelo o una interacción sobre la Vista. Se comunica con el modelo y la vista a través de una referencia al propio

modelo. Además, JSF opera como un gestor que reacciona ante los eventos provocados por el usuario, procesa sus acciones y los valores de estos eventos, y ejecuta código para actualizar el modelo o la vista.

Una aplicación JSF requiere un servlet, llamado FacesServlet, el cual actúa como controlador. La configuración de dicho servlet se debe especificar en el archivo **web.xml**, el cual se muestra a continuación:

```
<web-app>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Lo importante en el ejemplo anterior es el mapeo del servlet. Todas las páginas JSF son procesadas por un servlet especial que forma parte del código de implementación de JSF. El contenedor servlet usa la regla del mapeado servlet para activar el servlet JSF, quien elimina el sufijo **faces** y carga la página **index.jsp.** Esto se hace así para que el framework JSF, a través de su servlet principal tome el control. Si no se hiciera de esta manera, el servidor de aplicaciones mostraría una simple página JSP como tal, y la compilación de dicha página, fuera del marco JSF, provocaría errores.

4.1.2 Accediendo a un Bean desde una página JSF

En el contexto de JavaServer Faces, los Beans no se utilizan para nada relacionado con la interfaz de usuario. Los Beans se utilizan cuando se necesita conectar las clases Java con páginas web o archivos de configuración.

Una vez que un Bean ha sido definido, puede ser accedido a través de etiquetas JSF. Por ejemplo, la siguiente etiqueta lee y actualiza el atributo password del Bean usuario:

```
<h:inputSecret value="#{usuario.password}"/>
```

4.1.3 Navegación entre páginas JSF

Las aplicaciones JavaServer Faces usan las reglas de navegación para controlar la navegación entre páginas. Cada regla de navegación especifica cómo ir de una página a las demás dentro de la aplicación. En la arquitectura MVC, la navegación de la página es una de las responsabilidades del controlador. Las reglas de navegación de las aplicaciones JSF están contenidas en el archivo **faces-config.xml** bajo el directorio **WEB-INF**.

Para empezar, existen dos tipos diferenciados de navegación: navegación estática y dinámica.

Navegación estática

Consideremos el caso en el que un usuario llena un formulario de una página web. El usuario puede escribir en los campos del texto, puede hacer clic sobre enlaces, presionar botones o seleccionar elementos de una lista, etc. Todas estas acciones ocurren dentro del navegador del cliente. Cuando, por ejemplo, el usuario presiona un botón, envía los datos del formulario y éstos son gestionados por el servidor. Al mismo tiempo, el servidor JSF analiza la entrada del usuario y debe decidir a qué página ir para dar la respuesta. En una aplicación web simple, la navegación es estática. Es decir, presionar sobre un botón suele redirigir al navegador a una misma página para dar la respuesta. En este caso, simplemente, a cada botón se le da un valor para su atributo de acción (**action**), por ejemplo:

```
<h:commandButton label="Aceptar" action="login"/>
```

Esta acción desencadenante, debe concordar con la etiqueta **outcome** del archivo **faces-config.xml**, dentro de sus reglas de navegación como veremos a continuación. En esta simple regla de navegación, se indica que tras la acción **login**, se navegará a la página **hola.jsp**, si esta acción ocurre dentro de la página **index.jsp**. Se debe tener cuidado de no olvidar el slash (/), en la lineas **from-view-id** y **to-view-id.**

Se puede mezclar reglas de navegación con el mismo **from-view-id**, por ejemplo:

```
<navigation-rule>
     <from-view-id>/index.jsp</from-view-id>
     <navigation-case>
           <from-outcome>login</from-outcome>
           <to-view-id>/hola.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
           <from-outcome>signup</from-outcome>
           <to-view-id>/adios.jsp</to-view-id>
      </navigation-case>
</navigation-rule>
```

Navegación dinámica

En la mayoría de aplicaciones web, la navegación no es estática. El flujo de la página no depende de qué botón se presiona, sino que también depende de los datos que el cliente introduce en un formulario. Por ejemplo, una página de entrada al sistema puede tener dos resultados: el éxito o el fracaso.

El resultado depende de una computación, sea cual sea el nombre y la contraseña es legítima. Para implementar una navegación dinámica, el botón de aceptar debe tener un método referencia, por ejemplo:

```
<h:commandButton label="Aceptar"
action="#{loginControlador.verificarUsuario}"/>
```

En este caso, **loginControlador**, referencia un Bean, y éste debe tener un método denominado **verificarUsuario**. Un método de referencia, en un atributo de acción, no tiene parámetros de entrada y devuelve una cadena de caracteres, que será usada para activar una regla de navegación, por ejemplo, el método **verificarUsuario** debería parecerse a algo así:

```
String verificarUsuario() {
if (...)
return "exito";
else
return "fracaso";
}
```

El método devuelve un String, “éxito” o “fracaso” según sea la condición if. El manejador de navegación usa el String devuelto para buscar una regla de navegación que haga juego. De manera que en las reglas de navegación, podría encontrarse algo así:

```
<navigation-case>
        <from-outcome>exito</from-outcome>
        <to-view-id>/exito.jsp</to-view-id>
</navigation-case>
<navigation-case>
         <from-outcome>fracaso</from-outcome>
         <to-view-id>/fracaso.jsp</to-view-id>
</navigation-case>
```

4.1.4 Etiquetas básicas

Los componentes UI JavaServer Faces son elementos configurables y reutilizables que componen la interface de usuario de las aplicaciones JavaServer Faces. Un componente puede ser simple, como un botón, o compuesto, como una tabla, que puede estar compuesta por varios componentes. A continuación se detallan los más básicos e importantes:

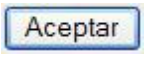
Código del componente	Descripción
`<h:form ... </h:form>`	Representa un formulario
`<h:commandButton value="Aceptar" action="siguiente"/>`	Un botón con una acción asociada 
`<h:inputText value="#{formulario.nombre}" />`	Un campo de texto 
`<h:inputSecret value="#{formulario.password}" />`	Un campo tipo password 
`<h:outputText value="Hola Mundo!"/>`	Muestra una línea de texto, en este caso: "Hola Mundo!"

Tabla 4.1: Componentes básicos de UI JSF

4.2 PrimeFaces

PrimeFaces es una librería open Source para JavaServer Faces cuyo objetivo principal es ofrecer un conjunto de componentes ricos visualmente y funcionalmente para facilitar la

creación de aplicaciones Web. PrimeFaces se divide en tres módulos: el primero es el set de componentes para las interfaces de usuario (cabe destacar que existen aproximadamente más de 100). El segundo módulo llamado Optimus utiliza Guice para poder crear Managed Beans utilizando anotaciones, simplificando así la navegación entre páginas e integrar PrimeFaces con JPA, transacciones y más. Y un tercer módulo llamado FacesTrace que permite monitorear aplicaciones JSF. La ventaja es que estos módulos son totalmente independientes lo que permite crear cualquier tipo de combinación de frameworks.

Otra de las ventajas de PrimeFaces es que usa como herramienta principal Ajax para responder a eventos que se encuentran del lado de la interface de usuario, sin embargo las últimas versiones, también responden a eventos disparados por el servidor. Esto quiere decir que PrimeFaces no solo actúa del lado del servidor sino también del lado del cliente. Para los efectos visuales, PrimeFaces incorpora JQuery nativo en sus componentes, JQuery es un framework visual para componentes web basado en JavaScript y CSS, pero veremos un poco más sobre JQuery en el punto 4.3 de este capítulo. Su versión más estable es la 3.4.

4.2.1 Configurando PrimeFaces

Antes de cualquier configuración, necesitamos bajar el .jar de la página oficial de PrimeFaces (primefaces.org/downloads) y ponerlo en el classpath de nuestro proyecto. Así como se configuró el servlet de JSF dentro del archivo **web.xml**, será necesario hacerlo también para el servlet de PrimeFaces de la siguiente manera:

```
<servlet>
  <servlet-name>Resource Servlet</servlet-name>
  <servlet-class>org.primefaces.resource.ResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>Resource Servlet</servlet-name>
   <url-pattern>/primefaces_resource/*</url-pattern>
</servlet-mapping>

<welcome-file-list>
   <welcome-file>index.xhtml</welcome-file>
</welcome-file-list>
```

Ahora solo debemos importar el espacio de nombres de PrimeFaces a nuestra página JSF y podremos comenzar a gozar de sus componentes. A continuación se presenta un ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<f:view xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:p="http://primefaces.org/ui">
<html>
   <h:head>
      <title>Prueba con Primefaces</title>
   </h:head>
   <h:body>
      <h:form>
         <p:panel header="Prueba con Primefaces">
           <h:panelGrid columns="2">
             <h:outputLabel value="Fecha" />
             <p:calendar pattern="dd/MM/yyyy" />
           </h:panelGrid>
         </p:panel>
      </h:form>
   </h:body>
</html>
</f:view>
```

Está página nos producirá el siguiente resultado para el usuario:

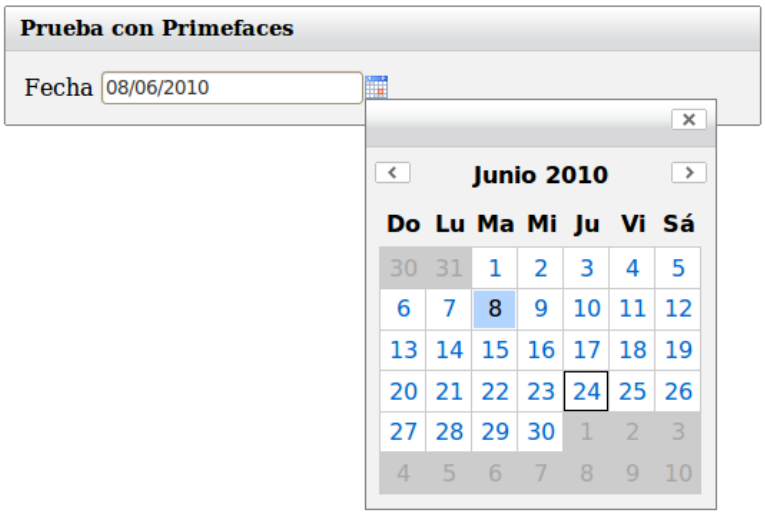


Figura 4.3: Ejemplo del componente Calendar de PrimeFaces

Autor: Diego Hinojosa

Del ejemplo anterior podemos rescatar los siguientes puntos:

- En la línea 7, se aprecia la importación de la librería PrimeFaces, de la cual, usando el prefijo **p**, podemos declarar componentes de dicha librería, tales como botones, calendarios, áreas de texto, gráficos de barras, widgets, etc.

