Universidad de las Ciencias Informáticas

FACULTAD 3

Título: Propuesta de Modelo Arquitectónico para Aplicaciones Empresariales sobre la Plataforma JEE

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autores: Yisel Jinoria Fernández
Daynier Ruiz Rodríguez

Tutor: Dr.C Pedro Yobanis Piñero Pérez

Mayo de 2007

DECLARACIÓN DE AUTORÍA

Declaramos que somos los únicos autores de este trabajo y autorizamos a la Infraestructura Productiva de la Universidad de las Ciencias Informáticas; así como a dicho centro para que hagan el uso que estimen pertinente con este trabajo.

Para que así conste firmamos la presente a los ___ días del mes de _____ del año 2007.

Yisel Jinoria Fernández	Daynier Ruiz Rodríguez	Dr.C. Pedro Yobanis Piñero Pérez
_____	_____	_____
Firma del Autor	Firma del Autor	Firma del Tutor

OPINIÓN DEL USUARIO DEL TRABAJO DE DIPLOMA

El Trabajo de Diploma, titulado PROPUESTA DE MODELO ARQUITECTÓNICO PARA APLICACIONES EMPRESARIALES SOBRE LA PLATAFORMA JEE, fue realizado en la Universidad de las Ciencias Informáticas. Esta entidad considera que en correspondencia con los objetivos trazados, el trabajo realizado le satisface

- □ Totalmente
- □ Parcialmente en un ____ %

Los resultados de este Trabajo de Diploma le reportan a esta entidad los beneficios siguientes (cuantificar):

_____
_____
_____
_____
_____
_____
_____

Como resultado de la implantación de este trabajo se reportará un efecto económico que asciende a _____.

Y para que así conste, se firma la presente a los ____ días del mes de _____ del año _____.

_____	_____
Representante de la entidad	Cargo
_____	_____
Firma	Cuño

OPINIÓN DEL TUTOR DEL TRABAJO DE DIPLOMA

Título: PROPUESTA DE MODELO ARQUITECTÓNICO PARA APLICACIONES EMPRESARIALES SOBRE LA PLATAFORMA JEE.

Autores: Yisel Jinoria Fernández y Daynier Ruiz Rodríguez.

Los estudiantes Yisel Jinoria Fernández y Daynier Ruiz Rodríguez durante el desarrollo del Trabajo de Diploma demostraron un elevado nivel de independencia y originalidad. A partir de las sugerencias y las orientaciones que recibieron, desarrollaron un trabajo profundo de investigación mostrando además una elevada creatividad. El trabajo escrito presenta también una elevada calidad técnica, constituye uno de los problemas principales que presenta la universidad hoy en la organización del proceso de producción de software. Con este trabajo se logra un aporte importante a la facultad y a la universidad en la estrategia de optar por la plataforma Java para el desarrollo de software comercial y para la informatización del país.

El trabajo escrito presenta también una elevada calidad y los resultados obtenidos son adecuados. En el documento escrito, como parte del análisis de los resultados, se presenta un interesante análisis comparativo entre la propuesta de la tesis y las soluciones de software más importantes implementadas anteriormente por la universidad con esta tecnología. Sobre la base de ejemplos concretos y la experiencia acumulada este trabajo muestra por que la arquitectura propuesta en esta tesis puede ser atractiva. Este trabajo no constituye un trabajo aislado sino que es parte de la estrategia de la facultad por estandarizar, mejorar la calidad y productividad en nuestra incipiente industria de software. Este trabajo debe ser continuado a partir de las recomendaciones en el mismo.

Este ejercicio representa el último de la carrera pero no constituye un esfuerzo aislado sino que se corresponde con la actitud ejemplar mantenida por los estudiantes durante los cinco años de la carrera. Este ejercicio final constituye un premio a su trabajo, que se ha caracterizado por una actitud destacada en la docencia y la producción, las dos misiones fundamentales de nuestra universidad.

Por todo lo anteriormente expresado considero que los estudiantes están aptos para ejercer como Ingenieros en Ciencias Informáticas; y propongo que se le otorgue al Trabajo de Diploma la calificación de Excelente 5 puntos.

Fecha: 11 de mayo de 2007

Tutor: Profesor Asistente Dr.C. Pedro Yobanis Piñero Pérez

Dedicatoria

A mami y papi, por la educación que me dieron, sin ustedes hubiera sido imposible llegar hasta aquí.

A mi abuelito Emy, gracias por confiar en mí siempre.

A mi hermano Jose, malanga estoy orgullosa de ti.

A mi abuela mima, vieja gracias por quererme tanto y apoyarme.

A mis dos tías queridas, Babey y Vivian.

A mi tío Omar por tenerme en un lugar muy especial.

A mi abuelita tata y mi abuelito Héctor.

A mi tío Rafe.

A mis primitas queridas: Dayana, Mary, Lesyani, Yanne, Marliuvys y Yaque.

A mi abuelo pipa y mi abuela cuca, no saben cuanto quisiera que me vieran graduarme.

A Daynier, pipo gracias por apoyarme en todo y ayudarme en mi formación.

Yisel

Para mamá y papá por guiarme y estar siempre a mi lado.
Para mimi y papa por todo lo grande que han hecho por mí, por darme el tamaño que tengo.
A mi hermana, vivi te quiero mucho.

De manera muy especial a TITA donde quiera que esté por quererme tanto y ayudarme siempre, hubiera dado cualquier cosa por tenerte en este momento.

Daynier

Agradecimientos

A nuestras familias por sabernos guiar en la vida por el camino correcto.

A nuestros profesores, en especial a nuestro tutor Pedro Yobanis Piñero Pérez, por su ayuda en nuestra formación como profesionales.

A nuestros amigos, en especial a George.

Al proyecto Registros y Notarías por fortalecer nuestros conocimientos.

A nuestros compañeros de grupo y de proyecto.

A mi pareja.

Resumen

En este Trabajo de Diploma se hace una exploración y estudio del estado del arte de los elementos de la plataforma Java Enterprise Edition y los modelos arquitectónicos tanto para aplicaciones distribuidas como no distribuidas que contribuyan a la selección adecuada de arquitecturas de software donde se apliquen buenas prácticas de programación y se obtengan soluciones robustas, que carezcan de complejidad innecesaria, con mejores propiedades de escalabilidad, portabilidad y reusabilidad. Se investigan los frameworks de desarrollo a aplicar en las diferentes capas de una solución empresarial, se valoran sus deficiencias y bondades, así como también los patrones de diseño que estos implementan. Se realiza un análisis crítico y comparación de modelos arquitectónicos híbridos donde se destacan las potencialidades y fortalezas de los frameworks de desarrollo que intervienen. Se presenta una propuesta arquitectónica para su utilización a la hora de la toma de decisiones en proyectos actuales que logra los objetivos de una arquitectura empresarial y permite un ágil desarrollo de la misma. Además se comparan las arquitecturas utilizadas en varios proyectos y los resultados alcanzados por estos con respecto a la propuesta.

Índice

Introducción

Actualidad del tema

Java es uno de los lenguajes de programación más elaborados y más utilizados para la creación de software de empresa en la actualidad. La tecnología Java se crea como una herramienta de programación en una pequeña operación secreta y anónima denominada "the Green Project" de Sun Microsystems en el año 1991. El equipo secreto llamado "Green Team", estaba compuesto por trece personas y dirigido por James Gosling[1].

El resultado fue un lenguaje de programación que no dependía de los dispositivos denominado "Oak". Poco tiempo después Internet estaba listo para la tecnología Java y, justo a tiempo para su presentación en público en 1995, el equipo pudo anunciar que el navegador Netscape Navigator incorporaría la misma[1]. La plataforma Java ha atraído alrededor de 4 millones de desarrolladores de software, se utiliza en los principales sectores de la industria de todo el mundo y está presente en un gran número de dispositivos, ordenadores y redes de cualquier tecnología de programación.

De hecho, su versatilidad y eficiencia, la portabilidad de su plataforma y la seguridad que aporta, la han convertido en la tecnología ideal para su aplicación a redes, de manera que hoy en día, más de 2.500 millones de dispositivos la utilizan[1]. Esta tecnología goza de una gran madurez, es extremadamente eficaz y sorprendentemente versátil, se ha convertido en un recurso inestimable ya que permite a los desarrolladores:

- Desarrollar software en una plataforma y ejecutarlo en cualquier otra.
- Crear programas para que funcionen en un navegador web.
- Desarrollar aplicaciones para servidores como foros en línea, tiendas, encuestas, procesamiento de formularios HTML.
- Desarrollar potentes y eficientes aplicaciones para teléfonos móviles, y prácticamente cualquier dispositivo digital.

La empresa informática Sun Microsystems, como propietaria del lenguaje, se ha encaminado a la creación de comunidades y a compartir innovaciones y tecnologías para fomentar una mayor participación en ellas. Entre los proyectos de la empresa está el de liberar el código fuente para la máquina virtual Java HotSpot,

el javac[1] y el JavaHelp[2], también el de proveer una versión de un kit completo de Java SE[3] Development Kit (JDK) basado completamente en código open source[2].

La empresa tiene como objetivo liberar en primer lugar el código fuente para las implementaciones Java ME[4] Feature Phone basadas en Connected Limited Device Configuration (CLDC), las que actualmente permiten servicios de datos móviles enriquecidos en más de 1.5 billones de handsets y el código fuente para el framework Java Micro Edition testing and compatibility kit (TCK)[2].

Al hacer open source sus implementaciones Java, Sun genera nuevas oportunidades de mercado y fomenta la innovación. Los desarrolladores pueden mejorar la calidad y funcionalidad de la plataforma contribuyendo a las mejoras de sus características y la reparación de fallas[2].

Con la evolución del lenguaje Java se han desarrollado tres plataformas, cada una de ellas orientadas a cubrir entornos diferentes:

- Java Standard Edition o Java SE (anteriormente J2SE) es una colección de APIs[5] del lenguaje de programación Java útiles para muchos programas de la plataforma.
- Java Enterprise Edition o Java EE (anteriormente J2EE) es una plataforma para crear aplicaciones de cliente-servidor.
- Java Micro Edition o Java ME (anteriormente J2ME) es una colección de APIs de Java para el desarrollo de software para dispositivos de recursos limitados, como PDA (Personal Digital Assistant o Asistentes Personales Digitales), teléfonos móviles y otros aparatos de consumo.

Java ha dado lugar a la creación de nuevos modelos y tecnologías de programación en diferentes campos, desde los dispositivos más sencillos hasta aplicaciones de empresa, pasando por telefonía. La plataforma de informática de empresa de Java, la plataforma Java Enterprise Edition (JEE), es uno de estos campos, la cual está especialmente pensada para la creación de aplicaciones web[3]. El Trabajo de Diploma se centra en esta plataforma.

El objetivo de JEE es simplificar algunas de las complejidades técnicas y se apoya en los siguientes elementos para crear aplicaciones de empresa:

- Un modelo de programación, consistente en un conjunto de interfaces de programación de aplicaciones (API) y en los enfoques para la creación de dichas aplicaciones.

[1] Compilador del lenguaje de programación Java.
[2] Software de ayuda en línea.
[3] Standard Edition.
[4] Micro Edition.
[5] Interfaces de programación de aplicaciones.

- Una infraestructura de aplicación, para respaldar las aplicaciones de empresa creadas utilizando los API.
- Frameworks.

Entre las interfaces de programación utilizadas por la plataforma para brindar una infraestructura de aplicación se encuentran:

- Java DataBase Connection (JDBC), especifica todo lo referente al trabajo con las bases de datos.
- Enterprise JavaBeans (EJB), especifica una estructura de componentes que encapsulan lógica de empresa para aplicaciones distribuidas.
- Java Servlets, proporciona abstracciones orientadas al objeto para construir aplicaciones web dinámicas.
- Java Server Pages (JSP), esta extensión abarca lo referente a la confección de las interfaces web del lado del servidor.
- Java Message Service (JMS), abarca los servicios orientados a mensajes.
- Java Transaction API (JTA), destinado a implementar aplicaciones transaccionales distribuidas.
- JavaMail, proporciona soporte para aplicaciones de e-mail basadas en Java.
- Java API XML Parsing (JAXP), proporciona abstracciones para analizadores XML y API de transformación[3].

Estos APIs constituyen la base de las aplicaciones JEE, los mismos van a formar eslabones fundamentales en las arquitecturas.

Como elementos esenciales mencionados anteriormente para la creación de aplicaciones de empresa se encuentran los frameworks. Los mismos constituyen implementaciones basadas en patrones de diseño, soluciones comunes normalmente empaquetadas para poder hacer más rápido el desarrollo haciendo uso de buenas prácticas, algo así como las "librerías de funciones" usadas constantemente en diversos proyectos.

Entre los principales frameworks de la plataforma se encuentran:

- Struts Framework, recomendado para aplicaciones de gran tamaño y complejidad, posee un conjunto de clases que usan los estándares JEE (Servlets, JSP, Tags Personalizados, XML) y ayudan a hacer el desarrollo más rápido cuando se trata de una aplicación web. Struts está basado en la arquitectura MVC (Modelo-Vista-Controlador), lo que garantiza la separación de cada una de estas capas, con vistas a lograr una aplicación escalable, reutilizable y con un nivel alto de profesionalidad[4].

- Java Server Faces (JSF, o simplemente “Faces”), posibilita desarrollar aplicaciones web de una manera fácil y brinda poderosos componentes de interfaz de usuario, tal es el caso de textboxes, listboxes, paneles y datagrids. Brinda una arquitectura de componentes, un grupo de elementos para la interfaz de usuario y una infraestructura de aplicación. Posee una poderosa paleta de componentes orientados a eventos, lo cual permite un manejo de estos[5].
- Spring, centrado a manejar los objetos de negocio. Facilita el desarrollo de buenas prácticas de programación. Ayuda a resolver muchos problemas evitando el uso de los EJB, provee una alternativa a los mismos que es apropiada para muchas aplicaciones. Spring puede usar Programación Orientada a Aspectos(AOP, por sus siglas en inglés) para el manejo de transacciones, seguridad, sin tener que usar el contendor de EJB[6].
- Acegi, posibilita el manejo de la seguridad de las aplicaciones. Está muy ligado al framework Spring. Lleva a cabo la autenticación de los usuarios de la aplicación, el acceso a los recursos y control de los permisos[6].
- Hibernate, hace posible el manejo de la capa de persistencia de cualquier aplicación. Realiza el mapeo entre el mundo orientado a objetos de las aplicaciones y el mundo entidad-relación de las bases de datos en entornos Java. El término utilizado es ORM (object/relational mapping) y consiste en la técnica de realizar la transición de una representación de los datos de un modelo relacional a un modelo orientado a objetos y viceversa. Constituye un motor de persistencia que implementa múltiples funcionalidades[7].

Formulación del problema

Mientras este conjunto de opciones da la posibilidad de diseñar las mejores soluciones para cada caso, también puede ser problemático. A pesar del avance de la plataforma JEE y el surgimiento de poderosos frameworks que al implementar patrones dan la posibilidad de hacer el trabajo de una manera más fácil, rápida y con la elegancia y profesionalidad que se requiere, pueden agobiar a los desarrolladores producto del desconocimiento y la falta de experiencia de los mismos, o tentarlos a usar infraestructuras inapropiadas para dar respuesta a las situaciones planteadas, simplemente porque se encuentran disponibles.

La selección por parte de los desarrolladores de arquitecturas poco factibles puede traer dificultades de escalabilidad, donde un aumento de recursos dedicados supone modificaciones en su comportamiento;

las aplicaciones dejan de ser portables, van perdiendo el poder de adaptarse a las distintas arquitecturas físicas y hay un aumento de la necesidad de modificar el código de la misma ante dichas situaciones. Además puede no tener presente la reutilización de código como uno de los objetivos más claros e importantes y no contemplarla desde dos perspectivas distintas: no desarrollar elementos o componentes que ya existan y estén probados, y desarrollar pensando en que mañana el código implementado pueda ser reutilizado en distintos proyectos. Es posible que no se logre una independencia entre los componentes del sistema y que se encuentren muy acoplados. La adopción de una arquitectura compleja innecesariamente, puede traer consigo grandes esfuerzos y costos en el desarrollo de funcionalidades que son requisitos irrelevantes para el software, además de dificultades en las pruebas y el despliegue del sistema.

Una vez mostrada la situación a la que se enfrentan los desarrolladores se está en condiciones de plantear el problema:

Las dificultades en la selección de modelos arquitectónicos híbridos a partir de estrategias y tecnologías existentes sobre la plataforma JEE, provoca en ocasiones la selección errónea de modelos de arquitectura y por consiguiente la creación de soluciones poco factibles; dificultades de escalabilidad, portabilidad, reusabilidad y le atribuye a las mismas una complejidad innecesaria que aumenta el costo del proyecto.

Objeto de estudio

Desarrollo de software de gestión empresarial en la plataforma JEE.

Campo de acción

Arquitecturas empresariales en la plataforma JEE.

Hipótesis

Si se desarrollan modelos arquitectónicos híbridos que potencien las fortalezas de diferentes estrategias para el desarrollo sobre la plataforma JEE se obtendrá una guía para la selección adecuada de arquitecturas en los proyectos y la construcción de soluciones con mejores propiedades de escalabilidad, portabilidad, reusabilidad y con una mayor productividad en su implementación.

Objetivo general

Desarrollar un modelo arquitectónico para su uso en proyectos de software de gestión sobre la plataforma JEE.

A partir de este objetivo general se trazaron los siguientes objetivos específicos:

- Analizar las tecnologías disponibles para el desarrollo de aplicaciones en la plataforma JEE.
- Realizar un estudio de modelos arquitectónicos para soluciones empresariales sobre la plataforma JEE.
- Desarrollar un modelo de arquitectura para su uso en aplicaciones empresariales sobre la plataforma JEE.

Tareas de la investigación

- Estudiar las tecnologías presentes en la plataforma JEE.
- Estudiar los frameworks de desarrollo y los patrones de diseño que estos implementan.
- Estudiar la actualidad de modelos arquitectónicos para su uso en aplicaciones empresariales sobre la plataforma JEE.
- Brindar como resultado del análisis un modelo que sea el más factible de utilizar en soluciones empresariales.
- Realizar análisis crítico y comparación de arquitecturas existentes con respecto a la propuesta.

Metodología de la investigación

La estrategia de investigación utilizada fue la exploratoria. Se realizó una exploración de la plataforma JEE, sus elementos y los diferentes modelos arquitectónicos de software empresarial para soluciones distribuidas y no distribuidas, con vistas a realizar un análisis de la factibilidad y finalmente proponer un modelo híbrido que sea el más indicado.

Métodos teóricos: Histórico lógico, Hipotético deductivo.

La investigación se centra en el estudio del estado del arte de los modelos arquitectónicos de software de gestión para aplicaciones distribuidas y no distribuidas, se analizaron los diferentes frameworks de trabajo sobre la plataforma JEE y los patrones de diseño que estos implementan, se profundizó en la esencia de los mismos, sus deficiencias y bondades. La investigación sigue además un método hipotético deductivo porque a partir del problema se realizó el planteamiento de una hipótesis que en el transcurso de la investigación se verifica, comprueba y arriba a conclusiones.

Valor práctico

La propuesta de modelo arquitectónico para aplicaciones no distribuidas tiene un alto valor práctico para la ayuda a la toma de decisiones a la hora de plantear una arquitectura para soluciones de software

empresariales sobre la plataforma JEE, pues se realiza análisis crítico de cada uno de sus componentes, tecnologías, herramientas, los pilares arquitectónicos que cumple, entre otros.

Resultados esperados

Como resultado de la investigación, después del análisis de la plataforma JEE, sus elementos y la valoración de los diferentes modelos arquitectónicas para soluciones empresariales sobre la plataforma JEE en aplicaciones tanto distribuidas como no distribuidas, se obtiene una propuesta de arquitectura que sea la indicada para utilizar en proyectos de software, propicie la portabilidad del producto, la escalabilidad y la reutilización de sus componentes; haga posible que se eliminen los grandes esfuerzos y costos en el desarrollo de las funcionalidades del mismo.

Estructura del trabajo

En el capítulo 1 se realiza un análisis del estado del arte de los elementos de la plataforma JEE, destacando las diferentes tecnologías que posee la misma, los patrones de diseño, los frameworks de desarrollo y los servidores de aplicaciones y contenedores web. El capítulo 2 hace un estudio del estado del arte de los estilos arquitectónicos tradicionales, los pilares por los que debe regirse cualquier arquitectura empresarial, se analizan los diferentes modelos arquitectónicos tanto para aplicaciones distribuidas como no distribuidas exponiendo sus deficiencias y bondades. El capítulo 3 se apoya en el análisis de los modelos arquitectónicos realizado en el capítulo 2 para hacer una propuesta de un modelo arquitectónico híbrido que potencie las fortalezas del uso de los frameworks de desarrollo de la plataforma JEE. El capítulo 4 analiza los resultados alcanzados con la propuesta respecto a una comparación realizada con otras arquitecturas de proyectos.

Capítulo 1: Análisis de los elementos de la plataforma JEE.

1.1 La plataforma JEE

La plataforma empresarial de Java provee un conjunto de librerías, interfaces, frameworks que brindan una infraestructura para el desarrollo de arquitecturas empresariales, abarcando cada uno de los elementos como las transacciones, la mensajería, las conexiones a base de datos, interfaces de usuario, manejo de recursos, seguridad, entre otros. Además JEE es una plataforma con experiencia y amplio soporte en el mundo. Dentro de las principales empresas en el mundo que desarrollan para la plataforma se encuentran Sun, IBM, Oracle, Apache, BEA, JBOSS, entre otras, las cuales aportan gran cantidad de componentes y herramientas que apoyan y facilitan el trabajo de los desarrolladores en los proyectos.

1.2 APIs JEE

La plataforma JEE define un grupo de extensiones estándar Java que son la base para el desarrollo de arquitecturas empresariales, las APIs. A continuación se exponen algunas de ellas:

Servlets

Son bloques de construcción básicos para construir interfaces de base web para aplicaciones. La tecnología de servlets proporciona un modelo de programación común que es también la base de las Páginas JavaServer.

Los servlets se ejecutan dentro de un contenedor web [6] como Tomcat, WebLogic, entre otros, y proporcionan la comunicación entre el cliente y el servidor web. Los servlets pueden comunicarse entre sí, por tanto es posible la reasignación dinámica de procesos; trabajan transparentemente con la filosofía get y post de los formularios HTML[7], manejando peticiones y respuestas; se entienden perfectamente con los Applets[8]. Además, permiten la generación dinámica de páginas HTML. Los servlets no son adecuados para construir las páginas de la aplicación, ellos se utilizan generalmente como controladores. En grandes aplicaciones el desarrollo con servlets puede ser agotador por la gran cantidad de tipos de solicitudes que se pueden generar en ellas y por tanto las complejas reglas de navegación que se generan, para estos casos es mucho más recomendable la utilización de otras herramientas o frameworks que implementen soluciones de este tipo[3].

---

6 Entorno de ejecución Java que implementa el API Servlet y el API JSP.
7 **H**yper**T**ext **M**arkup **L**anguage (lenguaje de marcas hipertextuales).
8 Componente de software que corre en el contexto de otro programa, por ejemplo en un navegador web.

Java Server Pages (JSP)

El objetivo de la especificación Java Server Pages (JSP) es simplificar la creación de páginas web dinámicas proporcionando un sistema de composición más conveniente que los servlets. Las páginas JSP se ejecutan en un servidor web y combinan anotación estática, como HTML y XML[9], con librerías de etiquetas y scriptlets[10]**.** Las páginas JSP son similares a documentos de directivas pero cada página JSP es traducida a un servlet cada vez que es invocada por primera vez. El servlet resultante es una combinación de la anotación del archivo JSP y el contenido dinámico integrado especificado por las etiquetas de directiva. JSP ha sido un método sólido y de uso muy extendido para la generación de contenido web[3].

Los servlets se ejecutan en el servidor e interceptan solicitudes de navegador, actuando como una especie de capa intermedia entre clientes y aplicación de menor nivel. Los servlets son adecuados para decidir cómo manejar solicitudes de cliente e invocar otros objetos de lado servidor pero no son tan adecuados para generar contenido. La experiencia ha demostrado que la generación de anotaciones a partir de código Java es difícil de implementar y de mantener. Los servlets deben ser escritos por desarrolladores familiarizados con Java[3].

Las páginas JSP, por otro lado, pueden ser diseñadas y desarrolladas no tanto como programas sino más como páginas web. Las páginas JSP son ideales para situaciones en las que se necesita mostrar anotaciones con contenido dinámico integrado. Sin embargo, aunque generar HTML es mucho más fácil con JSP que con un servlet, las páginas JSP son menos adecuadas para manejar la lógica de procesamiento. Las páginas JSP pueden utilizar JavaBeans[11] con un alcance o extensiones de etiquetas especificados para alcanzar una clara separación entre contenido estático y el código Java que produce aplicaciones web dinámicas. Esto permite que las páginas JSP sean creadas y mantenidas por diseñadores con aptitudes de presentación que no necesitan tener conocimientos de Java[3].

Aunque hay una coincidencia de capacidad, los servlets son vistos como objetos controladores y las páginas JSP como objetos de vista. Son tecnologías complementarias y una aplicación web hace uso de ambas. Los inconvenientes del trabajo con JSP están dados por contar con un reducido grupo de librerías de etiquetas, lo que es traumático para construir las vistas en aplicaciones grandes, una alternativa a esto podría ser el desarrollo de etiquetas personalizadas aunque podría tomar un poco de tiempo.

[9] e**X**tensible **M**arkup **L**anguage (lenguaje de marcas extensibles).
[10] Bloque de código Java que es ejecutado durante el período de procesamiento de solicitud.
[11] Componente que se puede reutilizar y manipular visualmente por una herramienta de programación en lenguaje Java.

Generalmente los desarrolladores incorporan otras librerías que incluyen muchas prestaciones, estas generalmente vienen incorporadas en frameworks de presentación como Struts, JSF, Tapestry, entre otros.

Enterprise JavaBeans (EJB)

Los EJBs proporcionan un modelo de componentes distribuido estándar para el lado del servidor. Su objetivo es encapsular lógica de empresa, es decir, dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes permite que éstos sean flexibles y sobre todo reutilizables. Son concebidos por muchos el centro de JEE. Un EJB se ejecuta en un contenedor EJB. El contenedor EJB se ejecuta en un servidor de aplicación y se responsabiliza de los temas de nivel de sistema. Los EJB se benefician de los servicios que aporta el contenedor EJB como seguridad, comunicación, manejo de transacciones, entre otros, muchos de estos bastan con ser configurados en un fichero XML que posee cada uno de los EJB y que se conoce como descriptor de despliegue. Los EJB pueden ser remotos (distribuidos en varias máquinas virtuales) o locales (ejecutándose en la misma máquina virtual que el resto de la aplicación)[3].
La especificación Enterprise JavaBeans ofrece tres modelos diferentes para Enterprise JavaBeans, beans de entidad, beans de sesión y beans controlados por mensaje.

- Beans de entidad

  Representan los objetos entidad del modelo de análisis. Pueden corresponder a conceptos del mundo real, como compradores o productos, o pueden corresponder a abstracciones, como procesos de fabricación. Son una representación de datos orientados hacia el objeto en una base de datos, manejan la persistencia de los mismos. Al igual que una base de datos, múltiples clientes pueden acceder simultáneamente a ellos. La vida de un EJB de entidad es tan larga como la de los datos que representa en la base de datos.

- Beans de sesión

  El bean de sesión se utiliza para representar flujo de trabajo, lógica de empresa y estado de aplicación. Existen dos tipos de beans de sesión: con estado y sin estado. La diferencia fundamental entre los dos tipos de beans es cómo tratan el estado de cliente. Un bean de sesión con estado puede mantener datos entre diferentes accesos de cliente. Un bean de sesión sin estado, no. La vida de un bean de sesión no se corresponde con la de su cliente. Cuando el cliente abandona el sitio web o cierra la

aplicación, el bean de sesión tiene libertad para desaparecer. Ya no está disponible para su acceso por otros clientes.

- Beans controlados por mensaje

El bean controlado por mensaje es, esencialmente, un oyente de mensajes que puede consumir mensajes de una cola a través del contenedor JEE; el contenedor invoca al bean como resultado de la llegada de un mensaje. A diferencia de otros tipos de EJB, los beans controlados por mensaje no tienen interfaces iniciales ni remotas. Debido a la carencia de estas interfaces del lado cliente, los clientes no pueden interactuar con ellos; el bean es invocado con la llegada de un mensaje[3].

