Modelos de desarrollo de aplicaciones distribuidas con J2EE

La plataforma J2EE es a menudo críticada por su complejidad lo que dificulta su adopción por parte de los desarrolladores menos experimentados. La cantidad de modelos de desarrollo, frameworks, APIs oficiales, etc., ofrecen una gran flexibilidad y funcionalidad pero añaden más confusión a este grupo de desarrolladores. Por si fuera poco, se pueden distinguir claramente entre los estándares oficiales promocionados por las empresas impulsoras de J2EE y los estándares de facto utilizados en el día a día del desarrollador. En este artículo se intentará exponer de manera clara las alternativas de diseño más recomendables a la hora de crear una aplicación J2EE, creando así un catálogo de modelos y buenas prácticas para la creación de aplicaciones en esta plataforma.

1 . El modelo de desarrollo de J2EE

J2EE es una especificación abierta que define una plataforma para el desarrollo de aplicaciones distribuidas orientadas a la empresa. El desarrollo de aplicaciones bajo J2EE, al igual que otras plataformas como .NET, se basa en la separación de capas. Esta separación de capas permite una delimitación de responsabilidades a la vez que satisface los requisitos no funcionales de este tipo de aplicaciones (escalabilidad, extensibilidad, flexibilidad, etc.) y disminuye el acoplamiento entre las diferentes partes de las mismas.

El número de capas de una aplicación J2EE variará según su complejidad y/o necesidades. Aún así, la estructura sugerida como modelo para el desarrollo de aplicaciones J2EE es la que se aprecia en la figura 1 :

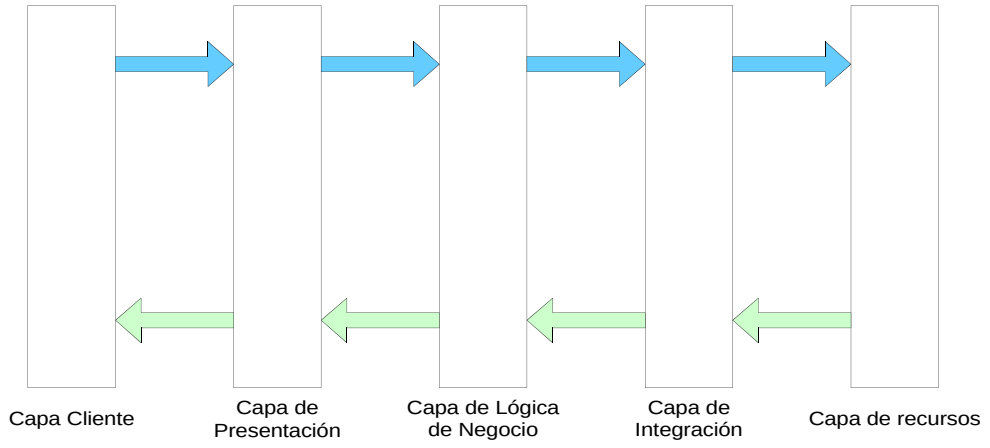

Figura 1 : Diseño en capas de una aplicación distribuida[1]

Esta estructura en capas se ha constituido en un estándar a la hora de desarrollar aplicaciones distribuidas para sistemas empresariales dejando obsoleto el clásico modelo cliente-servidor. Las dos plataformas empresariales más importantes de la actualidad, J2EE y .NET, proponen este esquema de desarrollo de aplicaciones. Esta estandarización es bastante importante ya que si la arquitectura es clara y está diseñada en términos de alto nivel sin basarse en código explícito de cada plataforma será mucha más fácil una posible migración.

El rol de cada una de estas capas es similar en cada ambas plataformas variando únicamente la tecnología sobre la que se sustentan cada una de dichas capas :

- **Capa de cliente :** Es la capa donde se localizan los diferentes clientes de nuestras aplicaciones. Estos clientes normalmente serán de tipos y funcionalidades muy diversos.
- **Capa de presentación:** La capa de presentación contiene toda la lógica de interacción entre el usuario y la aplicación. Además, es la capa encargada de controlar la interacción entre el usuario y la lógica de negocio generando las vistas necesarias para mostrar información al usuario en la forma y formatos más adecuados.
- **Capa de lógica de negocio** : En esta capa se localiza el código y las reglas que sirven como núcleo de nuestras aplicaciones empresariales. Como tal es importante que cumpla una serie de características fundamentales como la fácil extensibilidad y mantenibilidad, alta reutilización, alta flexibilidad y fácil adopción de tecnologías, etc.

- **Capa de integración** : Esta es la capa donde se realizan diferentes tareas de integración con otros sistemas como

1 Este es el diseño recomendado. Los nombres pueden variar de una publicación a otra. En este caso los nombres han sido extraidos directamente del libro *Core J2EE Patterns 2nd Edition* por tratarse de la referencia oficial de SUN

son el acceso a datos, el acceso a sistemas legacy, la aplicación de motores de reglas o de *workflow*, etc. Es importante que estos sistemas sean especialmente extensibles de modo que sea fácil añadir nuevas fuentes sin que esto afecte a la capa de lógica de negocio.

- **Capa de sistemas legacy** : En esta capa se localizan los diferentes sistemas de información disponibles en nuestra empresa. Bases de datos, sistemas de ficheros COBOL, sistemas 4GL, Tuxedo, Siebel, SAP, etc., son sólo algunos de los inquilinos que nos encontraremos en este nivel de la arquitectura. Es fundamental tener algún sistema que nos permita el acceso flexible a sistemas tan heterogéneos.

