5 DISEÑO DE APLICACIONES CON J2EE

La plataforma J2EE nos ofrece distintas posibilidades de diseño. La elección de una arquitectura u otra condicionará en cierto modo nuestro diseño. Sin embargo, la utilización del patrón arquitectónico MVC y los Patrones de Diseño J2EE, dotarán a nuestras aplicaciones de gran escalabilidad facilitando posibles futuras ampliaciones, migraciones y cambios en general.

Como anteriormente hemos comentado, J2EE se basa en una arquitectura multicapa y utiliza un patrón arquitectónico MVC (Model – View – Controller). Con este modelo se permite una fácil separación de la interfaz gráfica y del modelo de negocios, gracias a un controlador que los mantiene desacoplados. Con esto podríamos tener configuraciones en las que el cliente tan sólo disponga de interfaz gráfica y acceden a un servidor donde se implementa el modelo. De este modo, cambios en el modelo sólo afectarían al servidor. Aunque nos centramos en aplicaciones Web, este modelo no es sólo útil en entornos Internet, sino en todos los entornos empresariales.

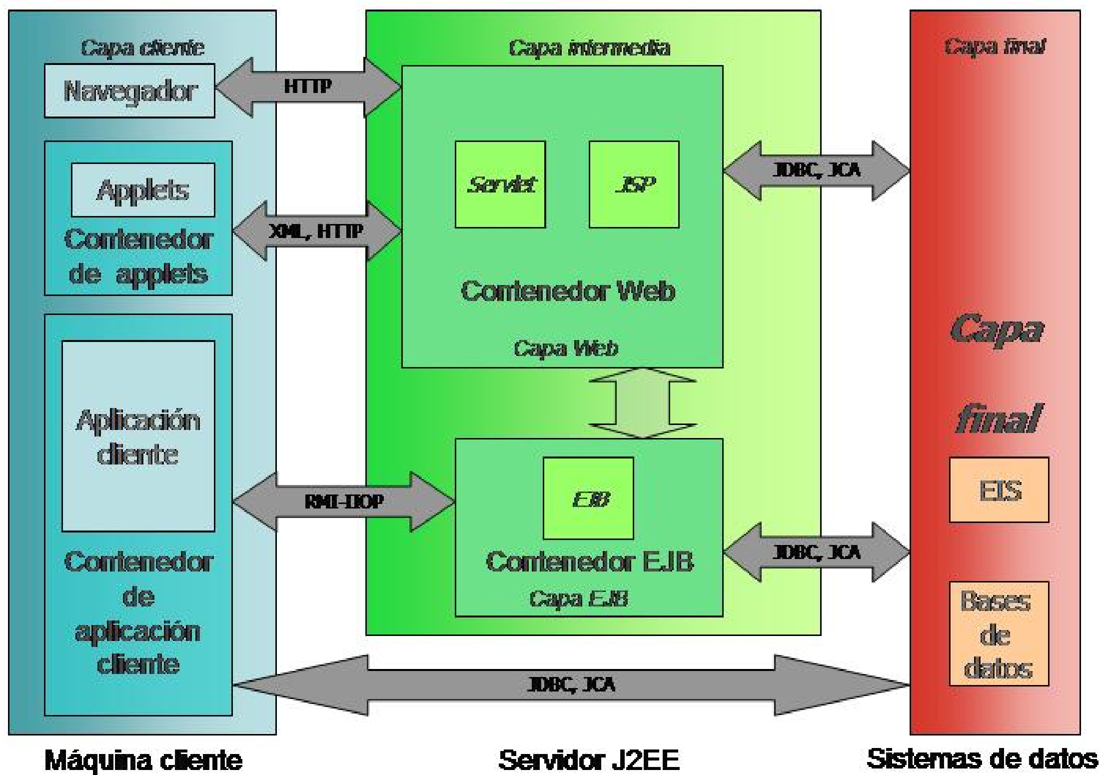
Para el diseño de aplicaciones además, utilizaremos soluciones basadas en los Patrones de Diseño J2EE, utilizando ciertos elementos que combinados tienen la estructura ya conocida MVC.

Tanto Servlets como JSP son soluciones para el desarrollo de aplicaciones empresariales. Difieren en su modelo de programación. Una página JSP es esencialmente un documento que especifica contenido dinámico, mientras que un servlet se puede comparar más a un programa que produce este tipo de contenido. Aunque ambas tecnologías pueden presentarse como soluciones independiente para el desarrollo de aplicaciones, son realmente potentes en un contexto MVC trabajando conjuntamente.

5.1 POSIBILIDADES DE DISEÑO

Debido a que el estándar J2EE nos ofrece varias posibilidades de arquitectura, en las primeras fases del diseño de aplicaciones según el estándar es necesario especificar que arquitectura utilizar de acuerdo con nuestras necesidades. En torno a la arquitectura elegida girarán las siguientes fases del diseño.

El modelo de una arquitectura completa J2EE se muestra en el capítulo 4 SOLUCIÓN PROPUESTA: J2EE.



F. 5-1: Modelo de componentes en la arquitectura J2EE.

Uno de los elementos a tener en cuenta es el tipo de cliente de la aplicación, en el capítulo anterior, capítulo 4 SOLUCIÓN PROPUESTA: J2EE, en el apartado 4.1 ARQUTECTURA FÍSICA MULTICAPA se muestran las distintas posibilidades que nos ofrecen los distintos tipos de clientes.

Nos centraremos en una arquitectura multicapa de servidor intermedio, arquitectura de tres capas o cuatro capas con clientes Web, es decir, desde un navegador Web se accede al servidor y éste es el que accede a los datos. Esta arquitectura nos permite abarcar los conceptos básicos de la capa Web y la capa EJB siendo fácilmente escalable.

La fuente de datos también es otro factor a tener en cuenta en el diseño de una aplicación. En esta presentación consideraremos el caso más común en el que los datos se almacenan en una base de datos relacional.

El acceso a datos condicionará, por ejemplo la decisión de utilizar sólo capa Web en el servidor, o la necesidad de utilizar también la capa EJB. Presentamos las posibles configuraciones que se nos pueden plantear según las necesidades.

5.1.1 Acceso a datos con JDBC

Esta opción es adecuada si no hay mayoría de casos de uso transaccionales y los clientes son sólo clientes Web, no aplicaciones stand-alone. Para tener independencia de la base de datos se recomienda implementar programáticamente una capa de acceso a datos utilizando el patrón de diseño DAO. Esto permitirá en un futuro una fácil migración a otro tipo de almacenamiento y acceso a datos.

En algunos casos puede ser necesaria la utilización de JDBC junto a EJB.

5.1.2 Acceso a datos con J2EE Connector Arquitecture

Una alternativa a la conexión con JDBC es la arquitectura de conector J2EE. Esta alternativa actualmente se utiliza principalmente en conexiones a EIS, sistemas de información empresariales. Actualmente, para acceso a bases de datos relacionales la opción de JDBC representa una alternativa más madura y fácil. Además, la mayoría de fabricantes de bases de datos, ofrecen una implementación del driver JDBC adecuado.

5.1.3 Acceso a datos con RMI

El uso de RMI con JDBC sería la opción más adecuada cuando se utilizan clientes stand-alone y existe una carga pequeña en un entorno distribuido.

5.1.4 Acceso a datos con EJB

Cuando existen aplicaciones clientes stand-alone, una carga alta en un entorno distribuido o un importante número de casos de usos transaccionales se recomienda realizar el acceso a datos a través de la capa EJB. La capa EJB, para el acceso a datos, hará uso de un driver JDBC o de la Arquitectura de Conector J2EE.

En algunos casos, puede ser necesaria la implementación EJB y JDBC. En estos casos, la arquitectura sería la misma que si se utilizase sólo EJB, pero existirían casos de uso, especialmente los correspondientes a búsquedas que devuelvan colecciones de objetos, que se expresan de un modo más eficiente con JDBC. Otro motivo para la utilización de ambos métodos para el acceso a datos, podría ser la existencia previa al diseño de la aplicación, de un esquema general de la base de datos que ya se utilizase por aplicaciones antiguas.