El problema de los EJB es que resultan muy difíciles de implementar, sobre todo los EJB remotos, además el hecho de ejecutarse en un contenedor EJB y por tanto en un servidor de aplicaciones también dificulta su desarrollo puesto que la configuración y administración de un servidor de aplicaciones es más compleja que la de un contenedor de servlet. En el caso de EJB de entidad, estos resultan más pesados que otros mecanismos de persistencia como Hibernate. Los EJB se encuentran acoplados a un contenedor el cual constituye su entorno de ejecución, a la implementación y extensión de interfaces y clases que le proporcionan funcionalidades especiales.

Java Data Base Connection (JDBC)

Una base de datos relacional es normalmente el recurso primario de datos en una aplicación de empresa. El API JDBC ofrece a los desarrolladores un modo de conectar con datos relacionales desde el interior del código Java. Utilizando el API JDBC, los desarrolladores pueden crear un cliente (que puede ser desde un Applet hasta un EJB) que se pueda conectar con una base de datos, ejecutar instrucciones de Structured Query Language (SQL) y que procese el resultado de esas instrucciones[3]. El trabajo con JDBC resulta ligero, su dificultad radica en las aplicaciones grandes donde se vuelven interminables las consultas SQL para realizar todas las operaciones con la base de datos, además de la complejidad de muchas, en estas aplicaciones más de la mitad del código fuente es de acceso a datos. Además, en caso de trabajar con varios gestores de bases de datos o en caso de migrar de gestor, habría que codificar las sentencias SQL para cada uno o para el nuevo, esto le resta portabilidad a las aplicaciones.

Java Message Service (JMS)

El Servicio de Mensajes Java es un API que proporciona interfaces a las aplicaciones para que creen, reciban y lean mensajes utilizando cualquier implementación que se adapte al mismo[3].

Web Services

Los servicios web (Web Services) pueden definirse como componentes de aplicación accesibles a través de protocolos web estándar. Son unidades de lógica de aplicación. Proporcionan servicios y datos a clientes remotos y otras aplicaciones. Los clientes y las aplicaciones remotas acceden a los mismos a través de protocolos de Internet omnipresentes. Utilizan XML para el transporte de datos y SOAP (Simple Object Access Protocol) para hacer uso de servicios. Debido al uso de XML y SOAP, el acceso al servicio es independiente de la implementación[3].

Los mismos solucionan el problema de interoperabilidad entre aplicaciones, sin embargo las llamadas a estos servicios web son más lentas que a métodos locales, por lo que no se deben utilizar desmedidamente en una aplicación.

1.3 Patrones de diseño

Los patrones J2EE describen típicos problemas encontrados por desarrolladores de aplicaciones empresariales y proveen soluciones para los mismos. Estos patrones, a diferencias de los clásicos, están enfocados a los API JEE. En esencia, ayudan a diseñar y construir aplicaciones para la plataforma. La elección de un modelo arquitectónico que tenga en cuenta estos patrones puede cumplir en gran medida los objetivos en los que debe basarse cualquier arquitectura empresarial.

Estos patrones se agrupan en tres:

- Grupo de presentaciones: Todo lo necesario para presentar los datos de la aplicación y los elementos de la interfaz de usuario está dentro del grupo de presentaciones de la aplicación. Las tecnologías fundamentales en uso son Java Server Pages (JSP) y los Java Servlet.
- Grupo de negocios: El grupo de negocios es donde tiene lugar todo el proceso de negocios. Las principales tecnologías JEE de este grupo son los Enterprise JavaBean (EJBs).
- Grupo de integración: El grupo de integración proporciona conexiones al grupo de recursos. El grupo de recursos incluye elementos como las colas de mensajes, bases de datos y sistemas heredados. Las tecnologías JEE más destacadas son Java Message Service (JMS) y Java Database Connectivity (JDBC).

Patrones de presentación

Front Controller (Controlador Frontal)

Contexto: La gestión de las peticiones http a la aplicación web puede ser centralizada o distribuida.

Problema: Tener distribuida la gestión de peticiones en las páginas jsp lleva a aplicaciones de baja calidad, en las que se genera mucho código similar y distribuido por todas las páginas (vistas).
Solución: Usar un controlador como punto inicial para la gestión de las peticiones. El controlador gestiona estas peticiones, y realiza algunas funciones como: comprobación de restricciones de seguridad, manejo de errores, realiza un mapeo y delegación de las peticiones a otros componentes de la aplicación que se encargan de generar la vista adecuada para el usuario.
El controlador ayuda a reducir la cantidad de código java que se encuentra embebido en las páginas jsp (a este código se le suele llamar scriptlet). Lo que se hace es centralizar control en el controlador y reducir lógica de negocio en las vistas. La implementación típica de un controlador suele ser un servlet.
Consecuencias: Se centraliza el control, mejora la manejabilidad de la seguridad, y aumenta la reusabilidad del código[10].

Dispacher View

Contexto: El sistema controla el flujo de ejecución y acceso al procesamiento de la presentación, el cual es responsable de la generación dinámica de contenidos.
Problema: No hay un componente centralizado para la gestión del control de acceso, la recuperación de contenidos o la gestión de las vistas, y existe código duplicado a través de múltiples vistas. Adicionalmente, la lógica de negocio y la lógica de formateo de la presentación están mezcladas en estas vistas, haciendo al sistema menos flexible, menos reusable, y generalmente menos resistente al cambio.
Solución: Combinar un controlador y un despachador con vistas para manejar las peticiones de los usuarios y preparar una presentación dinámica como respuesta. Un dispatcher es responsable de la gestión de vistas y de la navegación y puede ser encapsulada tanto dentro de un controlador, una vista, o como un componente separado.
Consecuencias: Centraliza control, mejora la reusabilidad, la mantenibilidad y el particionamiento de la aplicación[10].

Composite View

Contexto: Las aplicaciones web sofisticadas presentan contenido de numerosas fuentes de datos, usando múltiples subvistas, que conforman una única página a visualizar.
Solución: Usar vistas compuestas que estén formadas por múltiples subvistas atómicas. Cada componente de la plantilla puede ser incluido dinámicamente en el total, y el esquema de la página puede ser gestionado de forma independiente al contenido.

Consecuencias: Mejora la modularidad, la reutilización, la flexibilidad, el mantenimiento, la manejabilidad. Puede tener un impacto negativo (aunque muy pequeño) en el rendimiento[10].

View Helper

Contexto: El sistema crea presentaciones del contenido, lo que requiere procesamiento dinámico de datos de negocio.

Problema: Es frecuente hacer cambios en la capa de presentación de una aplicación, y esto es difícil de hacer cuando la lógica de acceso a datos de negocio y la lógica de formateo de la presentación están mezcladas en esta capa.

Solución: Usar clases ayudadoras, implementadas generalmente como JavaBeans o tags personalizados para que la vista delegue sus responsabilidades de procesamiento. Los helpers o ayudadores también guardan los modelos de datos intermedios de la vista y sirven como adaptadores de los datos de negocio.

Consecuencias: Mejora el particionamiento de la aplicación, su reutilización y mantenibilidad[10].

Patrones de negocio

Business Delegate

Contexto: Un sistema multi-capa distribuido requiere invocación remota de métodos para enviar y recibir datos entre las capas. Los clientes están expuestos a la complejidad de tratar con componentes distribuidos.

Problema: Los componentes de la capa de presentación interactúan directamente con servicios de negocio. Esta interacción directa expone los detalles de la implementación del API del servicio de negocio a la capa de presentación. Como resultado, los componentes de la capa de presentación son vulnerables a los cambios en la implementación de los servicios de negocio: cuando cambia la implementación del servicio de negocio, la implementación del código expuesto en la capa de presentación también debe cambiar.

Solución: Utilizar un Business Delegate como una abstracción de negocio para reducir el acoplamiento entre los clientes de la capa de presentación y los servicios de negocio, así como para ocultar los detalles de la implementación del mismo[11].

Service Locator

Contexto: La búsqueda y creación de servicios implican interfaces complejas y operaciones de red.

Problema: Los clientes JEE interactúan con componentes de servicio, como componentes Enterprise JavaBeans (EJB) y Java Message Service (JMS). Para interactuar con estos componentes, los clientes deben localizar el componente de servicio (referido como una operación de búsqueda) o crear un nuevo componente. Localizar un objeto servicio administrado JNDI[12] es una tarea común para todos los clientes que necesiten acceder al objeto de servicio, muchos tipos de clientes utilizan repetidamente el servicio JNDI, y que el código JNDI aparece varias veces en esos clientes. Esto resulta en una duplicación de código innecesaria en los clientes que necesitan buscar servicios. Se necesitan muchos recursos para localizar a este objeto de servicio y si muchos clientes lo solicitan a la vez esto puede causar impacto negativo en el rendimiento de la aplicación.

Solución: Utilizar Service Locator para abstraer todas las dependencias y la utilización JNDI. Varios clientes pueden reutilizar el Service Locator para reducir la complejidad del código, la duplicación del mismo y el consumo de demasiados recursos[11].

Session Facade

Contexto: Los Beans Enterprise encapsulan lógica y datos de negocio y exponen sus interfaces, y con ellos la complejidad de los servicios distribuidos, a la capa de cliente.

Problema: Acoplamiento fuerte, que provoca la dependencia directa de los clientes a la implementación de los objetos de negocio, los clientes de la aplicación necesitan acceder a los objetos de negocio para cumplir sus responsabilidades y para cumplir los requerimientos del usuario. El cliente se vuelve muy susceptible a los cambios en la capa de objetos de negocio. Se produce una degradación del rendimiento de la red porque el gran volumen de llamadas a métodos remotos incrementa la cantidad de interacciones sobre la capa de red. Como los objetos de negocio se exponen directamente a los clientes, no hay una estrategia unificada para acceder a ellos.

Solución: Usar un bean de sesión como una fachada (facade) para encapsular la complejidad de las interacciones entre los objetos de negocio participantes en un flujo de trabajo. El Session Facade maneja los objetos de negocio y proporciona un servicio de acceso uniforme a los clientes. El Session Facade va a constituir una interfaz de negocio[11].

Transfer Object

Contexto: Las aplicaciones cliente necesitan intercambiar datos con los Beans Enterprise.

Problema: Toda llamada a método hecha al objeto de servicio de negocio, ya sea a un bean de entidad o a un bean de sesión, potencialmente es una llamada remota. Así, en una aplicación de

[12] Java Namming Directory Interface, API de JEE que se encarga de los servicios de búsqueda (nombrado).

JavaBeans Enterprise (EJB) dichas llamadas remotas usan la capa de red sin importar la proximidad del cliente al bean, creando una sobrecarga en la red. Según se incrementa la utilización de estos métodos remotos, el rendimiento de la aplicación se puede degradar significativamente.

Solución: Utilizar un Transfer Object para encapsular los datos de negocio. Se utiliza una única llamada a un método para enviar y recuperar el Transfer Object. Los clientes normalmente solicitan más de un valor a un bean enterprise. Para reducir el número de llamadas remotas y evitar la sobrecarga asociada, es mejor el Transfer Object para transportar los datos desde el bean enteprise al cliente[11].

Patrones de integración

Data Access Object

Contexto: El acceso a los datos varía dependiendo de la fuente de los datos. El acceso al almacenamiento persistente, como una base de datos, varía en gran medida dependiendo del tipo de almacenamiento (bases de datos relacionales, bases de datos orientadas a objetos, ficheros planos, entre otros) y de la implementación del vendedor.

Problema: Muchas aplicaciones de la plataforma JEE en el mundo real necesitan utilizar datos persistentes en algún momento. Para muchas de ellas, este almacenamiento persistente se implementa utilizando diferentes mecanismos, y hay marcadas diferencias en los API utilizados para acceder a esos mecanismos de almacenamiento diferentes. Otras aplicaciones podrían necesitar acceder a datos que residen en sistemas diferentes. Existe una dependencia directa entre el código de la aplicación y el código de acceso a los datos, que hace difícil y tedioso migrar la aplicación de un tipo de fuente de datos a otro.

Solución: Utilizar un Data Access Object (DAO) 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. Como la interfaz expuesta por el DAO no cambia cuando cambia la implementación de la fuente de datos subyacente, este patrón permite al DAO adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio. Esencialmente, el DAO actúa como un adaptador entre el componente y la fuente de datos[11].

Dificultad de los patrones

Los patrones JEE no son triviales de implementar en aplicaciones empresariales, los desarrolladores necesitan conocer metodologías para una correcta implementación de los mismos. Una aplicación

compleja puede demandar el uso de cada uno de estos patrones, por lo que llevaría grandes esfuerzos por parte de los desarrolladores en su implementación, y por consiguiente consumo de tiempo. Además, un error en su interpretación o implementación puede desencadenar un mal diseño de la arquitectura.

1.4 Frameworks de desarrollo

Los frameworks son soluciones que implementan patrones y ayudan a desarrollar funciones dentro de las aplicaciones a un nivel superior. Tienen la funcionalidad de abstraer de complejas implementaciones y hacer el desarrollo más ágil, ya que proveen soluciones a problemas comunes, son como un conjunto de librerías que se usan en las aplicaciones. Constituyen elementos a tener en cuenta a la hora de modelar una arquitectura pues hacen que se cubran en gran medida los objetivos con los que debe cumplir la misma. Entre los frameworks más populares de JEE se encuentran los siguientes:

Struts

Cuando se afronta una aplicación seria y de gran tamaño, surgen ciertas problemáticas sobre cómo separar la lógica de negocio de la de presentación, controlar el flujo de navegación y normalizar las acciones que procesa el sistema.

Struts es un framework open source creado por Craig R. McClanahan y donado a la Apache Software Fundation en el año 2000 quien lo acogió como parte de su proyecto Jakarta[12]. Tiene como objetivo ayudar a los desarrolladores a construir aplicaciones web de una forma rápida y fácil. Struts se apoya en un grupo de tecnologías estándar de JEE como los JavaBeans, Java Servlets y JavaServer Pages (JSP)[4].

Struts se encarga de normalizar el desarrollo de la capa Vista dentro de la arquitectura MVC. También proporciona mecanismos para trabajar con la capa Controller. Pero nunca Struts se mezcla con la capa del Modelo. Esta característica principal de Struts permite separar la lógica de presentación de la lógica del negocio, con las ya consabidas ventajas que este tipo de desarrollo supone.

Este framework tiene integrado una serie de patrones de diseño que conducen a la realización de buenas prácticas de programación: Front Controller, View Helper, Dispatcher View, Service to Worker y Composite View.

Posee un servlet controlador que se encarga del manejo del flujo entre la JSPs (Java Server Pages) y otros dispositivos de la capa de presentación, apoyándose en el uso de ActionForwards y ActionMappings para control del flujo de desiciones fuera de la capa de presentación (fig 1.1). A continuación se muestran algunos elementos que brinda Struts para implementar esta arquitectura:

- Action:

  Es una parte del controlador que gestiona los cambios de estados, invoca e interactúa con la lógica de negocio. Forma parte del controlador en el modelo MVC. Implementa el patrón de diseño Front Controller[4, 12, 13].

- ActionForward:

  Se encarga de la selección de las interfaces del usuario y el redireccionamiento. Maneja el flujo de trabajo de una aplicación. Forma parte del controlador en el modelo MVC. Implementa el patrón de diseño Dispatcher View.

- ActionForm:

  Gestiona el intercambio de datos entre las Java Server Pages (JSP) y el controlador. Existe una especialización de ActionForm, el DynaActionForm, que permite su definición dinámica a partir de un fichero XML. Se ubica dentro de la vista en el modelo MVC. Implementa el patrón de diseño View Helper.

- ActionMapping:

  Encapsula la información de mapeo entre las vistas y el controlador, permite el direccionamiento de cada solicitud hasta la acción indicada como su respectiva respuesta. Se encuentra ubicado dentro del controlador en el modelo MVC. Tiene implementado el patrón Dispatcher View.

- ActionServlet:

  Recibe las solicitudes, las empaqueta y enruta hacia el manejador apropiado.

  Forma parte del controlador en el modelo MVC. Implementa el patrón de diseño Front Controller.

- Struts Tags:

  Permiten manipular los datos en las JSPs, así como realizar lógica de procesamiento sin necesidad de utilizar los molestos y no recomendados scriplets. Se ubican dentro de la vista en el modelo MVC. Tiene implementado el patrón View Helper.

- Tiles:

  Permiten usar las plantillas en Struts como mecanismo de construcción de vistas compuestas. Forma parte de la vista en el modelo MVC. Implementa el patrón de diseño Composite View [4, 12, 13].

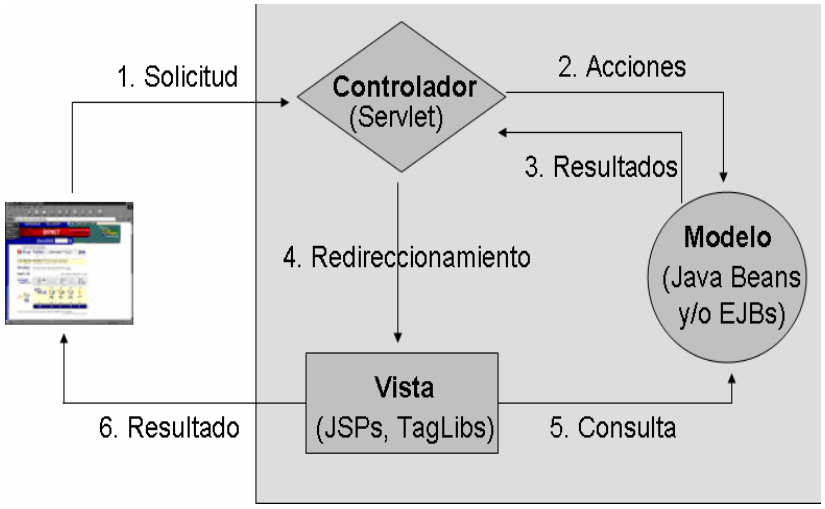


Fig 1.1 Funcionamiento de Struts

Además de todos estos elementos que constituyen el centro de la arquitectura, Struts brinda muchas otras opciones, a continuación se mencionan algunas:

- Framework Validator permite configurar todas las validaciones de los datos que se obtienen en los formularios (ActionForms), tomando los mensajes de los Resources Bundles[13] y con la opción de tratarlas en el cliente.
- El fichero (struts-config.xml) permite configurar el flujo de la aplicación, junto con los ActionMapping y ActionForward apoya el redireccionamiento del control hacia cualquier recurso. Además de controlar otras configuraciones como los Resources Bundles, Excepciones, Framework Validator, y opciones avanzadas como DynaActionForms, entre otras.
- Sistema de internacionalización de recursos, utilizando ficheros .properties (Resources Bundles), que permiten manejar varios idiomas en la aplicación haciendo uso de ellos dinámicamente.
- Sistema de manejo de excepciones que se registran en el fichero de configuración donde se direccionan las respuestas a los recursos correspondientes.
- ActionError y ActionErrors que encapsulan el tratamiento de errores conjunto con los Resources Bundles[4, 12, 13].

Las debilidades de Struts se hacen sentir en grandes aplicaciones donde el desarrollo de las páginas puede tornarse lento, puesto que este framework carece de herramientas que permitan un rápido

[13] Fichero que contiene el diccionario de los textos de la aplicación.

desarrollo de las interfaces web, además las librerías de etiquetas que trae el mismo pueden ser insuficientes para el desarrollo de complejas interfaces de usuario. Otros aspectos desfavorables con los que cuenta Struts es que las clases que manejan los formularios y ejecutan las acciones están acopladas al framework ya que heredan de las clases ActionForm y Action.

Java Server Faces (JSF)

JSF (Java Server Faces) es un estándar de JEE, además constituye un framework de desarrollo basado en el patrón MVC (Modelo Vista Controlador). Al igual que Struts, JSF pretende normalizar y estandarizar el desarrollo de aplicaciones web. Hay que tener en cuenta JSF es posterior a Struts, y por lo tanto se ha nutrido de la experiencia de este, mejorando algunas sus deficiencias. De hecho el creador de Struts (Craig R. McClanahan) también es líder de la especificación de JSF[5, 14, 15].

Este framework tiene un objetivo específico: lograr un rápido y fácil desarrollo de aplicaciones web, para esto permite a los desarrolladores pensar en términos de componentes, eventos, beans de respaldo, en lugar de solicitudes y respuestas, en otras palabras JSF abstrae múltiples complejidades del desarrollo web para que los desarrolladores se puedan enfocar en la lógica de negocio. Algunos de los servicios más importantes que aporta JSF son:

- Conversión de datos, facilita la conversión de los datos entrados como texto por el usuario en los formularios web a distintos tipos como fechas, números y otros. Permite definir nuevas reglas de conversión.
- Validación y manejo de errores, permite asociar reglas de validación a los campos, y mostrar los mensajes de error apropiados.
- Internacionalización, JSF maneja todos los aspectos de la internacionalización como la codificación de caracteres y la configuración de los ficheros de recursos.
- Desarrollo de componentes, los programadores pueden desarrollar sofisticados componentes de diseño de interfaz a la medida de sus necesidades.
- Alternativos renderizadores, por defecto los componentes de JSF son renderizados como HTML, pero facilita la producción y utilización de otros renderizadores para mostrar diferentes salidas en otros lenguajes.
- Soporte de herramientas, JSF está optimizado para ser desarrollado con herramientas automatizadas que agilicen el desarrollo de aplicaciones web[5, 14, 15].

Además de estos beneficios, JSF cuenta con otros aspectos que son claves en el desarrollo con este framework:

- El manejo de beans de respaldo los cuales son JavaBeans especializados que almacenan valores de los componentes de interfaz de usuario e implementan métodos de manejo de eventos, además pueden contener referencias de estos componentes.
- Fichero de configuracion (faces-config.xml) permite la configuración de la mayoría de los servicios del framework, desde este archivo se estructura la navegación, los beans de respaldo, la internacionalización, entre otros[5, 14, 15].

Hibernate

Trabajar con software orientado a objetos y bases de datos relacionales puede hacerle invertir mucho tiempo a los desarrolladores en los entornos actuales. Hibernate realiza el mapeo entre el mundo orientado a objetos de las aplicaciones y el mundo entidad-relación de las bases de datos en entornos Java. Es la solución ORM (Object-Relational Mapping) más popular en el mundo Java[7]. Fue liberado bajo la Lesser GNU Public License[16]. El término ORM (object/relational mapping) consiste en la técnica de realizar la transición de una representación de los datos de un modelo relacional a un modelo orientado a objetos y viceversa. Es un framework que constituye un motor de persistencia que implementa múltiples funcionalidades[7].

El lenguaje de consultas de Hibernate HQL (Hibernate Query Language), diseñado como una mínima extensión orientada a objetos de SQL, proporciona un puente entre los mundos objetual y relacional. Hibernate también permite expresar consultas utilizando SQL nativo o consultas basadas en criterios[17]. Soporta todos los sistemas gestores de bases de datos SQL y se integra de manera elegante y sin restricciones con los más populares servidores de aplicaciones J2EE y contenedores web, y por supuesto también puede utilizarse en aplicaciones standalone[7, 17].

Características clave:

- Persistencia transparente: Puede operar proporcionando persistencia de una manera transparente para el desarrollador.
- Modelo de programación natural: Soporta el paradigma de orientación a objetos de una manera natural (herencia, polimorfismo, composición y las librerías de colecciones de Java).
- Soporte para modelos de objetos con una granularidad muy fina: Permite una gran variedad de mapeos para colecciones y objetos dependientes.

- Sin necesidad de mejorar el código compilado (bytecode): No es necesaria la generación de código ni el procesamiento del bytecode en el proceso de compilación.
- Escalabilidad extrema: Hibernate posee un alto rendimiento, tiene una caché de dos niveles y puede ser usado en un clúster. Permite inicialización perezosa (lazy) de objetos y colecciones.
- Lenguaje de consultas HQL: Este lenguaje proporciona una independencia del SQL de cada base de datos, tanto para el almacenamiento de objetos como para su recuperación.
- Soporte para transacciones de aplicación: Hibernate soporta transacciones largas (aquellas que requieren la interacción con el usuario durante su ejecución).
- Generación automática de claves primarias: Soporta los diversos tipos de generación de identificadores que proporcionan los sistemas gestores de bases de datos (secuencias, columnas autoincrementales) así como generación independiente de la base de datos, incluyendo identificadores asignados por la aplicación o claves compuestas.

El centro de la arquitectura de Hibernate lo constituyen una serie de interfaces que realizan el grueso de las funcionalidades del framework, almacenar, obtener objetos persistentes y manejar las transacciones, dentro de ellas están:

- Session: Es la interfaz primaria y la más usada en las aplicaciones, es la manejadora de la persistencia, a través de ella se pueden almacenar y cargar objetos de la base de datos[7].
- SessionFactory: Es la interfaz mediante la cual se obtiene la Session. En una aplicación solo hay una SessionFactory que se crea cuando se inicializa la misma. Cada base de datos que se va a utilizar en una aplicación tiene una SessionFactory.
- Configuration: Es usada para configurar Hibernate, se utiliza para especificar la ruta de los documentos de mapeo.
- Transaction: Se utiliza para el control de las transacciones. Ayuda a mantener la portabilidad de las aplicaciones que usan Hibernate entre los diferentes tipos de entornos de ejecución y contenedores.
- Query y Criteria: Permiten la ejecución de consultas a la Base de Datos.

Además de estas interfaces, Hibernate brinda otros aspectos muy importantes y que son de gran utilidad en las aplicaciones, por ejemplo:

- Type: Interfaz fundamental y muy poderosa, posibilita el mapeo de tipos Java a columnas en bases de datos incluso a varias columnas, incluye tipos como Calendar, byte [ ], y permite

además definir tipos personalizados de datos. Las interfaces UserType y CompositeUserType permiten definir nuevos tipos de datos como Persona, Dirección, Nombre. El soporte de tipos de datos personalizados es considerado el comportamiento central de Hibernate.

- Callback: Interfaz que gestiona el ciclo de vida de los objetos, recibe notificación cuando un objeto es cargado, salvado o eliminado. Las interfaces Lifecycle, Validatable les permiten a los objetos persistentes reaccionar a los eventos relativos a su propio ciclo de vida.
- HQL: Un dialecto orientado a objetos fácil de manejar y familiar a SQL que aunque no es un lenguaje de manipulación de datos como este, puesto que es usado solamente para extraer datos no para borrar, insertar o actualizar, permite realizar complejas consultas.

Hibernate presenta un mecanismo persistente totalmente transparente, los objetos desconocen su capacidad de persistir, no necesitan extender comportamientos especiales de otra clase o interfaz, no necesitan un contenedor de aplicaciones, pueden tener lógica de aplicación, son de fácil manejo y pueden ser usados para transportar los datos a cualquier capa de la aplicación. Estas son algunas de las razones por las cuales este framework ha ganado tanto en popularidad y preferencia por encima de los EJBs de entidad de SUN y otras herramientas objeto/relacional.

Este framework es una clara implementación del patrón DAO, pues crea una capa separada que se ocupa del acceso a datos con total independencia del gestor y la base de datos, dando la oportunidad de trabajar con varios gestores y bases de datos dentro de la misma aplicación sin que esto cree ningún conflicto en el modelo objetual. Hibernate está basado en JDBC, y su filosofía es tener todo el código de acceso a datos del lado de la aplicación, esto puede afectar el rendimiento de algunas consultas muy complicadas como reportes especializados u otros, donde lo más recomendable sería utilizar procedimientos almacenados (fig 1.2).

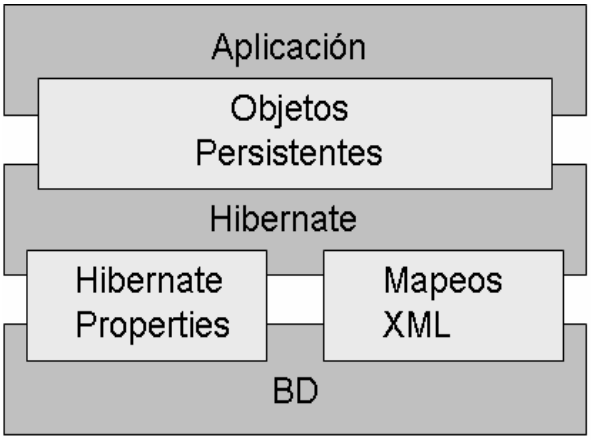


Fig 1.2 Framework Hibernate

Spring

Recientemente el mundo de Java fue testigo del cambio de las llamadas arquitecturas de pesos pesados basadas en Enterprise JavaBeans (EJBs) por frameworks de peso ligero como Spring. Rod Jonson es el inventor de Spring. Este framework se hizo open source en febrero del 2003.