En los siguientes apartados analizaremos cada una de estas capas, y cuales son los patrones de desarrollo más utilizados comúnmente en las mismas, centrándonos en las que sin duda presentan más variación: la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos. Además dedicaremos apartados especiales a aspectos como los servicios web o la generación de código que presentan cada vez un papel más importante dentro de las arquitecturas J2EE.

2. La capa cliente

A la hora de crear una aplicación empresarial con J2EE nos encontraremos habitualmente con los siguientes tipos de aplicaciones cliente:

- Aplicaciones de escritorio tradicionales.
- Navegadores web.
- Aplicaciones para pequeños dispositivos.

Los dos primeros tipos son, sin duda, los más habituales. El debate entre las ventajas y desventajas entre estos tipos, aún siendo uno de los factores más importantes a la hora de decidir los clientes que soportará nuestra aplicación, está totalmente fuera del contexto de este artículo y ya ha sido tratado en otras publicaciones. Nos centraremos pues en los siguientes párrafos en algunas de las diferencias más importantes entre estas alternativas a la hora de desarrollar aplicaciones distribuidas.

En el momento de crear aplicaciones en J2EE, la principal diferencia con la que se va a encontrar el desarrollador es que en una aplicación de escritorio, es la propia aplicación la encargada de manejar la lógica de presentación, mientras que en una aplicación web es el contenedor web el encargado de realizar dicha tarea. Obviamente, este último punto es un argumento importante a favor de las aplicaciones web ya que el renderizado en potentes servidores con gigas de memoria RAM será mucho más rápido y los clientes no se ven sobrecargados con esa tarea. Por otra parte, la arquitectura web nos puede resultar algo más costosa ya que nos puede obligar a la instalción de un servidor adicional para albergar el contenido web.

Otro factor a tener en cuenta es que en las aplicaciones de escritorio, la lógica de negocio es accesible únicamente a través de interfaces remotas. Esto obliga a la utilización del patrón Session FaÇade, ya que en caso contrario incurriríamos en grandes penalizaciones de rendimiento al tener que acceder remotamente a nuestros EJBs. En caso de usar una aplicación web, esto no es necesario cuando el contenedor web y el contenedor de EJBs se alojan en el mismo servidor físico.

Por último, existe un problema adicional asociado a las aplicaciones de escritorio y que resulta bastante importante. La creación de aplicaciones de escritorio que accedan a servidores J2EE puede conllevar la instalación de una serie de librerías en el cliente que le permitan acceder al servidor de aplicaciones. Esta instalación no supone ningún problema en aplicaciones web, pero puede ser algo realmente traumatizante en aplicaciones de escritorio, sobre todo teniendo en cuenta que un servidor de aplicaciones en concreto puede poner algunas restricciones a este proceso. IBM WebSphere, por ejemplo, obliga al uso de su JDK 1.3.1 para conseguir acceder a su servidor de aplicaciones desde una aplicación de escritorio al tiempo que el conjunto de librerías que necesitan estar desplegadas en el cliente es realmente abrumador, llegando en algunos casos (se pueden utilizar varios modos de ejecución) a superar los 30Mb.

Las consideraciones a tener en cuenta a la hora de desarrollar aplicaciones para pequeños dispositivos utilizando J2ME son las mismas que las que se han comentado anteriormente. Desde un dispositivo móvil podremos utilizar un navegador web o lanzar una aplicación propia que acceda al servidor de aplicaciones ya sea a través de servicios web, RMI, o cualquier otro sistema. La principal preocupación a la hora de desarrollar para estos dispositivos es sin duda sus limitaciones en cuanto a posibilidades, tipo y características del dispositivo, etc.

En caso de tener un aplicación para dispositivos móviles que no utilice para nada un navegador, tenemos una ligera variación del esquema que se describía para aplicaciones de escritorio. J2ME, utiliza HTTP como protocolo de comunicación para enviar y recibir datos desde un servidor de aplicaciones. Por ello la comunicación se hace siempre con un servlet que recibe el mensaje del dispositivo móvil, lo decodifica y a partir del resultado se pone en contacto con

la lógica de negocio para posteriormente enviar la respuesta al dispositivo móvil. Este servlet no realiza ninguna acción de presentación, tan sólo recibe y envía mensajes. Como consecuencia podíamos imaginarnos que tenemos un Servlet intermedio entre las capas de presentación y lógica de negocio y por el que pasan todas las peticiones y respuestas.