5.2 USO DE SERVLETS

En una estructura según el modelo MVC, Model – View – Controller, con EJB, JSP y Servlets, los servlets se utilizarán principalmente para la implementación del controlador.

En general, si trabajan conjuntamente con JSP, no se utilizarán como componentes visuales, excepto en los casos en los que sea necesario generar contenido binario. Un servlet tras proporcionar diversos servicios tales como plantillas, seguridad, personalización, control de la aplicación, filtros, debe seleccionar un componente de presentación, generalmente una página JSP, y remitirle la petición para visualizarla.

Las servlets implementan un controlador que activa operaciones de la aplicación y toma decisiones, determinan como manejar una respuesta y eligen la vista a presentar.

Generalmente, los Servlets sólo se utilizan como componentes visuales si es necesaria la generación de contenido binario. En este caso, los servlets deben dar el valor de MIME del contenido generado al campo de tipo de contenido de la cabecera HTTP.

Utilizar servlets para la creación de contenido textual se convierte en una tarea tediosa y difícil de mantener, ya que es necesario realizarlo programáticamente y actualizar la presentación visual supondría modificar y recompilar un programa. Para este tipo de contenido se recomienda el uso de JSP ya que sólo sería necesario actualizar una página de etiquetas.

Un servlet utiliza dos métodos del manejador de peticiones para crear una respuesta utilizando otros componentes. El método `forward` delega el proceso de una petición completa a otro componente. El método `include` construye una respuesta que contiene resultados de múltiples.

5.3 USO DE JSP

En una estructura según el modelo MVC, Model – View – Controller, con EJB, JSP y Servlets, las páginas JSP se utilizarán principalmente para generar la vista de la aplicación.

Las páginas JSP no deben utilizarse como controladores, ya que con esta funcionalidad se obtiene una mezcla de etiquetas y código que las hace difícil de leer y mantener. Sólo en aplicaciones muy sencillas se justifica el uso de JSP tanto para generar la vista, como controlador, y para implementar una lógica de negocio sencilla.

La tecnología JSP se crea especialmente para producir contenido dinámico que sólo cambia en los valores de datos y no en la estructura básica. Una página JSP es un documento que contiene una plantilla de texto fija, junto a unas etiquetas específicas para incluir texto dinámico o ejecutar la lógica embebida en el documento. Las etiquetas personalizadas y los scriptlets aparecen embebidos entre los datos estáticos y son sustituidos en tiempo de ejecución por el contenido dinámico. Puesto que es un lenguaje de servidor, una página JSP es siempre servida al cliente como código HTML tradicional.

Las páginas JSP se utilizan principalmente para producir contenido textual, en particular, es una tecnología excelente para generar mensajes XML en formato estándar. Pueden crear también contenido sin estructura, como texto ASCII, o bien actuar como plantillas para unir dato de múltiples fuentes. Si se necesita generar contenido binario, es necesario el uso de servlets, ya que las páginas JSP no pueden crear este tipo de contenido.

Aunque el etiquetado estándar de una página JSP no es un texto bien formado XML, la especificación define una alternativa para sus páginas de una sintaxis XML. Con el uso de esta alternativa, las páginas pueden ser validadas contra un esquema de definición de lenguaje, o XSDL, lo que permitiría, por ejemplo, validar cualquier posible error, que de otro modo, sólo se apreciaría en tiempo de ejecución. Esta sintaxis XML también facilita la integración con herramientas de desarrollo. Si se utiliza la alternativa de sintaxis XML para una página JSP, por motivos de integridad, no podría contener mezcla de sintaxis JSP estándar.

Independientemente de utilizar una sintaxis XML, o no, en el etiquetado de una página JSP, una página JSP puede generar contenido dinámico XML.

La tecnología JSP es compatible con una amplia gama de herramientas de diseño ya que al igual que HTML se basa en un lenguaje de etiquetas. Las etiquetas de una página JSP pueden ser de tres tipos: directivas, script, o etiquetas personalizadas. Las directivas son evaluadas en tiempo de compilación y se identifican por los delimitadores `<%@` y `%>`. Los scripts son bloques de código Java embebido en la página JSP entre los delimitadores `<%` y `%>` que generan contenido dinámico que es incluye en la respuesta cuando la página es servida.

Los comentarios en una página JSP se delimitan por `<%--` y `--%>`, estos comentarios no serán recibidos por el cliente ya que no se traducen a código HTML.

El desarrollador puede definir sus etiquetas personalizadas según guías estándar que se sustituyen por contenido dinámico cuando la página es servida. El programador define la sintaxis para una etiqueta e implementa el comportamiento de la etiqueta en la clase que la maneja, `Tag Handler Class`, la cual crea el contenido dinámico. Posteriormente se empaqueta en archivos de librerías de etiquetas, que pueden ser importadas y usadas en una página Web con las etiquetas correspondientes.

Junto a las propias etiquetas, se pueden utilizar una serie de etiquetas estándares que define la especificación JSP, JSTL (The Java Standard Tag Library), que están disponibles en todas las implementaciones de la plataforma J2EE. Estas etiquetas, según se vio en 4.3.2 JSTL, biblioteca de etiquetas para JSP, son una serie de etiquetas personalizadas que proporcionan funcionalidades básicas para las páginas JSP. Normalmente se le conocen como etiquetas lógicas. Entre las funciones típicas se incluyen inclusión de recursos Web, redireccionamiento de peticiones, lógica condicional, iteración de datos, transformaciones XSTL, internacionalización, formularios. Las librería estándares, a menudo, proporcionan más funcionalidades de las que una página JSP necesita.

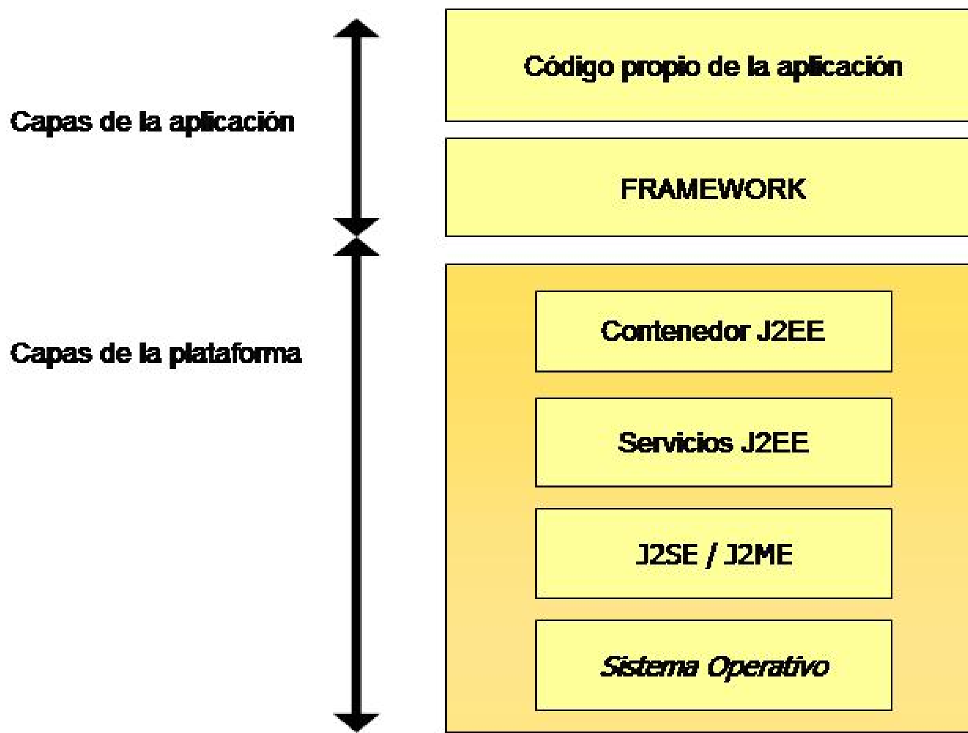
Conviene evitar soluciones que abusen de etiquetas estándares y scripting para implementar gran cantidad de lógica en las páginas JSP. En vez de ello, es una mejor opción la utilización de servlets y etiquetas personalizadas. Una potente técnica es definir etiquetas personalizadas

implementando el manejo de etiquetas con EJB, buscando e invocando sus métodos. Esta técnica permite a la vista, páginas JSP, el acceso directo al modelo a través de los EJB, manteniendo la separación de la vista y la funcionalidad.