Beneficios arquitectónicos de Spring:

- Spring puede organizar eficazmente la capa intermedia de cualquier aplicación ya sea web o standalone[18].
- Facilita el desarrollo de buenas prácticas, reduce el costo de programación en una aplicación.
- Las aplicaciones con Spring son fáciles de testear.
- Puede hacer el uso de la implementación de Enterprise JavaBeans (EJBs) una opción, siendo antes esta implementación determinante para la arquitectura de una aplicación. Ayuda a resolver problemas sin usar EJBs, provee una alternativa a ellos que es útil para muchas aplicaciones.
- Permite la integración entre varios frameworks, constituyendo un eslabón central en la arquitectura de las aplicaciones.
- Permite resolución de problemas típicos de aplicaciones empresariales de manera declarativa, estos servicios como manejo de transacciones, seguridad, entre otros, sólo eran brindados por servidores JEE.

El centro de Spring está basado en el principio de Inversion of Control (IoC) o inyección de dependencias. Esta técnica hace externa la creación y el manejo de las dependencias de los componentes, logrando mayor limpieza y claridad en el código pues provee, en tiempo de ejecución, de todas las instancias de las clases de la aplicación y las dependencias que estas necesitan[6, 19]. Con este mecanismo se obtienen los siguientes resultados:

- Reduce potencialmente el código de enlace entre los diferentes componentes de la aplicación.
- Externaliza las dependencias, lo cual permite la reconfiguración de las mismas si necesidad de compilar todo el código.
- Manejo de las dependencias en un solo lugar, facilitando la configuración de las mismas y disminuyendo el margen de errores.
- Fomenta un buen diseño de la aplicación. Todo el diseño de la inyección de instancias está basado en interfaces, y las clases que las implementan son creadas por el contenedor de IoC.

Además de IoC ser una de las características más relevantes de Spring lo constituye el soporte a la programación orientada a aspectos (AOP).

La programación orientada a aspectos es una de las tecnologías del momento en el mundo de Java, AOP permite efectuar procesamientos comunes en muchas partes de la aplicación (crosscutting) implementándolos en un solo lugar. En Spring AOP es usado para muchos propósitos como el manejo de transacciones, manejo de trazas, seguridad, y permite hacerlo en muchos casos de forma declarativa[6, 19]. Otros recursos que incorpora este framework:

- Excelente integración con una selección de herramientas de acceso a datos, incluyendo JDBC, Hibernate, iBATIS y Java Data Objects (JDO). Cuando se utiliza el API de Spring para el acceso a datos utilizando cualquiera de estas herramientas se obtienen las ventajas del excelente manejo de transacciones del mismo.
- Una capa de abstracción para el manejo de transacciones permitiendo el control de las mismas tanto programática como declarativamente.
- Un amplio conjunto de clases que permiten la creación de aplicaciones web. Además de proveer total integración con frameworks de desarrollo como Struts y JSF, Spring contiene su propio framework MVC, con un conjunto de tecnologías para las vistas como JSP, entre otros.
- Incluye un módulo que se encarga del manejo de los flujos de la páginas en la aplicaciones web, que constituye un poderoso controlador para ser usado cuando las aplicaciones demandan complejos controles de navegación que llevan al usuario por una serie de pasos que

constituyen una larga transacción. Este módulo puede ser integrado con otras tecnologías web como Struts y JSF.

- Un módulo que abarca el soporte para la internacionalización de mensajes, servicios de empresariales como e-mail, acceso a repositorios JNDI, integración con EJB, accesos remotos, entre otros[6, 18, 19].

Spring constituye una buena opción para las aplicaciones pues constituye un contenedor ligero que gestiona los objetos de las mismas, manteniendo el código limpio y claro, favoreciendo a un bajo acoplamiento. Los objetos gestionados no necesitan implementar o extender funcionalidades especiales, por cuanto es un mecanismo totalmente transparente. Además, los servicios que brinda Spring son piedra angular en cualquier aplicación y abstraen al programador de codificar cientos e incluso miles de líneas, esto hace que aumente la productividad en los proyectos actuales que lo utilicen.

Acegi

Acegi Security System es un framework creado por Ben Alex e íntimamente ligado al proyecto Spring. La arquitectura de Acegi está fuertemente basada en interfaces y en patrones de diseño, proporcionando las implementaciones más comúnmente utilizadas. Acegi provee seguridad declarativa para las aplicaciones basadas en Spring, de esta manera libera al código de la aplicación de implementaciones de seguridad[6].

Acegi está compuesto por cuatro componentes (fig 1.3):

- Security Interceptor: Previene el acceso a los recursos seguros de la aplicación.
- Authentication Manager: Es el encargado del proceso de autenticación.
- Access Decision Manager: Es el encargado de decidir si un usuario puede acceder a un recurso determinado.
- Run-As Manager: Permite remplazar los permisos para recursos más profundos en la aplicación.

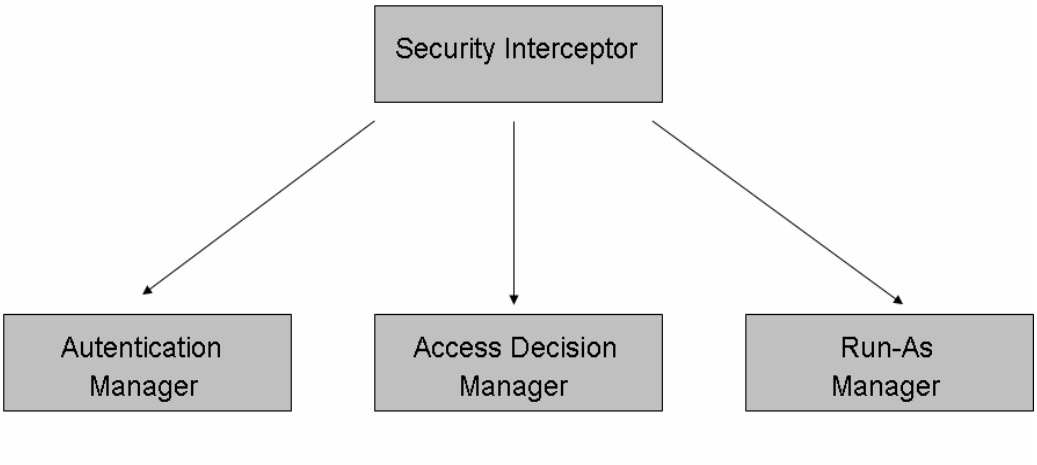


Fig 1.3 Componentes de Acegi

En las aplicaciones web Acegi usa filtros que interceptan las solicitudes, utilizando un único mecanismo para declararlos e inyectarlos con sus dependencias utilizando IoC, creando un desacople total e independencia de la aplicación. Además permite implementar la seguridad a bajo nivel controlando las invocaciones de métodos, usando AOP los objetos proxies de Acegi aplican aspectos que aseguran que el usuario que llama a los métodos contenga los permisos requeridos[6].

Otras ventajas que brinda Acegi:

- Provee configuración de protección del canal, permitiendo redireccionar hacia un canal adecuado (HTTP no seguro o HTTPS seguro) de acuerdo a la solicitud.
- Soporta autenticación HTTP BASIC, esta autenticación es adecuada para aquellas aplicaciones que prefieren una simple ventana de login del navegador en lugar de un formulario de login.
- Librería de etiquetas, proporciona una librería de etiquetas que puede ser utilizada en JSPs para garantizar que el contenido protegido como enlaces y mensajes sean únicamente mostrados a usuarios que posean los permisos adecuados.
- Distintos métodos de almacenamiento de la información de autenticación.
- Incluye la posibilidad de obtener los usuarios y permisos utilizando ficheros XML, fuentes de datos JDBC o implementando un interfaz DAO para obtener la información de cualquier otro lugar.

Acegi es una buena opción para el manejo de la seguridad en las aplicaciones pues permite implementar la seguridad desde nivel de filtros de solicitudes hasta nivel de métodos, sin tener que codificar líneas extras, pues realiza esto utilizando AOP (Programación Orientada a aspectos). De una manera poco intrusiva, realizando configuraciones en un fichero XML, se pueden definir las políticas de seguridad de la aplicación.

1.5 Servidores de aplicaciones y contenedores web

Servidores de aplicaciones

Los servidores de aplicaciones son el corazón de la plataforma JEE. Forman parte en las arquitecturas empresariales pues en ellos residen todos los componentes, ya sean objetos distribuidos accesibles remotamente, páginas web, o incluso aplicaciones completas accesibles utilizando Java Web Start. En un servidor de aplicaciones existe un contenedor de servlets, un contenedor de EJBs, sistemas de mensajería, y un gran número de herramientas.

La pieza más importante dentro de un servidor de aplicaciones es el contenedor de EJBs. Los contenedores de EJBs son servidores que se encargan de controlar los Enterprise JavaBeans (EJB), esto es muy importante ya que se automatizan tareas como la gestión del ciclo de vida, la gestión de las transacciones, la gestión de persistencia. Los desarrolladores, de este modo, no tienen que centrarse en crear servicios de bajo nivel y pueden centrarse en la creación de lógica de negocio empresarial[3].

En la actualidad el mercado de los servidores de aplicaciones es uno de los más activos y entre las empresas más destacadas se encuentran BEA WebLogic e IBM WebSphere, la única dificultad radica en que sus productos tienen un coste por licencia algo elevado.

Como una alternativa al uso de estos servidores propietarios surge JBoss, el cual es el servidor de aplicaciones libre por excelencia. Este servidor se encuentra avalado por un grupo de desarrollo formado por arquitectos con gran experiencia[20].

Los servidores de aplicación son mucho más grandes que los contenedores web, pero resultan más difíciles de configurar y de administrar.

Contenedores web

Son servidores, escritos normalmente en Java. Forman parte de las arquitecturas empresariales pues su funcionalidad radica en recibir peticiones HTTP y ejecutar las clases de Java (Servlets y JSPs). Muchos de los contenedores web incluyen un pequeño servidor web que les permite además de procesar las

clases de Java, a su vez utilizar recursos tales como páginas HTML para dar respuesta, formando lo que se llama una aplicación web. Pueden ser integrados con los servidores más conocidos como Apache, de modo que estos últimos se encarguen de realizar el procesado del contenido estático (HTML), labor para la que están optimizados.

Apache Tomcat es un proyecto de software libre creado a partir de código donado por SUN Microsystems a la Apache Software Foundation. Es el contenedor web que más se está utilizando en servidores de aplicaciones. Además de utilizarse como contenedor web de servlets y JSP, también hace función de servidor web[20].

La desventaja del uso de contenedores web radica en que estos no ofrecen servicios de manejo de transacciones y seguridad a las aplicaciones como caso de los contenedores de EJB en servidores de aplicación, estas funcionalidades resultan muy importantes pues libera a los desarrolladores de su implementación. Además no soportan los API EJB ni JMS (Servicio de mensajería) que en ocasiones son requeridos por las aplicaciones. En cambio resultan sencillos de configurar y administrar logrando funcionalidades importantes como clusterización, aportando escalabilidad al sistema o manejo de pool de conexiones que brinda gran eficiencia, pues crea una reserva de conexiones para las fuentes de datos que se puede utilizar cada vez que se desee acceder a las mismas sin necesidad de crear las conexiones en el momento.

Conclusiones

A través de este capítulo se ha podido llegar a las siguientes conclusiones:

- Existe una gran diversidad de elementos a tener en cuenta y que intervienen en el desarrollo de una arquitectura JEE, cada uno de ellos trata de exponer su genialidad.
- Los desarrolladores, debido a la falta de experiencia, pueden verse agobiados a la hora de tomar decisiones en la adopción de una arquitectura empresarial, los mismos pueden mostrarse indecisos y puede hacérseles difícil escoger las tecnologías a utilizar dadas sus necesidades.
- La gran disponibilidad de las tecnologías de la plataforma puede hacer que los desarrolladores escojan combinaciones poco factibles.

Capítulo 2: Análisis de modelos arquitectónicos.

2.1 Arquitecturas empresariales

Una posible definición abstracta de arquitectura empresarial sería: "El estudio de sistemas empresariales complejos desde el punto de vista de su estructura". Un arquitecto empresarial ha de ser capaz de estudiar un problema en concreto y de escoger una serie de componentes con los que modelar la arquitectura más adecuada para el problema en cuestión. Dichos componentes pueden ser servidores de aplicaciones, contenedores web, servidores de mensajería, etc. El arquitecto ha de ser capaz de establecer el modo de trabajo de dichos componentes, las herramientas utilizadas y las relaciones existentes entre los mismos[8].

La labor más complicada y con mayor responsabilidad para un arquitecto empresarial es la elección de la plataforma empresarial sobre la que 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 ha de 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.

La importancia de una plataforma empresarial es que todos estos componentes se 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 los 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.

Luego de seleccionar la plataforma de desarrollo, el arquitecto estudia cada uno de sus componentes, librerías, herramientas que ayuden al desarrollo de la aplicación y se apoya en ellos para conformar la arquitectura.

Entre los objetivos que debe cumplir una arquitectura empresarial se encuentran:

- Ser robusta:

  El software de empresa es importante en una organización. Sus usuarios esperan que este sea confiable y libre de errores. Los desarrolladores deben elegir una arquitectura que permita construir soluciones robustas y escribir códigos de calidad.
- Ser escalable y responder a las exigencias de rendimiento que plantea el usuario:

La arquitectura de empresa debe mostrar suficiente escalabilidad para que una aplicación soporte un potencial incremento de la carga dado por un hardware apropiado. La escalabilidad es particularmente importante para aplicaciones de Internet, en las cuales es imposible predecir el número de usuarios.

- Tomar ventaja de los principios de diseño orientado a objetos:
  Los principios de diseño orientado a objetos proveen beneficios para los sistemas complejos. Las buenas prácticas de diseño orientado a objetos son promovidas por el uso de patrones de diseño que son soluciones comunes a problemas.
- Evitar una complejidad innecesaria:
  Se debe ser cauteloso y no aplicar una arquitectura excesivamente compleja porque puede imposibilitar que la aplicación trabaje. Debido al gran conjunto de componentes en oferta es tentador para los ingenieros de soluciones utilizar modelos arquitectónicos complejos para requerimientos del negocio irrelevantes que realmente no lo necesiten. Esta complejidad solo trae como resultado aumento en el ciclo de vida del software y causa serios problemas.
- Ser mantenible y extensible:
  El mantenimiento es la fase más costosa del ciclo de vida del software. Las aplicaciones son una parte clave de una organización y deben permitir ser arregladas para las nuevas necesidades del negocio de la empresa. El mantenimiento y extensión dependen mayormente de un diseño limpio. Cada componente de la aplicación debe tener clara su responsabilidad.
- Ser fácil de probar:
  Las pruebas son una actividad esencial durante el ciclo de vida del software. Se deben tener en cuenta decisiones de diseño que sean fácil de testear.
- Ser reusable:
  Es importante fomentar la reusabilidad a fin de que la duplicación del código sea eliminada. La reusabilidad resulta una buena práctica de diseño orientado a objetos.
- Ser portable:
  Es de gran importancia para las aplicaciones su portabilidad entre recursos como bases de datos y servidores de aplicaciones, al no lograrse este objetivo podría verse afectado el código de la aplicación al introducirle nuevos cambios y el ciclo de vida del software tendría dificultades, es vital escoger una arquitectura que lo permita[9].

2.2 Arquitecturas distribuidas y no distribuidas

Entre los tipos de arquitecturas empresariales se encuentran las distribuidas y las no distribuidas. Las arquitecturas distribuidas surgen para dar solución a las necesidades de repartir el volumen de información y las de compartir recursos, ya sea en forma de software o hardware. La utilización de arquitecturas distribuidas permite que los desarrolladores no estén sujetos a las restricciones de la máquina, sino que puedan utilizar los recursos de toda una red[9].

Las arquitecturas no distribuidas permiten que todos los componentes se ejecuten en un solo servidor. Esto las hace simples y eficientes pues las llamadas a las funcionalidades pueden hacerse de forma local pero es limitada la flexibilidad de su despliegue[9].

Una arquitectura distribuida es más compleja, difícil de implementar, probar y mantener que una arquitectura no distribuida al encontrarse dispersos en la red los componentes que encapsulan las funcionalidades. Cuando se usan arquitecturas distribuidas debe hacerse cuidadosamente pues las mismas podrían exhibir sus fallas de funcionamiento. Las invocaciones remotas son muchas veces más lentas que invocaciones locales. La habilidad de las arquitecturas distribuidas para desplegarse en cualquier servidor físico es muy importante para el balance de carga. Las arquitecturas distribuidas pueden ser más robustas y escalables en algunos casos y pueden ser la única forma de resolver los requerimientos del negocio[9].

Es posible lograr en una arquitectura no distribuida los objetivos con los que debe cumplir una arquitectura empresarial utilizando un buen diseño y combinando las tecnologías apropiadas. De esta forma se alcanzarían buenos resultados con menos esfuerzo.

2.3 Patrones arquitectónicos

Los estilos arquitectónicos de las aplicaciones empresariales contemporáneas pueden dividirse en arquitecturas de nivel 2, nivel 3 y nivel n. En una aplicación tradicional de 2 niveles, la carga de procesamiento es facilitada al PC cliente mientras que el servidor actúa simplemente como controlador del tráfico entre la aplicación y los datos (fig 2.1). Como resultado, el rendimiento de la aplicación no sólo sufre debido a los recursos limitados de la PC sino que el tráfico de la red también tiende a aumentar. Cuando la aplicación completa es procesada en una PC, la aplicación es forzada a realizar múltiples peticiones de datos antes incluso de presentar algo al usuario. Estas múltiples peticiones de bases de datos pueden sobrecargar la red[3].

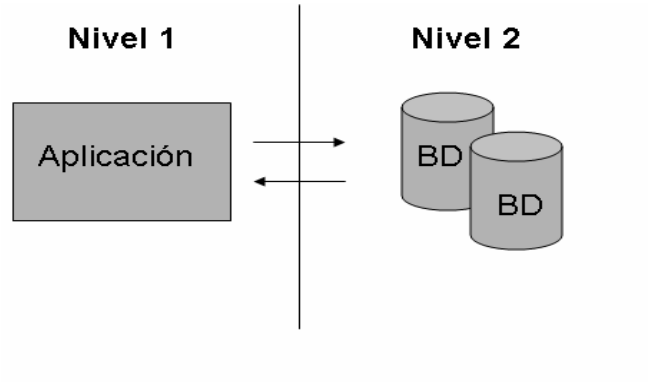


Fig 2.1 Arquitectura de 2 niveles

Otro problema típico relacionado con el enfoque de 2 niveles es el del mantenimiento. Incluso el menor cambio realizado a una aplicación puede conllevar una completa alteración en la base de usuario. Aunque sea posible automatizar el proceso, todavía debe enfrentarse a la actualización de cada instalación de cliente. Es más, algunos usuarios puede que no estén preparados para una alteración total y posiblemente ignoren los cambios mientras que otro grupo puede que insista en realizar los cambios de inmediato. Esto puede provocar que diferentes instalaciones de cliente utilicen diferentes versiones de la aplicación[3].

Para enfrentarse a estos temas, la comunidad de software desarrolló la noción de una arquitectura de 3 niveles. Una aplicación se divide en tres capas lógicas distintas, cada una de ellas con un grupo de interfaces perfectamente definidas. La primera capa se denomina capa de presentación y normalmente consiste en una interfaz grafica de usuario. La capa intermedia, o capa de empresa, consiste en la aplicación o lógica de empresa, y la tercera capa, la capa de datos, contiene los datos necesarios para la aplicación. La capa intermedia (lógica de aplicación) es básicamente el código al que recurre el usuario (a través de la capa de presentación) para recuperar los datos deseados. La capa de presentación recibe entonces los datos y los formatea para su presentación. Esta separación entre la lógica de aplicación de la interfaz de usuario añade una enorme flexibilidad al diseño de la aplicación. Pueden construirse y desplegarse múltiples interfaces de usuario sin cambiar en absoluto la lógica de aplicación, siempre que la lógica de aplicación presente una interfaz claramente definida a la capa de presentación[3].

La tercera capa contiene los datos necesarios para la aplicación. Estos datos consisten en cualquier fuente de información, incluido una base de datos de empresa o un conjunto de documentos XML (fig 2.2).

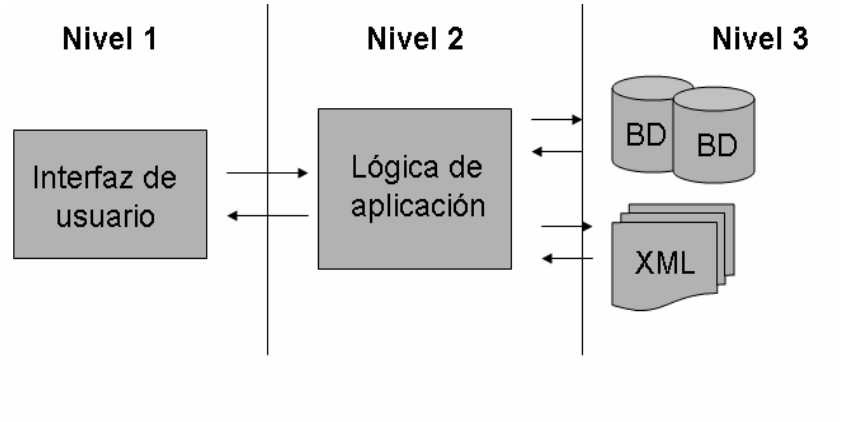


Fig 2.2 Arquitectura de 3 niveles.

Una arquitectura de n niveles se descompone en las siguientes partes:

- Una interfaz de usuario que maneja la interacción del usuario con la aplicación. Ésta puede ser un navegador web ejecutado mediante un cortafuego, o incluso un dispositivo inalámbrico.
- Una lógica de presentación que define lo que muestra la interfaz de usuario y cómo son gestionadas las demandas del usuario.
- Una lógica de empresa que modele las reglas de empresa de la aplicación, a menudo a través de la interacción con los datos de la aplicación.
- Servicios de infraestructura que proporcionen la funcionalidad adicional requerida por los componentes de la aplicación, tales como mensajería y apoyo transaccional.
- La capa de datos donde residen los datos de la empresa(fig 2.3)[3].

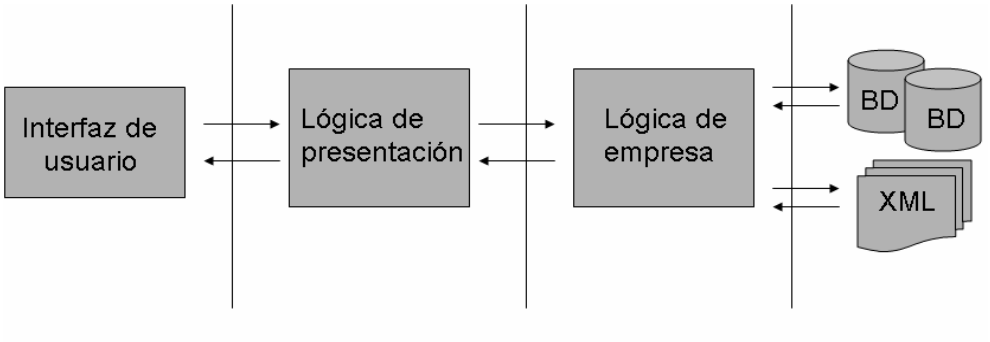


Fig 2.3 Arquitectura de n niveles.

Las aplicaciones basadas en esta arquitectura emplean esencialmente el patrón Modelo-Vista-Controlador (MVC), lográndose así un desacople de manera que los datos (el modelo), la presentación de la información (la vista) y la lógica de aplicación (el controlador) sean independientes (fig 2.4).

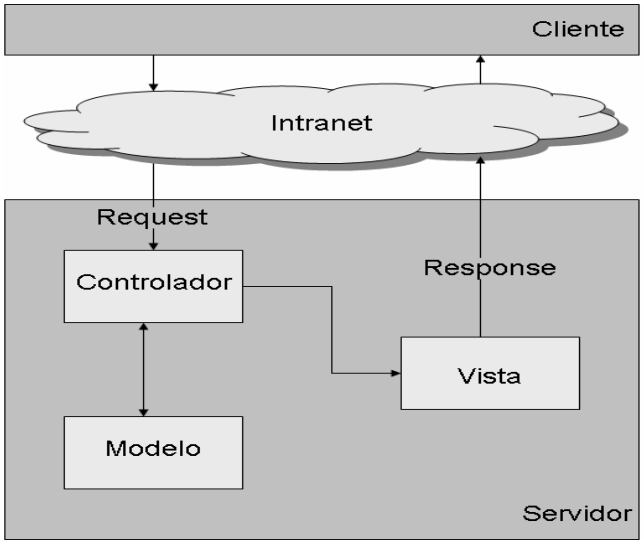


Fig 2.4 Patrón Modelo-Vista-Controlador

2.4 Modelos arquitectónicos en JEE

Con el desarrollo de la plataforma y el crecimiento de aplicaciones desarrolladas en la misma, se han extendido modelos arquitectónicos que han sido los más populares y usados en los proyectos. Estas arquitecturas están basadas en capas (Interfaz de usuario, Intermedia y Datos) y han servido de guía para el desarrollo de aplicaciones JEE. A continuación se analizan algunos de estos modelos.

2.4.1 Aplicaciones distribuidas

JEE provee un excelente soporte para la implementación de aplicaciones distribuidas. Los componentes de una aplicación distribuida pueden estar desplegados en múltiples máquinas virtuales ejecutándose sobre uno o varios servidores físicos. Estas aplicaciones están basadas en el uso de EJB con interfaces remotas, puesto que son los componentes que propone JEE para el desarrollo distribuido [9].

2.4.1.1 Aplicación distribuida con EJB

Esta es la clásica arquitectura JEE, donde la capa media está esparcida por diferentes máquinas virtuales. Es una arquitectura compleja que utiliza RMI para acceder a los componentes dispersos en la red. Todos los objetos que se intercambian deben ser serializados, estas llamadas son más lentas que las llamadas

locales por lo que se afecta el rendimiento de la aplicación. Los programadores se ven obligados a implementar una serie de patrones con el objetivo de amenizar las llamadas remotas. Los EJB son los encargados de implementar la lógica de la aplicación. Estas aplicaciones requieren de servidores de aplicaciones para poder ejecutar los EJB, y se benefician de los servicios del contenedor EJB, como el manejo de las transacciones y la seguridad [9] (fig 2.5).

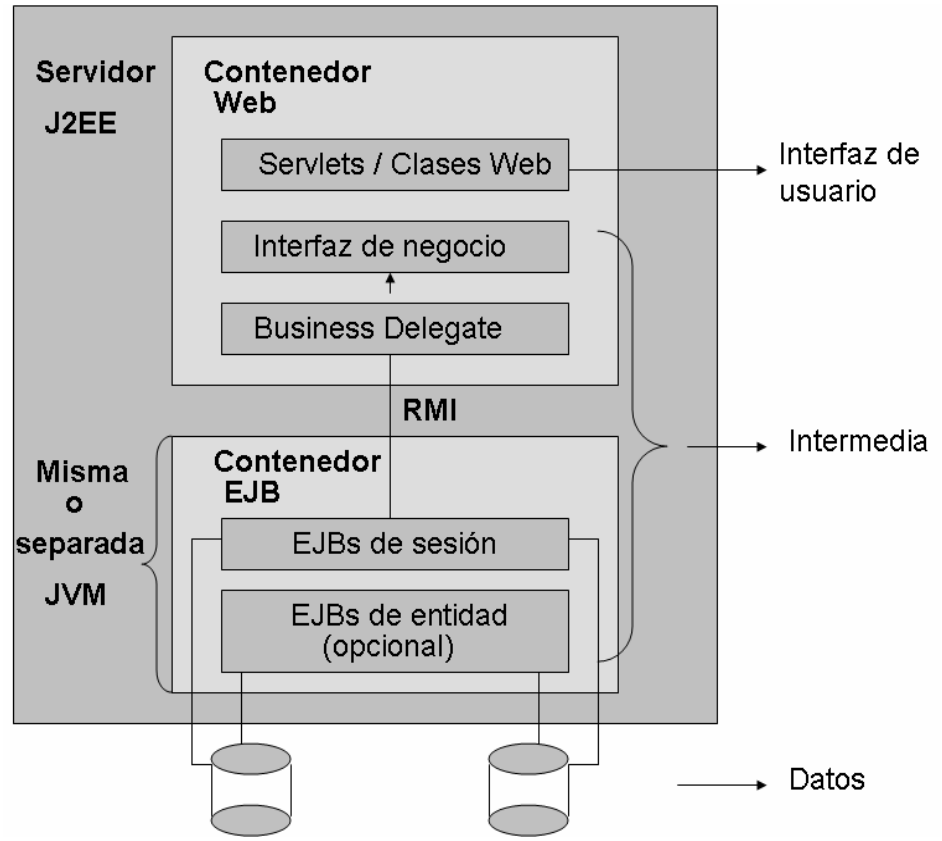


Fig 2.5 Modelo arquitectónico para aplicaciones distribuidas usando EJBs

Interfaz de usuario

La capa de interfaz de usuario en este modelo es implementada usando los APIs JSP y servlet, haciendo uso de los patrones de diseño para esta capa como: Front Controller, View Helper, Composite View, entre otros. Se utiliza un servlet como un controlador frontal que intercepte todas las solicitudes del cliente y llame a los métodos correspondientes, reciba los resultados de los mismos y muestre las páginas correspondientes. Las páginas JSP son las encargadas de mostrar las vistas apoyándose en las librerías de etiquetas y en java beans. Además, para registrar la autenticación de usuarios, asignar formato a datos

u otras tareas se pueden utilizar filtros, los cuales se aplican a los objetos solicitud (request) o respuesta (response) en cada una de sus salidas y entradas al contenedor web[9].