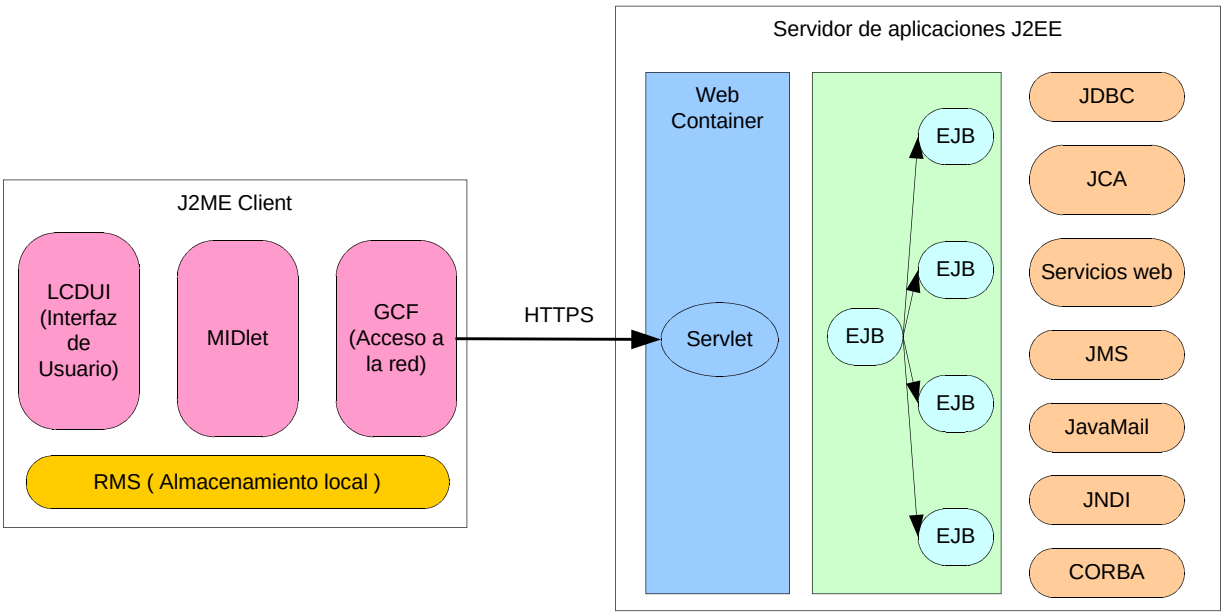

Figura 2 : Arquitectura de acceso a un servidor J2EE desde una aplicación J2ME

Está claro que ante tal variedad de sistemas cliente es necesario algún mecanismo que nos permita una sencilla migración de nuestros interfaces de cliente a diferentes plataformas. Ahí es donde se imponen los *frameworks* de generación dinámica de interfaces de usuario basados en XUL (XML User-Interface Language).

XUL es un lenguaje independiente de la plataforma que permite la definición de interfaces de usuario utilizando XML. Con XUL se pueden generar definiciones de interfaces de usuario totalmente independientes de la plataforma cliente y que después podrán ser fácilmente desplegadas en todo tipo de dispositivos: escritorios, navegadores web, PDAs, móviles, etc. Al estar basado en XML, XUL hereda algunas ventajas de éste como son la portabilidad, la facilidad de modificar e implementar interfaces, la posibilidad de incrustar XHTML u otros lenguajes basados en XML como MathML o SVG o la posibilidad de aplicar hojas de estilo para modificar fácilmente el aspecto del interfaz de usuario.

Existen multitud de librerías basadas en XUL. Entre las más conocidas destacan Luxor, XWT, Thinlets o SwingML.

2. La capa de presentación

Como se ha visto en el apartado anterior, la capa de presentación puede localizarse en una aplicación de escritorio o dentro de un contenedor web. Independientemente de su localización, la capa de presentación es la encargada de controlar la generación del contenido que se mostrará al usuario final. Esta generación comprende la comunicación con la capa de lógica de negocio para obtener los datos necesarios, el control del flujo de pantallas, el control de la interacción entre diferentes componentes, ensamblar las diferentes vistas que puedan formar una pantalla y limitar la información en base a perfiles de usuario y reglas de autorización, entre otras tareas.

Leido el párrafo anterior, es fácil comprender que la capa de presentación es una de las más complejas ya que engloba muchas tareas de diferente índole. Es por ello que hemos de ser cuidadosos en su creación para no penalizar en exceso factores como la extensibilidad. Esta capa ha de ser fácilmente mantenible y extensible ya que es la más sensible al cambio (la lógica de negocio de la empresa no es algo que cambie tan frecuentemente) y por lo tanto es muy importante que pequeñas modificaciones y añadidos no interfieran en el funcionamiento de las aplicaciones existentes.

Nos encontremos ante una aplicación de escritorio o una aplicación web, el patrón más común a la hora de implementar la capa de presentación es, sin duda alguna, el MVC (Model View Controller), que permite una separación cuasi-perfecta entre lo que se conoce como modelo (en nuestro caso será la lógica de negocio), el controlador y la vista.

Aunque no se pretenda en este artículo adentrarnos dentro del patrón MVC conviene explicar su funcionamiento básico para comprender mejor los *frameworks* de los que se hablará a continuación. MVC define tres roles diferentes: el modelo, la vista y el controlador. El modelo es un objeto que se encarga de almacenar información sobre los datos y comportamiento de éstos en el dominio de la aplicación. La vista es una representación visual de los datos del modelo o

de una parte de los mismos. El controlador se encarga de gestionar la interacción entre el modelo y las diferentes vistas, de modo que cuando el modelo cambie se actualicen automáticamente todas las vistas reflejando dichos cambios.

Las ventajas de este patrón son múltiples. Entre algunas de las más tratadas en la literatura que versa sobre este tema destacan:

- Separación de responsabilidades. Cada una de las partes de este patrón se encarga de tareas muy diferentes que se encuentran aisladas y no interfieren entre ellas.
- Múltiples vistas. El escaso acoplamiento entre las partes de este patrón permite que se puedan generar fácilmente múltiples vistas para un mismo modelo. Cada vez que el modelo se ve modificado todas las vistas se actualizan automáticamente.
- Tests de funcionalidad. La separación entre vista y modelo permite implementar de manera muy sencilla conjuntos de tests para probar el funcionamiento del modelo y del controlador utilizando una vista mucho más sencilla que la que tendrá la aplicación final. Una vez que se comprueba el correcto funcionamiento del modelo y el controlador se puede desarrollar una vista mucho más rica.

2.1 Aplicaciones de escritorio

Dentro de las aplicaciones de escritorio las dos tendencias actuales más importantes son utilizar *Swing* o *JFace*. Ambas son implementaciones del modelo MVC, siendo quizás Swing una implementación más a bajo nivel mientras que *JFace* facilita en mayor medida la realización de tareas comunes y el uso de componentes como árboles o tablas que generalmente son los más complejos.