El uso de etiquetas personalizadas es una elegante alternativa frente al abuso de scriptlets. Una de las características principales de las etiquetas personalizadas es la reusabilidad mientras que los scriptlets sólo aparecen en las páginas que los definen, proveen servicios de alto nivel para las páginas JSP portables entre contenedores ya que encapsulan la lógica permitiendo a la página JSP centrarse en la presentación, facilitan el mantenimiento al reducir la repetición de código, facilitan la separación de roles entre desarrolladores de aplicación y diseñadores Web, pueden proporcionar una sintaxis intuitiva para invocar la lógica de negocio, y permiten desacoplar la lógica de negocio de la presentación de datos. Otra ventaja frente a los sriptlets es que los errores en las etiquetas personalizadas no llegarían a compilarse y se utilizan herramientas de desarrollo que facilitan la identificación de errores mientras que los errores de compilación de los scriptlets pueden ser difíciles de interpretar y de testear.

5.4 USO DE UN FRAMEWORK

Todas las aplicaciones estructuradas en capas comparten un conjunto de requerimientos básicos que no son proporcionados por la plataforma J2EE. A menudo se utiliza una capa de software llamada framework que concentra estos requerimientos y puede ser compartida entre aplicaciones. Sobre la capa del framework tendríamos la capa que consistiría en el código específico de la aplicación, que interacciona con el framework.



F. 5-2: Modelo de capas de una aplicación. Framework.

El framework se asienta sobre la plataforma J2EE, proporcionando funcionalidades comunes de aplicación como el manejo de peticiones, invocación de métodos del modelo, y selección y composición de las vistas.

Las clases e interfaces del framework son estructurales. Los desarrolladores de aplicaciones usan, extienden, o implementan las clases e interfaces del framework para desarrollar funciones específicas de la aplicación. Por ejemplo, un framework puede ofrecer una clase abstracta que un desarrollador puede extender para ejecutar la lógica de negocio en respuesta a eventos de la aplicación.

Un framework hace más fácil el uso de tecnologías Web ayudando a los desarrolladores de aplicaciones en la lógica de negocio. El uso de un framework adecuado proporciona desacoplo de la presentación y la lógica, y por tanto separación de roles de desarrolladores y diseñadores de página. Proporcionan un punto central de control además de un conjunto de características personalizables como plantillas, etiquetas para la creación de vistas, localización, control de acceso y log in. Otra de las ventajas es la estabilidad que presentan, además de la facilidad que supone realizar pruebas de unidad y mantenimiento gracias a las interfaces consistentes del framework.

Se recomiendan utilizar frameworks existentes puesto que ahorra esfuerzo, tiempo y costes. Además existen frameworks de gran madurez en el mercado que nos permitirán centrarnos en la lógica de negocio propia de la aplicación. Los frameworks se pueden comprar, aunque también existen frameworks gratuitos muy potentes. Normalmente, los frameworks que podemos encontrar en el mercado son compatibles con herramientas existentes que pueden mejorar la productividad y fiabilidad.

Debido a la evidente necesidad de un framework, se ha creado JavaServer Faces, JSF. JSF es un framework normalizado en la JSR-127 la primera versión, y en JSR-252 en la versión 1.2. Este framework, aunque relativamente reciente, se apoya en la madurez de otras frameworks existentes como Struts. Este último se abarcará con más detalle en el próximo capítulo.

5.5 USO DE EJB

El utilizar un enfoque orientado a objeto, permite descomponer una función de negocio en un conjunto de componentes llamados objetos de negocio. La lógica de negocio, abarca el conjunto reglas que, con pre y post condiciones, ayudan a identificar la estructura y el comportamiento de los objetos de negocio.

Los requerimientos más comunes para un objeto de negocio son el mantenimiento del estado, compartición de datos, participación en transacciones, accesible por distintos tipos de clientes, disponibilidad, acceso remoto, control de acceso y reusabilidad.

Los EJB son componentes de servidor distribuidos que nos ofrecen estabilidad, seguridad, aspecto transaccional y persistencia de datos, de un modo transparente al programador. Simplifica la tarea de desarrollo de la lógica de negocio.

Los EJB a través de un contenedor EJB ofrecen servicios "middleware", entre ellos, servicios para el manejo de errores, alternativas de bases de datos, aspectos transaccionales. Esto permite que el desarrollador se centre en el diseño de la lógica de negocio, sin tener que preocuparse de estos aspectos comunes a la mayoría de objetos de negocio. Potencian la división del trabajo permitiendo al diseñador de componentes centrarse en la lógica de proceso sin tener que preocuparse de la lógica de servicios que implementa el contenedor.

La existencia de las especificaciones permite la diversidad de proveedores tanto de los propios contenedores como de componentes EJB que ya resuelvan algún tipo de lógica determinada. Los EJB además, pueden interactuar con varios tipos de clientes, como clientes Web y clientes stand-alone.

A pesar de las características que los EJB ofrecen, a la hora de desarrollar una aplicación es importante plantearse si el uso de EJB es justificado. La utilización de esta tecnología puede suponer un mayor tiempo de desarrollo, y en muchos casos el uso de EJB supondría una solución sobrada. Además, para el desarrollo de EJB, al ser uno de los principales componentes de J2EE, se requiere un exhaustivo conocimiento de Java y las dependencias con otras API J2EE.

Los EJB son especialmente útiles en sistemas distribuidos. Representan una alternativa al uso de Java RMI ya que sus características de estabilidad, seguridad, aspecto transaccional y persistencia son mucho mejores y más fáciles de programar con EJB que con RMI. EJB también usan IIOP por lo que pueden interoperar con CORBA.

La API de EJB aunque estandarizada, es lo suficientemente flexible como para implementar componentes según las necesidades del sistema. Existen distintos tipos de componentes EJB para representar distintos tipos de componentes de negocio:

a) Session Bean (Componentes de Sesión). Estos componentes no representan directamente datos compartido en una base de datos, aunque pueden acceder y actualizar estos datos. Permiten implementar las fachadas del modelo y realizar cierta lógica solicitada por un cliente. En el caso en que el contenedor de EJB cayera, estos objetos no permanecerían.

Se pueden a su vez diferenciar en dos tipos: con estado (Statefull Session Bean, SFSB), y sin estado (StateLess Session Bean, SLSB). Los primeros permiten mantener la sesión de un cliente y puede trabajar con ciertos datos específicos administrados por el contenedor. El segundo tipo no mantiene el estado en el contenedor y su uso puede ser por ejemplo operaciones matemáticas, búsquedas generales, etc.

b) Entity Bean (Componentes de Entidad). Son componentes persistentes que nos permiten implementar objetos persistentes del sistema. Trabajan en conjunción con un depósito de información y manipula la información. En realidad lo que manipula es una copia o reflejo de la información. Puede representar una alternativa a JDBC. Sobreviven a caídas del contenedor EJB. Si el estado de una entidad fue actualizado por una transacción mientras el contenedor estaba caído, el estado de la entidad es automáticamente reseteado al estado de la última transacción llevada a cabo.

Existen también dos tipos de Entity Beans. Los Bean Manager Persistent (BMP) requieren que la lógica de acceso a los sistemas de información se defina manualmente normalmente por JDBC. En cambio, con los Container Managent Persistent (CMP), es el contenedor el que genera toda la lógica de acceso. Se utiliza el lenguaje EJB-QL ("Enterprise Java Bean-Query Language"). Hay casos donde la lógica empleada es sumamente compleja y no puede ser implementada por CMP, en este caso se utilizan los BMP.

c) Message-Driven Bean (Componentes orientados a mensaje). Estos componentes son invocados asíncronamente, pueden ser transaccionales y pueden actualizar datos compartidos. Aunque pueden acceder y actualizar datos, no representan directamente los datos. Tienen un tiempo de vida corto y son sin estado. No permanecen tras una caída del contenedor. Estos componentes permiten que las aplicaciones u otros componentes, se comuniquen intercambiando mensajes de modo que el componente que envía el mensaje no necesita esperar q que el receptor de éste, reciba o procese el mensaje.