Intermedia

Lo más apropiado es dividir esta capa en dos, separando el procesamiento referente al negocio de la lógica de acceso a datos, creando una capa de abstracción que se encargue del intercambio de datos ya sea a una base de datos o un fichero, y exponer una interfaz con los métodos necesarios para que las clases de negocio realicen su tarea (patrón DAO).

Esta capa posee una interfaz de negocio situada delante de un Business Delegate que va a actuar como una abstracción de la implementación del negocio, el Business Delegate es el encargado de invocar al EJB indicado y para obtener la referencia a este utilizará un Service Locator.

Los EJB se encuentran distribuidos en la red, a los cuales se accede con llamadas remotas. Cada uno de estos componentes realizará una parte de la lógica de negocio. Desde estos componentes se accederá a la capa de acceso a datos que puede ser implementada utilizando EJB de entidad, JDBC u otro mecanismo de persistencia[9].

Datos

Esta capa puede ser representada por cualquier sistema gestor de base de datos como Oracle, SQL Server, MySQL, o por algún fichero de almacenamiento de texto, XML, entre otros[9].

Flujo de una solicitud

Cuando el usuario escribe la url en el navegador, el contenedor web recibe una solicitud la cual es tomada por el servlet controlador, no sin antes haber pasado por el filtro (si es que se decide la utilización de filtros, y se configura para que intercepte dicha url), el controlador dada la solicitud la direcciona bien puede ser directamente para una página JSP o para algún método de alguna clase que intervenga en la lógica de presentación con el objetivo de realizar alguna acción llamando a algún método de la interfaz de negocio, el Bussines Delegate obtiene la referencia al EJB remoto indicado utilizando para ello un Service Locator que realiza toda la lógica de localización de componentes. En este caso devolverá un Session Facade implementado como un EJB de sesión que expone una fachada para la interacción con la lógica de negocio presente en uno o varios componentes EJB, estos van a realizar el procesamiento, utilizando para el intercambio de datos o bien EJBs de entidad o una capa de abstracción DAO (fig 2.6).

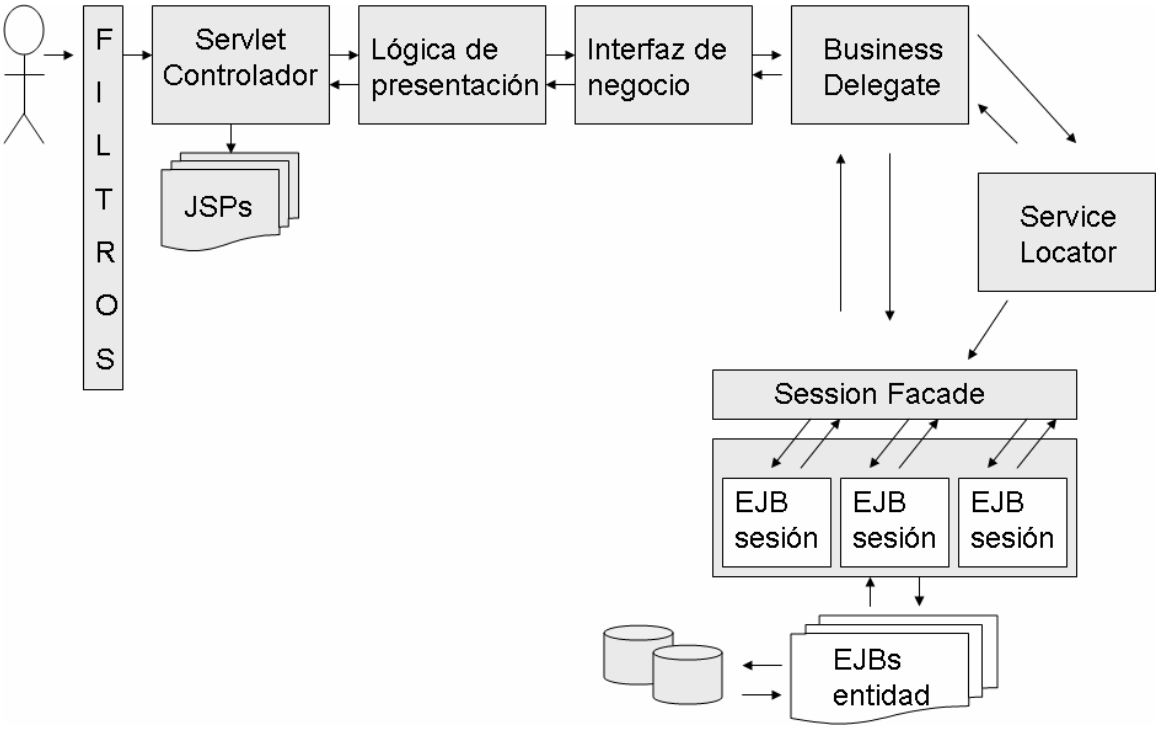


Fig 2.6 Flujo de la solicitud en una aplicación distribuida usando EJBs

Ventajas

- Esta arquitectura soporta cualquier cliente JEE puesto que la capa media se encuentra compartida.
- Permite distribuir componentes a través de diferentes servidores físicos, logrando mayor escalabilidad.
- Se obtienen los beneficios del contenedor EJB (manejo de transacciones declarativamente, seguridad, etc.)

Desventajas

- Es una arquitectura muy compleja, propensa a errores, difícil de implementar y de probar pues los componentes se encuentran distribuidos. También es difícil de desplegar.
- Afecta el rendimiento de la aplicación puesto que las llamadas remotas son mucho más costosas.
- El manejo de excepciones es muy complejo en aplicaciones distribuidas.
- Todos los componentes de negocio deben ejecutarse dentro de un contenedor EJB, lo cual ata a esta arquitectura al uso del mismo.

- La configuración y administración de los servidores de aplicaciones es más compleja que la de un simple servidor web con un contenedor de servlet.
- Se necesita hacer uso de las buenas prácticas y la utilización de patrones de diseño para lograr la separación de papeles y que esta arquitectura sea mantenible y extensible.

Aunque JEE presenta un excelente soporte para ellas, esto no quiere decir que todas las aplicaciones en esta plataforma tengan que ser distribuidas, ni que sea el único camino para lograr que sean robustas, escalables. Muchas arquitecturas JEE que usan interfaces remotas tienden a ser desplegadas con todos sus componentes en el mismo servidor físico para evitar los costos de rendimiento que traen las llamadas remotas.

2.4.2 Aplicaciones no distribuidas

Las arquitecturas no distribuidas son las más comunes en las aplicaciones JEE hoy en día pues son soluciones menos complejas y a la vez pueden llegar a ser robustas y escalables[9].

2.4.2.1 Aplicación web sin EJB

Muchas aplicaciones web pueden ser construidas sin usar EJB, en estas soluciones la lógica de negocio es implementada por clases comunes. En este caso se expone una arquitectura que se basa en el uso de los APIs de JEE (fig 2.7).

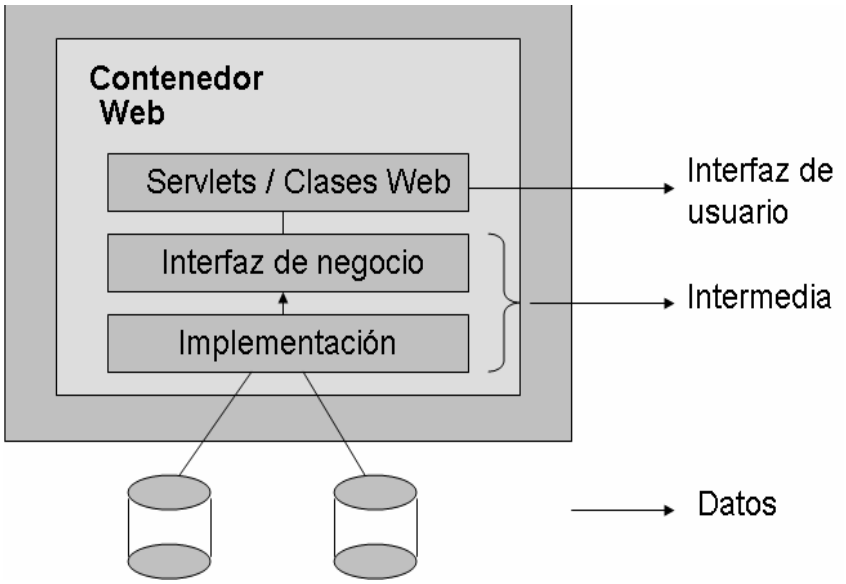


Fig 2.7 Modelo arquitectónico para aplicaciones no distribuidas sin usar EJB

Interfaz de usuario

La capa de interfaz de usuario en este modelo es implementada usando los APIs JSP y servlet. Se utiliza un servlet como un controlador frontal que intercepte todas las solicitudes del cliente y llame a los métodos correspondientes, reciba los resultados de los mismos y muestre las páginas correspondientes. Las páginas JSP son las encargadas de mostrar las vistas apoyándose en las librerías de etiquetas y en Java Beans. Además, para registrar la autenticación de usuarios, asignar formato a datos u otras tareas se pueden utilizar filtros, los cuales se aplican a los objetos solicitud (request) o respuesta (response) en cada una de sus salidas y entradas al contenedor web[9].

Intermedia

La capa media se implementa usando clases java, la interfaz de usuario se comunica con esta capa mediante una interfaz donde se exponen las funcionalidades necesarias, ocultando los detalles de la implementación de las mismas. Desde aquí se va a interactuar con la capa de acceso datos implementada utilizando el API JDBC mediante el cual y utilizando consultas SQL se realiza el intercambio con la base de datos[9].

Datos

Esta capa puede ser representada por cualquier sistema gestor de base de datos como Oracle, SQL Server, MySQL, o por algún fichero de almacenamiento de texto, XML, entre otros[9].

Flujo de una solicitud

Cuando el usuario escribe la url en el navegador el contenedor web recibe una solicitud la cual es tomada por el servlet controlador no sin antes haber pasado por el filtro (si es que se decide la utilización de filtros, y se configura para que intercepte dicha url), el controlador dada la solicitud la direcciona bien puede ser directamente para una página JSP o para algún método de alguna clase que intervenga en la lógica de presentación con el objetivo de realizar alguna acción llamando a algún método de la interfaz de negocio si es necesario, y la implementación de la capa de negocio interactuando a su vez con la interfaz de acceso a datos y esta con la capa de datos. Así es como se comporta el flujo de una solicitud en cada uno de los componentes de esta arquitectura (fig 2.8).

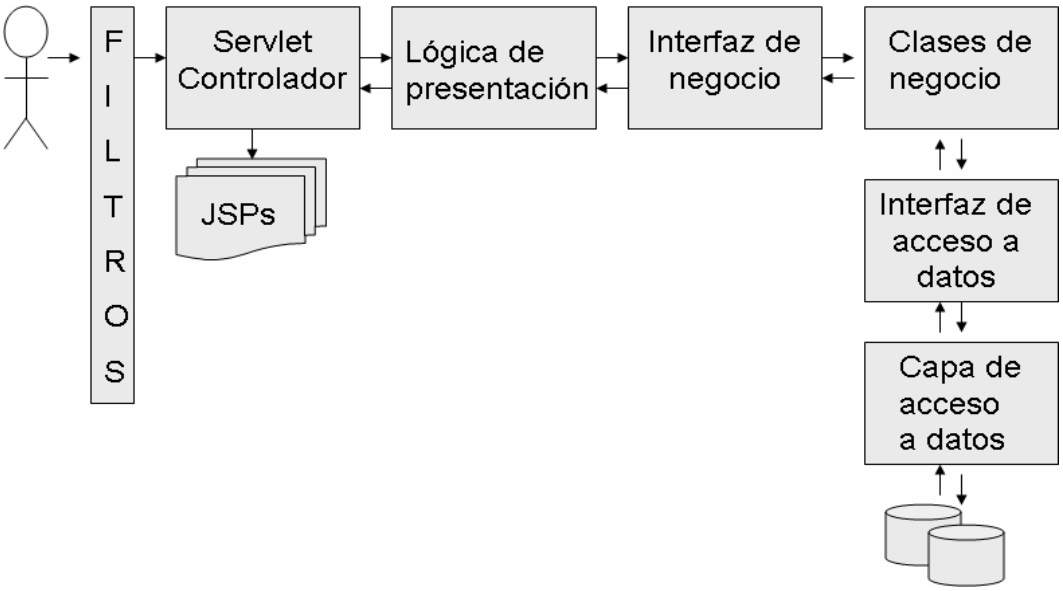


Fig 2.8 Flujo de la solicitud en una aplicación no distribuida sin usar EJB

Ventajas

- Estas aplicaciones se ejecutan dentro de un contenedor web, no necesitan de un contenedor EJB, por lo que son más sencillas de configurar y administrar.
- Son más rápidas, el contenedor web tiene menos carga, y no se realizan invocaciones remotas.
- Son más fáciles de probar, es posible probar directamente contra la interfaz de negocio sin utilizar la presentación web.
- Fácil de implementar, acceder a las clases java es mucho más sencillo que acceder a EJB.
- Pueden ser más robustas y menos propensas a errores que las arquitecturas distribuidas que usan componentes EJB.

Desventajas

- Esta arquitectura sólo soporta interfaz web, no soporta aplicaciones de escritorio como clientes.
- Toda la aplicación se ejecuta dentro de una misma máquina virtual, impidiendo la distribución de componentes en diferentes servidores.
- Las transacciones son tratadas en el código de la aplicación, a diferencia del soporte de transacciones manejadas por el contenedor EJB.

2.4.2.2 Aplicación web con EJB

Los EJBs son vistos con frecuencia como el núcleo de la plataforma JEE. Esto constituye un error, los EJBs deben ser una de las opciones que brinda la plataforma. Ellos son ideales para resolver algunos problemas pero sin embargo agregan altos costos en muchas aplicaciones[9].

En las aplicaciones web no distribuidas pueden utilizarse componentes EJB para implementar la capa de negocio. Como todos los componentes de la aplicación se ejecutan en una máquina virtual de Java se utilizan EJB locales, o sea no necesitan ser invocados remotamente, lo que simplifica su uso y a la vez se ganan los servicios del contenedor EJB, además de proveer a las aplicaciones de una completa infraestructura[9] (fig 2.9).

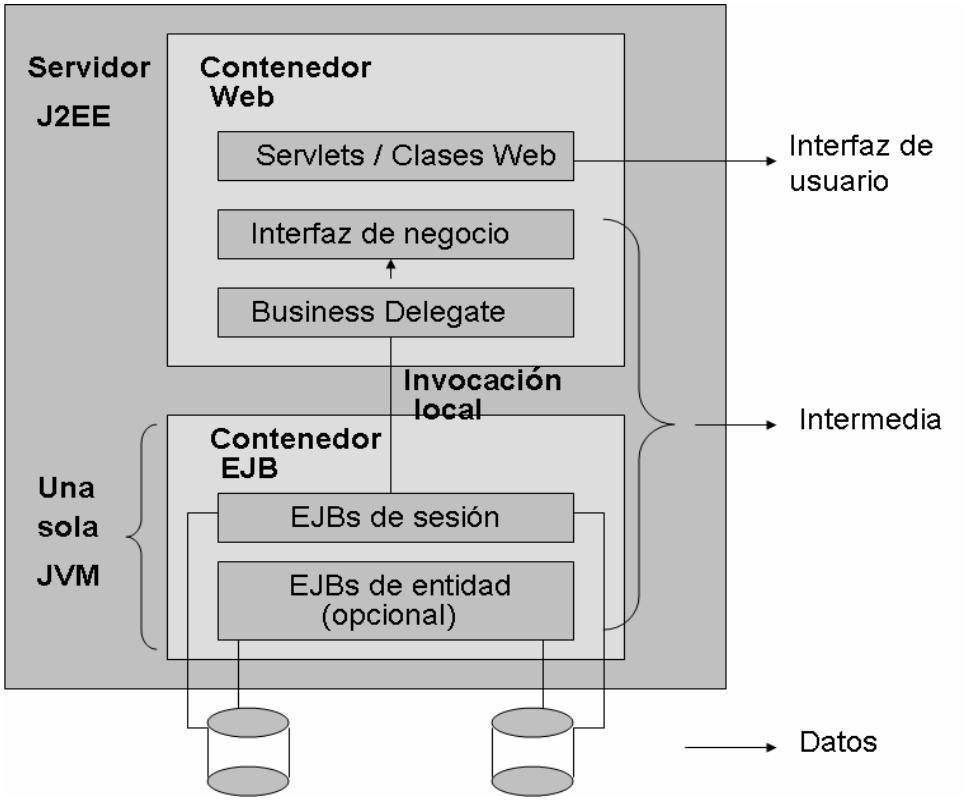


Fig 2.9 Modelo arquitectónico para aplicaciones no distribuidas usando EJB

Interfaz de usuario

La capa de interfaz de usuario en este modelo es implementada usando los APIs JSP y servlet. Se utiliza un servlet como un controlador frontal que intercepte todas las solicitudes del cliente y llame a los métodos correspondientes, reciba los resultados de los mismos y muestre las páginas correspondientes. Las

páginas JSP son las encargadas de mostrar las vistas apoyándose en las librerías de etiquetas y en java beans. Además, para registrar la autenticación de usuarios, asignar formato a datos u otras tareas se pueden utilizar filtros, los cuales se aplican a los objetos solicitud (request) o respuesta (response) en cada una de sus salidas y entradas al contenedor web[9].

Intermedia

La capa media posee una interfaz de negocio que va a actuar como una abstracción de la implementación del negocio, un Business Delegate que va a invocar al EJB local apropiado. EL Service Locator se encarga de la localización de componentes. El Session Facade sirve como fachada de los EJBs de sesión, los cuales pueden utilizar para el intercambio de datos con los sistemas de almacenamiento EJBs de entidad o una capa de abstracción DAO como el caso analizado anteriormente[9].

Datos

Esta capa puede ser representada por cualquier sistema gestor de base de datos como Oracle, SQL Server, MySQL, o por algún fichero de almacenamiento de texto, XML, entre otros[9].

Flujo de una solicitud

Cuando el usuario escribe la url en el navegador el contenedor web recibe una solicitud la cual es tomada por el servlet controlador no sin antes haber pasado por el filtro (si es que se decide la utilización de filtros, y se configura para que intercepte dicha url), el controlador dada la solicitud la direcciona bien puede ser directamente para una página JSP o para algún método de alguna clase que intervenga en la lógica de presentación con el objetivo de realizar alguna acción llamando a algún método de la interfaz de negocio, el Bussines Delegate obtiene la referencia al EJB local indicado utilizando para ello un Service Locator que realiza toda la lógica de localización de componentes. En este caso devolverá un Session Facade implementado como un EJB de sesión que expone una fachada para la interacción con la lógica de negocio presente en uno o varios componentes EJB, estos van a realizar el procesamiento, utilizando para el intercambio de datos o bien EJBs de entidad o una capa de abstracción DAO(fig 2.6).

Ventajas

- Es menos compleja que una aplicación distribuida con EJB.
- El uso de EJB no altera el diseño básico de la aplicación. En esta arquitectura sólo hace que los EJBs necesiten los servicios del contenedor de EJB.
- Ofrece los beneficios del contenedor EJB para el manejo de transacciones.

- Permite el uso de EJB de entidad si se desea.

Desventajas

- Es más compleja que una aplicación web pura que no tenga componentes EJB.
- La aplicación completa se ejecuta dentro de una sola Java Virtual Machine, lo que significa que todos los componentes se ejecutan dentro del mismo servidor físico.
- Los EJBs con interfaces locales son difíciles de probar. Se necesitan ejecutar casos de prueba dentro del servidor JEE (por ejemplo dentro de servlets).
- Aún con interfaces locales las invocaciones a EJBs son más lentas que llamadas a métodos comunes.

Conclusiones

Existen varios modelos arquitectónicos sobre la plataforma JEE, cada uno presenta ventajas y desventajas que a través de su análisis permiten llegar a las siguientes conclusiones:

- Las aplicaciones distribuidas son difíciles de implementar y de probar al contar con componentes remotos dispersos por la red.
- Los servicios del contenedor EJB (manejo de transacciones, seguridad, etc) agilizan el desarrollo de aplicaciones complejas, sin embargo el uso de componentes EJB puede causar problemas en la implementación.
- El uso de los API de JEE directamente requiere la implementación de varios patrones de diseño.

Capítulo 3: Propuesta de arquitectura JEE.

En el capítulo anterior se analizan variantes arquitectónicas y sus principales características. En este capítulo se presenta de manera detallada la propuesta de arquitectura JEE, basada en un modelo híbrido formado a partir del estudio de las ventajas y deficiencias de variantes anteriores y que se ajusta principalmente a al desarrollo de aplicaciones web que automaticen procesos de negocio disponibles en una intranet, que no requieran la utilización de los API EJB (componentes distribuidos de acceso remoto) y JMS (servicios de mensajería) y que el sistema de información sea manejado por un gestor de base de datos.

La recomendación se basa en una arquitectura no distribuida, evitando los problemas de complejidad y rendimiento que provoca la distribución de componentes, y sobre todo acogiéndose a la idea de lograr aplicaciones robustas y escalables de la manera más sencilla posible. Los componentes distribuidos (EJB) son difíciles de implementar, necesitan ejecutarse en un contenedor EJB que a la vez viene unido con un servidor de aplicaciones cuya configuración y administración es más compleja que la de un servidor web y consume más recursos, además los EJB al ejecutarse en distintas máquinas son mucho más difíciles de testear. La propuesta intenta lograr un mayor nivel de productividad combinando frameworks que brinden manejadores de persistencia, transacciones, seguridad entre otros y a la vez sean más sencillos de configurar que otras opciones.

Aplicación web utilizando frameworks (JSF, Spring, Hibernate)

La propuesta se apoya en los siguientes frameworks de desarrollo para implementar las principales capas de la aplicación (fig 3.1):

- Java Server Faces (JSF) para la capa de presentación.
- Spring para gestionar la lógica de negocio y otras funciones a nivel de aplicación como la seguridad, las trazas y como enlace entre la capa de acceso a datos y la capa de presentación.
- Hibernate, como mecanismo persistente.

Además se utiliza como entorno de ejecución el contenedor web Tomcat.

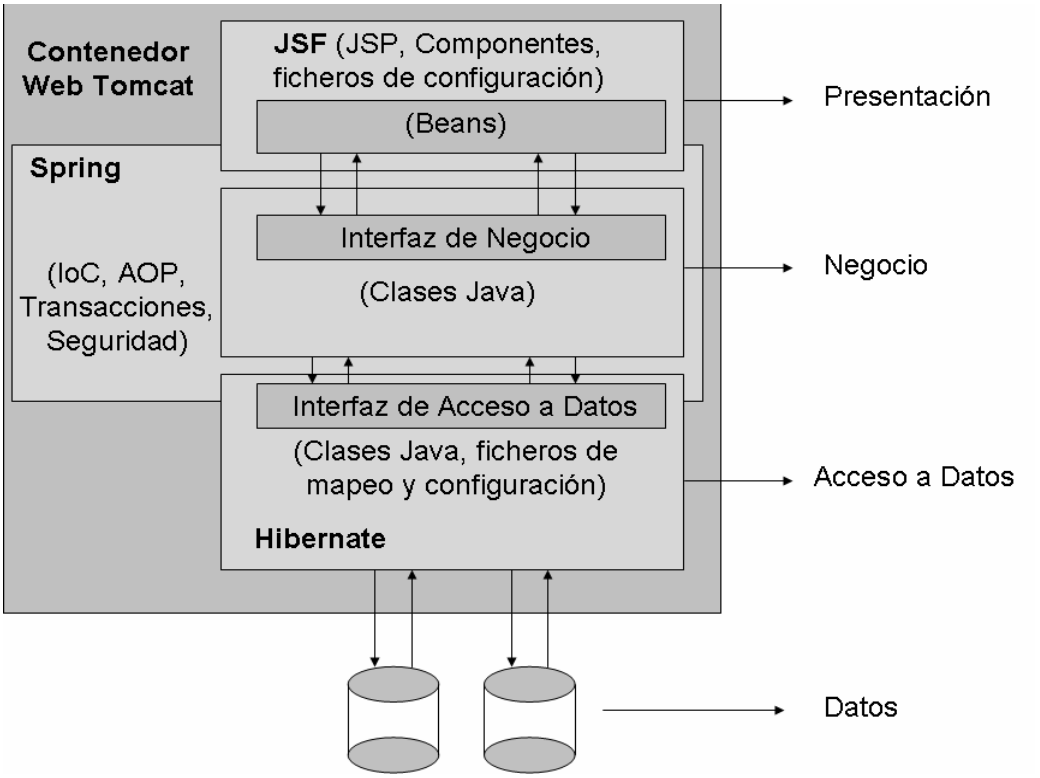


Fig 3.1 Propuesta arquitectónica

¿Por qué usar Java Server Faces?

No es menos cierto que el conocido Struts goza de un alto prestigio en la comunidad de desarrolladores debido a que es un framework maduro y con una larga base de desarrollo. Struts cuenta con un gran paquete de documentación y materiales de referencia, además de soporte de IDEs y herramientas. Pero, ¿por qué usar Java Server Faces si se cuenta con un framework con grandes características?

Con el objetivo de aprovechar el poderoso modelo estándar de componentes configurable y reutilizable, se pueden desarrollar componentes más complejos a partir de otros, a diferencia de Struts que tiene etiquetas personalizadas y un sistema de plantillas que permite dividir a las páginas JSP en pequeños pedazos. El modelo de componentes implementado por JSF hace que el mismo tenga un robusto manejo de eventos para la interfaz de usuario que son ejecutados cuando alguno de estos componentes cambie su estado actual y permite que se manejen funcionalidades para la capa de interfaz. Java Server Faces posee un desacoplado modelo de renderización permitiendo que el mismo componente sea renderizado de diferentes maneras para el mismo o diferentes clientes, por ejemplo, un componente de comando que

es usado para hacer un envío de un formulario (Submit) puede ser renderizado como un botón (Button) o como un hipervínculo (Hyperlink). Este framework está basado en beans de respaldo que son JavaBeans asociados con componentes utilizados en la página, estos beans son independientes del framework, a diferencia de Struts, los mismos no extienden su funcionalidad de ninguna clase. El control de beans de respaldo de JSF separa la definición de los objetos componentes de la interfaz de usuario de los objetos que realizan el procesamiento específico de la aplicación y que además contienen los datos, en estos beans se maneja la lógica de la aplicación, ya sea en funciones correspondientes a eventos así como también en funciones dedicadas a darle paso a la navegación. En el framework Struts no ocurre así pues la lógica de la aplicación requiere además de las clases Action. La implementación de JSF almacena y maneja estos ejemplares de beans de respaldo en el ámbito apropiado. El modelo de conversión y validación extensible de JSF hace posible que a partir de convertidores y validadores estándar se puedan desarrollar elementos personalizados que proporcionen un mejor modelo de protección. Java Server Faces puede hacer la validación de los datos de entrada en los beans de respaldo de la misma forma que lo hace Struts, pero a diferencia de este puede utilizar validadores estándar en las propias páginas JSP.

¿Por qué usar Spring?