2.2 Aplicaciones web

En este caso, el mundo de las aplicaciones web es muy diferente. Al ejecutarse en un servidor, pasa a ser necesario prestar atención a temas de sincronización de múltiples usuarios, prestación de calidad de servicio, tener mayor control transaccional, etc. En una aplicación de escritorio, estos factores no son tan importantes ya que la lógica de presentación se localiza en el mismo cliente. En las aplicaciones web, son múltiples los clientes que acceden simultáneamente a la lógica de presentación y por lo tanto es necesario prestar especial atención a todos esos factores.

2.2 Frameworks web

La primera decisión, y una de las más importantes a la hora de desarrollar una aplicación web, es decidir entre si se va a utilizar un *framework web* o si se van a implementar manualmente todas sus funcionalidades. Un *framework web*, es un conjunto librerías que proporcionan automáticamente una serie de servicios al desarrollador, de modo que éste ya no tiene que preocuparse de la implementación de los mismos. La mayor parte de estos frameworks implementan el modelo MVC y basan la creación de aplicaciones en dicho modelo. Los *frameworks* no basados en MVC es preferible descartarlos, ya que independientemente de su eficacia, aumentan la ligazón de dependencia. Recordemos que conviene mantener una estructura arquitectónica coherente con las tendencias y realizar nuestro diseño en base a dichas tendencias de modo que la migración en caso de presentarse problemas sea mucho más sencilla. Será mucho más fácil migrar una aplicación realizada en un framework MVC a otro framework de similares características, que intentar migrar una aplicación creada con un *framework spaguetti* que no siga ningún patrón de diseño claro.

Como muestra de la importancia de un *framework* web cabe destacar que tanto Pet *Store* como *Adventure Builder* recomiendan el uso de un framework web como base de las aplicaciones web creadas sobre J2EE. Ambos ejemplos de *Blueprints* utilizan un *framework* propio, denominado *WAF (Web Application Framework)*, que aunque básico ha servido como modelo de desarrollo para multitud de *frameworks*.

Los *frameworks web* no se quedan sólo en simples implementaciones de MVC sino que también ofrecen gran cantidad de servicios. Normalmente, cada *framework* posee un conjunto de *custom tags* propias que facilitan la creación de interfaces de usuario, formateo de XML, acceso a bases de datos, etc. No sólo eso, los servicios que ofrecen suelen ser múltiples y muy diversos: control del flujo entre páginas, *templates*, filtros de peticiones, etc. Todos estos servicios suponen un ahorro de tiempo excepcional a la hora de desarrollar aplicaciones web, y no sólo eso, sino que marcan claramente el camino a seguir ofreciendo una serie de componentes reusables y permitiendo que el desarrollador se centre en lo que realmente le interesa a la empresa, su lógica de negocio.

Existe una gran cantidad de *frameworks web* diferentes. Entre los más conocidos destacan el propio Struts, Tapestry,WebWork, WebMacro, Turbine, Barracuda o Expresso, todos ellos productos *Open Source*. Cabe destacar que entre todos estos sin duda el que se ha convertido en un estándar en el desarrollo web es *Struts*. Algunos de estos *frameworks* como *Expresso* están basados en *Struts*. Por otra parte, la mayor parte de servidores de aplicaciones propietarios como BEA WebLogic o IBM WebSphere utilizan internamente el *framework* de Apache.

Por último, es necesario hacer un pequeño hueco para el *framework* creado por Alberto Molpeceres llamado *cañamo* y que sirve como base del portal javaHispano. Este *framework* implementa el patrón MVC pero extendiendo el modelo MVC-II hasta permitir la salida en múltiples formatos, no sólo páginas JSPs sino también plantillas, XML, etc.

2.2 Soluciones sin un framework web

Como ya se ha señalado, no utilizar un *framework web* no es una solución recomendada, pero en determinados casos puede que nos veamos obligados a hacerlo. En estos casos, el modelo de desarrollo recomendado es el que se puede ver en la figura 3.