A continuación mostramos, de un modo general, las distintas partes de un EJB. Es necesario tener en cuenta que un message-driven bean no será directamente accesible por los clientes.

a) Enterprise Bean Class (Clase del Enterprise Bean). Esta clase es el componente medular de un EJB donde se definen todas las funciones realizadas por un EJB, ya sean lógica de proceso o de acceso a datos. Incluye además la activación del EJB y su destrucción. En el caso de Message-Driven bean, el contenedor invoca el método `onMessage` cuando llega un mensaje al bean a servir. Dependiendo del tipo de bean, esta clase implementará la interfaz `javax.ejb.EntityBean`, `javax.ejb.SessionBean` o bien, `javax.ejb.MessageDrivenBean`.

b) Home Interface (Interfaz local). Esta interfaz, como cualquier otra interfaz en Java, define un esqueleto para las funciones utilizadas por la clase definida anteriormente. Incluye por tanto métodos de activación y desactivación de un bean, junto a métodos de lógica de negocio. Debido al diseño distribuido de la arquitectura EJB, un cliente no comunica directamente con la clase del EJB, sino con esta interfaz. La interfaz de un EJB con un cliente remoto debe extender `javax.ejb.EJBHome`, si los clientes son locales extenderá a `javax.ejb.EJBLocalHome`.

c) Component Interface (Interfaz de Componente). Define una serie de métodos disponibles por los clientes, que pueden ser locales y/o remotos. Si el EJB ofrece una vista remota (clientes remotos) debe extender `javax.ejb.EJBObject`. Si la vista es local (clientes locales), debe extender `javax.ejb.EJBLocalObject`. Las implementaciones de esta interfaz son generadas por el contenedor de EJB y delegan la invocación de los métodos de negocio en una instancia de la clase del EJB (Enterprise Bean Class).

En el descriptor de despliegue de EJB se parametriza el código de la clase del EJB (Enterprise Bean Class) definiendo los parámetros que varían según el entorno de ejecución. Estos parámetros pueden ser bases de datos, servidores de páginas, privilegios de usuarios, etc. También indica al contenedor de qué tipo es cada EJB.

5.6 PATRONES DE DISEÑO

La utilización de los Patrones de Diseño en el desarrollo de aplicaciones mejora la calidad de las aplicaciones centralizando y encapsulando algunos mecanismos como servicios de seguridad, recuperación de contenidos y navegación, haciendo la aplicación mucho más fácil de mantener, sencilla y limpia.

Como se presentó en el capítulo 4 SOLUCIÓN PROPUESTA: J2EE, en el apartado 4.5 PATRONES DE DISEÑO, un patrón J2EE está documentado según una plantilla que define el contexto bajo el cual existe un determinado patrón; problema al que se enfrenta el desarrollador; motivación y razones a afectan al problema y a la solución; solución en detalle con estructuras, diagramas y clases que definen objetos, relaciones entre ellos y estrategias; consecuencias incluyendo las ventajas y desventajas de aplicar un patrón determinado; y patrones relacionados con el patrón en cuestión.

Existen multitud de patrones que proporcionan soluciones a distintos problemas. En este capítulo se presentan sólo algunos de ellos. Para un estudio más profundo de los Patrones de Diseño J2EE se recomienda acudir a la documentación que Sun Microsystems ofrece a través de su portal bajo el nombre de Core J2EE Patterns.

5.6.1 Intercepting Filter

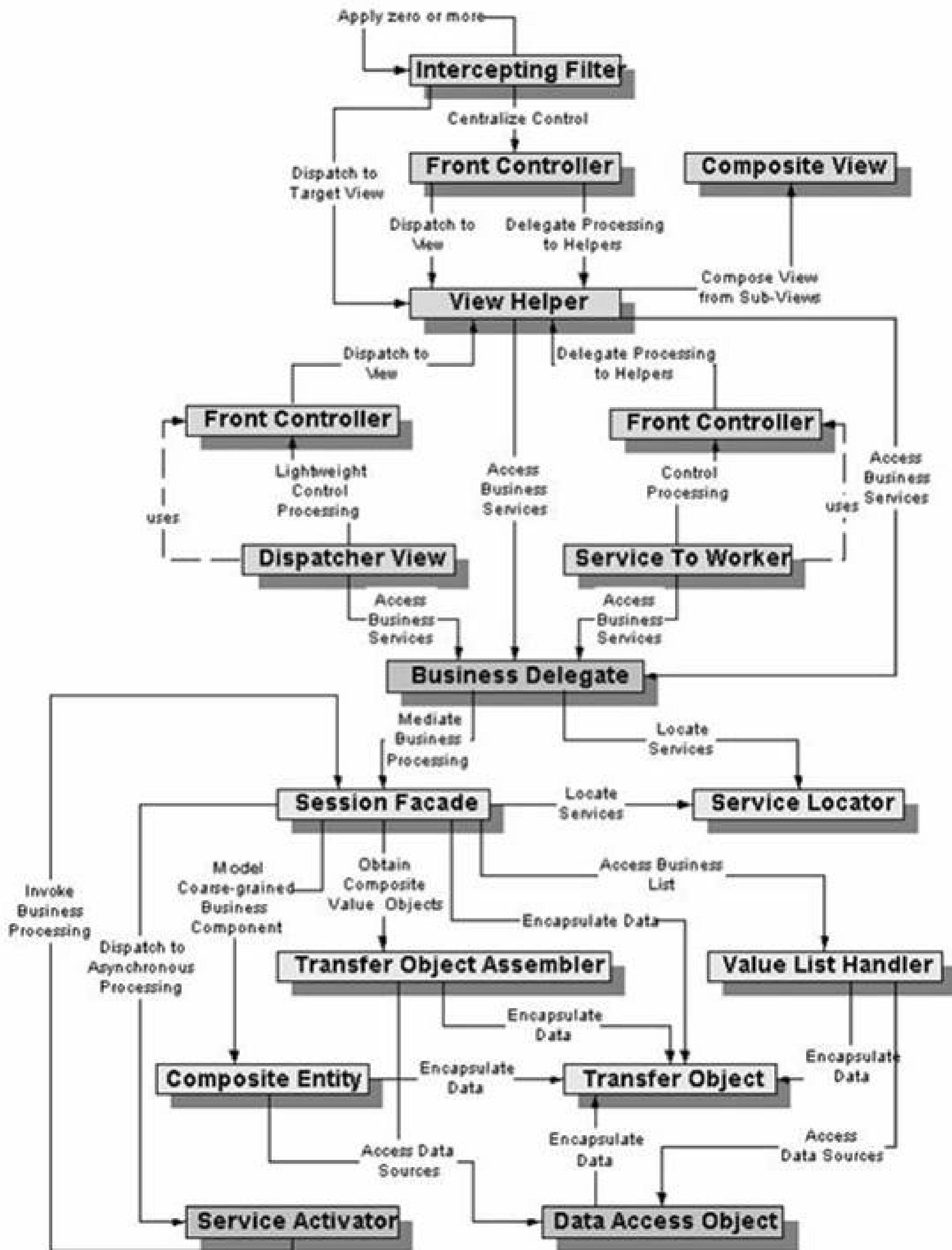
La solución que proporciona este patrón es el uso de filtros en cadena para procesar servicios comunes interceptando las peticiones entrantes y las respuestas salientes, permitiendo un pre y post-procesamiento, pudiendo añadir y eliminar estos filtros a discreción sin necesitar cambios en nuestro código existente. Los filtros pueden ser por ejemplo de registro, autenticación, validación de sesión, etc.

Este patrón está relacionado con el *Front Controller* y resuelven problemas similares. Este patrón está centrando en el filtrado de peticiones, mientras que el Front Controller elige la acción correspondiente al recibir una petición, esto podría realizar un filtrado si la petición no llega convenientemente filtrada.

5.6.2 Front Controller

Un controlador maneja el control de las peticiones, incluyendo la invocación de los servicios de seguridad como autenticación y autorización, delega el procesamiento de negocio, controla la elección de la vista adecuada, maneja errores, y selecciona las estrategias de creación de contenido.

Crea un punto de acceso centralizado, ya que recibe todas las peticiones entrantes remitiendo a su vez, si es necesario, cada petición al gestor de peticiones *(Dispatcher)* adecuado que se encargará de gestionar la construcción de las respuestas.