Spring es un framework que tiene el objetivo de facilitar la construcción de aplicaciones java. A diferencia de otros frameworks como Struts, Spring se puede utilizar en cualquier tipo de aplicación no solo en aplicaciones web. Spring es un framework ligero por el mínimo impacto que tiene en las aplicaciones. Constituye una alternativa al uso de componentes EJB y de servidores de aplicaciones, pues provee servicios similares a estos como es el caso del manejo de los objetos y la gestión de transacciones declarativamente, disminuyendo el tiempo y el esfuerzo en el desarrollo de las aplicaciones. Spring trae integrado el framework Acegi, que provee un mecanismo de seguridad potente y fácil de configurar permitiendo implementar las políticas de seguridad de manera transparente al código de la aplicación. Además Spring es un framework ligero y sus aplicaciones pueden ejecutarse en un contenedor de servlet. Spring está pensado para manejar los objetos de negocio de la aplicación con su técnica de inyección de instancias, todo desde un fichero de configuración, manteniendo el código limpio y disminuyendo el acoplamiento. Contiene un módulo para programar orientado a aspectos que permite implementar funciones que de hacerlas orientado a objetos serían largas e intrusivas en el código de la aplicación. Posee una capa de abstracción para el acceso a datos con JDBC que facilita la implementación en esta capa. Cuando se utilizan otros frameworks de desarrollo tanto en la capa de presentación como en la de

acceso a datos, Spring funciona como un framework integrador que facilita la comunicación entre estas capas.

¿Por qué usar Hibernate?

En aplicaciones complejas el uso del API JDBC puede traer consigo interminables líneas de código SQL para realizar todo el trabajo de acceso a datos. Una de las alternativas a esta problemática son los EJB de entidad. Esta variante tiene varios inconvenientes, entre ellos el hecho de que los EJB son complejos de implementar, se ejecutan dentro de un contenedor de EJB y por tanto en un servidor de aplicaciones. Es por eso que la propuesta se inclina al uso de un framework de persistencia como Hibernate. Hibernate crea un puente entre el mundo orientado objetos de las aplicaciones y el entorno relacional de las bases de datos, enajenando a los desarrolladores del código JDBC y de las consultas SQL. Es un framework ORM (Object Relational Mapping), que se basa en ficheros xml que mapean los objetos con las tablas en la base de datos. A diferencia de los EJB de entidad, Hibernate constituye un mecanismo persistente transparente pues las clases persistentes desconocen que tienen esa propiedad porque no tienen que implementar ninguna interfaz especial ni extender de ninguna clase, y pueden ser utilizadas en cualquier capa de la aplicación. Además, este framework no necesita ningún entorno especial para ejecutarse.

El uso de Hibernate evita la difícil implementación de EJBs de entidad y la costosa llamada a los mismos. Hibernate provee un lenguaje de consultas orientado a objetos que facilita la ejecución de estas, crea una capa de abstracción que permite migrar de gestor de base de datos sin cambiar una sola línea de código. Como conclusión este framework facilita el desarrollo de la lógica de acceso a datos en las aplicaciones.

¿Por qué usar Tomcat?

Tomcat es un contenedor web y por tanto es más sencillo de administrar y configurar. Consume menos recursos que un servidor de aplicaciones. Tomcat es gratis y posee un gran número de usuarios y soporte en la comunidad mundial. Además es compatible con la mayoría de los APIs de J2EE. En caso de aplicaciones con gran cantidad de usuarios, Apache Tomcat permite lograr escalabilidad mediante clúster. Además, este contenedor presenta un manejador de reserva de conexiones que se configura de manera que desde la aplicación se acceden a estas, este mecanismo es más eficiente que el de realizar la conexión en el momento que se requiera.

Presentación

El framework Java Server Faces, utilizado como propuesta para el manejo de la capa de presentación de la arquitectura, se apoya en el API JSP para la construcción de interfaces de usuario. Cuenta con dos librerías centrales de etiquetas, las cuales proveen etiquetas personalizadas para las tareas de validación y manejo de eventos, se debe incluir la declaración de estas librerías al inicio de las páginas JSP (fig 3.2).

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
```

Fig 3.2 Librerías centrales de etiquetas

Además, las páginas JSP pueden tener un grupo de componentes, estos son organizados dentro de familias, cada una reúne un conjunto de componentes con comportamiento similar. Los formularios y componentes que forman las páginas están respaldados por beans, estos poseen total independencia del framework pues no extienden comportamiento de ninguna otra clase proporcionada por JSF.

Los cambios de comportamiento en los componentes provocan el lanzamiento de eventos que son escuchados por oyentes, estos eventos se aprovechan para agregar algún tipo de funcionalidad a la interfaz de usuario, a los componentes se les especifica el bean de respaldo y el método en el cual se hace la implementación de la nueva funcionalidad (fig 3.3).

```
<h:commandLink actionListener="#{usuarioBean.cambiarlenguaje}">
     <h:outputText value="#{textos.vinculo}"/>
     <f:param name="idioma" value="#{textos.vinculo}"/>
</h:commandLink>
```

Fig 3.3 Vínculo que al ser accedido desencadena un evento dentro del bean de respaldo (usuarioBean) que cambia el idioma de la página (cambiarlenguaje)

Otros componentes de interfaz como botones e hipervínculos, al ser accedidos por el usuario, pueden desencadenar acciones que son especificadas dentro de los mismos e implementadas dentro de los beans de respaldo de los formularios. Dentro de estas acciones puede realizarse lógica de aplicación así como también acceso a la lógica de negocio. El sistema de navegación se apoya en las acciones dentro de los formularios, estas acciones desencadenan resultados de navegación. También dentro de las páginas se pueden ofrecer resultados de navegación estáticos sin necesidad de implementarlos dentro de las acciones en los beans de respaldo. Java Server Faces utiliza un fichero XML de configuración (*faces-config.xml*) en el cual se declaran los beans de respaldo que se utilizan en la aplicación con todas sus propiedades (el nombre por el cual se van a referenciar dentro de la aplicación, la clase, el alcance de los mismos) (fig 3.4).

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD
JavaServer Faces Config 1.0//EN"
                              "http://java.sun.com/dtd/web-
facesconfig_1_0.dtd">
<faces-config>
 <managed-bean>
    <description>
      Bean de prueba de usuario
    </description>
    <managed-bean-name>usuarioBean</managed-bean-name>
    <managed-bean-class>view.beans.UsuarioBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
 </managed-bean>
</faces-config>
```

Fig 3.4 Fichero xml donde se van a configurar los beans de respaldo con sus propiedades

En este fichero de configuración también se van a definir las reglas de navegación para las diferentes páginas JSP y los casos de navegación para las mismas, ahí se van a tomar como referencia las cadenas de resultados que salen de las acciones de los beans de respaldo, cada una de ellas arroja una salida diferente(fig 3.5).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems,
Inc.//DTD JavaServer Faces Config 1.0//EN"

"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
 <navigation-rule>
  <from-view-id>/usuario.jsp</from-view-id>
  <navigation-case>
   <from-outcome>inicio</from-outcome>
   <to-view-id>/usuario.jsp</to-view-id>
  </navigation-case>
 </navigation-rule>
</faces-config>
```

Fig 3.5 Fichero xml donde se va a configurar el sistema de navegación

El framework posee un servlet controlador central (*Faces Servlet*) que enruta las peticiones y respuestas, este proceso puede verse desde el momento en que el usuario hace alguna acción sobre un componente y el mismo está asociado a un bean de respaldo, el servlet controlador de JSF localiza en su fichero de configuración(*faces-config.xml*) el nombre del bean y la clase del mismo, dentro de esta clase busca el método que desencadena la acción sobre el componente y dentro del mismo realiza la funcionalidad especificada. Los componentes pueden tener la propiedad *binding* donde se especifica el bean de formulario y el atributo de la clase que representa el componente, esto permite que haya una

sincronización entre el componente en la interfaz de usuario y el atributo que lo representa dentro del bean de tal manera que si en un método desencadenado por un evento se puede cambiar el valor de este atributo y como tiene relación directa con el componente este modifique su valor también cuando se muestre nuevamente la interfaz. Este framework tiene implementado validadores estándar que pueden especificarse dentro de los componentes que se encuentran en las JSP (fig 3.6).

```
 <h:inputText value="#{usuarioBean.nombre}"
id="nombre" required="true">
     <f:validateLength minimum="3" maximum="20"/>
 </h:inputText>
 <h:message for="nombre"/>
```

Fig 3.6 Componente de entrada de texto con los validadores estándar *required* (valor de entrada requerido) y *validateLength* (valor de entrada con un máximo y un mínimo de caracteres)

Las validaciones de los datos de entrada también pueden hacerse dentro de los beans de respaldo. A cada componente puede asociársele uno o varios validadores estándar, esta validación se hará por cada validador y si el componente falla en cualquiera de ellos entonces los datos entrados no son válidos. También a los componentes se les puede asociar validaciones por expresiones regulares. JSF posee mecanismos de internacionalización de aplicaciones, estos mecanismos están basados en archivos de recursos *.properties* en los cuales se van a escribir todos los textos de la aplicación y mensajes, en estos archivos de recursos se especifican las mismas claves pero en cada uno de ellos estas tienen diferentes valores en dependencia del lenguaje (fig 3.7 y 3.8).

TextosApp_es_ES.properties

```
encabezado=Datos del usuario\:
texto2=No. Identidad\:
texto_boton=Aceptar
vinculo=Ingl\u00E9s
texto3=Tel\u00E9fono\:
texto1=Nombre\:
```

Fig 3.7 Archivo de recurso para el idioma Español

TextosApp_en_US.properties

```
encabezado=User Information\:
texto2=Identification number\:
texto_boton=OK
vinculo=Spanish
texto3=Telephone\:
texto1=Name\:
```

Fig 3.8 Archivo de recurso para el idioma Inglés

Dentro del archivo de configuración se expone el idioma por defecto que se utilizará (fig 3.9). Los archivos de recursos se deben nombrar de igual forma, especificando el idioma y el país después del nombre, que es común a todos los de la aplicación. El cambio de referencia al idioma se realiza en los métodos dentro de los beans de respaldo, en caso que el usuario a través de la interfaz solicite otro lenguaje.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems,
Inc.//DTD JavaServer Faces Config 1.0//EN"
                              "http://java.sun.com/dtd/web-
facesconfig_1_0.dtd">
<faces-config>
 <application>
  <locale-config>
   <default-locale>es_ES</default-locale>
  </locale-config>
  <message-bundle>view.boundle.TextosApp</message-bundle>
  <message-bundle>view.boundle.Mensajes</message-bundle>
 </application>
</faces-config>
```

Fig 3.9 Fichero de configuración donde se especifica el idioma de los archivos de recursos que se utilizará por defecto en la aplicación y los nombres de los archivos de textos y mensajes de la misma

El registro de convertidores puede desarrollarse declarativamente en las páginas. JSF tiene convertidores estándar como *DateTime* y *Number*. Se pueden construir nuevos convertidores además de los ya existentes de acuerdo a las necesidades de los desarrolladores y hacer referencia a ellos dentro de los componentes.

Como antes se había mencionado, en los métodos que desencadenan acciones dentro de los beans de respaldo puede realizarse la llamada a los métodos de negocio de la aplicación que son gestionados por el framework Spring (fig 3.10).

```
public String insertar_usuario(){
  try{
  gestion_usuario.insertar(this);
  }
  catch (Exception e){
  return "error";
  }
  return "success";
}
```

Fig 3.10 Llamada a los métodos de negocio de la aplicación gestionados por Spring

En el fichero de configuración de JSF se especifica la referencia a los beans de negocio, en este caso se especifica el bean de negocio utilizado para insertar el usuario como una propiedad del bean de formulario. Esta técnica se nombra inyección de dependencia (fig 3.11).

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD
JavaServer Faces Config 1.0//EN"
                              "http://java.sun.com/dtd/web-
facesconfig_1_0.dtd">
<faces-config>
 <managed-bean>
    <description>
      Bean de prueba de usuario
    </description>
    <managed-bean-name>usuarioBean</managed-bean-name>
    <managed-bean-class>view.beans.UsuarioBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>gestion_usuario</property-name>
      <value>#{gestion_usuario}</value>
    </managed-property>
 </managed-bean>
</faces-config>
```

Fig 3.11 Declaración del objeto de negocio como una propiedad del bean de respaldo

JSF usa una variable que resuelve la localización de las clases de negocio gestionadas por Spring de manera transparente, *FacesSpringVariableResolver,* que debe ser declarada en el fichero de configuración de JSF como lo muestra la fig 3.12. Es necesario también en este caso configurar en el fichero *web.xml* con una clase escuchadora que deberá cargar el contexto del archivo de configuración de Spring (fig 3.13).

```
<application>
  …
<variable-resolver>
de.mindmatters.faces.spring.FacesSpringVariable
Resolver
</variable-resolver>
</application>
```

Fig 3.12 Declaración de la variable que resuelve la localización de las clases de negocio

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

Fig 3.13 Declaración de la clase escuchadora que carga el contexto del archivo de configuración de Spring

Capa de Negocio

Como antes se había mencionado, esta capa está gestionada por el framework Spring. Spring presenta un contenedor, el cual usa IoC para manejar los componentes que estructuran la aplicación.

El centro del desarrollo con Spring lo constituyen los ficheros de configuración donde se definen todos los objetos tratados por el contenedor. La raíz de este fichero XML es el elemento <*beans*> en su interior se definen los beans (objetos) que serán manejados por el contenedor. Cada objeto será definido por el elemento <*bean*>, el cual contiene un atributo *id* que será el identificador del bean, y un atributo *class* donde se especifica la clase a la cual pertenece (fig 3.14).

Por defecto, todos los objetos definidos en los ficheros de configuración son tratados como *singleton*, por tanto cada vez que el contenedor devuelve una instancia está devolviendo exactamente el mismo objeto. Si se desea cambiar este comportamiento se debe modificar el atributo *singleton* en la definición, y de esta manera será nuevo cada objeto devuelto por el contenedor.

Para inicializar atributos en los objetos se utiliza el subelemento <*property*>, se le especifica el nombre del mismo en el atributo *name* y el valor en el subelemento <*value*> (fig 3.14).

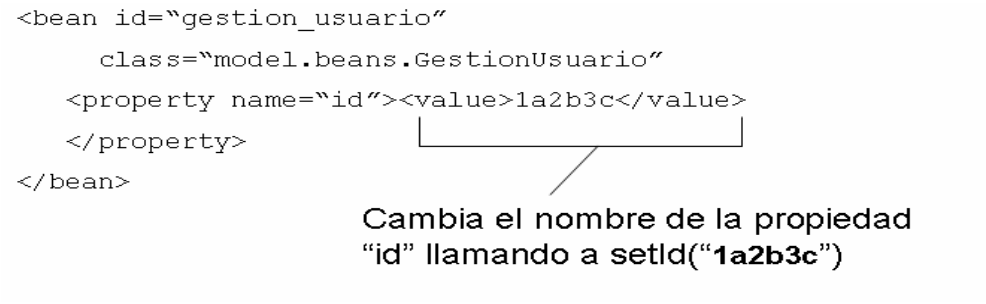


Fig 3.14 Forma de inicializar valores de los atributos en Spring

En el caso de que el atributo haga referencia a otro bean se utiliza el subelemento *<ref>* y se especifica en el atributo *bean* el identificador que le fue definido a esta clase dentro del fichero de configuración (fig 3.15).

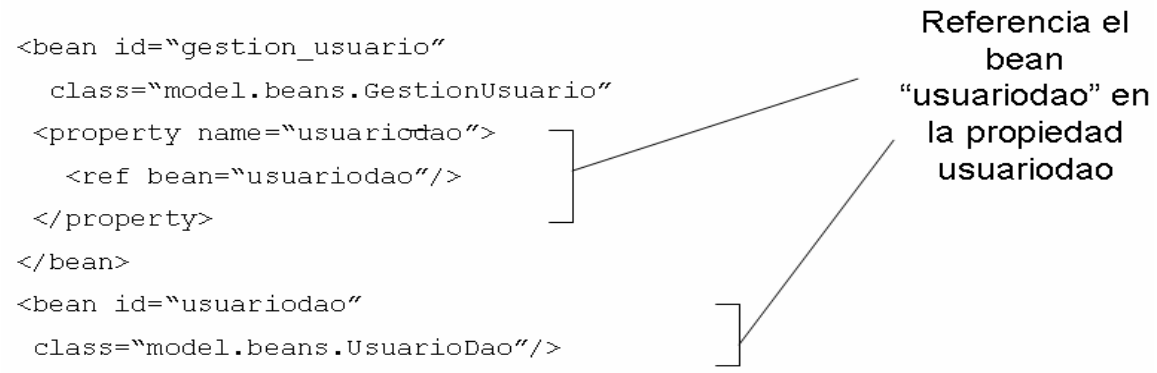


Fig 3.15 Forma que utiliza Spring para hacer referencia mediante una propiedad de un bean a otro

Para realizar la inyección de los valores de las propiedades en la clase del bean deben estar especificados los métodos *"get"* y *"set"* de cada una de las propiedades que serán inyectadas. De esta forma cuando se obtiene un objeto, este contiene todas sus referencias inicializadas, mediante los métodos de acceso (esta inicialización se puede lograr de forma similar mediante el constructor de la clase bean). De esta manera se configuran los objetos para que sean gestionados por el contenedor de Spring.

En esta arquitectura la lógica de negocio se expone mediante una interfaz y es implementada mediante clases java, las mismas contienen referencias a la interfaz de la capa de acceso a datos, que será implementada por clases que utilizan las funcionalidades de Hibernate. Tanto las clases de negocio como las de acceso a datos se definen en el fichero de configuración de Spring, por tanto las primeras reciben las referencias de las segundas inyectadas por el contenedor de Spring.

Capa de Acceso a Datos

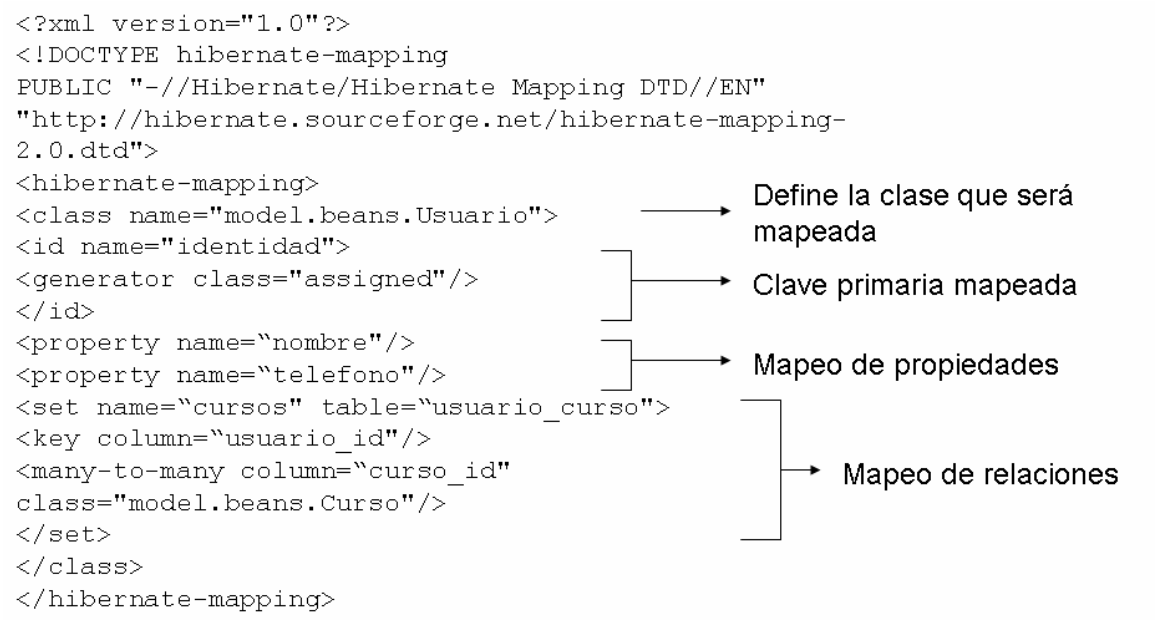
La capa de accesos a datos cuenta con interfaces, estas son implementadas por clases java que contienen la lógica de acceso a datos (DAO), para la implementación de estas clases se utilizan las funcionalidades de Hibernate.

Para su funcionamiento Hibernate utiliza principalmente las siguientes interfaces:

- *Session*, es la interfaz principal de Hibernate, su noción oscila entre una conexión y una transacción, es quien realiza todas las operaciones de persistencia.
- *SessionFactory*, mediante ella se obtiene la instancia del objeto *Session*, es una interfaz pesada, generalmente se implementa como un *singleton*, si se utilizan varias bases de datos se deben utilizar también varias *SessionFactory*.
- *Configuration*, se utiliza para configurar Hibernate y especificar la localización del fichero de configuración para crear la *SessionFactory*.
- *Transaction*, se utiliza para manejar las transacciones.
- *Query* y *Criteria*, se utilizan para la ejecución de consultas, utilizando para ello HQL o lenguaje nativo.

Este framework se basa en el mapeo objeto relacional, para esto utiliza un fichero xml por cada clase persistente de la aplicación. Estas clases están totalmente desacopladas del framework gracias a que el mecanismo de Hibernate es transparente, por tanto son utilizadas como portadoras de datos que se mueven desde la capa de acceso a datos hasta la capa de presentación.

En el fichero de mapeo se especifica la correspondencia entre cada clase persistente y la tabla en la base de datos, así como los atributos de la clase y los campos de la tabla correspondientes a estos, de igual forma se especifican las relaciones. Generalmente estos ficheros al igual que las clases persistentes, son generados a partir del esquema de la base de datos utilizando herramientas (fig 3.16).




```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-
2.0.dtd">
<hibernate-mapping>
<class name="model.beans.Usuario">
<id name="identidad">
<generator class="assigned"/>
</id>
<property name="nombre"/>
<property name="telefono"/>
<set name="cursos" table="usuario_curso">
<key column="usuario_id"/>
<many-to-many column="curso_id"
class="model.beans.Curso"/>
</set>
</class>
</hibernate-mapping>
```

Fig 3.16 Mapeo de correspondencia entre la clase persistente y la tabla de la base de datos

Además de los ficheros de mapeo, Hibernate cuenta con un archivo de configuración donde se especifican los detalles de la conexión a la base de datos:

- Driver de conexión.
- URL de conexión.
- Usuario y contraseña de la base de datos.
- Dialecto (se refiere al dialecto SQL a utilizar el cual es referente al gestor de base de datos).

Además, aquí se registran todos los ficheros de mapeo de la aplicación. El archivo de configuración de Hibernate es cargado cuando se construye la *SessionFactory*.

Ahora, como se utiliza Hibernate integrado con Spring, este último abstrae a los desarrolladores del uso de las interfaces de Hibernate directamente, puesto que provee un mecanismo que permite utilizar todas las funcionalidades de este de una manera más sencilla y transparente.

Spring provee los siguientes puntos de integración con Hibernate:

- Manejo de recursos.
- Plantillas de clases.
- Manejo de transacciones.

En cuanto al manejo de recursos, Hibernate mantiene un solo objeto *SessionFactory* (por cada base de datos) a través de toda la aplicación, por lo que se va a utilizar el contenedor de Spring para manejar dicho objeto y para esto se usa la clase de Spring *LocalSessionFactoryBean* (fig 3.17).

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSession
FactoryBean">
<property name="dataSource">
  <ref bean="dataSource"/>
</property>
</bean>
```

Fig 3.17 Configuración de la SessionFactory de Hibernate

A esta clase se le especifica la propiedad *dataSource*, la cual representa la forma de obtener la conexión a la base de datos. En este caso se va a utilizar un mecanismo que es recomendable usar por su eficiencia, la reserva de conexiones. Al utilizar una reserva de conexiones se garantiza que cuando la aplicación necesite conectarse a la base de datos solo tiene que utilizar una conexión ya disponible y no tratarse de conectar en ese momento.

Para implementar esta técnica la propuesta se apoya en el manejador de reserva de conexiones que incluye el contenedor web Apache Tomcat (fig 3.18).

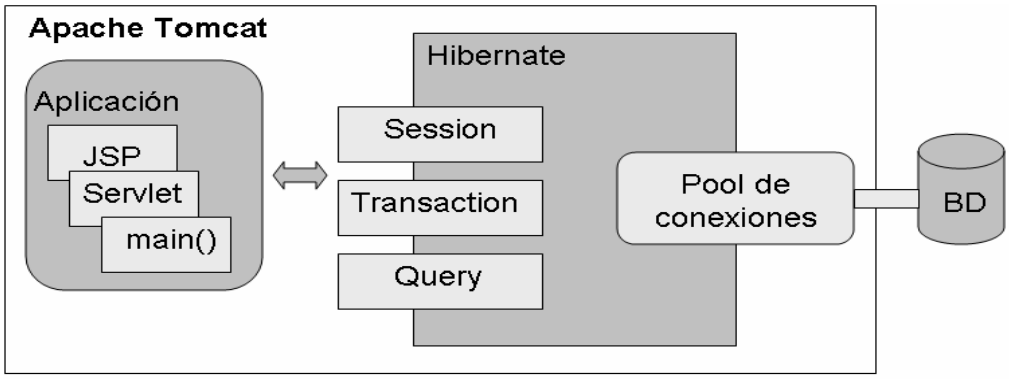


Fig 3.18 Reserva de conexiones

El Tomcat expone las conexiones en un repositorio *JNDI* al cual se accede desde Spring utilizando la clase *JndiObjectFactoryBean* (fig 3.19).

```
<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
 <property name="jndiName">
  <value>java:comp/env/jdbc/MiFuentedeDatos</value>
 </property>
</bean>
```

Fig 3.19 Configuración del dataSource

En este caso se obtienen los valores del contenedor web Tomcat que es el servidor donde se ejecuta la aplicación, el cual crea una reserva de conexiones (connection pool) que permite siempre tener conexiones disponibles para la *SessionFactory*, este procedimiento es más eficiente que crear conexiones cuando se necesiten, esta reserva se realiza en cuanto levanta el contenedor web. De esta manera Spring maneja los recursos de Hibernate.

El otro punto de integración es la utilización de las plantillas de Spring para el acceso a datos. Para implementar las clases que realizan el acceso a datos, Spring contiene la clase *HibernateDaoSupport* de la cual deben heredar las mismas. Esta herencia permite desde las clases DAO acceder al método *getHibernateTemplate()*, que devuelve una plantilla con las operaciones básicas de persistencia y consulta de objetos en la base de datos (fig 3.20). Entre estos métodos se encuentran:

- *save(Object objeto)*, persiste el objeto en la base de datos
- *saveOrUpdate(Object objeto),* persiste el objeto en caso de no existir en la base de datos, si existe actualiza la tupla.
- *saveOrUpdateAll(Collection lista)*, persiste en caso de no existir los objetos, si alguno existe actualiza dicha tupla.
- *get(Class clase ,String id)*, devuelve el objeto dado la clase y el identificador.
- *findByExample(Object objeto)*, devuelve una lista con los objetos que cumplan con los criterios que se especifican en los atributos del objeto entrado.
- *findByNamedParam(String consulta, String[] nombreParametros, Object[] valorParametros)* , devuelve una lista con los objetos que cumplan con la consulta en HQL dada y con los parámetros y valores especificados.

Además de estos métodos, la plantilla contiene otros similares que agrupan casi todas las posibles operaciones que se puede realizar y por si fuese necesario, también contiene los métodos *getSession()* y *closeSessionIfNecessary()* para realizar operaciones de Hibernate sin usar los métodos de la plantilla.

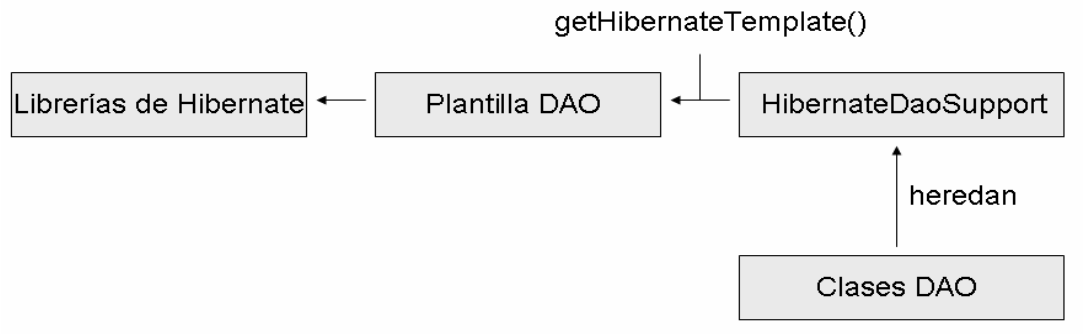


Fig 3.20 Relación entre las clases de acceso a datos (Clases DAO), la clase HibernateDaoSupport, la plantilla DAO y las librerías de Hibernate

Capa de Datos

Hibernate como mecanismo de persistencia permite un alto grado de portabilidad en lo que respecta al gestor de base de datos de la aplicación, permitiendo tener total independencia del mismo. Por tanto, en esta capa se puede utilizar cualquier sistema sea SQL Server, Oracle, PostGres, lo único que va a variar en la aplicación es la configuración de la propiedad dataSource el driver y la URL de conexión y en la *SessionFactory*, el dialecto que utilizará Hibernate.