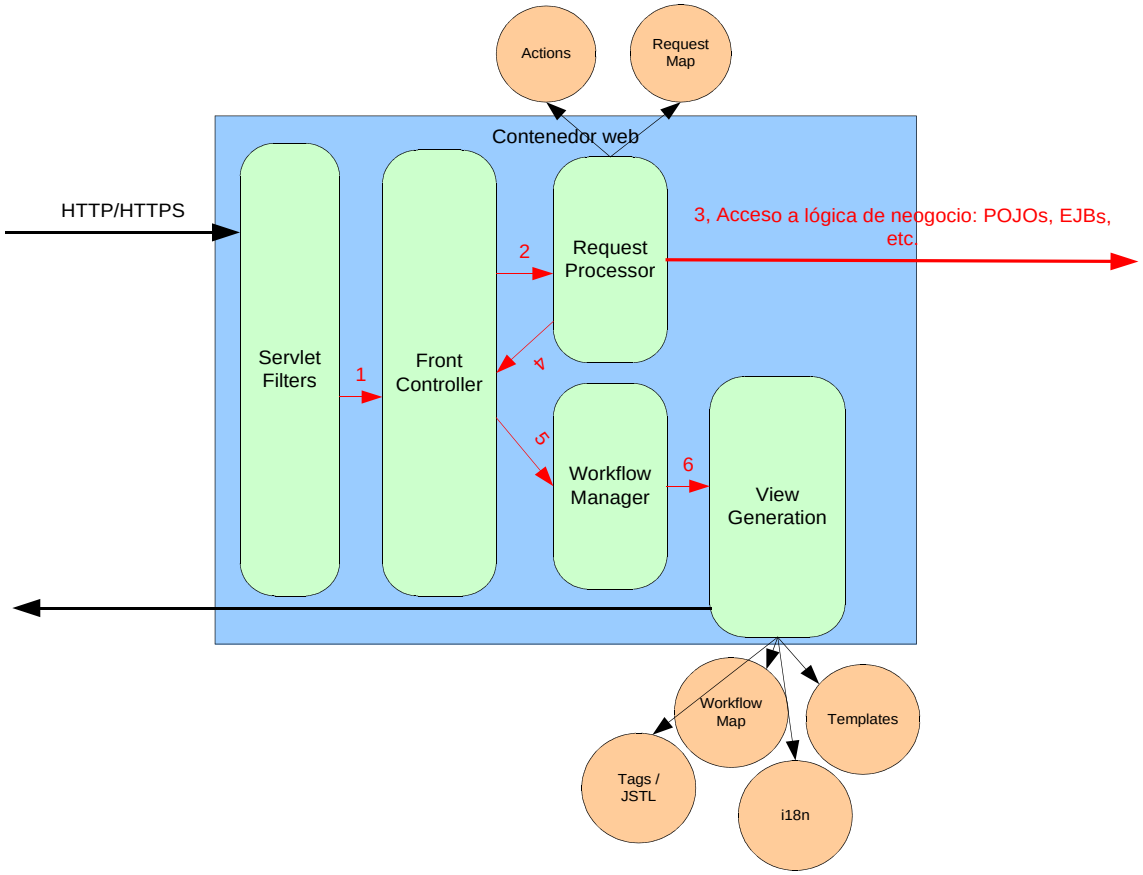

Figura 3 : Arquitectura web típica

En la figura se puede ver lo que se conoce como modelo MVC-II[2], que es el implementado por la mayoría de *frameworks web*. Normalmente tendremos una serie de Servlets (uno sólo si implementamos el patrón *Request Controller*) que se encargarán de recibir las peticiones. Estos servlets conectarán con la lógica de negocio y recogerán los datos en *JavaBeans* para por último, en base a los valores devueltos desde la lógica de negocio, decidir la vista (página JSP) que se mostrará y renderizar los datos obtenidos de desde la susodicha capa de lógica de negocio.

En el primer paso de la figura anterior se observa como la petición HTTP o HTTPS es procseada por una serie de *Servlets* que actúan de filtro realizando tareas comunes como puedan ser la autenticación, la auditoría, etc. Una vez que se aplicacn todos los filtros de manera secuencial se dirige la petición hacia el *Front Controller*. El *Front Controller* es el encargado de recibir las peticiones y procesarlas utilizando el *Request Processor*. Este componente, en base a un fichero de configuración de acciones decide la acción a ejecutar. La acción se encargará de realizar el proceso de negocio o de redirigir la petición a los componentes que realicen dicho proceso.

Una vez ejecutado el proceso de negocio el *Front Controller* utiliza un componente denominado *Workflow Manager* que es el encargado de decidir cual será la página de destino en base a otro fichero de configuración. Una vez que se conoce la página de destino se procede a generar la vista utilizando *Templates*, *custom tags* e internacionalizando la salida.

---

2 MVC-II es una evolución de MVC-I que consistía en una implementación del patrón MVC utilizando únicamente JSPs. MVC-II es muy parecido al patrón MVC tradicional con mínimas diferencias, como por ejemplo la imposibilidad técnica de implamentar el patrón *Observer* para notificar al cliente debido a las limitaciones del protocolo HTTP.

Realmente, la mayor parte de los componentes que aparecen en la figura 3 no son necesarios para la implementación de un modelo MVC-II, pero son recomendable ya que permiten una perfecta separación de roles y dan lugar a aplicaciones más flexibles, extensibles y fácilmente mantenibles.

Una de las recomendaciones más importantes a la hora de crear las páginas JSP que se mostrarán al usuario es el utilizar etiquetas personalizadas o *custom tags*. Estas etiquetas, son de muy fácil uso para diseñadores web con poco conocimiento de programación. En los primeros años de la tecnología JSP se tendió a colocar *scriptlets*, es decir, código Java, en el interior de las páginas JSP lo que tenía dos consecuencias importantes: primero, los diseñadores no sabían hacer estas páginas, y segundo, los diseñadores no comprendían las páginas creadas por otros diseñadores o programadores.

Las *custom tags* son etiquetas, con una síntaxis similar a las etiquetas HTML, por lo que su comprensión por parte de diseñadores web es mucho mayor. Además, estas etiquetas se integran muy bien con entornos de desarrollo web como *Macromedia Dreamweaver* con lo que la programación de JSP se simplifica enormemente. Hoy en día, la mayor parte de los *frameworks web* han implantado las *custom tags* como método de generación de vistas recomendado y casi todos ofrecen su propio conjunto de etiquetas personalizadas.

El conjunto de *custom tags* recomendado es JSTL (JSP Standard Tag Library *)*, un conjunto estándar de etiquetas personalizadas, resultado de la evolución del JSR-52, y formado por etiquetas de control de código (sentencias de iteración, sentencias condicionales, etc.), etiquetas de formateo e internacionalización, etiquetas de acceso y formateo de ficheros XML y por último un conjunto de etiquetas de acceso a base de datos.

2.3 Java Server Faces

Java Server Faces ha aparecido en el 2003 como un nuevo actor dentro del mundo de las aplicaciones web. Se trata de la implementación de referencia del JSR-127, que define un API estándar para la creación de interfaces de usuario desde el servidor.