F. 5-3: Patrones de Diseño J2EE.

(Core J2EE Patterns: Best Practices and Design Strategies authored by architects from the Sun Java Center.)

Utilizando este patrón centralizamos el control logrando reducir el acoplamiento entre lógica de negocio y vista, aplicando directamente el modelo MVC.

Existen varias estrategias para la implementación de este patrón. La utilización de páginas JSP como controlador, o la utilización de Servlets como controlador. Tal y como se ha insistido constantemente en este capítulo se recomienda la estrategia que utilizar un servlet como controlador, este servlet haría uso de las clases de ayuda *(Helpers)* en los que el controlador delegaría las responsabilidades.

La implementación típica del controlador suele ser un servlet que extiende a `HttpServlet`, en el caso de aplicaciones Web.

5.6.3 View Helper

Encapsula lógica correspondiente a la presentación y el acceso a datos. Suelen ser JavaBeans y evita el uso de scriptlets en la vista. El uso de etiquetas personalizadas o estándares podría considerarse implementaciones de este patrón con las estrategias JavaBeans Helper y JSP View respectivamente.

Cuando se utilizan de hojas de estilo para dar formato a la vista también se está haciendo uso de este patrón.

5.6.4 Composite View

Gestiona los diferentes elementos de la vista por medio de una plantilla que hace la representación de las vistas más manejable. Con este patrón se tiende a utilizar vistas compuestas que están formadas por múltiples subvistas.

El abuso de estas subdivisiones puede tener un efecto negativo en el rendimiento por lo que es conveniente alcanzar un compromiso entre modularidad y mantenimiento frente a rendimiento.

5.6.5 Dispatcher View

Este patrón es en realidad la solución de utilizar conjuntamente los patrones de Front Controller, Dispatcher (gestor de peticiones) y View Helper.

5.6.6 Business Delegate

Este patrón es utilizado en sistemas multi-capa distribuidos cuando es necesaria la invocación de métodos remotos para enviar y recibir datos entre las distintas capas del sistema.

Evita que los componentes de la capa de presentación interactúen directamente con los servicios de negocio y sean vulnerables a los cambios en la capa de negocio. Si no se empleara este patrón sería necesario exponer la implementación de la interfaz del servicio de negocio a la capa de presentación obteniendo una fuerte dependencia entre ambas capas.

Este patrón reduce el acoplamiento entre los clientes y la capa de negocio, ocultando los detalles de la implementación de los servicios de negocios y búsquedas y accesos asociados en los sistemas distribuidos.

Otras de las ventajas del uso de este patrón es la posibilidad de dotarlo de la capacidad de almacenar resultados y referencias a servicios remotos que puede mejorar considerablemente el rendimiento.

Puede utilizarse como interfaz entre distintas capas, ya que no se restringe su uso a la separación de las capas de presentación y modelo.

Normalmente el Business Delegate oculta a otro patrón, el *Session Façade*, ocultando la tecnología utilizada en el modelo.

5.6.7 Session Façade

Es una aplicación del patrón *Façade* en el entorno de sesiones. El patrón Façade, o Fachada, sustituye las interfaces de una serie de clases bajo una sola interfaz por lo que provee un acceso unificado a un subsistema de una aplicación.

Los sistemas se estructuran en subsistemas formados por patrones y clases que los implementan, con dependencias entre ellos. Con la utilización de este patrón se intenta evitar que un cliente, al acceder un sistema (o subsistema) necesite acceder a más de una clase provocando dependencia de cada una de ellas.

Este patrón representa una capa controladora entre los clientes y la capa de negocio, abstrae la complejidad de esta última y le presenta al cliente una interfaz sencilla reduciendo el acoplamiento y dependencia entre ellos. Encapsula la complejidad de las interacciones entre objetos exponiendo sólo las interfaces requeridas y proporcionando un acceso uniforme a los clientes.

Se trata de una capa sencilla en el modelo que da soporte directo para implementar los casos de uso y oculta la tecnología de acceso a datos realizando las funciones de controlador dentro del modelo.

Las capas inferiores no tienen conocimiento de la capa Façade. Este punto de acceso a las capas inferiores del modelo proporciona un desacoplo que facilita la posibilidad de cambios en las capas inferiores del modelo sin que los clientes se vean afectados.

5.6.8 Service Locator

En sistemas distribuidos se utiliza para abstraer la utilización de JNDI y ocultar las complejidades asociadas a la creación del contexto, búsquedas de objetos y creación de objetos tipo EJB.

5.6.9 Value List Handler (Page–by–Page Iterator)

Se utiliza para controlar la búsqueda y hacer caché de resultados que se proporcionan al cliente con un tamaño y desplazamiento que cumpla con los requisitos. Se utiliza un `Iterator` que nos ofrece una interfaz estándar para recorrer una estructura de datos sin que nos tengamos que preocupar por la representación interna de los datos y proporcionando a nuestro código independencia de la estructura de los datos.

Puede ayudar a acceder a una gran lista de *Transfer Objects* (patrón que representa los objetos de negocios) de manera eficiente. Este patrón controla la funcionalidad de la ejecución de consulta y el caché de resultados. Puede acceder directamente a un objeto de acceso a datos (implementación del patrón *Data Access Object*, DAO) y almacenar los resultados obtenidos como una colección de Transfer Objects.

Este patrón, al solicitar objetos por bloques, proporciona una mayor flexibilidad de consulta y mejora el rendimiento de la red.

5.6.10 Transfer Object (Value Object, VO)

Es uno de los patrones más utilizado. Se utiliza para la comunicación, transporte e intercambios de datos negocio especialmente útil en aplicaciones distribuidas.

Se implementa con un objeto que extiende la interfaz `Serializable`. Es una representación estado/valor de un objeto y se utiliza para encapsular datos de negocio. Se trata de un JavaBeans, con los atributos propios del objeto de dominio con carácter privado, y métodos `get` y `set` para acceder y modificar sus valores.

Existen varios enfoques de este patrón. Cuando el Transfer Object modela una tabla relacional y los atributos de la clase son los campos de la tabla, se está utilizando un Domain Value Object. Si se encapsulan atributos de varias tablas, creando una clase que combine sólo los atributos requeridos, se está utilizando un Custom Value Object.

Este patrón, en el contexto JDBC, nos permite representar un conjunto de atributos procedentes de uno o varios objetos de dominio; en el contexto EJB representa eficiencia. Sin embargo, hay que tener cuidado en el diseño porque hay casos en el que puede contener información obsoleta.

5.6.10.1 Interfaz Serializable

La interfaz `java.io.Serializable`, convierte automáticamente el estado de una instancia de una clase serializable en un vector de bytes y a partir de éste puede regenerar una instancia con el mismo estado pudiendo estar en cualquier máquina virtual lo que facilita su utilización en sistemas distribuidos. Puesto que los tipos básicos y muchas clases comunes son serializables, no suele ser necesario implementar ningún método para que el Transfer Object sea serializable, suele bastar con extender la interfaz.

5.6.11 Data Access Object (DAO)

Este patrón es muy utilizado en el contexto JDBC. Abstrae y encapsula todos los accesos a la fuente de datos aislando los objetos de negocio de una implementación particular de una implementación particular de la persistencia. El objeto DAO maneja la conexión con la fuente de datos para obtener y modificar datos implementando los mecanismos de acceso requeridos para trabaja con la fuente de datos.

Para acceder a datos resistentes en bases de datos relacionales, el API JDBC proporciona acceso y manipulación de datos utilizando sentencias SQL. Sin embargo, incluso en estos entornos de bases de datos relacionales, la actual sintaxis y formato de las sentencias SQL pueden variar dependiendo de la base de datos en particular.

Pero las diferencias son mayores cuando se utilizan distintos mecanismos para el almacenamiento de datos. Para evitar dependencias directas de nuestro código con los mecanismos de almacenamiento de datos es especialmente útil la aplicación de este patrón de diseño.