Manejo de transacciones

Otro punto de integración de Hibernate con Spring lo constituye el manejo de las transacciones. Este aspecto es uno de los más importantes dentro de las aplicaciones pues puede llegar a ser muy compleja su implementación en algunos sistemas.

Por mucho tiempo el manejo de transacciones declarativas fue una característica de los contenedores EJB, Spring rompió con esto. Spring provee de un mecanismo de manejo de transacciones limpio y sencillo de implementar, utilizando programación orientada a aspectos (AOP) y un conjunto de librerías que permiten definirlas declarativamente en el fichero de configuración del framework. Una transacción es concebida como un aspecto que envuelve a un método.

Spring no gestiona las transacciones directamente, sino que el mismo contiene un conjunto de manejadores que interactúan y delegan esta responsabilidad en uno específico, puede ser el API *Java Transaction* (JTA) u otro mecanismo persistente. Entre los manejadores que trae Spring se encuentran:

- *DataSourceTransactionManager*, para manejar las transacciones cuando se utiliza JDBC.
- *HibernateTransactionManager*, para manejar las transacciones cuando se utiliza Hibernate como mecanismo persistente.

Estos manejadores actúan como una fachada de la implementación de un gestor de transacciones específico, de esta forma facilitan su implementación dejándola en manos de Spring.

Un elemento importante para definir las transacciones son los atributos de las mismas. Los atributos de una transacción constituyen la descripción de cómo una transacción debe ser aplicada a un método. Esta descripción puede contener uno o varios de estos parámetros:

- Comportamiento de propagación.
- Nivel de aislamiento.
- Indicio de solo lectura.
- Período de expiración de la transacción.

El comportamiento de propagación define lo que ocurre en una llamada transaccional dependiendo si existe alguna transacción activa. Algunas de las reglas de propagación de Spring son las siguientes:

Comportamiento de Propagación	Descripción
PROPAGATION_REQUIRED	Indica que el método debe ser ejecutado dentro de una transacción. Si alguna transacción está en progreso, el método puede ejecutarse dentro de ella, sino se creará una nueva.
PROPAGATION_SUPPORTS	Indica que el método no requiere un contexto transaccional, pero puede ejecutarse dentro de una transacción si alguna está en progreso.
PROPAGATION_MANDATORY	Indica que el método debe ejecutarse dentro de una transacción. Si no existe ninguna activa entonces lanza una excepción.
PROPAGATION_REQUIRES_NEW	Indica que la ejecución del método siempre va a comenzar una nueva transacción, y si alguna ya está activa será suspendida.
PROPAGATION_NOT_SUPPORTED	El método nunca se va a ejecutar en un ambiente transaccional, si alguna transacción está activa será suspendida mientras dure

	el método.
PROPAGATION_NEVER	Nunca el método se va a ejecutar en una transacción, si alguna existe se lanzará una excepción.

El comportamiento de propagación responde cuando una transacción debe ser iniciada o suspendida, o si un determinado método debe ejecutarse en un ambiente transaccional.

En una aplicación múltiples transacciones se ejecutan simultáneamente y a menudo trabajan con los mismos datos. Esta concurrencia puede desencadenar problemas como:

- Mala lectura, una transacción lee datos que han sido escritos por otra pero que no se les ha dado commit aún, si más tarde a esta transacción se le aplica rollback, los valores de la primera transacción serán inválidos.
- Lectura no repetida, una transacción ejecuta una consulta dos o más veces y en cada momento obtiene distintos datos. Esto ocurre usualmente cuando otra transacción actualizó los datos en el intermedio de las consultas.
- Lectura de nuevas filas, es similar al anterior, una transacción T1 lee varias filas en una consulta, una transacción concurrente T2 inserta nuevos datos, en consultas posteriores T1 encuentra nuevos resultados.

Para evitar estos problemas lo mejor sería que cada transacción fuera independiente de las demás. Esta total independencia puede afectar el rendimiento pues desencadena bloqueo de tuplas y a veces tablas enteras, y normalmente no todas las transacciones necesitan ser totalmente aisladas. Los niveles de aislamiento que provee Spring para las transacciones son los siguientes:

Niveles de aislamiento	Descripción
ISOLATION_DEFAULT	Usa el nivel de aislamiento por defecto que presenta la base de datos que se utiliza.
ISOLATION_READ_UNCOMMITTED	Permite leer cambios que aún no se les ha dado commit. Permite que ocurran los problemas anteriores. Es el nivel más bajo de aislamiento.
ISOLATION_READ_COMMITTED	Permite leer de transacciones concurrentes que ya han dado commit. Se evita que ocurra la mala lectura.
ISOLATION_REPEATABLE_READ	Permite que varias lecturas sobre el mismo campo

	devuelvan los mismos resultados, a menos que sean modificados por la propia transacción, se evita los problemas de mala lectura y de lectura no repetida.
ISOLATION_SERIALIZABLE	Es el nivel más confiable y más costoso, evita todos los problemas, todas las transacciones son ejecutadas independientes o sea una a continuación de la otra.

Utilizando estos valores se puede realizar el diseño del comportamiento de las transacciones en cuanto al nivel de aislamiento entre ellas.

Otro de los parámetros es el de marcar una transacción como de solo lectura. Los gestores de base realizan algunas optimizaciones en las transacciones que son de solo lectura. Al marcar la transacción como solo lectura le abre las puertas al gestor para realizar cualquier tipo de optimización a la transacción. Como esto se aplica cuando una transacción comienza solo tiene sentido en métodos cuyo comportamiento de propagación le permita iniciar una transacción (*PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW*). Además en el caso de la propuesta donde se usa Hibernate como mecanismo persistente, al marcar una transacción como solo lectura se evitan sincronizaciones de objetos innecesarias, dejándose todas para el final de la transacción.

Algunas transacciones se pueden tornar largas y de acuerdo a sus políticas de aislamiento pueden crear caos en la base de datos, pues si implican el bloqueo de varias filas o de tablas pueden crear grandes atascamientos. Para evitar esto, se le asigna a la transacción un tiempo de expiración, de manera que transcurrido este tiempo si no ha concluido se realiza un roll back. Como el conteo de tiempo comienza cuando se inicia una transacción esto solo tiene sentido en los métodos que pueden hacerlo de acuerdo a su comportamiento de propagación (*PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW*).

Para tratar las transacciones con Spring se utilizan las siguientes clases:

- *TransactionAttributeSourceAdvisor,* esta clase en resumen encierra dentro las funciones que son ejecutadas para manejar la transacción y los puntos dentro de la ejecución de la aplicación donde se deben aplicar, es decir los métodos que deben ser transaccionales y el momento de ejecución (ante, durante o después de la ejecución) .
- *TransactionInterceptor*, es referenciado dentro de la clase *TransactionAttributeSourceAdvisor* y va a contener el manejador a utilizar, en el caso de la propuesta *HibernateTransactionManager*, para coordinar las transacciones con el mecanismo de persistencia. Además va a contener el atributo

transactionAttributeSource donde se especifica las políticas transaccionales a aplicar y los métodos donde se va a aplicar, se definen los atributos de las transacciones de cada método (Comportamiento de propagación, Nivel de aislamiento, etc.).

- *DefaultAdvisorAutoProxyCreator*, esta clase es la encargada de crear los proxies automáticamente a todos los métodos que presenten atributos de transacción, a los cuales se les va a aplicar el aspecto, o sea el código que ejecuta la transacción.
- *HibernateTransactionManager* es la clase manejadora de las transacciones que sirve de interfaz con la plataforma de implementación de estas.

Las tres primeras de estas clases son parte de las librerías de Spring para facilitar el trabajo con la programación orientada a aspectos.

Con estas clases basta para gestionar las transacciones de una aplicación por muy grande que sea, las mismas se configuran el fichero xml de Spring, y utilizando la misma filosofía de definición de beans que se analiza en la gestión de los objetos de negocio.

Primero se declara el manejador de transacciones que corresponde a la clase *HibernateTransactionManager*, se le especifica la propiedad *sessionFactory* que debe ser la misma que está utilizando Hibernate para gestionar el acceso a datos, como se analiza en la sección **Capa de Acceso a Datos**, esta *sessionFactory* se define en el fichero de configuración de Spring y por esta razón se referencia desde el bean manejador de transacciones (*transactionManager)* (fig 3.22).

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate.HibernateTran
sactionManager">
  <property name="sessionFactory">
    <ref bean="sessionFactory"/>
  </property>
</bean>
```

Fig 3.22 Configuración del bean manejador de transacciones

Seguidamente se define en el mismo fichero el *TransactionAttributeSourceAdvisor*. Esta clase contiene un constructor al cual se le pasa como parámetro un objeto *TransactionInterceptor*, es por eso que se declara un argumento de constructor y se le pasa una referencia al bean *transactionInterceptor*, de esta manera se inicializa el bean *transactionAdvisor* utilizando la inyección de instancias (fig 3.23).

```
<bean id="transactionAdvisor"
class="org.springframework.transaction.interceptor.Tran
sactionAttributeSourceAdvisor">
 <constructor-arg>
   <ref bean="transactionInterceptor"/>
 </constructor-arg>
</bean>
```

Fig 3.23 Configuración del advisor

En la definición del bean *transactionInterceptor*, se inicializa la propiedad *transactionManager* haciendo referencia a este bean, se especifican cada uno de los métodos que se ejecutarán en un ambiente transaccional y los atributos de transacción de cada uno. Se utiliza un mapeo del nombre completo (incluyendo los paquetes) de la clase y el nombre del método para lo cual se pueden usar comodines y de esta forma agrupar varios, seguido se coloca el signo '=' y se especifican separados por coma cada uno de los atributos transaccionales de los mismos (fig 3.24).

```
<bean id="transactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionInter
ceptor">
 <property name="transactionManager">
   <ref bean="transactionManager"/>
 </property>
 <property name="transactionAttributeSource">
   <value>
     model.beans.GestionUsuario.cargar*=PROPAGATION_SUPPORTS,
readOnly
    model.beans.GestionUsuario.SALVAR*=PROPAGATION_REQUIRED
   </value>
 </property>
</bean>
```

Fig 3.24 Configuración del interceptor

Por último se declara el *DefaultAdvisorAutoProxyCreator,* el cual buscará por todo el contexto de la aplicación los *advisors* que existan y creará los proxy para cada uno de los métodos en los que se vaya a aplicar el aspecto, en este caso encuentra el *transactionAdvisor*, y crea un proxy (fig 3.25).

```
<bean id="autoproxy"
class="org.springframework.aop.framework.autoproxy.De
faultAdvisorAutoProxyCreator">
 …
</bean>
```

Fig 3.25 Configuración del proxy

De esta forma se configuran las transacciones declarativamente en la propuesta de arquitectura.

Gestión de Seguridad

La seguridad es un aspecto esencial en la arquitectura que a veces no se le da la importancia requerida. Generalmente la implementación de las políticas de seguridad es una tarea tediosa y resulta muchas veces en código ligado con las funciones de negocio. El framework Acegi combinado con programación orientada a aspectos (AOP) y la inyección de instancias de Spring (IoC) brinda un mecanismo de seguridad potente que permite definirlo de manera declarativa.

La gestión de la seguridad lleva consigo dos procesos:

- Manejo de la autenticación, consiste en determinar la identidad del usuario, generalmente mediante un nombre de usuario y una contraseña.
- Control del acceso, consiste en determinar si un usuario tiene acceso a un determinado recurso seguro.

Para manejar la autenticación este framework provee la interfaz *AuthenticationManager* que resulta sencilla de implementar (fig 3.26).

```
public interface AuthenticationManager{
  public Authentication authenticate(Authentication authentication)
         thows AuthenticationException;
}
```

Fig 3.26 Interfaz que usa Acegi para el manejo de la autenticación

Aunque, para evitar este trabajo, Acegi contiene la clase *ProviderManager* como una implementación de dicha interfaz. Esta clase es un manejador de autenticación que delega responsabilidades en otros manejadores específicos que permiten autenticar al usuario contra diferentes fuentes de recursos.

Entre estos manejadores sobresalen:

Manejador	Descripción
DaoAuthenticationProvider	Diseñado para obtener los datos del usuario de una base de datos.

RemoteAuthenticationProvider	Para la autenticación contra un servicio remoto.
CasAuthenticationProvider	Autenticación contra servicio central de autenticación (CAS)

En la propuesta se va a analizar la autenticación utilizando una base de datos por ser una de las más utilizadas.

El *DaoAuthenticationProvider* usa un objeto de una clase de acceso a datos (DAO) para recuperar la contraseña dado el nombre del usuario. Con el nombre de usuario y la contraseña provenientes de la base de datos el *DaoAuthenticationProvider* ejecuta la autenticación comparándolos con el usuario y contraseña pasados por el *ProviderManager* en el objeto *Authentication* (fig 3.27).

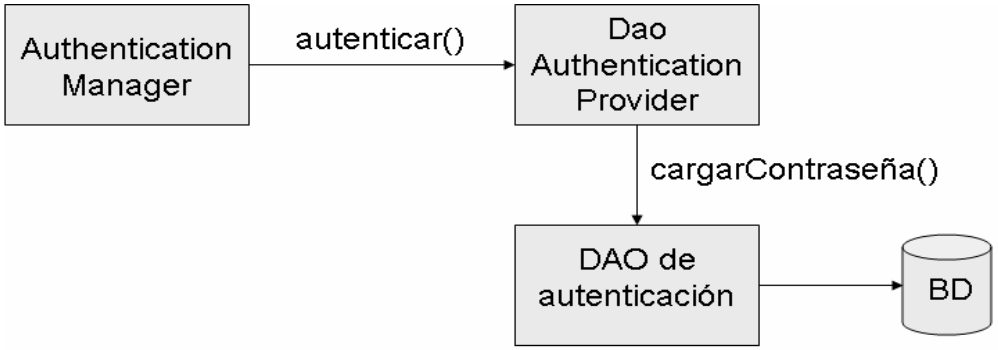


Fig 3.27 Relación entre las clases encargadas de realizar el manejo de la autenticación

La configuración de la seguridad se realiza en el fichero de configuración de Spring. Lo primero que se declara es el manejador de la autenticación, se le especifica en la propiedad *providers* los manejadores específicos que se utilizarán (fig 3.28).

```
<bean id="authenticationManager"
class="net.sf.acegisecurity.providers.ProviderManager">
<property name="providers">
<list>
<ref bean="daoAuthenticationProvider"/>
</list>
</property>
</bean>
```

Fig 3.28 Configuración del manejador de autenticación

Al manejador específico, se le declara en la propiedad *authenticationDao* la clase que va a ejecutar la consulta a la base de datos para recuperar la contraseña del usuario (fig 3.29). Acegi provee implementaciones para esta clase pero unas se rigen por un específico diseño de las tablas de la base de datos donde se almacena la información del usuario y otras demandan especificar la consulta SQL en la declaración del xml. Para evitar esta rigidez se puede implementar esta clase y así utilizar las funcionalidades de acceso a datos que brinda Hibernate.

```
<bean id="daoAuthenticationProvider"
class="net.sf.acegisecurity.providers.dao.DaoAu
thenticationProvider">
<property name="authenticationDao">
<ref bean="authenticationDao"/>
</property>
</bean>
```

Fig 3.29 Configuración del provider

La clase solo necesita implementar la interfaz *UserDetailsService* que contiene un solo método, *loadUserByUsername* el cual devuelve un objeto del tipo *UserDetails*. Como *UserDetails* es una interfaz, se utiliza la clase de Acegi *User* que la implementa. Esta clase contiene el nombre de usuario, la contraseña, un bool que indica si el usuario está activo o no y un arreglo del tipo *GrantedAuthority*, clase del framework que representa los roles o permisos que contiene el usuario.

El otro proceso dentro de la gestión de la seguridad es el control del acceso, el mismo decide si el usuario puede acceder a un determinado recurso o no. Similar a la autenticación, para este Acegi provee la interfaz manejadora de acceso *AccessDecisionManager* (fig 3.30).

```
public interface AccessDecisionManager{
  public void decide(Authentication authentication, Object
object, ConfigAttributeDefinition config)
  throws AccessDeniedException;
  public boolean supports(ConfigAttribute attribute);
  public boolean supports(Class clazz);
}
```

Fig 3.30 Declaración de la interfaz manejadora de acceso

Acegi presenta tres implementaciones de dicha interfaz que cubren casi cualquier tipo de situación en las aplicaciones. Estos manejadores de acceso son los responsables los derechos de acceso a un usuario autenticado, sin embargo ellos no toman esta decisión directamente. Se auxilian en uno o más objetos los

cuales votan cada uno si el usuario está autorizado a acceder al recurso o no. Una vez que todos votaron el manejador de acceso analiza el resultado y toma una decisión. Estos manejadores de acceso son los siguientes:

Manejadores de acceso	Descripción
AffirmativeBased	Permite el acceso si al menos uno de los votantes votó permitiendo el acceso.
ConsensusBased	Permite el acceso si la mayoría de los votantes votó permitiendo el acceso.
UnanimousBased	Permite el acceso si todos los votantes votaron permitiendo el acceso.

El manejador de acceso se declara en el fichero de configuración de Spring, se le especifica en la propiedad *decisionVoters* los objetos que van a ser los votantes. El trabajo de estos objetos es analizar los permisos que contiene el usuario y los requeridos para acceder a un recurso seguro, se pueden construir los objetos implementando la interfaz *AccessDecisionVoter* o se puede utilizar la clase *RoleVoter* que contiene Acegi, este objeto vota cuando el recurso seguro presenta un atributo de configuración que comienza con ROLE_(fig 3.31).Este objeto compara todos los atributos de configuración que presenta un recurso que comienza con ROLE_ con todos los permisos de acceso que presenta el usuario(contenidos en el arreglo de objetos *GrantedAuthority*), si alguno coincide autoriza el acceso de lo contrario lo deniega.

```
<bean id="accessDecisionManager"
class="net.sf.acegisecurity.vote.UnanimousBased">
 <property name="decisionVoters">
  <list>
    <ref bean="roleVoter"/>
  </list>
 </property>
</bean>
<bean id="roleVoter"
class="net.sf.acegisecurity.vote.RoleVoter"/>
```

Fig 3.31 Configuración del manejador de acceso y los objetos votantes

Hasta ahora se ha configurado la autenticación y el acceso a los recursos seguros. Como la propuesta de arquitectura es de una aplicación web, estos procesos van a ser aplicados desde filtros. Los filtros son los encargados de interceptar las solicitudes a recursos web y aplicarles los procesos de seguridad, entre ellos pasarles las solicitudes al manejador de autenticación y de acceso.

Acegi provee de un conjunto de filtros que permiten implementar a cualquier nivel la política de seguridad, entre ellos se encuentran:

Filtros	Descripción
ChannelProcessingFilter	Es el encargado de analizar si la solicitud necesita ser entregada en canal seguro (HTTPS) o inseguro (HTTP), redireccionando para el canal adecuado.
AuthenticationProcessingFilter	Es el encargado de procesar la autenticación basada en un formulario como vía de entrada de los datos.
AuthenticationProcessingFilter EntryPoint	Es el encargado de impulsar al usuario a la autenticación, específicamente redireccionar al usuario a una página HTML con un formulario de login.
FilterSecurityInterceptor	Intercepta las solicitudes, utiliza el manejador de autenticación para determinar cuando un usuario está logueado y utiliza el manejador de acceso para determinar cuando un usuario tiene permiso para acceder a un recurso. En él se especifican los permisos necesarios para acceder a cada uno de los recursos web de la aplicación.

Gracias al contenedor de Spring y el uso de programación orientada a aspectos, es posible realizar toda la configuración de estos filtros en el fichero de configuración, a diferencia de cuando se implementan filtros sin Spring que lleva a declararlos en el fichero *web.xml* de la aplicación.

Para lograr esto se utiliza la clase *FilterChainProxy* la cual va a mapear cada uno de los filtros con el patrón de URL que estos interceptarán evitando declarar para cada uno las entradas <*filter-mapping*> y <*filter*> en el archivo *web.xml* de la aplicación. Para configurar todos los filtros se declara primeramente la clase *FilterChainProxy*, a la cual se le especifica en el atributo *filterInvocationDefinitionSource* el patrón de URL y los filtros que interceptarán las solicitudes que cumplan con dichos patrones. El orden en que se coloquen los filtros significará el orden en que interceptarán la solicitud y luego se desencadenarán.

Normalmente cuando se trabaja con expresiones regulares para referenciar un determinado patrón de comportamiento se señala que previo de la comparación se convierta la expresión a minúscula para evitar confusión y de esta forma utilizar todas las cadenas en la expresión regular en minúsculas y además que se utilice el tipo de patrón de Apache Ant puesto que es el que resulta más familiar (fig 3.32).

```
<bean id="filterChainProxy"
  class="org.acegisecurity.util.FilterChain.Proxy">
  <property name="filterInvocationDefinitionSource">
   <value>
    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
    PATTERN_TYPE_APACHE_ANT
/**channelProcessingFilter,authenticationProcessingFilter,au
thenticationProcessingFilterEntryPoint,filterInvocationInter
ceptor
   </value>
  </property>
</bean>
```

Fig 3.32 Configuración de la clase FilterChainProxy

Seguidamente se declara cada uno de los filtros. En la declaración del *ChannelProcessingFilter*, se le especifica en la propiedad *filterInvocationDefinitionSource* cada una de las URL de las solicitudes y el canal que requieren (seguro o inseguro), para ello se utilizan expresiones regulares (fig 3.33).

```
<bean id="channelProcessingFilter"
class="org.acegisecurity.securechannel.ChannelProcessingFilter">
  <property name="channelDecisionManager">
    <ref local="channelDecisionManager"/>
  </property>
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /login.do**=REQUIRES_SECURE_CHANNEL
      /cambiarcontrasenna.do**=REQUIRES_SECURE_CHANNEL
      /j_acegi_security_check= REQUIRES_SECURE_CHANNEL
      /**=REQUIRES_INSECURE_CHANNEL
    </value>
  </property>
</bean>
```

Fig 3.33 Configuración del ChannelProcessingFilter

En la propiedad *channelDecisionManager* se referencia a un bean de la clase *ChannelDecisionManagerImpl*, esta clase de Acegi delega en los manejadores de canal

(*SecureChannelProcessor* o *InsecureChannelProcessor*) para que examinen la solicitud y sus requerimientos de seguridad y de acuerdo a ello redireccionarla al canal adecuado (fig 3.34).

```
<bean id="channelDecisionManager"
class="org.acegisecurity.securechannel.ChannelDecisionMan
agerImpl">
  <property name="channelProcessors">
    <list>
      <ref local="secureChannelProcessor"/>
      <ref local="insecureChannelProcessor"/>
    </list>
   </property>
</bean>
<bean id="secureChannelProcessor"
class="org.acegisecurity.securechannel.SecureChannelProce
ssor"/>
<bean id="insecureChannelProcessor"
class="org.acegisecurity.securechannel.InsecureChannelPro
cessor"/>
```

Fig 3.34 Configuración del ChannelDecisionManager

En el *AuthenticationProcessingFilter* se referencia en la propiedad *authenticationManager* al manejador de autenticación que se está utilizando, en la propiedad *authenticationFailureUrl* la URL a donde se redireccionará en caso de que falle la autenticación y en *filterProcessesUrl* la URL a la que se envía el formulario de autenticación (fig 3.35).

```
<bean id="authenticationProcessingFilter"
class="org.acegisecurity.ui.webapp.AuthenticationProcessin
gFilter">
  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
  </property>
  <property name="authenticationFailureUrl">
    <value>/login.do?login_error=1</value>
  </property>
  <property name="filterProcessesUrl">
    <value>/j_acegi_security_check</value>
  </property>
</bean>
```

Fig 3.35 Configuración del AuthenticationProcessingFilter

Cuando se declara el *AuthenticationProcessingFilterEntryPoint* se le especifica en la propiedad *loginFormUrl* la dirección de la página que contiene el formulario de autenticación y en la propiedad *forceHttps* se le da valor *true* para señalar que se use canal seguro cuando se trate la solicitud de autenticación (fig 3.36).

```
<bean id="authenticationProcessingFilterEntryPoint"
class="org.acegisecurity.iu.webapp.AuthenticationProc
essingFilterEntryPoint">
 <property name="loginFormUrl">
   <value>/login.do</value>
 </property>
 <property name="forceHttps">
   <value>true</value>
 </property>
</bean>
```

Fig 3.36 Configuración del AuthenticationProcessingFilterEntryPoint

Por último en el *FilterSecurityInterceptor* se especifica en la propiedad *authenticationManager* el manejador de autenticación que se utiliza, en la *accessDecisionManager* se referencia el manejador de acceso y por último en la propiedad *objectDefinitionSource* se declara, auxiliándose de expresiones regulares, las URL de las solicitudes y los roles o permisos para acceder a estas (fig 3.37).

```
<bean id="filterInvocationInterceptor"
class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager">
    <ref bean="authenticationManager">
  </property>
  <property name="accessDecisionManager">
    <ref local="httpRequestAccessDecisionManager"/>
  </property>
  <property name="objectDefinitionSource">
    <value>
       CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
       PATTERN_TYPE_APACHE_ANT
     /configuracion.xml**=ROLE_ACCESO_DENEGADO
     /**=ROLE_ACCESO_DENEGADO
    </value>
   </property>
</bean>
```

Fig 3.37 Configuración del FilterSecurityInterceptor

De esta forma se configura la seguridad de la aplicación de una manera robusta, declarativa y no intrusiva que permite que en los proyectos, por muy grande que sea esta tarea, pueda ser llevada a cabo por pocas personas, además con total independencia del desarrollo de los procesos de negocio. En este caso se implementan las políticas de seguridad basadas en filtros de solicitudes, si se desea querer reforzarla con otras medidas se pueden llevar estas políticas hasta nivel de métodos y no se diferenciaría mucho en lo que se ha presentado anteriormente.

Configuración del Apache Tomcat

El contenedor web Tomcat constituirá el entorno de ejecución de la propuesta, además de que va a gestionar la reserva de conexiones exponiéndolas como un recurso JNDI.

Para configurar el Tomcat es necesario editar el fichero *server.xml* que se encuentra en el directorio de instalación de este, específicamente dentro de la carpeta *conf*. En este archivo dentro de *Host* se agrega el elemento <*Context*> que representa el contexto de la aplicación que será cargado por el Tomcat. Se le especifica en el atributo *docBase* el camino hasta la carpeta *WebContent* de la aplicación donde se

encuentran todos los recursos de la misma. Además, en el atributo *path* se especifica el nombre por el que será llamada la aplicación para ser cargada por el navegador (fig 3.38).

```
<Host appBase="webapps" name="localhost">
  <Context docBase="D:/MiAplicacion/WebContent"
   path="/MiAplicacion"/>
</Host>
```

Fig 3.38 Configuración del contexto de la aplicación en el Tomcat

Con esta declaración ya el Tomcat es capaz de cargar la aplicación. Para configurar la reserva de conexiones es necesario declarar dentro de <*Context*> el elemento <*Resource*> que representa un recurso JNDI al cual se le especifica el nombre por el cual va a ser localizado y el tipo de dato que representa en los atributos *name* y *type* respectivamente. Seguidamente se especifican los parámetros del recurso declarando un elemento <*ResourceParams*>. Los parámetros que se le declaran son los siguientes (fig 3.39).

Parámetros	Descripción
driverClassName	Clase del driver de conexión
url	URL de conexión a la base de datos
username	Usuario de la base datos
password	Contraseña del usuario de la base de datos
maxIdle	Máxima cantidad de conexiones que pueden estar inactivas en la reserva.
maxActive	Máxima cantidad de conexiones que pueden estar activas.
maxWait	Máximo número de milisegundos que la reserva va a esperar por una conexión antes de lanzar una excepción.