Realmente, Java Server Faces es un *framework* cuya funcionalidad se asemeja muchísimo a *Struts*, no en vano el arquitecto líder de Java Server Faces es Craig R. McClanahan, creador de la susodicha librería de Apache. No es extraño por lo tanto encontrar que Java Server Faces ofrece gran parte de la funcionalidad de *Struts*: *custom tags* para la creación de componentes de interfaz de usuario, control declarativo de errores, validación automática de los formularios, control de la internacionalización, control del flujo de páginas mediante un fichero de configuración en XML, etc.

Aún así, *Java Server Faces* añade alguna funcionalidad que resulta especialmente interesante en comparación con la ofrecida por *Struts*, como un modelo de componentes propio frente al modelo de formularios/campos de *Struts*, o una arquitectura con soporte de eventos. Además *Java Server Faces* se convertirá en el estándar de desarrollo web de SUN y es de esperar que en un futuro consiga mucho mayor apoyo de los fabricantes de entornos de desarrollo. En favor de *Struts* tenemos que tiene más “momento”, tiene mejor soporte actual en cuanto a herramientas y entornos de desarrollo, y posee conceptos de alto nivel no presentes en *Java Server Faces* como son los *Tiles* o los *Validators*.

Java Server Faces no depende para nada de JSP por lo que es factible crear un conjunto de etiquetas personalizadas orientadas a diferentes dispositivos de modo que podamos exportar nuestras interfaces web a otro tipo de clientes como pudieran ser applicaciones de escritorio creadas con *Swing*. Java Server Faces, es un *framework* que permite la reutilización de componentes de interfaz de usuario además de permitir extender cualquier componente para crear componentes nuevos.

¿ Cómo afecta Java Server Faces a los patrones de desarrollo actuales ?. Bien, por una parte podemos implantar Java Server Faces como nuestro *framework web* por lo que estaríamos ante el patrón de desarrollo que vimos en el apartado 2.1. Es importante tener en cuenta que Java Server Faces no es un *framework* tan completo como otros ya que no ofrece etiquetas de tratamiento de XML, de SQL, ni tampoco se encarga de la gestión de transacciones, seguridad, etc. Se trata de un *framework* orientado principalmente a la creación de interfaces de usuario y el control del flujo de la aplicación.

Por otra parte, al tratarse de una especificación estándar se recomienda su integración dentro del esquema que vimos en el apartado 2.2 para las aplicaciones que no utilicen un *framework web*. En este caso solo utilizaríamos Java Server Faces para renderizar componentes en el servidor sin utilizar los servicios adicionales que ofrece este *framework* ya que de otro modo nos encontraríamos de nuevo en el caso expuesto en el apartado 2.1.

3. La capa de lógica de negocio

La capa de lógica de negocio es sin duda la más importante dentro de una aplicación empresarial ya que es la que va a

contener todo el conjunto de entidades, relaciones y reglas que se encargan de la implementación de los procesos de negocio de la empresa. Los datos, por si mismos, rara vez caracen de significado y es necesario tratarlos utilizando reglas y procesos propios de cada empresa. Estas reglas y procesos se conocen como lógica de negocio.

Los *blueprints* de Java para J2EE recomiendan la implementación de los procesos de negocio como beans de sesión sin estado y los datos como beans de entidad. En este apartado sin embargo analizaremos otras alternativas para representar la lógica de negocio.

3.1 POJOs

En un principio, antes de la aparición de J2EE, cuando el paradigma de aplicación empresarial continuaba siendo la creación de aplicaciones cliente-servidor, la lógica empresarial se implementaba en lo que históricamente se ha venido a llamar como POJOs (Plain Old Java Objects o viejos objetos Java). Con el tiempo, con la aparición de las tecnologías web y los Enterprise Java Beans, esta técnica perdio adeptos, aunque pronto los recuperaría.

Un POJO es una clase simple de Java que contiene código que representa un proceso de negocio de la empresa. Normalmente los POJOs representan tanto los procesos de negocio como las entidades de nuestra lógica de negocio y por lo tanto ambos conceptos (procesos y entidades) pudiendo alojarse en la misma capa o en distintas capas, al contrario de lo que pasa con los EJBs donde los procesos (beans de sesión) se situan dentro de la lógica de negocio y las entidades (beans de entidad) se sitúan en la capa de acceso a datos.

La gran ventaja de la alternativa de POJOs es que se trata de objetos muy ligeros. Los POJOs no añaden ningún tipo de carga de proceso adicional (gestión de transacciones, gestión de seguridad, control de sesiones, etc.) y por lo tanto resultan muy fáciles de implementar, presentan un sencillo mantenimiento y tienen un gran rendimiento. Por el contrario, la gran desventaja de los POJOs es que no ofrecen la funcionalidad que nos pueden ofrecer los Enterprise Java Beans por lo que tendremos que utilizar terceras librerías o implementar manualmente la gestión de todos esos servicios de los que se encarga automáticamente el contenedor de EJBs.

Obviamente, si no necesitamos ninguno de esos servicios, utilizar POJOs se convierte en una recomendación básica para la creación de aplicaciones empresariales con un buen rendimiento.

Otro de los puntos fuertes de los POJOs que debemos recalcar es el de su sencillez. En los últimos años nos hemos encontrado con una EJBtitis que ha llevado al fracaso de muchísimos proyectos. Muchos directores de proyectos se empeñaban en utilizar EJBs de sesión y entidad para proyectos que no lo necesitaban, simplemente porque se trataba de la última tecnología de moda. La escasa madurez años atrás de estas tecnologías unido a su mal rendimiento hizo que gran parte de estos proyectos fracasasen.