- Se puede ver la riqueza visual y el acabado elegante que tiene el componente Calendar utilizado en este caso.

4.2.2 Etiquetas básicas

Tal como se vio en el punto 4.1.4 con JavaServer Faces, PrimeFaces presenta elementos configurables y reutilizables que componen la interface de usuario. Actualmente posee más de 100 componentes de los cuales solo vamos a revisar los más relevantes:

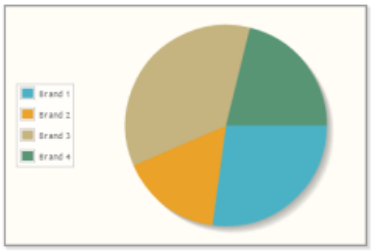
Código del componente	Descripción
`<p:commandButton value="Submit" update="display"/>`	Un botón con una acción asociada 
`<p:inputText id="name" value="#{pprBean.firstname}" />`	Un campo de texto 
`<p:password id="feedback" value="#{passwordBean.password}" feedback="true"/>`	Un campo tipo password con feedback 
`<p:keyboard value="#{keyboardBean.value1}"/>`	Teclado desplegable 
`<p:pieChartid="sample" value="#{chartBean.pieModel}" title="Sample Pie Chart" />`	Gráfico tipo pastel 

Tabla 4.2: Componentes básicos de PrimeFaces

Se pueden revisar más elementos de PrimeFaces en su página oficial (primefaces.org/showcase) donde inclusive nos ofrecen la implementación de los Beans para cada componente.

Lo importante a destacar es que con una o dos líneas de código (literalmente), tenemos un componente completamente desarrollado, que por ejemplo con Java Swing, tal como un gráfico tipo pastel, un marco de imágenes, un teclado desplegable, sería muy difícil de programar, y así un sinnúmero de componentes más, por lo que el uso de PrimeFaces, en mi experiencia, se hace muy necesario si se desea darle esa mezcla perfecta entre elegancia y funcionalidad a la interface de usuario.

4.3 jQuery

jQuery es un framework para desarrollo de interfaces de usuario basado en JavaScript, el cual permite manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web. jQuery es software libre y de código abierto. Ofrece una serie de funcionalidades basadas en JavaScript que de otra manera requerirían de mucho más código, es decir, con las funciones propias de esta biblioteca se logran grandes resultados en menos tiempo y espacio.

jQuery UI es el módulo de componentes para front end el cual posee una herramienta llamada ThemeRoller, la cual es de nuestro interés para integrarlo con PrimeFaces

ThemeRoller

Es una herramienta que se encuentra en la página oficial de jQuery UI (jqueryui.com/themeroller/) que permite ajustar y definir colores, tipografías, etc. de los componentes para interfaz o widgets que ofrece esta biblioteca.

Posee un panel que organiza sus funcionalidades en tres pestañas. La primera Roll Your Own que permite ajustar cada aspecto como: tipografías, esquinas redondeadas y sus radios, colores, etc. y también el comportamiento. La segunda es una galería con estilos ya definidos, para escoger uno entre varios disponibles y la última, es de ayuda.

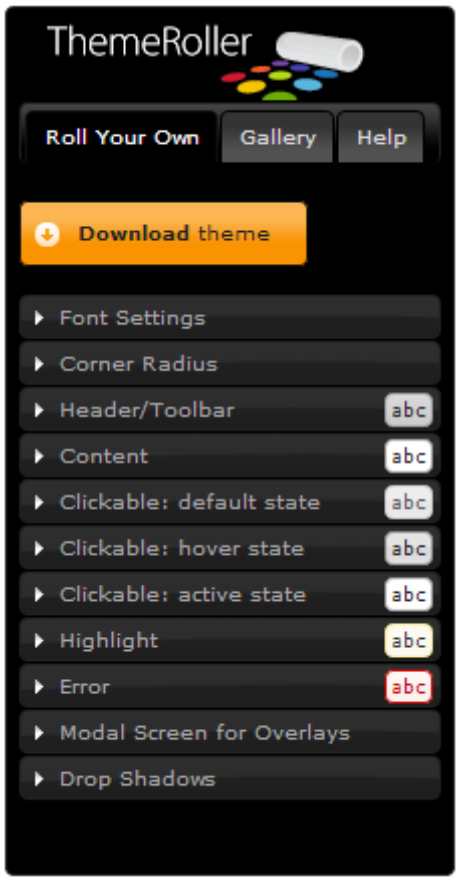


Figura 4.4: Panel de ThemeRoller

Fuente: jqueryui.com/themeroller/

PrimeFaces y ThemeRoller

Como se vio en el punto 4.2, PrimeFaces incorpora jQuery nativo en sus componentes UI. Otra característica interesante es que PrimeFaces viene integrado con ThemeRollerCSS Framework, por lo que se puede escoger entre 37 temas prediseñados y descargarlos desde la página oficial (primefaces.org/themes.html) o crearlos directamente online desde la herramienta ThemeRoller.

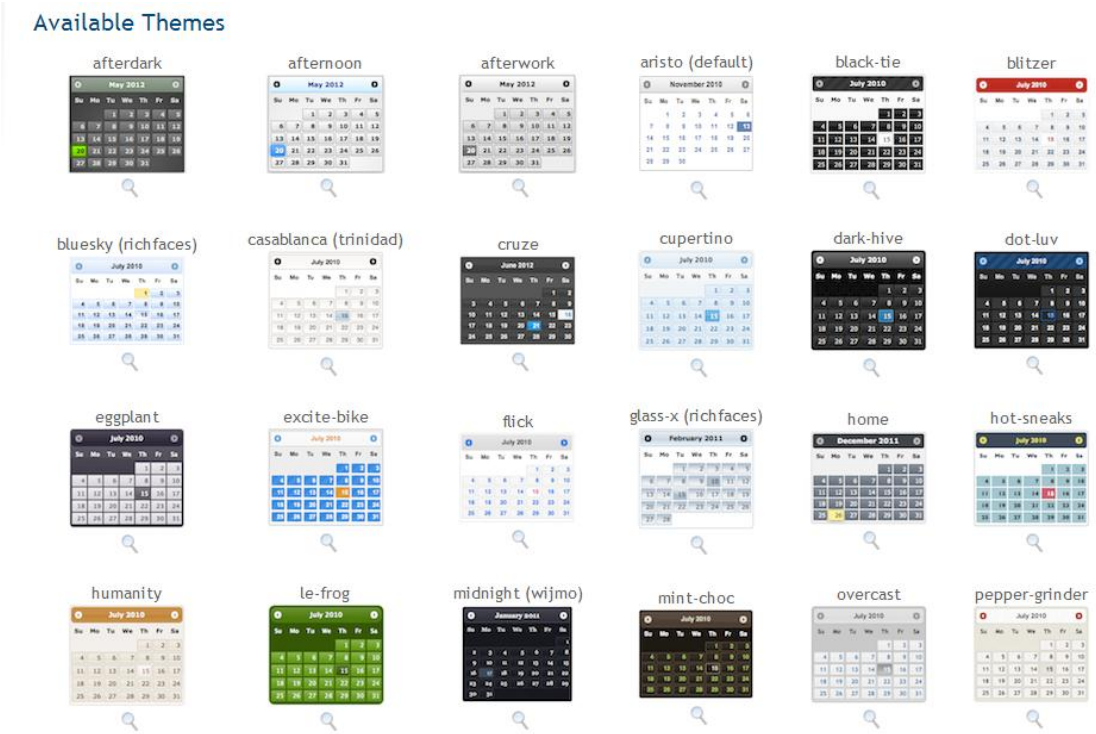


Figura 4.5: Temas disponibles para PrimeFaces

Fuente: primefaces.org/themes.html

Para añadir un tema a un proyecto con PrimeFaces, debemos descargar el archivo .jar el cual añadiremos a nuestras librerías para finalmente poner su referencia en el archivo **web.xml** como se muestra a continuación:

```
<context-param>
   <param-name>primefaces.THEME</param-name>
   <param-value>afterdark</param-value>
</context-param>
```

Revisemos las etiquetas que aparecen en el ejemplo:

- **<context-param>:** Delimita la definición de un parámetro de contexto.
- **<param-name>:** Se especifica el nombre del parámetro, en este caso el nombre del parámetro para añadir un tema a PrimeFaces es **primefaces.THEME.**
- **<param-value>:** Se especifica el valor que toma el parámetro especificado, en este caso el valor del parámetro es el nombre del tema elegido (**afterdark**).

4.4 Propuesta de la capa de presentación

Reuniendo todos los conceptos vistos en este capítulo, el objetivo es diseñar una arquitectura de presentación la cual muestre al usuario una interface amigable, elegante y a la vez funcional sin consumir recursos innecesarios tanto del lado del cliente como del servidor.