Este patrón, al evitar estas dependencias, hace mucho menos complicada y tediosa la migración de la aplicación de un tipo de fuente de datos a otro. Permitirá una fácil configuración para que componentes EJB, JSP o servlets trabajen con la información ubicada en sistemas B2B, LDAP, ya sean del tipo bases de datos relacionales (RDBMS), bases de datos orientadas a objeto (OODBMS), documentos XML, ficheros planos, etc. Esto podrá ser así aun cuando las API para el almacenamiento varíe según el vendedor del producto o sean API no estándares.

El uso de un objeto de acceso a dato (DAO), proporciona una solución frente a la diversidad de almacenamiento persistente abstrayendo y encapsulando todo acceso a la fuente de datos. Este objeto controla y maneja la conexión con la fuente para almacenar y obtener datos.

Este patrón de acceso a datos proporciona una capa que oculta completamente posibles cambios en la implementación de la fuente de datos, permitiéndole adoptar diferentes esquemas de almacenamiento sin afectar a los clientes o componentes de negocio. Actúa como un adaptador entre un componente y la fuente de datos, permitiendo un acceso transparente a la fuente de datos.

Se puede implementar la estrategia de DAO *Factory*. Una clase abstracta define los métodos por cada DAO que pueda ser creado y las factorías concretas implementarán este método. También se puede crear un método get en esta clase abstracta, que crea las factorías concretas. Estas factorías permiten obtener una instancia de un DAO apropiado para la aplicación, facilitando la instalación y la configuración. Además estas factorías pueden configurarse para ser accesibles vía JNDI.

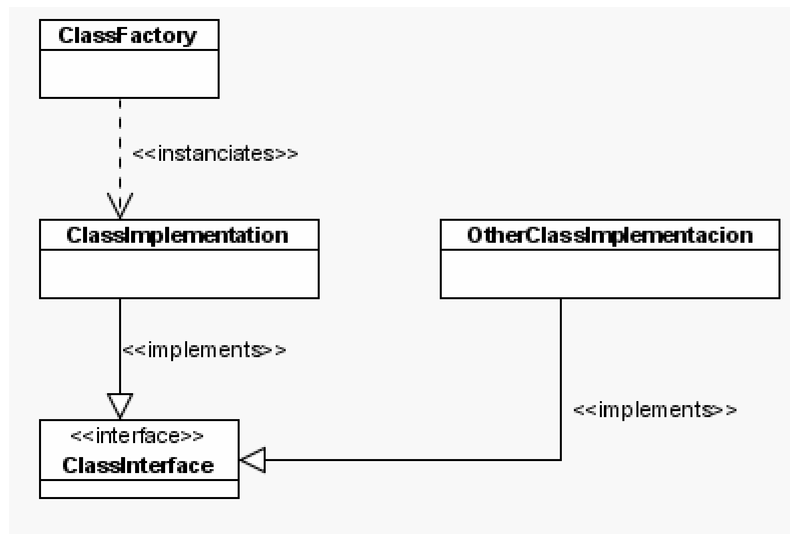
5.6.11.1 Factorías

Este patrón de factorías o factory, nos facilitará el desarrollo en capas y la independencia de éstas. Se utilizan interfaces abstractas para poder configurar la implementación a utilizar según necesidades de la aplicación.

A continuación se muestra un diagrama general de este patrón. Se crean las interfaces abstractas en las que se definirán los métodos utilizados. Podrían existir varias

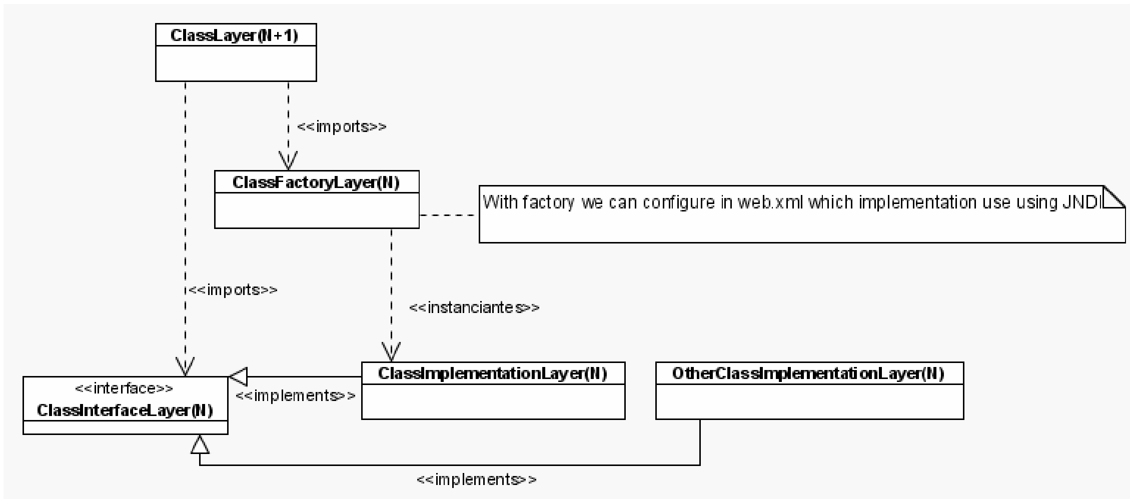
implementaciones de cada interfaz. Por último, otra clase es la que crea la instancia de la clase de implementación adecuada.

En la capa de acceso a datos este patrón es sumamente útil. Una clase interfaz definirá los métodos de interactuación a la base de datos y se crearán distintas implementaciones para los distintas fuentes de datos utilizados. De este modo, se podrían tener distintas implementaciones según se utilice almacenaje en ficheros planos, LDAP, bases de datos relacionales u otros. Incluso, puede ser necesario proporcionar distintas implementaciones según la base de datos relacional utilizada, ya que en muchos casos las sentencias SQL válidas pueden variar en cada caso.



F. 5-4: Parón Factory (Factorías).

Junto a las interfaces e implementaciones, será necesaria la clase factory que invoca una instancia de la implementación adecuada de cada interfaz. Esta clase factory, podría obtener el nombre de la clase implementación adecuada, por ejemplo, desde un fichero de configuración o vía JNDI.



F. 5-5: Uso del patrón Factory.

El uso de este patrón, como muestra la figura F. 5-5: Uso del patrón Factory., permite realizar cambios en capas inferiores sin que estos afecten a capas superiores. En la figura se observa, como capas superiores utilizan las clases factory e interfaz, sin embargo, no referencian en ningún caso la implementación utilizada. Esto permite utilizar cualquier implementación de la interfaz sin provocar cambios en el resto del sistema.

En el caso de la capa de acceso, permite migrar a otras fuentes de almacenamiento de información sin afectar al resto de la aplicación. En la mayoría de los casos, tan sólo es necesario proporcionar la nueva implementación de la interfaz de acceso a datos y configurar los parámetros JNDI necesarios.

5.7 CONFIGURACIÓN

A continuación se muestran algunas de las características que pueden configurarse en tiempo de despliegue gracias a los descriptores de despliegue de la aplicación. Estos descriptores son `web.xml` relacionado con el contenedor Web y `ejb-jar.xml` relacionado con el contenedor EJB.

Inicialmente estos documentos XML se definían en una DTD, en las actuales versiones se utiliza XSDL (Schemas) que permite más flexibilidad en la definición de tipos.

Junto a estos ficheros pueden existir otros ficheros de configuración. Por ejemplo, si se utiliza un framework o para definir las bases de datos. También pueden configurarse algunas características de los contenedores en sus propios ficheros de configuración, pero estos ficheros no siempre pueden ser modificados por los desarrolladores.

5.7.1 Descriptor de despliegue de la aplicación, `web.xml`

El contenedor Web puede proveer a una aplicación con balance de carga y recuperación ante fallos migrando las sesiones de usuarios a nodos distribuidos. Los servlets distribuidos deben cumplir una serie de restricciones adicionales y se etiquetan como `<distributable>` en el descriptor de despliegue de la aplicación.

También pueden definirse los servlets utilizados en la aplicación y sus valores iniciales. Se utiliza la etiqueta `<servlet>` para definir cada servlet y dentro de ella se pueden utilizar, entre otras, `<servlet-name>` que da un nombre lógico al servlet (nombre para identificarlo en el contexto), `<servlet-class>` que relaciona el nombre lógico con la clase que corresponde al servlet, y con `<init-param>` pueden definirse varios parámetros de inicialización del servlet.