El uso de POJOs además nos permite implementar rápidamente los procesos de negocio de nuestra empresa sin la necesidad de aprender una nueva tecnología. Nuestros desarrolladores pues se vuelven más productivos y nuestro código es más mantenible ya que todos los desarrolladores pueden comprenderlo sin necesidad de aprender una nueva API tan compleja como es la de los Enterprise Java Beans.

La arquitectura típica donde los POJOs contienen la lógica de negocio se encuentra en la figura 4. Obviamente se trata únicamente de una arquitectura web. Una aplicación de escritorio puede utilizar una aproximación similar pero entonces la capa de lógica de negocio se localizaría en el cliente con sus ventajas y desventajas.

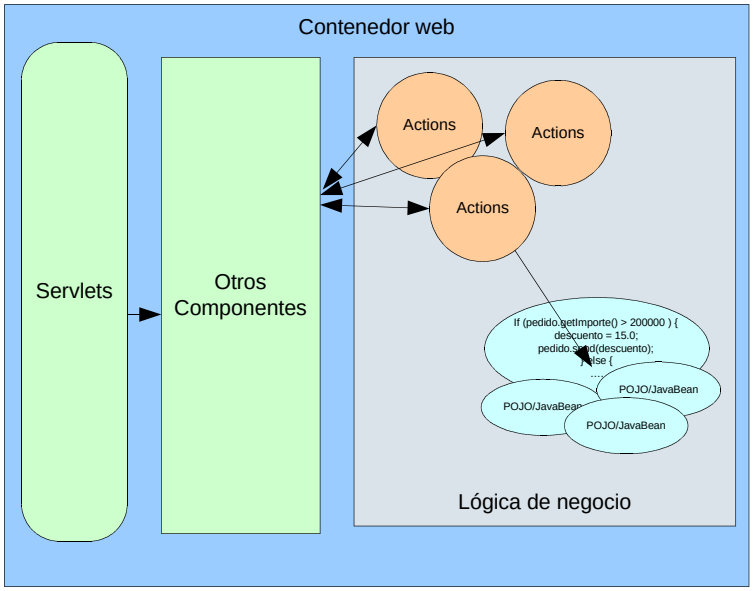

Figura 4 : Arquitectura web con lógica de negocio en POJOs/*JavaBeans*

La mayor parte de *frameworks web*, presentan una arquitectura donde la lógica de negocio se encapsula dentro de un conjunto de acciones y POJOs. Las acciones son una implementación del patrón GOF *Command* y se busca que sean procesos reutilizables. La lógica de negocio, puede estar dentro de las acciones, dentro de los POJOs mismos o en ambos. Por su parte, los POJOs en una arquitectura web suelen ser JavaBeans que además de poder contener lógica pueden contener datos y son recuperables fácilmente por los otros componentes del framework, especialmente por aquellos que se encargarán de la generación de la vista. Una vez ejecutadas las acciones, se devuelven el control al *Front Controller* que se encarga de generar la vista utilizando dichos *JavaBeans* si es necesario.

3.2 Beans de sesión

Los beans de sesión son la opción recomendada por los blueprints de *SUN Microsystems* para la implementación de la lógica de negocio de las aplicaciones empresariales basadas en J2EE 1.3. Se trata de un tipo especial de EJBs que representan procesos o conjuntos de tareas de nuestra aplicación. Como tal, se trata de una opción ideal para la representación de una arquitectura orientada al servicio, SOA o Service Oriented Architecture, donde cada bean de sesión ofrece un servicio al exterior. El extremo contrario sería una arquitectura DOA o Domain Oriented Architecture donde la lógica de negocio en lugar de estar contenida en beans de sesión, se encontrará dentro de los objetos de dominio, es decir, dentro de los beans de entidad.

Existen dos tipos de beans de sesión, los beans de sesión con estado y los beans de sesión sin estado. En los primeros, una vez que se establece una conexión entre el cliente del bean y éste, el servidor de aplicaciones se encargará de mantener el estado interno del bean entre las sucesivas peticiones que se produzcan. Esto nos plantea una duda respecto al diseño de sistemas de control de sesión en aplicaciones web, ¿ es mejor controlar la sesión a nivel de presentación (web) o a nivel de lógica de negocio (beans de sesión) ?

Como recomendación general siempre que nos encontremos ante una aplicación que utilice *Enterprise Java Beans* se debería utilizar un bean de sesión con estado para mantener el control de la sesión con el usuario. Cuando nos encontremos ante una aplicación web entonces lo recomendado es mantener la sesión dentro del objeto *HttpSession*. La siguiente tabla muestra las ventajas y desventajas de una y otra alternativa:

	HttpSession	*Beans de sesión con estado*
Ventajas	• Implementación muy sencilla • Optimización. Suele ser un aspecto muy optimizado en los contenedores web • Los servidores suelen añadir valores extra como soporte de *clustering*, tolerancia a fallos, etc. • Buena escalabilidad a través de cachés y/ o *clusters* de servidores. • Portabilidad. Suele ser un aspecto muy probado en los tests de compatibilidad	• Soporta múltiples tipos de clientes. Pueden acceder al estado de la sesión no solo clientes web sino también otros tipos de clientes como aplicaciones *stand-alone*, *EJBs*, etc. • *Thread Safety*. Los EJBs son *thread-safe* mientras que crear *Servlets* protegidos requiere un esfuerzo adicional por parte del desarollador • Gestión del ciclo de vida. El contenedor de EJBs controla automáticamente el ciclo de vida de los beans optimizando su rendimiento.
Desventajas	• Limitado a clientes web. Otro tipo de clientes no pueden acceder al estado de la sesión • Tolerancia a fallos no garantizada. La tolerancia a fallos no es algo que exija la especificación de Servlets por lo que no todos los servidores la soportan	• Se trata de una solución más compleja • El acceso a los datos de sesión es menos intuitivo • El rendimiento no es tan bueno como en el caso de *HttpSession* ya que existe una sobrecarga impuesta por la gestión del bean

Tabla 1 : Ventajas y desventajas de las alternativas de gestión de la sesión del usuario

Obviamente este tipo de beans se utilizará para recordar información sobre el usuario. Un uso típico por ejemplo sería un bean de sesión con estado que almacenase información sobre el carrito de la compra del usuario en una aplicación empresarial para un portal de venta por Internet.