Dado este enfoque, esta propuesta incluye el framework JavaServer Faces del lado del servidor. Como se vio en el punto 4.1.1, JSF maneja Beans en la capa del modelo, los cuales conectaremos con la capa de negocio nuevamente con la ayuda de Spring. Se utilizarán los componentes UI tanto de JSF como de PrimeFaces dado que su trabajo en equipo genera mejores resultados. En el lado del cliente nos encontramos con jQuery el cual está integrado dentro de los componentes de PrimeFaces, y AJAX nativo como escucha de eventos tanto del lado de cliente como del servidor.

A continuación se detalla de forma gráfica el diseño para esta capa:

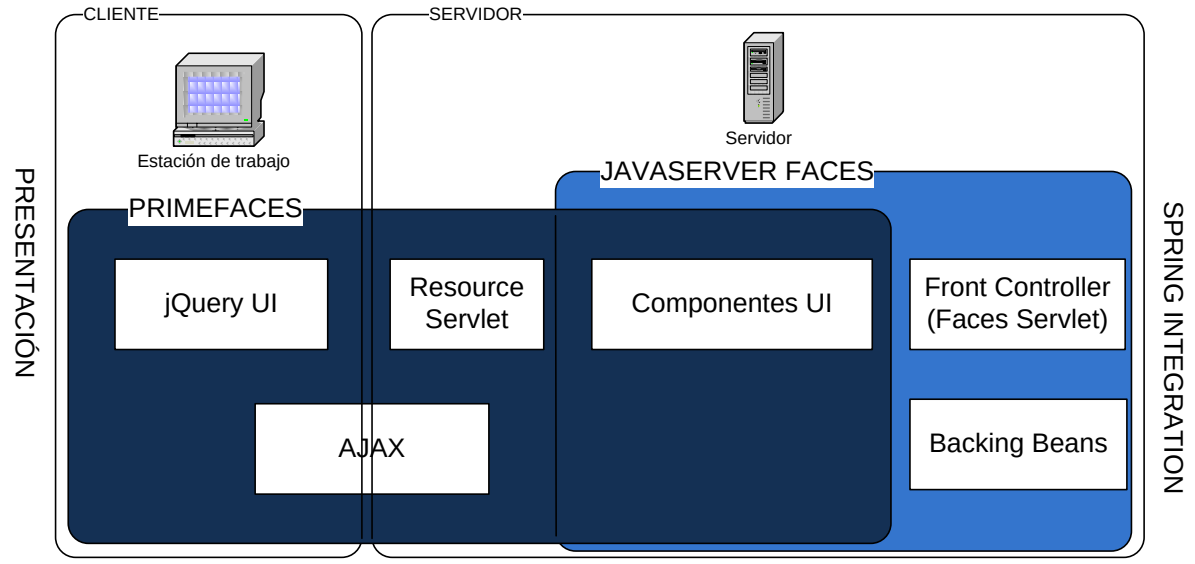


Figura 4.6: Propuesta de capa de presentación

Autor: Diego Hinojosa

Ahora veamos como implementar los Backing Beans con la ayuda de Spring.

Implementando los Managed (Backing) Beans con Spring

A continuación vamos a ver un ejemplo de una implementación sencilla de un managed-bean para lo cual vamos a hacer referencia a la clase de servicio ContactServiceImpl del punto 3.5:

```
@Service
public class ContactServiceImpl implements ContactService {

    @Autowired
    private ContactDAO contactDAO;

    @Transactional
    public void addContact(Contact contact) {
        contactDAO.addContact(contact);
    }
     @Transactional
    public List<Contact> listContact() {
         return contactDAO.listContact();
    }

    @Transactional
    public void removeContact(Integer id) {
        contactDAO.removeContact(id);
    }
}
```

La implementación del managed-bean se presenta a continuación, el cual podrá usarse en un formulario de ingreso de datos para generar un nuevo registro. Para seguir un estándar de desarrollo el nombre estará conformado por la acción o la vista que controla seguido de la palabra Bean (ViewBean):

```
@Controller
@Scope("session")
public class NewContactBean {

    @Autowired
    private ContactService contactService;

    private String firstname;
    private String lastname;
    private String number;
    private String address;
    private Contact contact

   public void setFirstname(String firstname) {
       this.firstname=firstname;
   }

   public String getFirstname() {
       return firstname;
   }

   public void setLastname(String lastname) {
       this.lastname=lastname;
   }

   public String getLastname() {
       return lastname;
   }
```

```
   public void setNumber(String number) {
       this.number=number;
   }

   public String getNumber() {
       return number;
   }

   public void setAddress(String address) {
       this.address=address;
   }

    public String getAddress() {
       return address;
   }

   public String storeContact() {
      contact.setFirstname(firstname);
      contact.setLastname(lastname);
      contact.setNumber(number);
      contact.setAddress(address);
      contactService.addContact(contact);
      return "index.xhtml";
    }
}
```

Hagamos hincapié en las nuevas anotaciones que aparecen en el ejemplo anterior:

- En la línea 1 nos encontramos con **@Controller**. Esta es una anotación de Spring, similar a @Repository o @Service que ya habíamos visto antes. Estamos indicando que esta es una clase relacionada con la capa de control.
- En la línea 2 nos encontramos con **@Scope("session")**. Esta es una anotación de Spring. Con ella estamos sobrescribiendo el comportamiento por defecto de Spring, que es hacer Singletons, y le estamos diciendo que nos cree una instancia diferente de esta clase por cada sesión Http. Es decir, cada usuario tendrá su propio managed-bean.

Para integrar JSF con Spring y hacer posible la inyección de nuestras clases de servicio de la capa de negocio, vamos a delegar el manejo de los managed-bean a Spring, para lo cual debemos modificar el archivo **faces-config.xml** de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer
Faces Config 1.1//EN"
"http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
    <application>
        <el-resolver>
           org.springframework.web.jsf.el.SpringBeanFacesELResolver
```

```
        </el-resolver>
    </application>
</faces-config>
```

Lo importante a destacar es que no ponemos ninguna definición de los Beans en este archivo a menos que queramos generar un Bean manejado por el propio JSF. La definición de <el-resolver> tiene soporte para las versiones de JSF 1.2 y superiores.

Ahora echaremos mano nuevamente de Spring y su contenedor IoC para registrar los managed-beans como Spring Beans añadiendo la siguiente línea al archivo **contextApplication.xml**:

```
<context:component-scan base-package="beans"/>
```

En este punto prácticamente hemos terminado de utilizar Spring, pero para ponerlo en marcha se deben añadir las siguientes líneas al archivo **web.xml:**

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
   <listener-class>
   org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>

<listener>
   <listener-class>
  org.springframework.web.context.request.RequestContextListener
   </listener-class>
</listener>
```

Lo importante a destacar del la configuración anterior es lo siguiente:

- Se crea un parámetro de contexto y a través de **contextConfigLocation** especificamos la ruta del archivo de configuración de Spring (**applicationContext.xml**).
- Se crean dos listener: el primero un **ContextLoaderListener** el cual será el encargado de levantar todo el contexto de Spring, y el segundo un **RequestContextListener** que se encarga de escuchar las solicitudes y enlazarlas al thread actual.