Con <servlet-mapping> se configura el mapeo entre peticiones HTTP y los distintos servlets de la aplicación. De esta forma, según la petición realizada, el contenedor pasará el control a un servlet u a otro.

Las etiquetas `<context-param>` y `<session-config>` permiten configuraciones de contexto y de sesión. Con `<welcome-file-list>` se puede definir una lista de posibles páginas de inicio de la aplicación.

Junto a los parámetros anteriores, en `web.xml`, también pueden definirse filtros, mapeo de filtros, mapeo de tipos MIME, páginas de error, uso de librerías de etiquetas y algunas características de internacionalización.

Otra característica importante es la posibilidad de definir valores disponibles vía JNDI utilizando las etiquetas `<env-entry>`, `<env-entry-name>`, `<env-entry-value>` y `<env-entry-type>`.

No todas las etiquetas que la especificación de servlets define son de uso obligatorio. Todas la etiquetas utilizadas estarán dentro del a etiqueta `<web-app>`.

5.7.2 Descriptor de despliegue de EJB, `ejb-jar.xml`

En el descriptor de despliegue `ejb-jar.xml`, se definen las características de los beans que debe gestionar el contenedor EJB.

Todas las etiquetas utilizadas se situarán dentro de la etiqueta `<ejb-jar>`. Con `<enterprise-beans>` se definirán los beans utilizados y sus características. Cada bean definido se iniciará con la etiqueta correspondiente según el tipo de bean, estas son, `<entity>`, `<session>` o `<message-driven>`. Dentro de estas etiquetas se definirán para cada bean características como nombre con `<ejb-name>`, interfaz local y remota con `<home>` y `<remote>`. También podrán definirse subtipos, parámetros accesibles vía JNDI, características de seguridad y de transacciones.

Si se utilizan EJB de entidad, tipo CMP, la lógica de acceso también se define el fichero `ejb-jar.xml`.

5.8 DESPLIGUE DE APLICACIONES J2EE

Los archivos JAR son usados para distribuir un conjunto de clases Java. Se utiliza para almacenar clases java compiladas. Junto a las clases compiladas incluye el fichero `META-INF/MANIFEST.MF` que determina como se utilizará el archivo .jar. Este formato otorga un nivel de compresión y reduce la carga administrativa de distribuir clases en java.

Para el despliegue de aplicaciones J2EE existen otros tipos de archivos, aunque estos son también archivos .jar que siguen una serie de convenciones.

Los archivos WAR (Web Archives), permiten agrupar un conjunto de clases y documentos que conforman una aplicación Web en Java para ser utilizado por el contenedor Web. Para empaquetar una aplicación (o parte de ella), ésta debe seguir la estructura de directorios que se muestra a continuación.

Estructura de un archivo WAR (Web-Archive):

- `/ *.html`, `*.jsp`, `*.css`, etc: En la raíz o subdirectorios, se incluyen los documentos que conforman la vista de la aplicación. Comúnmente se utilizan documentos HTML (`*.htm`,`*.html`), páginas JSP (`*.jsp`), hojas de estilo en cascada (`*.css`), etc.

- `/WEB-INF/web.xml` : El descriptor de despliegue `web.xml` se encuentra directamente bajo el directorio `WEB-INF`.

- `/WEB-INF/classes/` : Contiene las clases Java de la aplicación.

- `/WEB-INF/lib/` : Contiene las librerías de clases en .jar que serán utilizados por la aplicación.

- `/WEB-INF/` : En esta carpeta se pueden crear otras subcarpetas para organizar la aplicación. Por ejemplo se pueden utilizar otras carpetas para

almacenar descriptores de librerías de etiquetas utilizados, o bien para incluir los elementos necesarios si se utiliza un framework para el desarrollo.

Los EJB se distribuyen en un EJB-JAR y al igual que los anteriores deben seguir una estructura determinada.

La estructura de un EJB-JAR es la siguiente:

- `/ *.class` : Bajo el directorio raíz se encuentran las clases creadas para los EJB.
- `/META-INF/ejb-jar.xml` : Este archivo contiene ejb-jar.xml, descriptor de despliegue de EJB descrito en el apartado anterior.
- `/META-INF/*` : Bajo este directorio pueden incluirse otros ficheros de configuración utilizados para el contenedor EJB.

Tanto los `.war` como los `.ejb-jar` se utilizan también para modularizar el desarrollo de aplicaciones. Los primeros para aplicaciones Web con servlets y jsp, y los segundos para los desarrollos con EJB.

Por último nos encontramos con los archivos EARS (Enterprise Archives) son archivos `.war` y `.ejb-jar` agrupados. Se utilizan porque en muchos casos los fabricantes no diferencian entre su contenedor Web y su contenedor EJB, ofreciendo un producto con ambos integrados.

5.9 IMPLEMENTACIONES

A continuación mostramos algunas de las implementaciones de componentes y tecnologías J2EE. Existe una gran variedad de implementaciones de distintos fabricantes gracias al carácter estandarizado de la plataforma. Aunque en el mercado se pueden encontrar tanto implementaciones comerciales como libres, nos centraremos en éstas últimas. Es importante resaltar que estas implementaciones, aunque libres, son realmente potentes y ampliamente utilizadas en entornos empresariales exigentes.

5.9.1 Contenedores

Aunque existen servidores J2EE completos, es decir, que integran servidor Web y servidor EJB, hemos elegido implementaciones de estos contenedores individualmente. Recordamos, que un servidor Web integra un contenedor de servlets y un contenedor de JSP.

A veces, estos contenedores integran un servidor HTTP y otras no. En el caso de no integrarlo, para servir páginas que utilicen tecnologías distintas a HTML, JSP y Servlets, será necesario configurar también un servidor HTTP. Unos de los servidores HTTP más utilizados es el servidor Apache que es además de licencia libre.

5.9.1.1 Contenedor de servlets/JSP

5.9.1.1.a ***Tomcat***

Tomcat es una implementación libre y de código abierto de las tecnologías Servlets y JavaServer Pages desarrollada bajo el proyecto Jakarta de la Fundación Apache (Apache Software Foundation).

Sun ha adaptado e incluido el código de Tomcat como implementación de servlets y JSP en su implementación referencia de J2EE (J2EE RI). Mientras Sun ofrece una implementación de referencia para desarrollo de aplicaciones J2EE con todos los API incluidos en la especificación en su J2EE SDK, Tomcat ofrece sólo la implementación de los API de Servlets y JSP, la cual es

la que incluye el J2EE SDK de Sun. La licencia de la implementación de Sun J2EE SDK no permite su uso comercial o redistribución, aunque sí admite su uso gratuito para demostraciones, prototipos y fines educativos.

Tomcat si se puede utilizar para usos comerciales y no comerciales bajo la licencia de Apache (ASF license). Ofrece buenas prestaciones en cuanto a seguridad y potencia. La documentación oficial es un poco escasa sin embargo cantidad de usuarios ofrecen tutoriales en Internet.

Puesto que implementa sólo, y completamente, contenedor de servlets y JSP, si se desea proporcionar otro tipo de servicios se recomienda utilizar junto un servidor de HTTP, sin embargo, si sólo se utiliza HTML, JSP y Servlets, Tomcat puede trabajar en solitario. El servidor Apache, por ejemplo, sirve páginas web estáticas, CGI, y programas ejecutados por el servidor, tales como PHP (e incluso ASP).

Existen muchas versiones de Tomcat, además usualmente lo encontramos incluido en las IDE de desarrollo como Netbeans. Si se quiere utilizar stand-alone, en la documentación de este contenedor encontramos toda la información necesaria para su instalación. La configuración de Tomcat se hace sobre un fichero XML. En la propia web de desarrolladores de Sun aparecen cursos prácticos de JSP y servlets usando Tomcat.

Normalmente, si utilizamos servidores o contenedores, e incluso IDE, escritos en Java necesitaremos instalar el JDK (Java Development Kit) no es suficiente con el JRE (Java Runtime Environment). Sun nos proporciona los JDK actualizados y gratuitos, IBM también los ofrece pero aunque son algo más rápidos no siempre están actualizados.