```
<Context docBase="D:/MiAplicacion/WebContent"
  path="/MiAplicacion">
  <Resource auth="Container" name="jdbc/MiFuentedeDatos"
type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/MiFuentedeDatos">

<parameter><name>url</name><value>jdbc:oracle:thin:@10.3.5.101:1521:Mi
BasedeDatos</value></parameter>
<parameter><name>maxIdle</name><value>10</value></parameter>
<parameter><name>maxActive</name><value>20</value></parameter>
<parameter><name>driverClassName</name><value>oracle.jdbc.driver.Oracl
eDriver</value> </parameter>
<parameter><name>maxWait</name><value>40</value> </parameter>
<parameter><name>username</name><value>MiUsuario</value> </parameter>
<parameter><name>password</name><value>saren</value> </parameter>

</ResourceParams>
</Context>
```

Fig 3.39 Configuración del repositorio de conexiones

De esta manera se configura el contenedor web Apache Tomcat para que cuando levante ejecute la aplicación, cree un repositorio de conexiones y permita acceder al mismo desde la aplicación.

Conclusiones

La combinación de diferentes componentes, herramientas y frameworks que implementan patrones bien definidos en un diseño bien estructurado puede aportar grandes beneficios:

- Logrando una mayor agilidad e independencia en la implementación al contar con poderosas funcionalidades.
- Permitiendo la abstracción de problemas complejos en las aplicaciones.
- Sirviendo de guía en buenas prácticas de programación.
- Permitiendo que grandes tareas puedan ser llevadas a cabo por reducidos grupo de personas.
- Posibilitando el cumplimiento de pilares arquitectónicos como la escalabilidad, robustez, portabilidad.

Capítulo 4: Análisis de los resultados alcanzados con la propuesta.

En el capítulo anterior se expusieron todos los elementos que conforman la propuesta arquitectónica y las razones por las cuales se escogieron los mismos. En el presente capítulo se realiza un análisis de los resultados alcanzados con la propuesta que está basado en la comparación de la misma con arquitecturas utilizadas en diferentes proyectos.

4.1 Sistema Encuestas

El sistema Encuestas, proyecto que surge por idea de la Dirección de la Infraestructura Productiva de la universidad para gestión de los recursos humanos, tenía el objetivo de realizar un levantamiento de los estudiantes y clasificarlos en niveles de acuerdo a sus conocimientos respecto a las herramientas, tecnologías, experiencia en proyectos, entre otros. Los resultados que arrojó la implantación de este sistema y su realización por los estudiantes, fueron tomados a la hora de ubicar a los mismos en un determinado rol dentro de un proyecto.

A continuación se presenta la arquitectura que se utilizó en el Sistema Encuestas (fig 4.1):

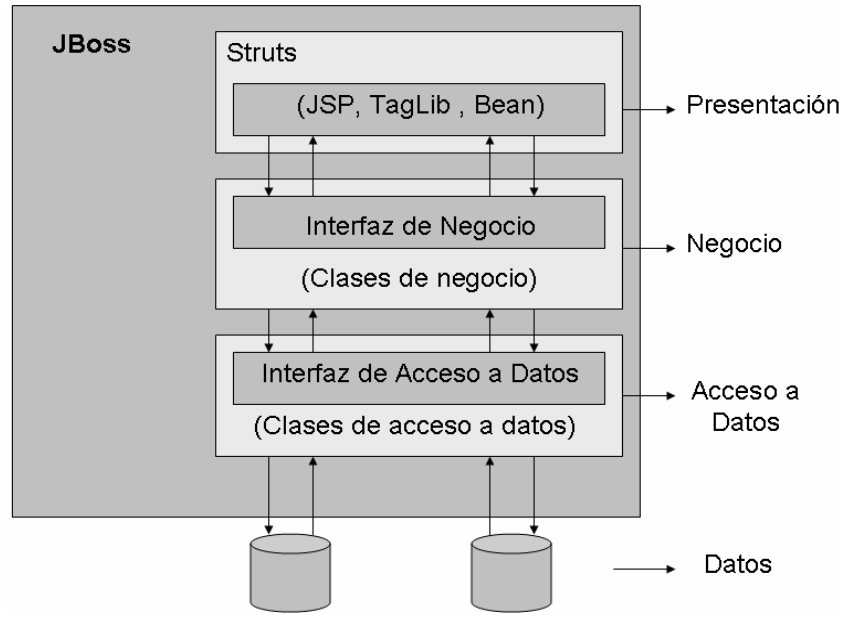


Fig 4.1 Arquitectura utilizada en el sistema Encuestas

Capa de Presentación

En esta capa se ubicó el framework Struts, para gestionar la vista y control dentro del patrón MVC. Para la presentación se utilizaron páginas JSP y un conjunto de librerías de etiquetas que ayudaron a manejar el contenido dentro de las mismas. Los formularios de datos fueron representados por beans(*ActionForms*) los cuales se utilizaron para encapsular los datos que se entran en los mismos, manejar la validación de los campos y a través de *ActionErrors* enviar los mensajes de error a las páginas JSP a través de etiquetas. Los mensajes de error se cargaron del fichero *.properties*, el cual se encarga de gestionar todos los textos de la aplicación. Para el control se utilizó el servlet controlador de Struts (*ActionServlet*), el cual maneja todas las solicitudes direccionándolas hacia las acciones apropiadas (*Action*) donde se implementa la lógica de presentación e interactúa con la capa de negocio. Se utilizaron los *ActionsMapping* que brinda el framework para definir las clases *Action* con sus *ActionsFroward*, excepciones y relación con los formularios en el *struts-config.xml*. A la hora de definir la navegación de la aplicación ya sea hacia una página JSP u otra funcionalidad, se usaron los *ActionsForward*.

Ventajas

- Se implementa un diseño MVC en poco tiempo:

  Este framework, al igual que JSF, tiene un conjunto de elementos que implementan patrones de diseño para la capa de presentación, tal es el caso del *FrontController* que es implementado por el servlet controlador, entre otros, esto hace posible que haya un desarrollo de buenas prácticas, exista una correcta asignación de responsabilidades y separación de papeles. Estos elementos tienen implementadas funcionalidades que permiten el manejo de la vista y el control de la aplicación.
- El desarrollo de la capa de presentación es más rápido que en las variantes donde no se usa framework:

  El nivel de rapidez es mayor cuando se utilizan frameworks como Struts y JSF que cuando no se usan, pues estos traen consigo una serie de elementos y librerías de clases que tienen funcionalidades implementadas que agilizan el desarrollo de esta capa.
- Se obtienen los beneficios del framework como el manejo de errores, la internacionalización, validación, entre otros:

  Struts, al igual que JSF, trae consigo elementos que tienen implementadas funcionalidades para el manejo de la validación, la internacionalización, los errores que ahorrarían tiempo de realizarlos a mano.

Desventajas

- Carencia de una infraestructura de componentes de presentación y manejo de eventos:
  Es importante contar con mecanismos que brinden un manejo de eventos pues esto permite que, a través de un cambio de comportamiento en los componentes, poder gestionar alguna funcionalidad ya sea en la presentación así como también en la lógica de negocio de la aplicación. Estos mecanismos los trae bien implementados el framework JSF que se propone para esta capa.
- Alto acoplamiento de los formularios (*ActionForms*) y las acciones (*Action*) con el framework:
  Existe una gran dependencia entre los beans de formulario y las acciones con el framework, esto imposibilita la reutilización de estas clases fuera del ámbito de Struts. Lo contrario sucede en la propuesta pues los beans de respaldo de JSF están desacoplados del framework y pueden usarse en cualquier otro lugar.

Capa de Negocio

En la capa de negocio del sistema de Encuestas se utilizaron interfaces de negocio que exponen las funcionalidades del mismo. Dichas funcionalidades son implementadas en clases java. Desde los métodos de las clases de negocio se accede a la interfaz que expone las funcionalidades de acceso a datos.

Capa de Acceso a Datos

En la capa de acceso a datos del sistema de Encuestas se utilizaron interfaces de que exponen las funcionalidades de la misma. Las clases que implementan estas funcionalidades utilizaron el API JDBC, realizando la interacción con la base de datos codificando sentencias SQL.

Desventajas

- Se utilizaron numerosas y extensas sentencias SQL utilizando el API JDBC dentro de las clases de acceso a datos que fueron costosas de implementar. Si se hubiera utilizado el framework Hibernate para el manejo de la persistencia, este proceso hubiera sido más fácil y robusto.

Capa de Datos

Se utilizó como gestor de bases de datos Access.

Seguridad

- Carencia de mecanismos para el manejo de filtros y la seguridad de la aplicación:

En el sistema de Encuestas la declaración de los filtros de la aplicación fue trabajosa al no disponer de frameworks con grandes librerías de funciones para el manejo de los mismos como Acegi, condujo a declararlos a mano y en el fichero *web.xml*.

Entorno de ejecución

Se utilizó como entorno de ejecución el servidor de aplicaciones JBoss producto que la cantidad de usuarios que podían conectarse simultáneamente era grande. Se pretendió lograr soporte para múltiples conexiones de usuarios utilizando un servidor más potente ejecutándose en una máquina con grandes recursos. En cambio se podría haber utilizado Tomcat ya que no se necesitaba soporte para los API EJB, ni JMS; y se hubiera logrado el rendimiento necesario clusterizándolo con máquinas que presentaran menos recursos.

Pilares de una arquitectura empresarial

Analizando el comportamiento de la arquitectura del sistema de Encuestas y comparándola con la propuesta de acuerdo a los pilares con los que debe cumplir una arquitectura empresarial se puede decir lo siguiente:

Pilares	Sistema Encuestas	Propuesta
Portabilidad	Esta arquitectura pierde portabilidad pues las líneas de código JDBC que están en las clases de acceso a datos están destinadas a acceder al gestor en Access, puede ser que posteriormente se necesite cambiar esta fuente de datos por otra, pero como ese código solo permite la comunicación con este gestor habría que cambiar las instrucciones para poder interactuar con la nueva fuente de datos.	Si en la capa de acceso a datos se hubiera ubicado el framework Hibernate como en la propuesta, la aplicación ganaría en portabilidad pues el mismo permite migrar de fuente de datos sin que se vea afectado el código, lo único que hay que cambiar es el driver, dialecto de la base de datos, entre otros elementos de configuración.
Reusabilidad	Su reusabilidad se ve afectada en los beans de formulario y en las	Lo contrario sucede en la propuesta al usar JSF como framework para la

	acciones utilizadas para el manejo de la capa de presentación, puesto que las mismas están muy ligadas al framework, extienden comportamiento de clases del mismo y en un ámbito donde no se encuentre presente Struts no sería posible utilizarlas.	capa de presentación, el mismo cuenta con beans de formularios que son independientes del framework y pueden ser usados por cualquier otro sistema.
Fácil de probar	El sistema tiene la propiedad de ser fácil de probar pues se pueden realizar casos de prueba que se validan contra la presentación, la interfaz de negocio y la interfaz de acceso a datos de la aplicación.	

4.2 Proyecto SAFRE

El proyecto SAFRE estuvo basado en la automatización de un sistema de gestión de seguros sociales para una empresa mexicana. Desde el principio se planteó como requerimiento del cliente que se desarrollara en la plataforma J2EE. El sistema era de complejidad alta y la arquitectura fue definida por el cliente. Se propuso la utilización del framework Expreso el cual contiene dos partes, una encargada de la parte de presentación implementando el MVC y la otra es un mecanismo de persistencia basado en mapeo objeto/relacional. Además se utilizaron componentes EJB para implementar la lógica de negocio.

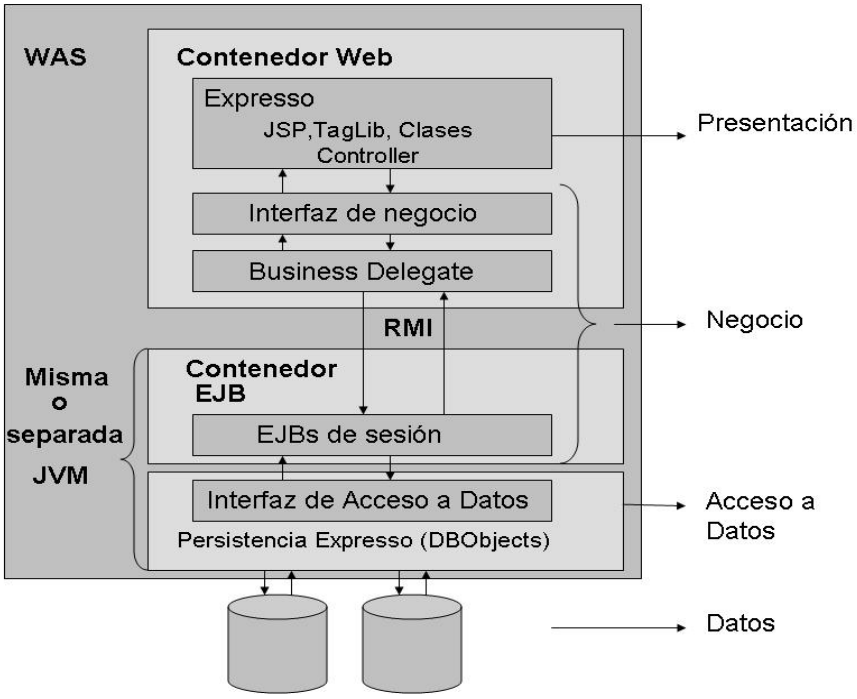


Fig 4.2 Arquitectura utilizada en el proyecto Safre

Capa de Presentación

En la capa de presentación se utilizó Expresso, este framework está basado en Struts, utiliza JSP para las vistas con librerías de etiquetas, algunas de ellas presentes en Struts. Presenta manejo de errores y de internacionalización de recursos. La lógica de presentación fue ejecutada en las clases *Controller* y desde aquí se interactuó con la capa de negocio. Presenta un fichero de configuración al igual que Struts donde se configura la navegación y las clases formularios.

Ventajas

- Se implementa un diseño MVC en poco tiempo:

  Este framework, al igual que JSF, tiene un conjunto de elementos que implementan patrones de diseño para la capa de presentación y hacen posible que haya un desarrollo de buenas prácticas, exista una correcta asignación de responsabilidades y separación de papeles. Estos elementos tienen implementadas funcionalidades que permiten el manejo de la vista y el control de la aplicación.
- El desarrollo de la capa de presentación es más rápido que en las variantes donde no se usa framework:

El nivel de rapidez es mayor cuando se utilizan frameworks como Expresso y JSF que cuando no se usan, pues estos traen consigo una serie de elementos y librerías de clases que tienen funcionalidades implementadas que agilizan el desarrollo de esta capa.

Desventajas

- Este framework presenta muy poco soporte y documentación en la comunidad internacional:
  Actualmente el framework Expresso cuenta con muy poca documentación para su uso y herramientas que agilicen el desarrollo con el mismo. No sucede así con el framework JSF de la propuesta.
- Las clases *Controller* que realizan la lógica de presentación están acopladas al framework:
  Existe una gran dependencia de las clases controladoras para manejar la lógica de presentación con el framework, las mismas extienden de clases del mismo. Todo esto trae consigo que en un entorno donde no esté presente Expresso estas clases controladoras no podrían funcionar. Esto no sucede así en la propuesta.

Capa de Negocio

En esta capa se utilizaron componentes EJB de sesión, pues se requería distribuir la aplicación en varios servidores físicos y así ganar en escalabilidad. Se utilizó una interfaz de negocio implementada por un *Business Delegate* encargado de abstraer los detalles de la invocación de los métodos remotos en los EJB, este a su vez utiliza un *Service Locator* que realiza la localización del componente en la red. Los EJB de sesión implementan la lógica de negocio e interactúan con la capa de acceso a datos.

Ventajas

- Se logró una clara separación entre la capa de negocio y la capa de presentación:
  La implementación de los métodos de lógica de negocio de la aplicación era manejada por los EJBs de sesión, estos métodos se exponían en una interfaz de negocio implementada por un *Business Delegate* que se apoyaba en un *Service Locator* para la localización de los componentes. Desde las clases controladoras se accedía a esta interfaz de negocio. Aquí se logró una clara separación de papeles pues cualquier cambio en los métodos dentro de los componentes EJBs no afectaba la lógica de presentación de la aplicación. En la propuesta se logra también esta separación pues desde los métodos de los beans se accede a los de la interfaz de negocio y los mismos son implementados por las clases de negocio.

- Los componentes distribuidos lograban distribuir la carga en varios servidores físicos:
  Los componentes EJBs, teniendo la posibilidad de ejecutarse en diferentes contenedores EJBs localizados en distintas máquinas, pudieron distribuir las funcionalidades de la capa de negocio y aumentar el rendimiento de la aplicación.
- Se implementaron patrones que mejoraban el diseño y el rendimiento de la capa (*Business Delegate*, *Service Locator*):
  La utilización de un *Business Delegate* como una abstracción de negocio para reducir el acoplamiento entre la capa de presentación y los servicios de negocio, así como para ocultar los detalles de la implementación del mismo trajo como resultado buenas prácticas de diseño, limpieza, aumento de las facilidades de hacer cambios en las funcionalidades de negocio sin que se viera afectada la lógica de presentación. En el caso de la implementación del *Service Locator* esto trajo consigo que se pudiera reutilizar el mismo para reducir la complejidad del código de localización de componentes en la red, la duplicación del mismo y el consumo de demasiados recursos. En la propuesta el contenedor de Spring se encarga del manejo de los objetos de esta capa realizando la inyección de dependencias.

Desventajas

- La implementación de esta capa fue lenta y difícil de probar:
  Al encontrarse los componentes que manejan las funcionalidades de la lógica de negocio distribuidos por la red era engorroso realizarle pruebas. Resultó lenta la implementación de esta capa pues por cada componente EJB se necesitó implementar dos interfaces (*Home,Remote*), la clase del EJB y un fichero xml descriptor de despliegue. Sin embargo la propuesta no es difícil de probar pues se pueden realizar casos de prueba y verificarlos contra la interfaz de negocio. También el desarrollo de esta capa, auxiliándose de las potencialidades de Spring para el manejo de los objetos de negocio, resulta más rápido que la implementación de esta capa en la aplicación de Safre.

Capa de Acceso a Datos

Esta capa contiene una interfaz que es implementada por clases que utilizan las funcionalidades de Expresso. Expresso provee un mecanismo de mapeo objeto/relacional a partir de la herencia de la clase *DBObject* y la implementación de algunos métodos para describir su relación con la tabla de la base de datos. Luego de esto dichos objetos tienen la propiedad de ser persistentes, permitiendo realizar

operaciones comunes con la base de datos con un mínimo de esfuerzo. Permite realizar consultas complejas. Es un mecanismo transparente al gestor de base de datos.

Ventajas

- Se agilizó el desarrollo de la capa de acceso a datos evitando codificar sentencias SQL:
  Desde las clases de acceso a datos fue posible el trabajo con los mismos a través de un lenguaje orientado a objetos, gracias a las funcionalidades que brindó Expresso, sin necesidad de usar código SQL. En el caso de la propuesta en las clases de acceso a datos utilizando las plantillas de Spring y extendiendo las funcionalidades de la clase *HibernateDaoSupport* es posible realizar la interacción con la base de datos sin usar código SQL.

Desventajas

- No es un mecanismo transparente, los objetos persistentes tuvieron que heredar de clases del framework e implementar métodos específicos. En la propuesta sin embargo los objetos persistentes están desacoplados del framework y desconocen que tienen la propiedad de persistir. Pueden navegar además por todas las capas de la aplicación.
- El mapeo de los objetos persistentes con las tablas de la base de datos se realizó en el código de la clase. En la propuesta el mapeo de los objetos persistentes no se realiza en el código de la aplicación, sino que se hace dentro de ficheros xml de mapeo, provocando limpieza en el código de la misma.
- Presenta poco soporte y documentación en la comunidad java.

Transacciones

En esta arquitectura el manejo de las transacciones fue implementado declarativamente utilizando las bondades del contenedor EJB. Esto trajo como consecuencia las siguientes ventajas:

- Rápida configuración de las transacciones:
  La configuración de las transacciones se hizo en los ficheros xml descriptores de despliegue, lo que provocó que se realizara de una manera rápida, no intrusiva, sin afectar el código de la lógica de negocio. En la propuesta se logra esta rapidez también. Gracias al uso de Spring este manejo se realiza de manera declarativa en ficheros xml como en esta aplicación. Además se aprovechan las funcionalidades que trae el framework para esto.

Seguridad

La seguridad fue implementada utilizando filtros que interceptaron las solicitudes y aplicaron los procesos de seguridad. Este mecanismo fue implementado manualmente por lo que resulto difícil, demandó un gran esfuerzo y tiempo de desarrollo, a diferencia del uso de mecanismos bien definidos que agilicen el desarrollo de este proceso como es el caso del framework Acegi utilizado en la propuesta.

Entorno de ejecución

Se utilizó WebSphere Application Server (WAS), el cual es un servidor de aplicaciones robusto y de gran experiencia que es propiedad de IBM. Este servidor presenta gran soporte en la comunidad internacional. Se usó dados los requerimientos de un servidor que soportara los componentes EJBs.

Pilares de una arquitectura empresarial

Al analizar la arquitectura del proyecto Safre respecto a la propuesta de acuerdo a los objetivos con los que debe cumplir una arquitectura empresarial se puede decir lo siguiente:

Pilares	Safre	Propuesta
Portabilidad	La aplicación gana portabilidad al utilizar los objetos persistentes (*DBObjects*) que brinda el framework Expresso, los mismos crean una capa de abstracción de la base de datos que permite la posibilidad de migrar de fuente de datos sin que se vea afectado el código, lo único que hay que cambiar es la configuración de la misma en un fichero xml. El mapeo es realizado a partir de la herencia de clases del framework (*DBObjec*t) y la implementación de varios métodos.	Este proceso es muy similar, con la diferencia que el mecanismo de persistencia es más transparente, los objetos persistentes son independientes del framework Hibernate, ellos son mapeados a partir de ficheros xml.
Extensibilidad	A la aplicación se le pueden agregar	El diseño de la propuesta permite

	diferentes comportamientos que vayan surgiendo a lo largo del ciclo de vida del software. Por ejemplo se le pueden sumar nuevos componentes distribuidos que estén en otros servidores físicos e implementen nuevos procesos de negocio.	que se puedan crear nuevas funcionalidades en la capa de negocio y exponer las mismas en la interfaz e incluso como servicios web sin afectar el funcionamiento ni la estructura de la aplicación. Además posibilita con el uso de Hibernate la interacción con varias bases de datos a la vez sin modificaciones en la arquitectura.
Fácil de probar	La aplicación es difícil de probar pues los componentes se encuentran en diferentes máquinas físicas y además los EJBs deben ser probados en el contenedor EJB pues están atados al mismo.	Los métodos de la interfaz de negocio pueden ser probados fácilmente pues no necesitan ejecutarse dentro del contenedor web.
Escalabilidad	La aplicación gana escalabilidad puesto que los EJBs pueden distribuir la carga de la aplicación por distintos servidores físicos.	La propuesta puede lograr escalabilidad clusterizando el contenedor web Tomcat y de esta forma agregar nodos físicos y balancear la carga de la aplicación.

4.3 Módulo Servicio Autónomo del proyecto Registros y Notarías

Registros y Notarías es un proyecto colaborativo con la Republica Bolivariana de Venezuela. El mismo tiene la misión de lograr informatizar todo el proceso de gestión documental que se realiza en las oficinas de registros de ese país. El proyecto consta de 4 módulos y entre ellos se encuentra el de Servicio Autónomo, el mismo tiene la labor de crear un producto de software que va a ser el encargado de la gestión y el control de los procesos en las oficinas de registros. Su funcionalidad se centra en definir los datos de las oficinas, usuarios, conceptos de pago, atributos, entre otros; gestionar reportes consolidados, exenciones de pago, prohibiciones y medidas.

A continuación se presenta la arquitectura que se utilizó en el módulo Servicio Autónomo (fig 4.3):

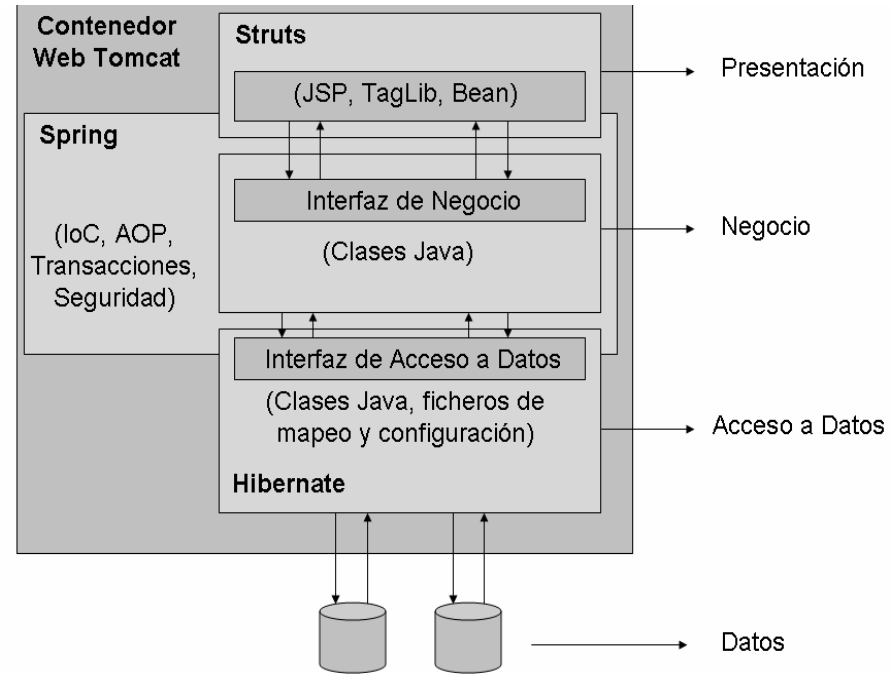


Fig 4.3 Arquitectura utilizada en el módulo Servicio Autónomo

Capa de Presentación

En esta capa se ubicó el framework Struts, para gestionar la vista y control dentro del patrón MVC. Para la presentación se utilizaron páginas JSP y un conjunto de librerías de etiquetas que ayudaron a manejar el contenido dentro de las mismas. Los formularios de datos fueron representados por beans(*ActionForms*) los cuales se utilizaron para encapsular los datos que se entran en los mismos, manejar la validación de los campos y a través de *ActionErrors* enviar los mensajes de error a las páginas JSP a través de etiquetas. Los mensajes de error se cargaron del fichero *.properties*, el cual se encarga de gestionar todos los textos de la aplicación. Para el control se utilizó el servlet controlador de Struts (*ActionServlet*), el cual maneja todas las solicitudes direccionándolas hacia las acciones apropiadas (*Action*) donde se implementa la lógica de presentación e interactúa con la capa de negocio. Se utilizaron los *ActionsMapping* que brinda el framework para definir las clases *Action* con sus *ActionsFroward*, excepciones y relación con los formularios en el *struts-config.xml*. A la hora de definir la navegación de la aplicación ya sea hacia una página JSP u otra funcionalidad, se usaron los *ActionsForward*.

Ventajas

- Se implementa un diseño MVC en poco tiempo:

  Este framework, al igual que JSF, tiene un conjunto de elementos que implementan patrones de diseño para la capa de presentación, tal es el caso del *FrontController* que es implementado por el servlet controlador, entre otros, esto hace posible que haya un desarrollo de buenas prácticas, exista una correcta asignación de responsabilidades y separación de papeles. Estos elementos tienen implementadas funcionalidades que permiten el manejo de la vista y el control de la aplicación.

- El desarrollo de la capa de presentación es más rápido que en las variantes donde no se usa framework:

  El nivel de rapidez es mayor cuando se utilizan frameworks como Struts y JSF que cuando no se usan, pues estos traen consigo una serie de elementos y librerías de clases que tienen funcionalidades implementadas que agilizan el desarrollo de esta capa.

- Se obtienen los beneficios del framework como el manejo de errores, la internacionalización, validación, entre otros:

  Struts, al igual que JSF, trae consigo elementos que tienen implementadas funcionalidades para el manejo de la validación, la internacionalización, los errores que ahorrarían tiempo de realizarlos a mano.

Desventajas

- Carencia de una infraestructura de componentes de presentación y manejo de eventos:

  Es importante contar con mecanismos que brinden un manejo de eventos pues esto permite que, a través de un cambio de comportamiento en los componentes, poder gestionar alguna funcionalidad ya sea en la presentación así como también en la lógica de negocio de la aplicación. Estos mecanismos los trae bien implementados el framework JSF que se propone para esta capa.

- Alto acoplamiento de los formularios (ActionForms) y las acciones (Action) con el framework:

  Existe una gran dependencia entre los beans de formulario y las acciones con el framework, esto imposibilita la reutilización de estas clases fuera del ámbito de Struts. Lo contrario sucede en la propuesta pues los beans de respaldo de JSF están desacoplados del framework y pueden usarse en cualquier otro lugar.

Capa de Negocio

En esta capa se ubicó una interfaz de negocio donde se expusieron las funcionalidades del mismo. Las clases java implementaron los métodos de lógica de negocio de la aplicación de Servicio Autónomo. En estas clases se realizó la interacción con la interfaz de la capa de acceso a datos. Todo el manejo de referencias estuvo gestionado por Spring y su técnica de inyección de instancias que permitió crear los objetos e inicializarlos desde un XML, de esta manera los objetos no se preocuparon por buscar o crear sus referencias sino que el contenedor de Spring se encargó de hacerlo. Tanto las clases de negocio como las de acceso a datos se definieron en el fichero de configuración de Spring, por tanto las primeras recibieron las referencias de las segundas inyectadas por el contenedor de Spring.

Ventajas

- Se creó un enlace en el centro de la arquitectura que unió a la presentación con el acceso a datos:
  Al usar Spring como framework para gestionar la capa de negocios de Servicio Autónomo, se pudo lograr un enlace entre la presentación y el acceso a datos pues el mismo se integra con ambos y su contenedor con su técnica de inyección de dependencias permitió el manejo de referencias sin que los objetos tuvieran que preocuparse por esto. Esta ventaja se vio principalmente a la hora de declarar los beans de respaldo de los formularios de presentación y su dependencia con los beans de negocio y de estos últimos con los de acceso a datos, aquí se observó como el contenedor de Spring buscó en estos ficheros las dependencias declaradas y fue capaz de inicializarlas, logrando así una integración de las capas. En la propuesta al utilizar este framework también se obtienen estos beneficios.
- Se gestionaron los objetos de la aplicación y mejoraron las prácticas de programación:
  El contenedor de este framework permitió la gestión de los objetos de la aplicación a través de la inyección de instancias, los objetos y sus propiedades eran declarados en ficheros xml y el contenedor se encargaba de inicializarlos. Esto permitió el desarrollo de buenas prácticas de programación, limpieza en el código de la aplicación. En la propuesta presentada en el capítulo anterior se aprovecha también esta funcionalidad que brinda Spring.

Capa de Acceso a Datos

En esta capa se ubicó una interfaz de acceso a datos donde se expusieron las funcionalidades que fueron implementadas utilizando clases java. En estas clases de acceso a datos se obtuvieron funcionalidades que brinda la integración de Spring con Hibernate como por ejemplo el uso de plantillas con una gran

cantidad de métodos implementados para el acceso a datos. Las entidades persistentes y sus diferentes características fueron representadas mediante documentos XML. Se utilizó el lenguaje orientado a objetos de Hibernate que permitió realizar consultas para extraer datos (*HQL*). Principalmente se aprovechó en esta capa el mecanismo que ofrece Hibernate totalmente transparente que permite que los objetos persistentes puedan viajar transportando información desde la capa de acceso a datos hasta la presentación.

Ventajas

- Se implementó el patrón DAO con un mínimo de esfuerzo:

  En esta capa hubo una correcta separación de papeles, no hubo código de acceso a datos embebido en el código de las clases de negocio, se utilizó una interfaz de acceso a datos para exponer los mismos, ocultando su implementación en las clases de acceso a datos. En estas últimas, al heredar de la clase que brinda Spring para el manejo de las librerías de Hibernate, se pudo realizar todo el procesamiento de acceso a datos. Esto posibilitó que al hacer un cambio en el acceso a datos no se modificó el código en las otras capas. Este comportamiento se implementa en la propuesta.
- Se agilizó el intercambio de información con la base de datos con lenguajes de consultas orientados a objetos (*HQL*):

  El lenguaje de consultas *HQL*, con su carácter orientado a objetos, permitió de una manera más fácil e intuitiva que el SQL manejar el intercambio de información de la aplicación con la base de datos. Como se usa Hibernate en la propuesta y el mismo lo trae consigo se puede desarrollar el acceso a datos de una forma bastante rápida.
- Se aprovechó la posibilidad de los objetos persistentes de poder viajar por todas las capas de la aplicación:

  Con el uso de Hibernate fue posible en la aplicación de Servicio Autónomo que los objetos persistentes navegaran a través de todas las capas como encapsuladores de datos (patrón *TransferObject*) para que los mismos no viajaran en pequeñas porciones por toda la aplicación.

Capa de Datos

Se utilizó como gestor de bases de datos Oracle Enterprise Edition.

Seguridad

Para gestionar la seguridad de la aplicación se utilizó el framework Acegi que viene integrado con Spring. La utilización de Acegi hizo posible un manejo de la autenticación del usuario y el control de acceso del mismo a través de manejadores. Se usaron filtros para que trabajaran junto con el manejo de la autenticación y el control de acceso.

Ventajas

- Manejo de la seguridad de forma declarativa:

  Se aprovechó la facilidad de Acegi de manejar la seguridad de una aplicación de forma declarativa, evitando la incómoda configuración manual de esta. Este mecanismo hizo posible que este proceso se realizara de una forma limpia dentro del archivo de configuración de Spring y que una sola persona implementara las políticas de seguridad en toda la aplicación en un corto período de tiempo. Este mecanismo se retoma en la propuesta.

Transacciones

En la aplicación de Servicio Autónomo se realizó la gestión de las transacciones a través del framework Spring. Esto permitió que se manejaran declarativamente utilizando las librerías del módulo de programación orientada a aspectos (*AOP*) como proxies que interceptan los métodos y los ejecutan en un ambiente transaccional, se utilizó el manejador de transacciones que provee Spring como interfaz con las librerías de Hibernate.

Ventajas

- Se manejaron las transacciones declarativamente:

  En esta aplicación se realizó un manejo de transacciones declarativamente que trajo como consecuencia un código limpio pues todas las declaraciones se realizaron en ficheros xml. En la propuesta se implementa este mecanismo.

- Rápido y fácil manejo de transacciones:

  El manejo de las transacciones en esta aplicación se realizó de una manera fácil, aprovechando las librerías del framework con funciones ya implementadas y la Programación Orientada a Aspectos que el mismo usa para la gestión de estas. También en la propuesta realizada se logra esto.

Entorno de ejecución

Se utilizó como entorno de ejecución el contenedor web Tomcat.

Ventajas

- Fácil de configurar y administrar:
  Este contenedor fue fácil de configurar y administrar en el sistema de Servicio Autónomo.
- Posibilitar la creación de un repositorio de conexiones para la aplicación:
  Este contenedor permitió la configuración de un repositorio de conexiones que hizo posible que estas fueran almacenadas de tal manera que cada vez que se quisiera acceder a la fuente de datos no se tuviera que realizar la conexión desde la aplicación, sino que se aprovechara estas conexiones ya abiertas en el contenedor web.

Desventajas

- No soporta los API EJB (componentes remotos) ni JMS (servicios de mensajería), lo que impide usar estas tecnologías en aplicaciones que usen esta arquitectura.

Pilares de una arquitectura empresarial

Analizando el comportamiento de la arquitectura de Servicio Autónomo y comparándola con la propuesta de acuerdo a los pilares con los que debe cumplir una arquitectura empresarial se puede decir lo siguiente:

Pilares	Servicio Autónomo	Propuesta
Portabilidad	La aplicación gana portabilidad al usar Hibernate como framework para el manejo de la capa de acceso a datos. Es posible migrar de fuente de datos sin que se vea afectado el código, lo único que hay que cambiar es el driver, dialecto de la base de datos, entre otros elementos.	
Reusabilidad	Su reusabilidad se ve afectada en los beans de formulario y en las acciones utilizadas para el manejo de la capa de presentación, puesto que las mismas están muy ligadas al framework, extienden comportamiento de clases del mismo y en un ámbito donde no se	Lo contrario sucede en la propuesta al usar JSF como framework para la capa de presentación, el mismo cuenta con beans de formularios que son independientes del framework y pueden ser usados por cualquier otro sistema.

	encuentre presente Struts no sería posible utilizarlas.	
Extensibilidad	A la aplicación se le pueden agregar diferentes comportamientos que vayan surgiendo a lo largo del ciclo de vida del software.	
Fácil de probar	La aplicación tiene la propiedad de ser fácil de probar pues se pueden realizar casos de prueba que se validan contra la presentación, la interfaz de negocio y la interfaz de acceso a datos de la aplicación sin ninguna dificultad.	
Escalabilidad	En la aplicación se puede lograr escalabilidad gracias al uso del contenedor web Tomcat utilizado que permite la clusterización.	
Mantenibilidad	Esta aplicación es mantenible, al tener una separación de papeles por capas apoyándose en las buenas prácticas que traen consigo los frameworks, en la exposición de los métodos de negocio y de acceso a datos a través de interfaces. De esta manera, cualquier nuevo comportamiento que haga falta agregar puede ser fácil de implementar.	

4.4 Proyecto Prisiones

Este proyecto consiste en la automatización del sistema penitenciario de la República Bolivariana de Venezuela. Es un proyecto de gran magnitud y se encuentra actualmente en desarrollo. Abarca toda la gestión de los procesos en las penitenciarías, control de la información de los reclusos, entre otros. Para este proyecto se utilizó una arquitectura que presenta varios puntos en común con respecto a la propuesta realizada en el capítulo anterior (fig 4.4).

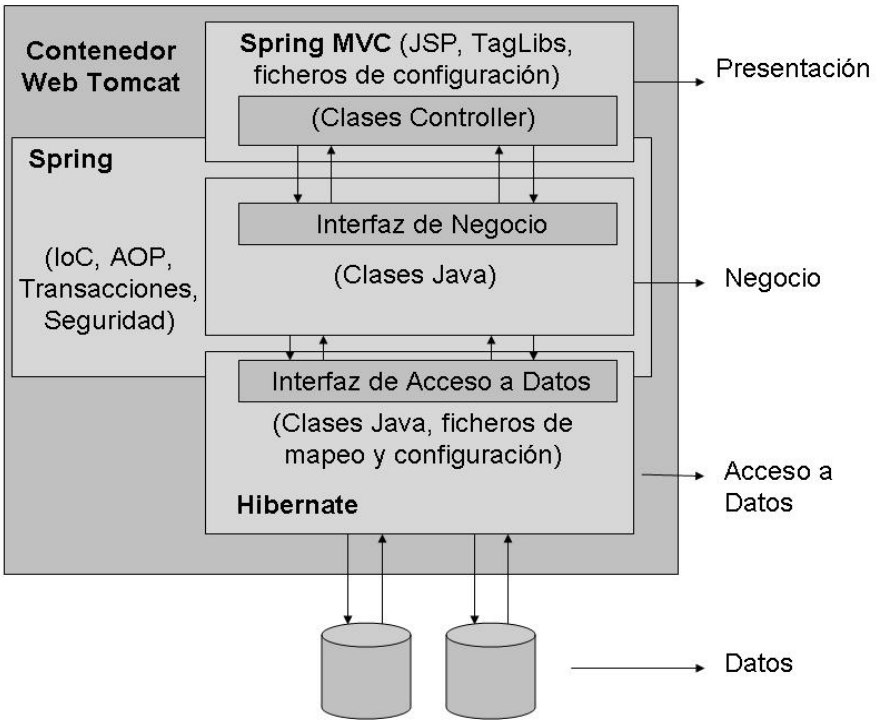


Fig 4.4 Arquitectura del proyecto Prisiones

Capa de Presentación

La capa de presentación está gestionada por el framework MVC que contiene Spring. Este framework ya viene integrado con Spring, se asemeja a Struts y está basado en páginas JSP y en librerías de etiquetas. Las clases que ejecutan la lógica de presentación implementan la interfaz *Controller*. Al utilizar este framework se logra una integración natural con Spring en la capa media. Toda la configuración de la navegación se realiza en los ficheros de configuración de Spring.

Ventajas

- Se logra una buena implementación del patrón MVC:

  Este framework, al igual que JSF, tiene un conjunto de elementos que implementan patrones de diseño para la capa de presentación, tal es el caso del *FrontController* que es implementado por el servlet controlador, entre otros, esto hace posible que haya un desarrollo de buenas prácticas, exista una correcta asignación de responsabilidades y separación de papeles. Estos elementos tienen implementadas funcionalidades que permiten el manejo de la vista y el control de la aplicación.

- Es un framework bien concebido y aceptado por la comunidad de desarrolladores.
- Provee una integración natural con la capa media de la aplicación cuando esta es manejada por Spring:

  Permite que la configuración de la capa de presentación, navegación, objetos vista, entre otros, sean declarados en los ficheros de configuración de Spring y manejados por el contenedor del mismo.

Desventajas

- Las clases que manejan la lógica de la presentación están acopladas a Spring:
  Estas implementan la interfaz *Controller* del framework.
- No está basado en eventos ni abstrae de la lógica de solicitudes y respuestas como lo hace JSF, pues en la implementación de los métodos de la interfaz *Controller* se utilizan los objetos *Request* y *Response*.
- No presenta soporte de herramientas que permitan un rápido desarrollo de las interfaces.

Capa de Negocio

En esta capa se ubicó una interfaz de negocio donde se expusieron las funcionalidades del mismo. Las clases java implementan los métodos de lógica de negocio de la aplicación. En estas clases se realiza la interacción con la interfaz de la capa de acceso a datos. Todo el manejo de referencias está gestionado por Spring y su técnica de inyección de instancias que permite crear los objetos e inicializarlos desde un xml, de esta manera los objetos no se preocupan por buscar o crear sus referencias sino que el contenedor de Spring se encarga de hacerlo. Tanto las clases de negocio como las de acceso a datos se definen en el fichero de configuración de Spring, por tanto las primeras reciben las referencias de las segundas inyectadas por el contenedor de Spring.

Ventajas

El uso del framework Spring para gestionar esta capa en la aplicación trae consigo grandes ventajas:

- Se crea un enlace en el centro de la arquitectura que une a la presentación con el acceso a datos:

  Al usar Spring como framework para gestionar la capa de negocios del proyecto Prisiones, se logra un enlace entre la presentación y el acceso a datos pues el mismo se integra con ambos y su contenedor con su técnica de inyección de dependencias permite el manejo de referencias sin que

los objetos tengan que preocuparse por esto. Esta ventaja se ve principalmente a la hora de declarar los beans de respaldo de los formularios de presentación y su dependencia con los beans de negocio y de estos últimos con los de acceso a datos, aquí se observa como el contenedor de Spring busca en estos ficheros las dependencias declaradas y es capaz de inicializarlas, logrando así una integración de las capas. En la propuesta al utilizar este framework también se obtienen estos beneficios.

- Se gestionan los objetos de la aplicación y mejoran las prácticas de programación:
  El contenedor de este framework permite la gestión de los objetos de la aplicación a través de la inyección de instancias, los objetos y sus propiedades son declarados en ficheros xml y el contenedor se encarga de inicializarlos. Esto permite el desarrollo de buenas prácticas de programación, limpieza en el código de la aplicación. En la propuesta presentada en el capítulo anterior se aprovecha también esta funcionalidad que brinda Spring.

Capa de Acceso a Datos

En esta capa se ubica una interfaz de acceso a datos donde se exponen las funcionalidades que son implementadas utilizando clases java. En estas clases de acceso a datos se obtienen funcionalidades que brinda la integración de Spring con Hibernate como por ejemplo el uso de plantillas con una gran cantidad de métodos implementados para el acceso a datos. Las entidades persistentes y sus diferentes características son representadas mediante documentos xml. Se utiliza el lenguaje orientado a objetos de Hibernate que permite realizar consultas para extraer datos (*HQL*). Principalmente se aprovecha en esta capa el mecanismo que ofrece Hibernate totalmente transparente que permite que los objetos persistentes puedan viajar transportando información desde la capa de acceso a datos hasta la presentación.

Ventajas

- Se implementa el patrón DAO con un mínimo de esfuerzo:
  En esta capa hay una correcta separación de papeles, no hay código de acceso a datos embebido en el código de las clases de negocio, se utiliza una interfaz de acceso a datos para exponer los mismos, ocultando su implementación en las clases de acceso a datos. En estas últimas, al heredar de la clase que brinda Spring para el manejo de las librerías de Hibernate, se realiza todo el procesamiento de acceso a datos. Esto posibilita que al hacer un cambio en el acceso a datos no se modifique el código en las otras capas. Este comportamiento se implementa en la propuesta.

- Se agiliza el intercambio de información con la base de datos con lenguajes de consultas orientados a objetos (HQL):

  El lenguaje de consultas *HQL*, con su carácter orientado a objetos, permite de una manera más fácil e intuitiva que el SQL manejar el intercambio de información de la aplicación con la base de datos. Como se usa Hibernate en la propuesta y el mismo lo trae consigo se puede desarrollar el acceso a datos de una forma bastante rápida.
- Se aprovecha la posibilidad de los objetos persistentes de poder viajar por todas las capas de la aplicación:

  Con el uso de Hibernate es posible que los objetos persistentes naveguen a través de todas las capas como encapsuladores de datos (patrón *TransferObject*) para que los mismos no viajen en pequeñas porciones por toda la aplicación.

Capa de Datos

Se utiliza como gestor de bases de datos Oracle Enterprise Edition.

Seguridad

Para gestionar la seguridad de la aplicación se utiliza el framework Acegi que viene integrado con Spring. La utilización de Acegi hace posible un manejo de la autenticación del usuario y el control de acceso del mismo a través de manejadores. Se usan filtros para que trabajen junto con el manejo de la autenticación y el control de acceso.

Ventajas

- Manejo de la seguridad de forma declarativa:

  Se aprovecha la facilidad de Acegi de manejar la seguridad de una aplicación de forma declarativa, evitando la incómoda configuración manual de esta. Este mecanismo hace posible que este proceso se realice de una forma limpia dentro del archivo de configuración de Spring y que una sola persona implemente las políticas de seguridad en toda la aplicación en un corto período de tiempo. Este mecanismo se retoma en la propuesta.

Transacciones

Se realiza la gestión de las transacciones a través del framework Spring. Esto permite que se manejen declarativamente utilizando las librerías del módulo de programación orientada a aspectos (*AOP*) como

proxies que interceptan los métodos y los ejecutan en un ambiente transaccional, se utiliza el manejador de transacciones que provee Spring como interfaz con las librerías de Hibernate.

Ventajas

- Se manejan las transacciones declarativamente:
  En esta aplicación se realiza un manejo de transacciones declarativamente que trae como consecuencia un código limpio pues todas las declaraciones se realizan en ficheros xml. En la propuesta se implementa este mecanismo.
- Rápido y fácil manejo de transacciones:
  El manejo de las transacciones en esta aplicación se realiza de una manera fácil, aprovechando las librerías del framework con funciones ya implementadas y la Programación Orientada a Aspectos que el mismo usa para la gestión de estas. También en la propuesta realizada se logra esto.

Entorno de ejecución

Se utiliza como entorno de ejecución el contenedor web Tomcat.

Ventajas

- Fácil de configurar y administrar:
- Posibilitar la creación de un repositorio de conexiones para la aplicación:
  Este contenedor permite la configuración de un repositorio de conexiones que hace posible que estas sean almacenadas de tal manera que cada vez que se quiera acceder a la fuente de datos no se tenga que realizar la conexión desde la aplicación, sino que se aproveche estas conexiones ya abiertas en el contenedor web.

Desventajas

- No soporta los API EJB (componentes remotos) ni JMS (servicios de mensajería) lo que impide usar estas tecnologías en aplicaciones que usen esta arquitectura.

Pilares de una arquitectura empresarial

En cuanto al comportamiento de los aspectos con los que debe cumplir una arquitectura empresarial se puede decir lo siguiente:

Pilares	Prisiones	Propuesta
Portabilidad	La aplicación gana portabilidad al usar Hibernate como framework para el manejo de la capa de acceso a datos. Es posible migrar de fuente de datos sin que se vea afectado el código, lo único que hay que cambiar es el driver, dialecto de la base de datos, entre otros elementos.	
Extensibilidad	La aplicación gana extensibilidad pues se le pueden agregar diferentes comportamientos que vayan surgiendo a lo largo del ciclo de vida del software.	
Fácil de probar	La aplicación tiene la propiedad de ser fácil de probar pues se pueden realizar casos de prueba que se validan contra la presentación, la interfaz de negocio y la interfaz de acceso a datos de la aplicación sin ninguna dificultad.	
Escalabilidad	La aplicación gana escalabilidad gracias al uso del contenedor web Tomcat utilizado que permite la clusterización.	
Mantenibilidad	Esta aplicación es mantenible al tener una separación de papeles por capas apoyándose en las buenas prácticas que traen consigo los frameworks, en la exposición de los métodos de negocio y de acceso a datos a través de interfaces. De esta manera, cualquier nuevo comportamiento que haga falta agregar puede ser fácil de implementar.	

Como parte de las conclusiones del análisis de resultados, se hace una comparación de acuerdo a diferentes criterios de la propuesta con todas las arquitecturas de los proyectos anteriormente mencionados en este capítulo a partir de encuestas realizadas a algunos de sus miembros (ver Anexo 1): programador del proyecto Encuestas, programador del proyecto Safre, arquitecto del módulo Servicio Autónomo y arquitecto del proyecto Prisiones. Las mediciones se realizan en una escala del 1 al 5, donde 1 es muy baja, 2 es baja, 3 es media, 4 es aceptable y 5 es alta.

Criterios	Encuestas	Safre	Servicio Autónomo	Prisiones	Propuesta
Mantenibilidad	3	4	4	5	5
Escalabilidad	4	5	5	5	5

Portabilidad	1	3	4	5	5
Facilidad de prueba	5	2	5	5	5
Extensibilidad	3	5	5	5	5
Robustez	2	5	5	5	5
Necesidad de capacitación de los recursos humanos	2	5	3	3	3
Productividad	2	3	4	4	5
Complejidad	2	5	4	4	4

Conclusiones

Al finalizar este capítulo se puede llegar a las siguientes conclusiones:

- La propuesta se destaca por encima de las arquitecturas utilizadas en proyectos anteriores pues fortalece los elementos claves de una arquitectura empresarial como la escalabilidad, portabilidad, extensibilidad.
- Su desarrollo es complejo, pero la combinación de los elementos de los frameworks, tecnologías y herramientas escogidas y su integración juegan un papel fundamental en el aumento de la productividad y la disminución del esfuerzo de los desarrolladores.
- La separación de papeles e implementación de patrones de diseño y buenas prácticas que la propuesta logra hacen que la misma tenga un código limpio, robusto y fácil de probar.
- La propuesta arquitectónica realizada mejora los puntos débiles de arquitecturas anteriormente utilizadas y hace suya las potencialidades de las mismas para obtener finalmente un modelo que permita a los proyectos actuales llegar a los mejores resultados.

Conclusiones generales

En el presente trabajo de diploma se cumplieron todos los objetivos propuestos y se arriba a las siguientes conclusiones:

A partir del estudio del estado del arte de las tecnologías disponibles para el desarrollo de aplicaciones en la plataforma JEE, donde se analizaron los APIs, frameworks, servidores web y de aplicaciones y sus características, se concluye:

- Que existen numerosas tecnologías y herramientas pero no están integradas en una sola por lo tanto se hace necesario proponer un modelo arquitectónico que reúna las tecnologías apropiadas para lograr un balance entre productividad, costo y portabilidad.
- A partir del estudio de las tecnologías y de otros modelos arquitectónicos, analizando sus ventajas, desventajas y puntos débiles, se propuso en la tesis un modelo de arquitectura que se destaca por encima de las arquitecturas utilizadas en proyectos anteriores.
- La nueva arquitectura propuesta constituye una combinación poderosa de tecnologías, frameworks y herramientas que permiten lograr a gran escala el cumplimiento de principios básicos de una arquitectura empresarial tales como: la escalabilidad, portabilidad y extensibilidad.
- El modelo propuesto constituye una guía para arquitectos y desarrolladores de la plataforma JEE en los proyectos productivos actuales.
- La implementación del modelo propuesto es compleja pero la combinación de los frameworks, las tecnologías y las herramientas seleccionadas y su integración juegan un papel fundamental en el aumento de la productividad y la disminución del esfuerzo de los desarrolladores.
- La clara definición de roles, la arquitectura basada en patrones de diseño y las buenas prácticas que la propuesta logra hacen que la misma tenga un código limpio, robusto y fácil de probar.

Recomendaciones

Se recomienda profundizar en otros elementos que enriquezcan la propuesta arquitectónica tales como:

- Herramientas para la realización de reportes y su integración con los frameworks utilizados en la propuesta.
- Herramientas de desarrollo que agilicen la producción en las aplicaciones basadas en esta arquitectura.

Realizar un estudio detallado de frameworks y herramientas de desarrollo así como buenas prácticas, que permitan definir una propuesta para aplicaciones de escritorio sobre la plataforma Java.

Analizar las crecientes tendencias a las arquitecturas orientadas a servicios (SOA) que permitan definir una línea que guíe las implementaciones de este paradigma en la plataforma Java.

Bibliografía

1. Microsystems, S. Acerca de la tecnología Java. [cited; Available from: http://www.java.com/es/about/.
2. [cited; Available from: http://www.comunidadjava.com.ar/news.html.
3. Allamaraju, S., Programación Java Server con J2EE Edición 1.3, A. Multimedia, Editor. 2002.
4. Husted, T., Struts in Action, M.P. Co., Editor. 2003.
5. Man, K.D., Java Server Faces in Action, M.P. Co., Editor. 2005.
6. Walls, C., Spring in Action, M.P. Co., Editor. 2005.
7. BAUER, C., Hibernate in Action, M.P. Co., Editor. 2005.
8. Touris, A.M. Arquitectura empresarial y software libre,J2EE. 2002 [cited; Available from: http://www.javaHispano.org.
9. Johnson, R., Expert One-on-One J2EE Design and Development, W. Press, Editor. 2003.
10. Diseño de aplicaciones internet usando los Patrones de diseño J2EE(los Core J2EE Patterns). [cited.
11. Microsystems, S. Catálogo de Patrones de Diseño J2EE. Capas de Negocio y de Integración. 1999-2006 [cited; Available from: http://www.programacion.net/java/tutorial/patrones2/1/.
12. Cavaness, C., Programming Jakarta Struts, O'Reilly, Editor. November 2002.
13. Hightower, R., Jakarta-Struts Live, SourceBeat, Editor. 2004.
14. JavaServer Faces, O'Reilly, Editor. 2004.
15. DAVID GEARY, C.H., Core JavaServer Faces A. Wesley, Editor. 2004.
16. Heudecker, N. Introduction to Hibernate. 2003 [cited; Available from: http://www.TheServerSide.com.
17. González, C.S. (2004) ONess: un proyecto open source para el negocio textil mayorista desarrollado con tecnologías open source innovadoras. Volume,
18. Johnson, R. Introduction to the Spring Framework. 2005 [cited; Available from: http://www.TheServerSide.com.
19. Harrop, R., Pro Spring, Apress, Editor. 2005.
20. Alberto Molpeceres Touris, M.P.M. Arquitectura empresarial y software libre, J2EE. 2002 [cited; Available from: http://www.javaHispano.org.

Anexo 1

Encuesta

Nombre: _____

Proyecto: _____

Rol que ocupa: _____

1- ¿Qué facilidades de mantenibilidad considera usted que tiene la aplicación?

_____Muy baja _____Baja _____Media _____Aceptable _____Alta

2- Indique el grado de portabilidad de la aplicación.

_____Muy bajo _____Bajo _____Medio _____Aceptable _____Alto

3- En caso que se desee agregar nuevas funcionalidades seleccione el nivel en que la aplicación puede soportarlas.

_____Muy bajo _____Bajo _____Medio _____Aceptable _____Alto

4- ¿La aplicación es fácil de probar?

_____Muy bajo _____Bajo _____Medio _____Aceptable _____Alto

5- Grado de robustez de su aplicación.

_____Muy bajo _____Bajo _____Medio _____Aceptable _____Alto

6- Diga en que grado la aplicación es capaz de responder a un aumento de la carga.

_____Muy bajo _____Bajo _____Medio _____Aceptable _____Alto

7- Indique el grado de complejidad al implementar la arquitectura de la aplicación.

_____Muy bajo _____Bajo _____Medio _____Aceptable _____Alto

8- ¿Qué productividad se alcanza con la arquitectura de la aplicación?

_____Muy baja _____Baja _____Media _____Aceptable _____Alta

9- Nivel de dificultad en el dominio de las tecnologías utilizadas en la arquitectura.

_____Muy bajo _____Bajo _____Medio _____Aceptable _____Alto