Continuando con el desarrollo, una posible implementación de una página **newContact.xhtml** con PrimeFaces que utilice el managed-bean **NewContactBean** del ejemplo anterior sería la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<f:view xmlns="http://www.w3.org/1999/xhtml"
            xmlns:f="http://java.sun.com/jsf/core"
            xmlns:h="http://java.sun.com/jsf/html"
            xmlns:p="http://primefaces.org/ui">
<html>
 <h:body>
  <h:form id="form">
   <p:panel header="Details">
    <h:panelGrid columns="2">
      <h:outputText value="First Name: " />
      <p:inputText id="firstname" value="#{newContactBean.firstname}"/>
      <h:outputText value="Last Name:" />
      <p:inputText id="lastname" value="#{newContactBean.lastname}">
      <h:outputText value="Number: " />
      <p:inputText id="number" value="#{newContactBean.number}"/>
      <h:outputText value="Address:" />
      <p:inputText id="address" value="#{newContactBean.address}">
  <p:commandButton value="guardar" action="#{newContactBean.storeContact}">
    </h:panelGrid>
   </p:panel>
  </h:form>
 </h:body>
</html>
```

4.5 Resumen de patrones aplicados

En resumen, la propuesta para esta capa está manejada en base al framework JavaServer Faces, del cual se aprovecha su patrón MVC con managed-beans como parte del **modelo** accediendo a los servicios de las capa de negocio a través del conocido DI, páginas JSF como front end conformando la **vista** hacia el usuario final y un FacesServlet como **controlador**. Para poder seguir usando el paradigma de inyección de recursos, necesitamos delegar el manejo de los manged-beans hacia Spring y su contenedor IoC. Complementario a JSF y sus componentes UI se integra la librería PrimeFaces la cual incluye jQuery en sus componentes UI para darle un mejor aspecto visual y funcionalidad al cliente.

4.5.1 Frameworks Utilizados

Spring

En esta propuesta de la capa de presentación utilizamos Spring nuevamente como contenedor de Beans y utilizando su paradigma DI establecemos las referencias desde los managed-beans hacia las clases de servicio de la capa inferior.

JavaServer Faces

JavaServer Faces es un framework que facilita la construcción de aplicaciones web que gestiona las acciones producidas por el usuario en su página HTML y las traduce a eventos que son enviados al servidor con el objetivo de regenerar la página original y reflejar los cambios pertinentes provocados por dichas acciones.

Otras alternativas descartadas:

- **Struts.-** Aunque es el framework que ofreció un camino hacia el MVC en Java para desarrollos Web, es también uno de los que más ha acusado el paso de los años en tiempo de desarrollo y flexibilidad. Hoy existen frameworks que comparten los mismos principios pero con más potencia, elegancia y flexibilidad.
- **Spring MVC.-** Una de las alternativas a Struts que ha incorporado una lógica de diseño más sencilla y que cuenta con todo el abanico de librerías de Spring. No obstante sigue también la misma filosofía y no ofrece nuevas mejoras que sean comparables a JSF.
- **Tapestry.-** Reconocido como uno de los frameworks más potentes para la web, es también uno de los más complejos y de los que tienen una curva de aprendizaje más pronunciada. Ofrece un sistema de componentes al igual que JSF pero no dispone de la comunidad, la documentación, el soporte y por supuesto la estandarización de JSF.

PrimeFaces

PrimeFaces es una librería de componentes para JSF. Estos componentes aportan, frente a los componentes estándar de JSF, una abstracción para el uso de la tecnología AJAX ya soportada en JSF 2. Es decir, el desarrollador puede centrarse en la funcionalidad ofrecida sin tener que preocuparse del JavaScript que se ejecutará en el cliente o de que partes de la pantalla serán necesarias refrescar en respuesta de un evento en la interfaz de usuario.

No siendo PrimeFaces parte del estándar JEE, ahora es la única librería de componentes visuales que podemos decir que soporta de manera estable la versión 2 de JSF.

Otras alternativas descartadas:

- **RichFaces.-** Es otra implementación de JSF. Posee cerca de 39 componentes y no ha crecido mucho en los últimos años. Tiene soporte Ajax, pero no se compara a la variedad de componentes de PrimeFaces y tampoco tiene la integración con jQuery para mejorar sus elementos visualmente, además el modelo que sigue obliga al desarrollador a tener más control sobre lo que quiere refrescar, también es menos flexible al estar claramente orientado su uso junto con Seam.
- **IceFaces.-** Es una muy buena implementación de JavaServer Faces que con alrededor de 70 componentes ha sido el estándar de facto y forma parte de la solución RIA de Sun (Oracle) para JSF 1.2, pero a su vez aún no proporciona un soporte estable para JSF 2.0.

CAPITULO V

INTEGRACION DE LAS CAPAS PROPUESTAS

A lo largo de los capítulos 2, 3 y 4 se han visto las propuestas para las distintas capas que conforman la arquitectura propuesta por este trabajo. A pesar de que en cada capa se vio como realizar su integración con la capa de nivel inferior, es necesario reunir todos los conceptos vistos y demostrar su funcionamiento en conjunto dentro de un ámbito real en un caso de uso.

En el presente capítulo se definirán los componentes necesarios para construir una aplicación que permita insertar registros de un contacto en una base de datos, así como la definición e implementación de las distintas capas que forman parte de esta arquitectura.

Como ya se ha mencionado, la integración de las capas está a cargo de Spring a través de su contenedor IoC de Beans y su paradigma de Inyección de Dependencias. Cabe destacar que Spring es un framework que le puede aportar aún más funcionalidades a esta arquitectura además de su comportamiento como contenedor, por lo que las posibilidades sobre esta arquitectura quedan abiertas para la integración de nuevos módulos e inclusive seguridades de autenticación y autorización tales como las que provee Spring Security.

A continuación se presenta el resultado final de la arquitectura estructurada:

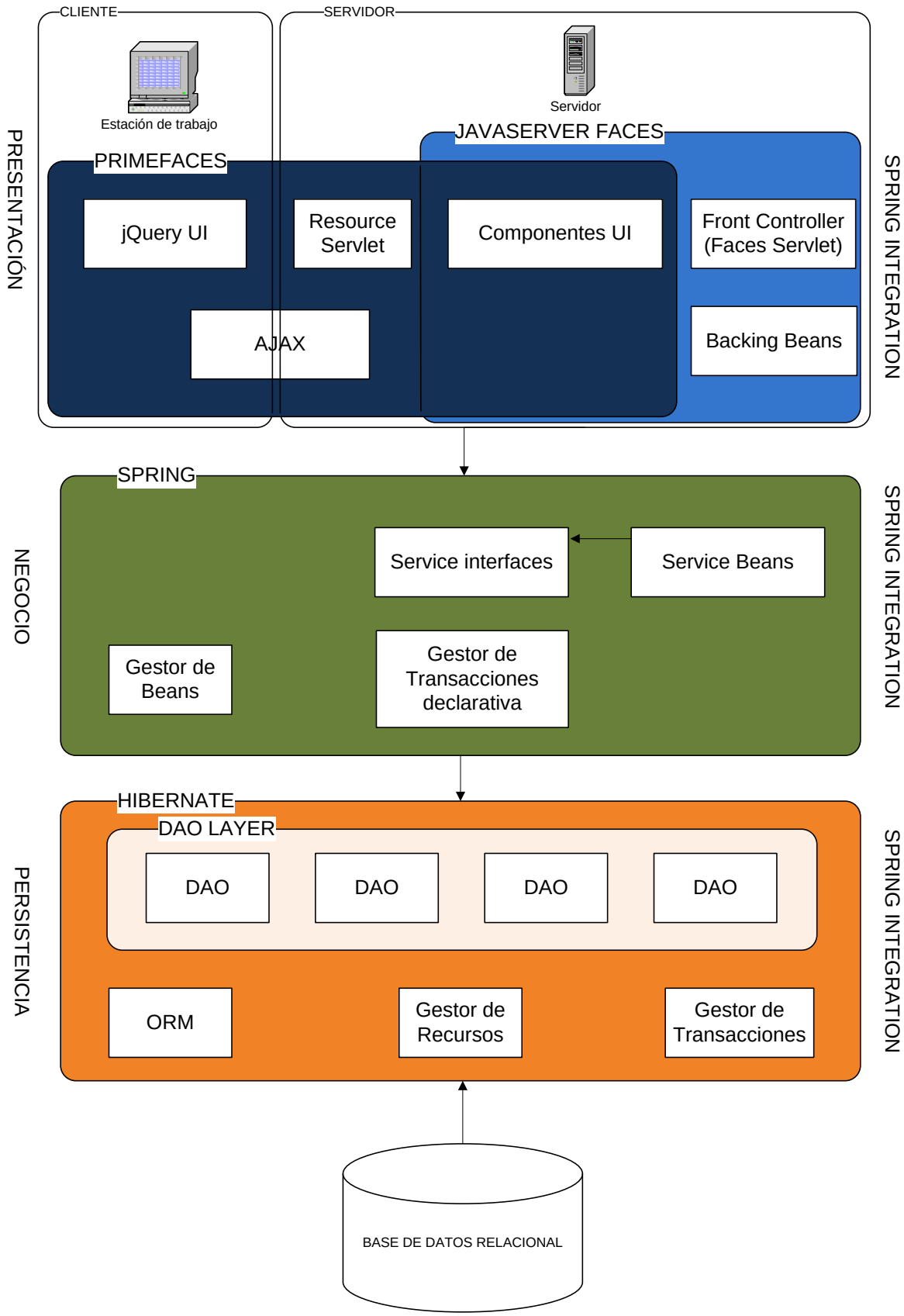


Figura 5.1: Propuesta de la arquitectura completa

Autor: Diego Hinojosa

Definamos las herramientas a utilizar para estructurar un escenario para su utilización:

- Base de datos: MySQL 1.2.3 Beta (Descargable desde mysql.com)
- Entorno de desarrollo: NetBeans IDE 6.9.1 (Descargable desde netbeans.org)
- Plataforma Empresarial: JEE 6 (Integrado en NetBeans)
- Mapeador Objeto-Relacional: Hibernate (Integrado en NetBeans)
- Herramienta de Integración General: Framework Spring 3.0.2 (Integrado en NetBeans)
- Herramienta de Presentación: JavaServer Faces 2.0 (Integrado en NetBeans)
- Librería Auxiliar de JSF: PrimeFaces 3.0.1 (Descargable desde primefaces.org)
- Tema jQuery para PrimeFaces: Start 1.0.1 (Descargable desde primefaces.org)
- Servidor de Aplicaciones: GlassFish Server 3.1 (Integrado en NetBeans)

Ahora veamos la estructura general de la aplicación:

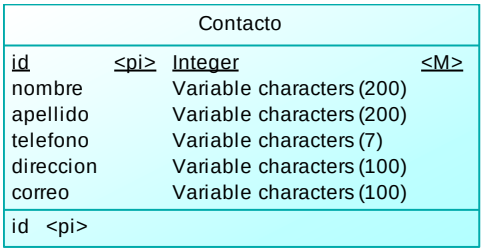


Figura 5.2: Diagrama conceptual de la tabla Contacto

Autor: Diego Hinojosa

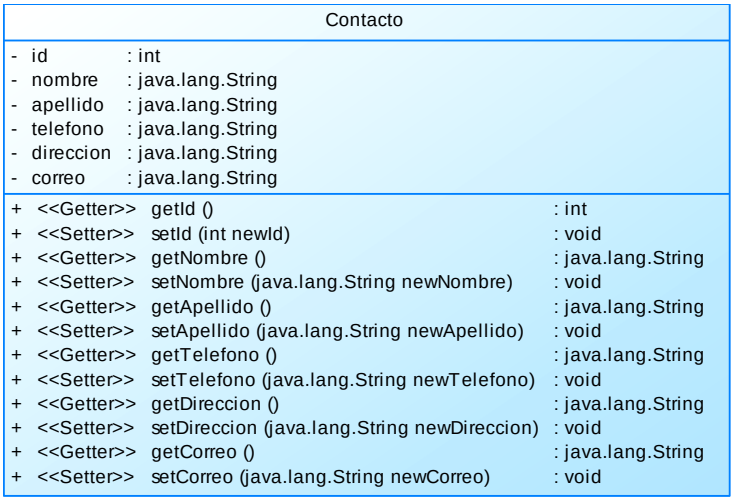


Figura 5.3: Diagrama de clase entidad Contacto

Autor: Diego Hinojosa

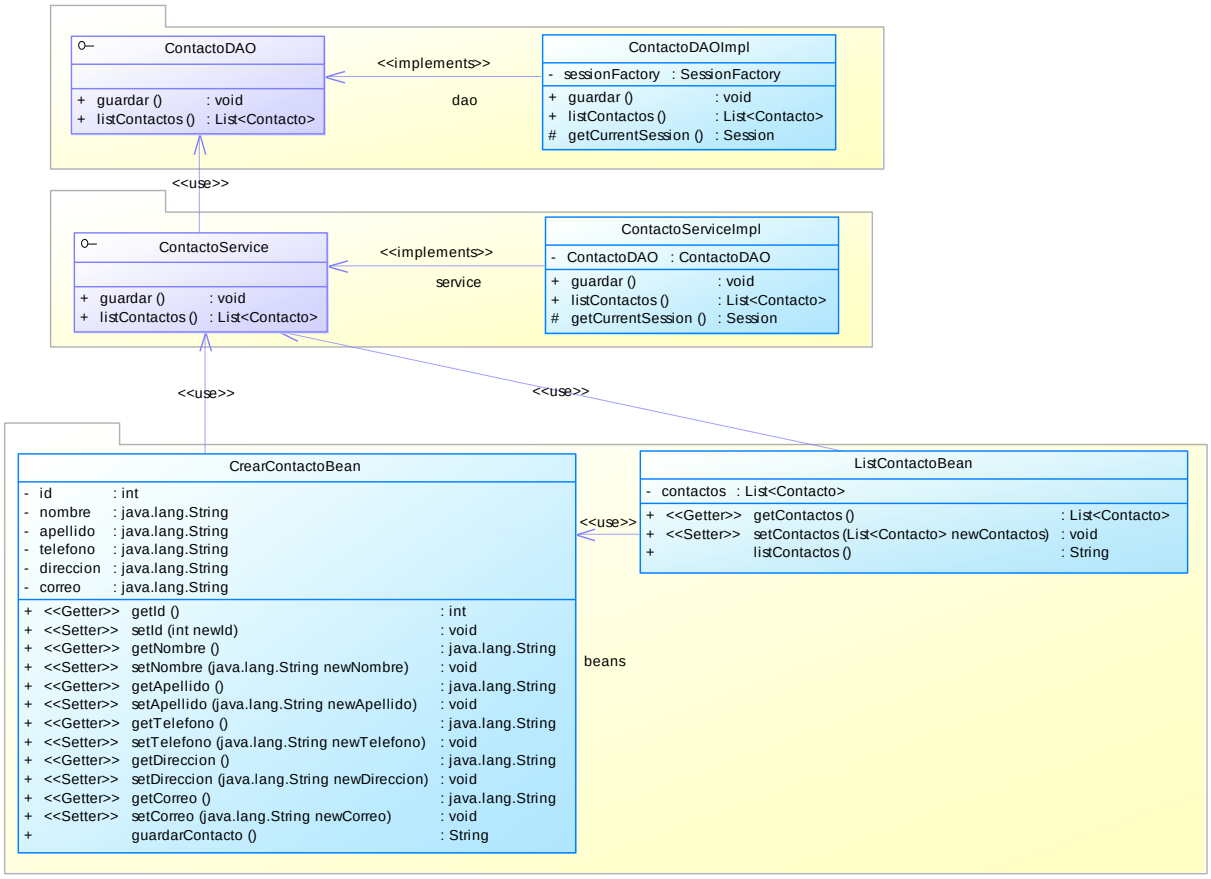


Figura 5.4: Diagrama de clases y paquetes general

Autor: Diego Hinojosa

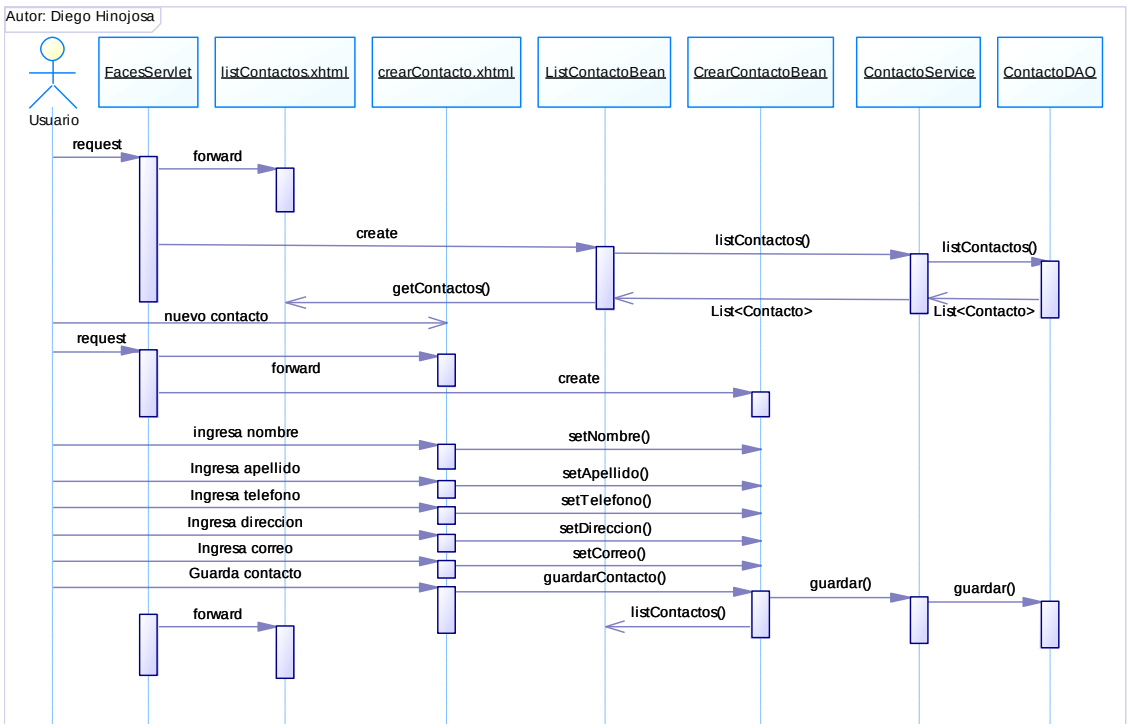


Figura 5.5: Diagrama de secuencia guardar contacto

A continuación se detalla la implementación de las clases por cada paquete (capa):

Paquete: Entities **- Archivo:** Contacto.java