Existen otros contenedores de servlets (no necesariamente servidores J2EE), los principales son los siguientes: Resin, de Cuacho Technologies, un motor especialmente enfocado al servicio de páginas XML, con licencia libre para desarrolladores, fácil de instalar, con soporte para Java y JavaScript e incluye un lenguaje de plantillas llamado XTP; BEA Weblogic, servidor de aplicaciones de alto nivel no gratuito, escrito íntegramente en Java y combinado con otros productos como un servidor de bases de datos para XML llamado Tuxedo; Jrun, de Macromedia, servidor de aplicaciones de Java, no gratuito pero de menor coste que el anterior (además existen versiones gratuitas de evaluación) y de prestaciones medias; Lutris Enhydra,

es también un servidor gratuito Open Source (aunque las versiones más actualizadas son de pago), enfocado también a servir XML y para plataformas móviles.

5.9.1.1.b ***Jetty***

La característica que hace este contendor distinto al anterior es la incorporación de un servidor HTTP. Este servidor por tanto, puede servir tanto servlets y JSP, como otras tecnologías tales como CGI, PHP, ASP. Es 100% java y también se distribuye bajo la licencia de Apache, por lo que es también libre.

Tiene la misma funcionalidad que Tomcat trabajando conjuntamente con el servidor Apache, sin embargo, el uso de Jetty no está tan extendido como el uso de Tomcat.

5.9.1.2 Contenedor EJB

Como implementaciones de contenedor EJB presentaremos tres de los más extendidos, los tres con certificado J2EE 1.4 y ampliamente utilizados en aplicaciones de e-business.

JBoss AS fue el primer servidor de aplicaciones de código abierto disponible en el mercado y el más popular de los tres que presentamos. JBoss AS puede ser descargado, utilizado, incrustado, y distribuido sin restricciones por la licencia. Recientemente JBoss ha sido comprado por Red Hat aunque se distribuye bajo los términos de la licencia de Apache.

JOnAS es un servidor de aplicaciones J2EE de Código abierto implementado en Java que forma parte de la iniciativa de Código abierto de ObjectWeb con colaboración de empresas como Bull, France Telecom, Red Hat y MySQL, entre otras asociaciones e individuos.

Geronimo es el servidor de aplicaciones J2EE de la Fundación de Software Apache y se distribuye bajo esta licencia. Empresas como IBM ofrecen soluciones basadas en este servidor.

5.9.2 Librerías de etiquetas

Algunas compañías proporcionan librerías de etiquetas que están íntimamente integradas con sus herramientas y línea de productos J2EE. Otras organizaciones proporcionan librerías de etiquetas para uso general en aplicaciones J2EE.

Apache, a través del proyecto Jakarta, ofrece una serie de soluciones Java de código abierto bajo una licencia de software abierto. El proyecto Jakarta TagLibs ofrece una gran variedad de etiquetas personalizadas JSP. Entre estas etiquetas incluye la librería de etiquetas estándar de Java, JSTL (The Standard Tag Library), estandarizadas por la Comunidad Java, JSR-52.

JSTL proporciona una rica capa de funcionalidad portable para páginas JSP y están disponibles para cualquier contenedor JSP.

No podemos olvidar que tanto JSP, como JSF, también incluyen un conjunto de librerías de etiquetas.

El proyecto Jakarta ofrece una gran variedad de etiquetas que proporcionan diversos beneficios. Quizás las más extendidas sean las etiquetas que incluye el framework de Struts. Otro framework, Bean Scripting Framework, ofrece etiquetas para la incorporación de scripting en otros lenguajes como JavaScript, VBScript, Perl, Python y otros, en una página JSP. Existen etiquetas para internacionalización, manejo de sesión, caché de páginas, transformaciones XSL, etc. Otros distribuidores de etiquetas son Coldjava, BEA WebLogic, JRun, OpenSymphony, Oracle, y muchas más. Podemos encontrar también etiquetas orientas a distintas áreas como librerías con funciones matemáticas. Sun Microsystem ofrece información sobre los distintos distribuidores de etiquetas.

5.9.3 Framework: JSF y Struts

La única implementación estándar de un framework de desarrollo basado en el patrón MVC es JavaServer Faces. Este framework es posterior y adopta mucha de las características de Struts, framework que inicialmente era un subproyecto del proyecto Jakarta de la Fundación de

Software Apache, pero que debido a la importancia que ha alcanzado se conformado como un proyecto independiente de Jakarta.

Aunque actualmente JavaServer Faces y Struts aportan soluciones similares, en un futuro pretenden seguir líneas de evolución distintas. El primero se centra en conseguir una interfaz de usuario más rica, y por lo tanto, se centra en la presentación. Struts pretende centrarse en las características propias del controlador.

Existen otros frameworks en el mercado, variados según las funcionalidades, sin embargo, estos dos frameworks son los más extendidos actualmente para J2EE.

5.9.4 Patrones de Diseño

Si hiciéramos un estudio más profundo se las distintas tecnologías y herramientas utilizadas, se observaría que muchas de ellas son implementaciones de los patrones de diseño anteriormente comentados.

El contenedor EJB implementa algunos de los patrones de diseño relacionados con el modelo de la aplicación. Por ejemplo, la capa de acceso a datos la implementa de un modo transparente al desarrollador, de forma que un cambio en la base de datos sólo requiere un cambio en los descriptores de despliegue. Aún así, los patrones de las capas de negocio, a menudo son implementados programáticamente.

Para formarnos una idea general, mostramos a continuación, ejemplos básicos de implementaciones de algunos de los patrones de diseño de las capas de vista y controlador. Estas implementaciones las encontramos en como Servlets y JSP, o en los frameworks y etiquetas de JSF, Struts y JSTL.

5.9.4.1 Intercepting Filter

Desde la especificación de Servlets 2.3, se incluye la posibilidad de utilización de este patrón implementando los métodos `init`, `destroy` y `doFilter` de la interfaz `javax.servlet.Filter`.

Una vez implementados se configuran en el descriptor de despliegue con las etiquetas `<filter>`, `<filter-name>`, `<filter-class>`, permitiendo, en este fichero, configurar valores iniciales con `<init-param>` accesibles desde el filtro por la interfaz `javax.servlet.FilterConfig`.

El mapeo de filtros con la URL adecuada se realiza con la etiqueta `<filter-mapping>`. Puesto que el mapeo se corresponde con patrones de URL, es interesente estudiar para cada aplicación las ventajas de utilizar los filtros de servlets o no. En algunos casos puede no interesar utilizar el patrón de Intercepting Filter y realizar el filtrado más adelante, por ejemplo en el Front Controller.

También es posible definir cadenas de filtros gracias a la interfaz `javax.servlet.FilterChain`.

5.9.4.2 Front Controller

El framework Struts proporciona una implementación del Front Controller, por lo que al utilizar este framework se estaría haciendo uso de este patrón.

También proporciona una serie de Helpers como por ejemplo las clases Action. Al utilizar Struts como controlador el programador extiende el método `execute` la clase `org.apache.struts.action.Action` por cada acción que requiere la invocación de la lógica de negocio. Esta estrategia de utilizar una clase básica que contenga las implementaciones comunes de las clases Helper, y estas clases extiendan esta clase básica, se le conoce como estrategia Base Font.

La estrategia que se utiliza para elegir el Helper adecuado a una petición es el de mapeo de URL en el fichero de configuración de Struts. Cuando se utiliza Struts como Front Controller, debe configurarse en el descriptor de despliegue `web.xml` el servlet `org.apache.struts.action.ActionServlet` (o una extensión de éste), la ruta del fichero de configuración de Struts y el mapeo de las URL que el contenedor pasará a Struts.

5.9.4.3 View Helper

Struts proporciona implementaciones de este patrón con sus librerías de etiquetas e interfaces JavaBeans para los formularios.

5.9.4.4 Composite View

La etiqueta `include` de JSP es una muestra de los beneficios de este patrón. Además, muchos frameworks proporcionan etiquetas para el uso de plantillas en las composiciones de las vistas.

Struts proporciona implementaciones de este patrón con sus librerías de etiquetas `template` y `tiles`. Ambas librerías proporcionan mecanismos para el manejo de plantillas en la composición de las vistas.