```
package entities;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "Contacto")
public class Contacto implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Integer id;
    @Column(name = "nombre",length=200)
    private String nombre;
    @Column(name = "apellido",length=200)
    private String apellido;
    @Column(name = "telefono",length=7)
    private String telefono;
    @Column(name = "direccion",length=100)
    private String direccion;
    @Column(name = "correo",length=100)
    private String correo;


    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getApellido() {
        return apellido;
    }

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    public String getCorreo() {
        return correo;
    }

    public void setCorreo(String correo) {
        this.correo = correo;
    }
```

```
    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }
}
```

Paquete: dao - **Archivo:** ContactoDAO.java

```
package dao;

import entities.Contacto;
import java.util.List;

public interface ContactoDAO {
    public void guardar(Contacto entity);
    public List<Contacto> listContactos();
}
```

Paquete: dao - **Archivo:** ContactoDAOImpl.java

```
package dao;

import entities.Contacto;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class ContactoDAOImpl implements ContactoDAO {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public void guardar(Contacto entity) {
        Transaction trans = this.getCurrentSession().beginTransaction();
```

```
        this.getCurrentSession().save(entity);
        trans.commit();
    }

    @Override
    public List<Contacto> listContactos(){
        Transaction trans = this.getCurrentSession().beginTransaction();
        List<Contacto> customers=this.getCurrentSession().createQuery("from
Contacto").list();
        trans.commit();
        return customers;
    }

    protected final Session getCurrentSession() {
        return this.sessionFactory.getCurrentSession();
    }
}
```

Paquete: service - **Archivo:** ContactoService.java

```
package service;

import entities.Contacto;
import java.util.List;

public interface ContactoService {
    public void guardar(Contacto entity);
    public List<Contacto> listCustomers();
}
```

Paquete: service - **Archivo:** ContactoServiceImpl.java

```
package service;

import dao.ContactoDAO;
import entities.Contacto;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class ContactoServiceImpl implements ContactoService {

    @Autowired
    private ContactoDAO contactoDAO;

    @Transactional
    public void guardar(Contacto entity) {
        contactoDAO.guardar(entity);
    }

    @Transactional
    public List<Contacto> listContactos() {
        return contactoDAO.listContactos();
    }
}
```

Paquete: beans - **Archivo:** ListContactoBean.java

```
package beans;

import entities.Contacto;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Controller;
import service.ContactoService;

@Controller
@Scope("session")
public class ListContactoBean {
    @Autowired
    private ContactoService contactoService;

    private List<Contacto> contactos;

    public String listContactos() {
        contactos = contactoService.listContactos();
        return "/listContactos.xhtml";
    }

    public List<Contacto> getContactos() {
        return contactos =contactoService.listContactos();
    }

    public void setContactos(List<Contacto> contactos) {
        this.contactos = contactos;
    }
}
```

Paquete: beans - **Archivo:** CrearContactoBean.java

```
package beans;

import entities.Contacto;
import javax.annotation.Resource;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Controller;
import service.ContactoService;

@Controller
@Scope("request")
public class CrearContactoBean {

    @Autowired
    private ContactoService contactoService;

    @Resource
    private ListContactoBean list;

    private String nombre;
    private String apellido;
    private String telefono;
```

```
    private String direccion;
    private String correo;

    public CrearContactoBean() {
    }

    public String getApellido() {
        return apellido;
    }

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    public String getCorreo() {
        return correo;
    }

    public void setCorreo(String correo) {
        this.correo = correo;
    }

    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public String guardarContacto() {
        FacesContext facesContext = FacesContext.getCurrentInstance();
     FacesMessage facesMessage = new FacesMessage("Guardado Exitosamente");
        Contacto contacto = new Contacto();
        contacto.setNombre(nombre);
        contacto.setApellido(apellido);
        contacto.setTelefono(telefono);
        contacto.setDireccion(direccion);
        contacto.setCorreo(correo);
        contactoService.guardar(contacto);
        facesContext.addMessage(null, facesMessage);
        return list.listContactos();
    }
}
```

Presentación - Archivo: listContactos.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui">
    <h:head>
        <title>Contactos</title>
    </h:head>
    <h:body>
        <h:form>
            <p:messages/>
            <p:commandButton action="/crearContacto.xhtml" value="Nuevo
Contacto" ajax="false"/>
            <p:dataTable var="contacto"
value="#{listContactoBean.contactos}">
                <p:column headerText="Id">
                    <h:outputText value="#{contacto.id}" />
                </p:column>
                <p:column headerText="Nombre">
                    <h:outputText value="#{contacto.nombre}" />
                </p:column>
                <p:column headerText="Apellido">
                    <h:outputText value="#{contacto.apellido}" />
                </p:column>
                <p:column headerText="Teléfono">
                    <h:outputText value="#{contacto.telefono}" />
                </p:column>
                <p:column headerText="Dirección">
                    <h:outputText value="#{contacto.direccion}" />
                </p:column>
                <p:column headerText="Correo">
                    <h:outputText value="#{contacto.correo}" />
                </p:column>
            </p:dataTable>
        </h:form>
    </h:body>
</html>
```

Presentación - Archivo: crearContacto.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui">
    <h:head>
        <title>Nuevo Contacto</title>
    </h:head>
    <h:body>
        <h:form>
            <p:messages/>
            <p:commandButton action="/listContactos.xhtml" value="Atrás"
ajax="false"/>
            <h:panelGrid columns="2">
                <h:outputText value="Nombre: "/>
```

```
                <h:inputText value="#{crearContactoBean.nombre}"/>
                <h:outputText value="Apellido: "/>
                <h:inputText value="#{crearContactoBean.apellido}"/>
                <h:outputText value="Teléfono: "/>
                <h:inputText value="#{crearContactoBean.telefono}"/>
                <h:outputText value="Dirección: "/>
                <h:inputText value="#{crearContactoBean.direccion}"/>
                <h:outputText value="Correo: "/>
                <h:inputText value="#{crearContactoBean.correo}"/>
                <p:commandButton
action="#{crearContactoBean.guardarContacto}" value="Guardar"
ajax="false"/>
            </h:panelGrid>
        </h:form>
    </h:body>
</html>
```

A continuación se detallan los archivos de configuración de Hibernate (hibernate.cfg.xml), Spring (applicationContext.xml), JavaServer Faces (faces-config) y web.xml respectivamente:

Configuración - Archivo: hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.current_session_context_class">
      Thread
    </property>
    <property name="hibernate.hbm2ddl.auto">update</property>
   </session-factory>
</hibernate-configuration>
```

Configuración - Archivo: applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:flow="http://www.springframework.org/schema/webflow-config"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:p="http://www.springframework.org/schema/p"
```

```
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
          http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
          http://www.springframework.org/schema/webflow-config
http://www.springframework.org/schema/webflow-config/spring-webflow-config-
2.0.xsd
          http://www.springframework.org/schema/jms
http://www.springframework.org/schema/jms/spring-jms-3.0.xsd
          http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-3.0.xsd
          http://www.springframework.org/schema/lang
http://www.springframework.org/schema/lang/spring-lang-3.0.xsd
          http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi-3.0.xsd
          http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
          http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.0.xsd
">
    <context:component-scan base-package="dao"/>
    <context:component-scan base-package="service"/>
    <context:component-scan base-package="beans"/>
    <context:annotation-config/>

    <bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean" >
        <property name="jndiName" value="ejemplo"/>
        <property name="lookupOnStartup" value="false"/>
        <property name="cache" value="true"/>
        <property name="proxyInterface" value="javax.sql.DataSource"/>
    </bean>

    <bean id="SessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFacto
ryBean">
        <property name="configLocation">
            <value>/WEB-INF/hibernate.cfg.xml</value>
        </property>
        <property name="dataSource">
            <ref local="dataSource"/>
        </property>
        <property name="packagesToScan" value="entities"/>
    </bean>

    <bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="SessionFactory"/>
    </bean>
</beans>
```

Tomando en cuenta que se va a desplegar la aplicación en un entorno empresarial, se hace indispensable recoger la conexión hacia la base de datos desde el propio servidor de

aplicaciones, en este caso GlassFish, lo cual se hace a través de la configuración de un pool de conexión. Para esto se ha cambiado el Bean dataSource de la siguiente manera:

```
<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean" >
        <property name="jndiName" value="ejemplo"/>
        <property name="lookupOnStartup" value="false"/>
        <property name="cache" value="true"/>
        <property name="proxyInterface" value="javax.sql.DataSource"/>
</bean>
```

Veamos la propiedad más relevante de este Bean:

- **jndiName.-** A través del atributo value, se especifica el sistema de nombres y directorios bajo el cual se encuentra el pool de conexión a la base de datos.

En este caso el nombre de JNDI es "ejemplo", el cual debe estar registrado en el servidor GlassFish de la siguiente manera para que Spring haga uso de él:

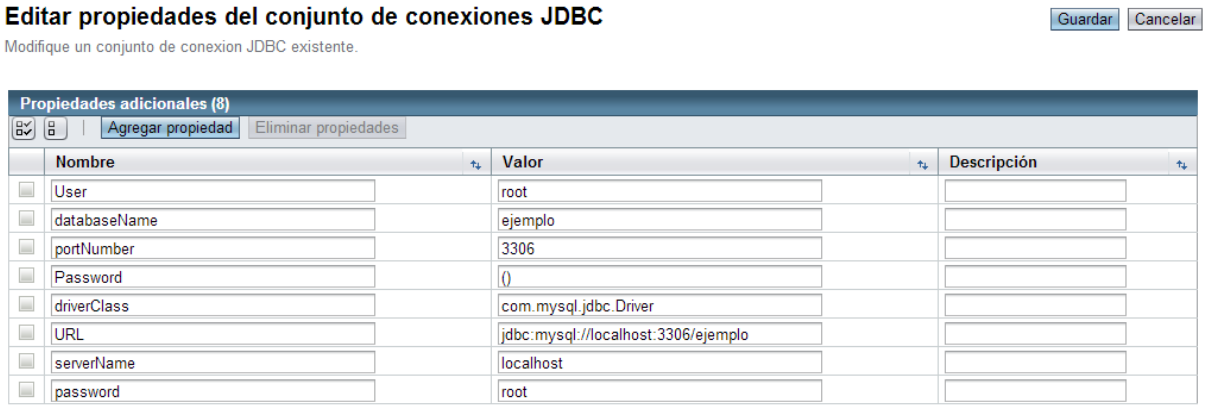


Figura 5.6: Configuración de pool de conexiones GlassFish

Autor: Diego Hinojosa

Como se ve en la figura anterior, en el pool se especifican propiedades como el driverClass, la URL, el usuario, la contraseña, etc., por lo que estas ya no deben especificarse en el Bean dataSource de Spring. Este pool de conexiones se guardó bajo el nombre **mysql_ejemplo_rootPool.**

Luego se asigna un nombre JNDI al conjunto de conexiones de la siguiente manera:

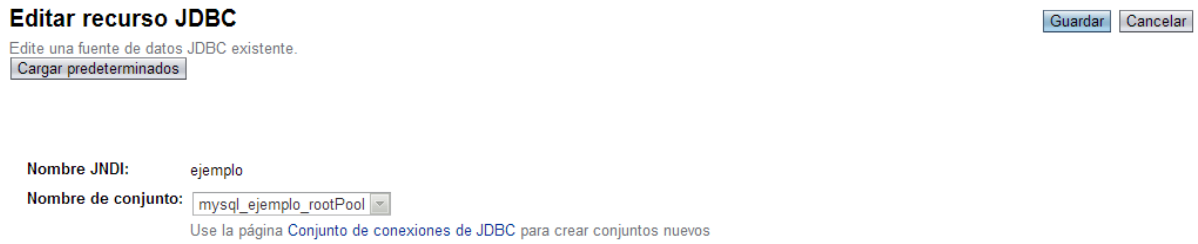


Figura 5.7: Configuración de nombre JNDI con conjunto de conexiones

Autor: Diego Hinojosa

Configuración - Archivo: faces-config.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
   <application>
    <el-resolver>
     org.springframework.web.jsf.el.SpringBeanFacesELResolver
    </el-resolver>
   </application>
</faces-config>
```

Configuración - Archivo: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
     <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/applicationContext.xml
        </param-value>
    </context-param>
     <listener>
        <listener-class>
         org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <listener>
        <listener-class>
         org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
```

```
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
     <servlet>
        <servlet-name>Resource Servlet</servlet-name>
        <servlet-class>
         org.primefaces.resource.ResourceServlet
        </servlet-class>
     </servlet>
    <servlet-mapping>
        <servlet-name>Resource Servlet</servlet-name>
        <url-pattern>/primefaces_resource/*</url-pattern>
    </servlet-mapping>
    <context-param>
        <param-name>primefaces.THEME</param-name>
        <param-value>start</param-value>
    </context-param>
    <welcome-file-list>
        <welcome-file>faces/listContactos.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

La estructura de archivos del proyecto es la siguiente:

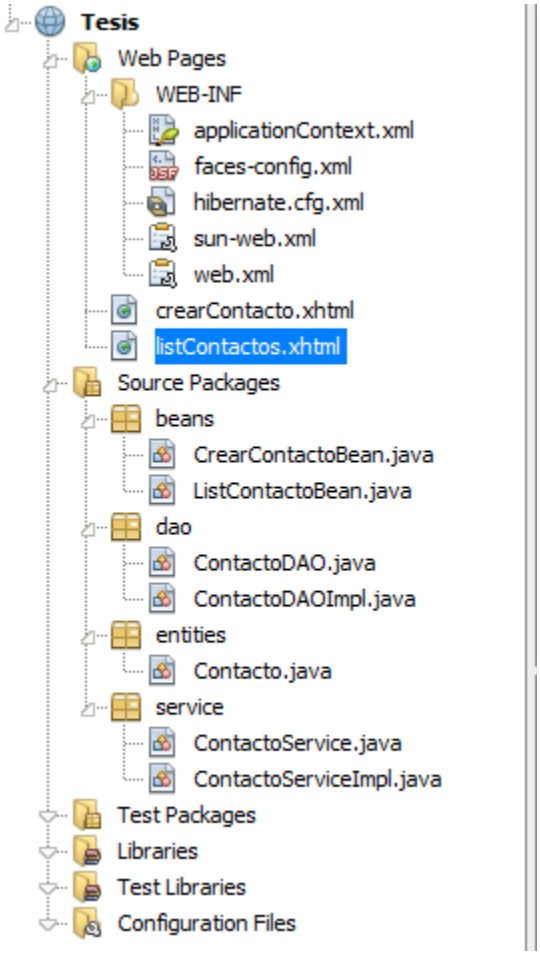


Figura 5.8: Estructura de archivos de la aplicación

Autor: Diego Hinojosa

Una vez realizada la implementación en el servidor GlassFish, podemos ver a toda la arquitectura en acción representada en la aplicación:

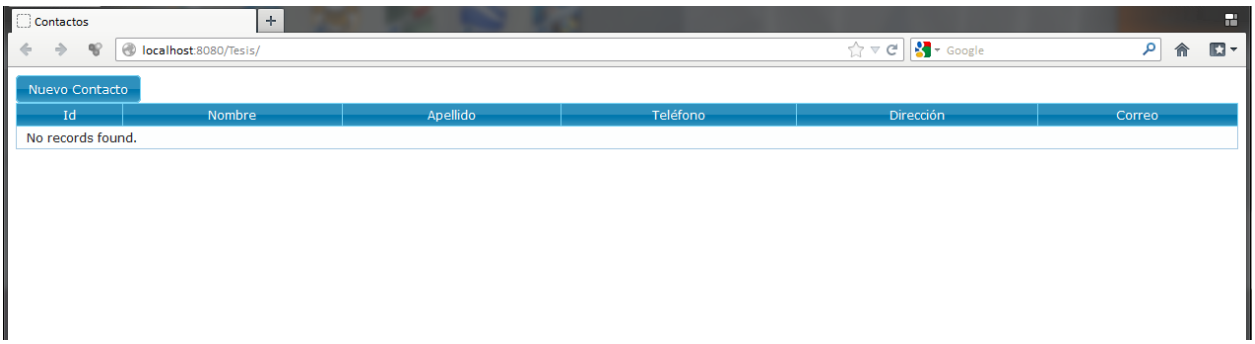


Figura 5.9: Pantalla de inicio sin registros listContactos.xhtml

Autor: Diego Hinojosa

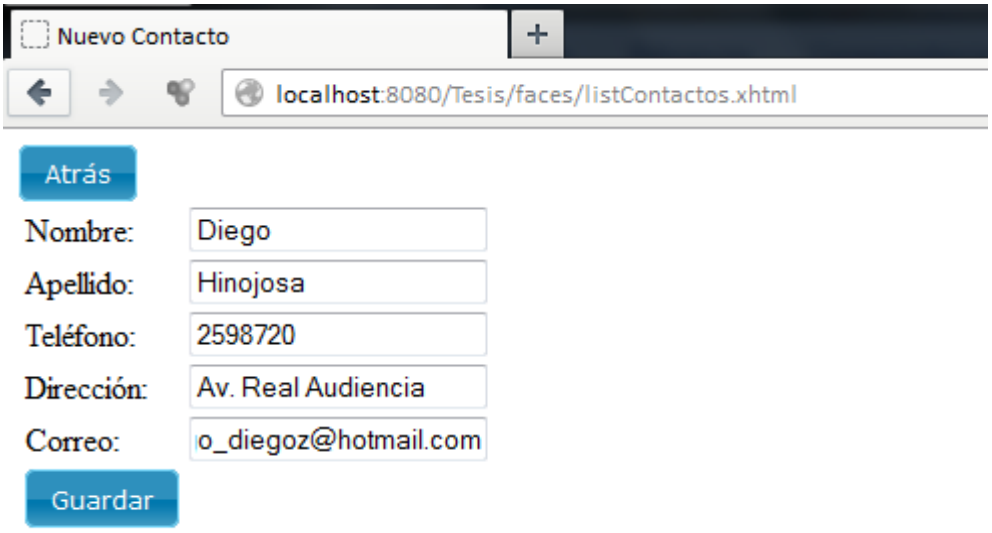


Figura 5.10: Pantalla de ingreso de contacto crearContacto.xhtml

Autor: Diego Hinojosa

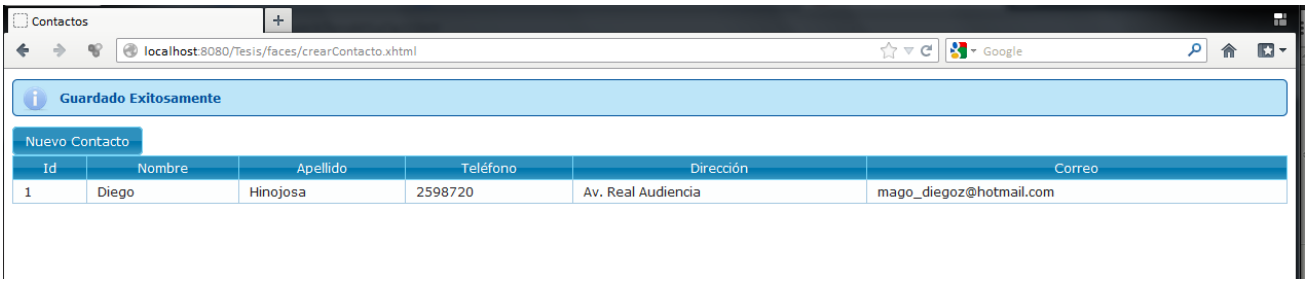


Figura 5.11: Pantalla de inicio después de guardar contacto listContacto.xhtml

Autor: Diego Hinojosa

5.1 Elección del servidor de aplicaciones

Como se vio con anterioridad, el servidor de aplicaciones en el que se desplegó la aplicación fue GlassFish, pero nace la pregunta: ¿por qué?, y es que para la construcción de una aplicación web y al utilizar la versión empresarial de Java (JEE) para construir aplicaciones que respondan a una arquitectura más especializada, no podemos utilizar un simple servidor de aplicaciones, si no que necesitamos un Servidor de Aplicaciones JEE, que soporte los componentes para la creación de aplicaciones web empresariales utilizando los estándares sugeridos dentro de JEE.

Sun Microsystem ofrece GlassFish como el servidor de aplicaciones oficial para soporte JEE, el cual viene en una versión comercial llamada el Sun GlassFish Enterprise Server y en una versión de código abierto llamada simplemente GlassFish, la cual posee dos licencias la CDDL y el GPL.

Así que GlassFish es el servidor de aplicaciones para Java en su versión JEE, que permite el despliegue y soporte de aplicaciones empresariales con componentes web, transaccionales y de persistencia.

Entre las características más notables de GlassFish cabe destacar su velocidad, alta escalabilidad, manejo centralizado de clusters e instancias, bajo consumo de memoria, interoperabilidad con .NET 3 y un excelente panel de administración.

5.2 Resumen de patrones aplicados

Resumamos los patrones utilizados a lo largo de este trabajo:
La capa de persistencia la provee Hibernate, mapeando entidades persistentes a través de anotaciones JPA. Para poder realizar transacciones con la base de datos se utiliza el patrón Data Access Objects (DAO's), estos objetos esconden toda la lógica de Hibernate a través de sentencias HQL y otras funciones proporcionadas de guardado, borrado y actualización. Cabe destacar que se opta por el uso de interfaces e implementaciones para poder intercambiar la implementación de los métodos, valga la redundancia, y garantizar un bajo acoplamiento. El framework Spring se hace presente para manejar el SessionFactory de Hibernate a través de un Bean el cual es inyectado como dependencia a las clases de manejo de datos.

La capa de la lógica de negocio maneja objetos de servicio registrados marcados como Beans en el contenedor de control de inversión (IoC) de Spring, el cual nos garantiza llamar los recursos solo cuando estos sean necesitados. Dichos objetos toman referencia de los objetos de acceso a los datos a través del paradigma de Inyección de Dependencias para desarrollar la lógica del negocio como tal.

Finalmente la capa de presentación se basa en el patrón MVC que provee JavaServer Faces, con managed-beans como parte del modelo pero registrados también en el

contenedor IoC de Spring de manera que puedan acceder a los servicios de la capa de negocio a través de la inyección de dependencias, páginas JSF como front end conformando la vista hacia el usuario final y un FacesServlet como controlador. Como un complemento a los componentes UI de JSF se integra la librería PrimeFaces la cual incluye jQuery en sus componentes UI para darle un mejor aspecto visual y funcionalidad al cliente.

CAPITULO VI

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

- A lo largo de este trabajo se logró estructurar una arquitectura con modularidad en su diseño. Cada una de las partes empleadas (Hibernate para la persistencia, Spring para la integración, y JavaServer Faces para la presentación) es intercambiable de forma fácil y limpia. Por ejemplo, para la vista se emplea JavaServer Faces, pero nada impide emplear también una aplicación de escritorio mediante Swing o SWT sin tener que tocar ni una sola línea de código de las capas restantes. También, nada impediría que se pudiese disponer de una aplicación con una parte de la capa de presentación en JSF y otra parte, para otro tipo de usuarios, en Swing, ambas funcionando a la vez y compartiendo todo el resto del código (lógica de negocio, persistencia, integración, seguridad, etc). De igual forma, si se desean cambiar elementos de la capa de persistencia empleando otro framework para el mapeo que sea diferente de Hibernate o no utilizar ninguno tan sólo serían necesarios cambios en esa capa. De la misma manera se podrían sustituir cualquiera de las otras capas. El diseño se ha hecho reduciendo al mínimo posible las dependencias entre ellas.

- Se pudo determinar que una aplicación empresarial debe tener las siguientes características: Ofrecer una buena escalabilidad tanto horizontal como vertical. Mantenibilidad para poder añadir o modificar los componentes sin modificar el comportamiento del sistema. Garantizar su disponibilidad. Tener la posibilidad de añadir nuevos componentes y capacidades al sistema sin que se vean afectados los demás componentes. El sistema debe ser fácilmente manejable y configurable. Debe tener buenos sistemas de seguridad tanto a nivel de autenticación, como de autorización y transporte. Y ofrecer automáticamente mecanismos que permitan aumentar el rendimiento de manera transparente al usuario.

- Los patrones de diseño ofrecen una solución ya probada y documentada a problemas de desarrollo de software, y aún más cuando se trata de sistemas empresariales, que están sujetos a contextos similares. Es decir, nos podemos valer de estos patrones y

aplicarlos al desarrollo de cualquier sistema que describa una problemática similar.

- La plataforma JEE ofrece una arquitectura multicapa para el desarrollo de aplicaciones del tipo Enterprise. JEE incluye varias especificaciones de API's, tales como JDBC, RMI, mail, JMS, Servicios Web, XML, etc. y define cómo coordinarlos. La plataforma también configura algunas especificaciones únicas para JEE para componentes, estas incluyen Enterprise JavaBeans, servlets, portlets, JavaServer Pages (jsp) y varias tecnologías de servicios web. Esto permite al desarrollador crear una Aplicación Empresarial portable entre plataformas y escalable, a la vez que integrable con tecnologías anteriores. Otros beneficios adicionales son, por ejemplo, que el servidor de aplicaciones puede manejar transacciones, la seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, significando que los desarrolladores pueden concentrarse más en la lógica de negocio de los componentes en lugar de ocuparse en tareas de mantenimiento de bajo nivel.

- Las ventajas al usar una herramienta de mapeo objeto-relacional son las siguientes: Rapidez en el desarrollo mediante la creación del modelo por medio del esquema de la base de datos. Se consigue una correcta abstracción de la base de datos. Facilita el mantenimiento del código debido a la correcta disposición de la capa de datos. Y ofrece un lenguaje propio para realizar las consultas.

- Hibernate es una muy buena herramienta en lo que se refiere a mapeo de clases en una base de datos relacional. Soporta la mayoría de los sistemas de bases de datos SQL. El Hibernate Query Language (HQL), proporciona un puente elegante entre los mundos objeto y relacional. Hibernate ofrece facilidades para recuperación y actualización de datos, control de transacciones, repositorios de conexiones a bases de datos, consultas programáticas y declarativas, y un control de relaciones de entidades declarativas.

- Spring demostró ser un framework potente y flexible al momento de realizar la integración de las capas de la arquitectura propuesta. Brinda la posibilidad de integrarse con otras herramientas e incluso otros frameworks con el fin de obtener beneficios que el desarrollador busca en cada una de ellas. Spring facilita el desarrollo ya que está diseñado con interfaces las cuales pueden utilizarse, promoviendo así la reutilización de código.

- El framework de la capa de presentación depende mucho de la elección y hasta cierto punto del conocimiento del desarrollador sobre la herramienta escogida. JavaServer Faces forma parte del estándar de JEE y fue una elección dentro de esta propuesta arquitectónica por 3 razones: La primera por la facilidad que ofrece para crear páginas complejas con componentes UI, la segunda por la integración sencilla y que hasta cierto punto parece natural con Spring, y finalmente por que posee implementaciones muy interesantes en cuanto a mejoras en el aspecto visual y funcional de sus páginas tales como IceFaces, RichFaces, PrimeFaces, etc.

- PrimeFaces fue una grata revelación en mi experiencia con JSF, ya que provee soporte de AJAX nativo y transparente para el usuario y además utiliza jQuery para los efectos visuales del lado del cliente. ¿Y el resultado?, una página de altísima calidad visual y funcional. PrimeFaces tiene un catálogo de más de 100 componentes UI, algunos de ellos van desde los más simples hasta los más complejos. En conclusión, una experiencia agradable para el usuario final y un manejo limpio y sencillo para el desarrollador.

Recomendaciones

- El entorno de desarrollo a escoger es una decisión propia del programador del sistema basado en los frameworks o herramientas que va a utilizar. NetBeans IDE es un software bastante completo en lo que a la plataforma JEE y aplicaciones web se refiere, pero si se desea entrar en el mundo de Spring en profundidad, vale la pena decidirse por MyEclipse, no solo por su facilidad de configuración e implementación, también porque provee una herramienta basada en scaffolding (construcción de aplicaciones basada en bases de datos), la cual permite desarrollar CRUD de una manera fácil y rápida en base a la plataforma JEE.

- PrimeFaces es una tecnología relativamente nueva y que tiene un largo camino por recorrer. Esta implementación tiene algunos problemas en ciertos componentes UI por lo que es muy recomendable su uso junto con una tecnología un poco más madura como lo es JavaServer Faces. Juntos forman un gran equipo de trabajo por así decirlo, y es ese rol en conjunto el cual se cumple dentro de esta propuesta arquitectónica.

- Se debería pensar en incluir el estudio de las técnicas y patrones de desarrollo que se

han presentado en este trabajo dentro de las materias relativas a la programación. También, se podría considerar añadir una materia en la cual se oriente al estudiante dentro del mundo de las aplicaciones empresariales y todo lo que esto implica (bases de datos relacionales, uso de Frameworks, servicios web, plataformas para desarrollo empresarial, etc), las cuales forman parte del mundo real y competitivo al cual uno se enfrenta cuando ingresa al mercado laboral.

BIBLIOGRAFÍA

- Alur, Deepak. Malks, Dan. Crupi, John. Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall, 2da Edición, 2003.
- Apache Software Foundation: Struts. Internet. http://struts.apache.org/ Acceso: 5 de Octubre de 2012.
- Biem, Adam. Real World Java EE Patterns, press.adam-bien.com, 2009.
- Construir Aplicaciones EJB con JBoss, Lomboz y Eclipse. Internet. http://www.programacion.com/articulo/construir_aplicaciones_ejb_con_jboss-lomboz_y_eclipse_267/3 Acceso: 5 de Octubre de 2012
- Esteban, A. Eloy. El API Struts. Internet http://www.programacion.com/java/tutorial/struts/4/ Acceso: 13 de Noviembre de 2012.
- Freeman, Eric. Freeman, Elisabeth. Head First Design Patterns, O'Reilly, 2004.
- Gamma, Erich. Helm, Richard. Johnson, Ralph. Vlissides, John. Patrones de Diseño, Madrid, Addison-Wesley, 2003.
- Geary, David. Horstmann, Cay. Core JavaServer Faces (Core Series), Prentice Hall PTR, 2007.
- Goncalves, Antonio. Beginning Java™ EE 6 Platform with GlassFish™ 3, New York, 2da Edición, 2010.
- Hibernate Architecture. Internet. http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/architecture.html Acceso: 15 de Octubre de 2012.
- Hibernate Developer Guide. Internet. http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html_single/ Acceso: 20 de Octubre de 2012
- Hibernate Getting Started Guide. Internet. http://docs.jboss.org/hibernate/orm/4.1/quickstart/en-US/html_single/ Acceso: 20 de Octubre de 2012
- Introduction to Spring Framework. Internet. http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/overview.html Acceso: 4 de Noviembre de 2012.

- Java Persistence API. Internet. http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html Acceso: 17 de Octubre de 2012.
- Java/J2ee Application Development Services- Company Expertise. Internet. http://www.globaxon.com/j2ee_technology_expertise.htm Acceso: 5 de Octubre de 2012
- JavaServer Faces Technology. Internet. http://aragorn.pb.bialystok.pl/~dmalyszko/PSS_Project/JavaServer%20Faces.htm Acceso: 20 de Noviembre de 2012
- Kumar, B. V. Narayan, Prakash. Ng, Tony. Implementing SOA using Java EE, Addison-Wesley, 2009.
- Man, Kito. JavaServer Faces in Action, Greenwich, Manning Publications Co, 2005.
- Mapping Entities. Internet. http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html Acceso: 22 de Octubre de 2012.
- Primefaces documentation. Internet. http://primefaces.org/documentation.html. Acceso: 29 de Noviembre de 2012.
- Rigada, Aitor. Model View Controller. Internet. http://blogdeaitor.wordpress.com/2008/10/20/model-view-controller/ Acceso: 22 de Noviembre de 2012.
- Singh, Inderjeet. Stearns, Beth. Johnson, Mark. Designing Enterprise Applications with the JEETM Platform, Addison-Wesley Professional, 2da Edición, 2004.
- Spring Framework Reference Manual. Internet. http://static.springsource.org/spring/docs/3.1.0.M2/spring-framework-reference/html/ Acceso: 10 de Noviembre de 2012.
- Sun Java Center - JEE Patterns. Internet. http://java.sun.com/developer/technicalArticles/JEE/patterns/ Acceso: 13 de Octubre de 2012.