El segundo tipo de beans de sesión son los beans de sesión sin estado. Este tipo de beans no mantiene ningún tipo de asociación con el cliente, de modo que cuando se termina la ejecución de un método se pierde toda la información relativa a la interacción con el usuario. Este tipo de beans es con diferencia el que menos recursos consume del servidor de aplicaciones. Por si fuera poco, se trata del tipo de bean que mejor rendimiento proporciona ya que además de requerir menos gestión por parte del servidor de aplicaciones, éste se encargará automáticamente de reutilizar las instancias creadas de un bean para servir peticiones de diferentes clientes de modo que con unos cuantos beans se pueda servir a cientos de usuarios diferentes.

3.2.1 Fachada de sesión

La fachada de sesión, o session faÇade, es uno de los patrones básicos en el diseño de aplicaciones empresariales que utilicen EJBs.

Hace años, al aparecer la tecnología de EJBs, la gente se lanzó a su uso sin tener en cuenta algunos factores que disminuían considerablemente el rendimiento. Uno de los factores más importantes era el conocido como *round trip* o tiempo de acceso al servidor de aplicaciones. Cada vez que se realiza una llamada a un EJB, es necesario transmitir y recibir información a través de la red desde el cliente al servidor de aplicaciones. Si el número de llamadas es excesivo, como comúnmente sucedía, la aplicación veía como decrecía alarmantemente su rendimiento. Los desarrolladores utilizaban los beans de sesión y de entidad como si se tratase de objetos locales y no se daban cuenta de las consecuencias que ésto tenía para el rendimiento. Obviamente, en el momento del despligue de las aplicaciones las consecuencias eran dramáticas.

La fachada de sesión es un patrón de diseño que evita este problema. Para solucionarlo, se intenta evitar a toda cosa las llamdas finas a EJBs utilizando llamadas mucho más gruesas, es decir, llamadas que realicen procesos completos. Afortunadamente, esa es justamente la mejor función para un bean de sesión, por lo que estos beans son los candidatos ideales para la implementación de este patrón. El cliente realizará una única llamda a través de la red y el bean de sesión se encargará de realizar múltiples llamadas locales a los diferentes componentes de la capa de integración hasta que complete todo el proceso de negocio que quería realizar el cliente.

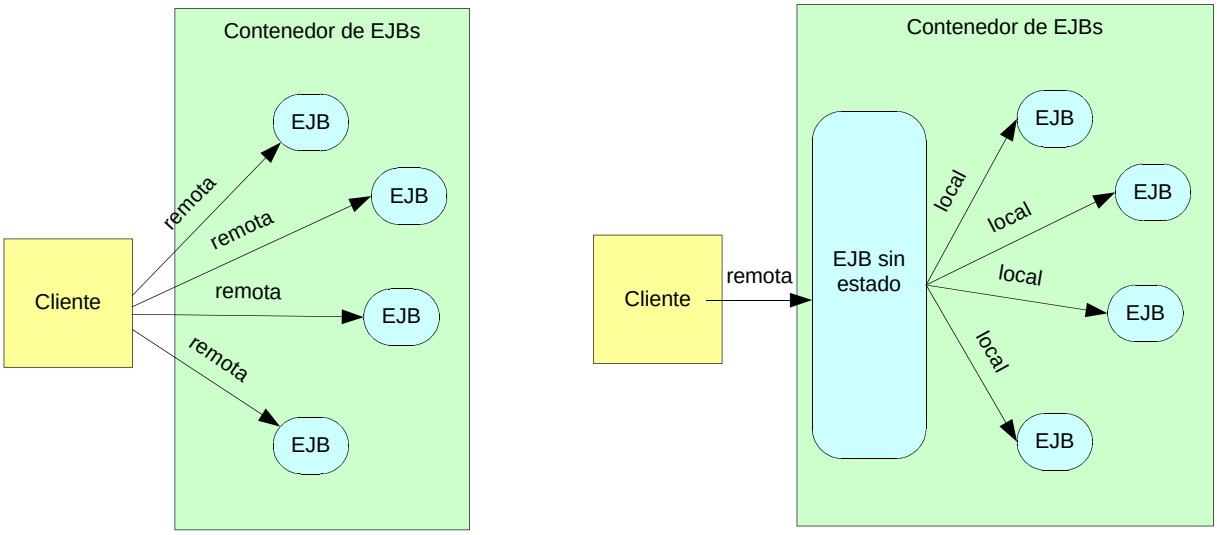

Figura 5 : Diferencia entre un acceso sin fachada de sesión y con ella

Los beans de sesión pueden ser accedidos tanto desde aplicaciones de escritorio como desde aplicaciones web. En el primer caso el acceso siempre es remoto. En el segundo caso el acceso puede ser remoto o local. Será local cuando el servidor de aplicaciones aloje tanto al contenedor web como al contenedor de EJBs o también cuando estemos trabajando en un cluster y sea un mismo nodo el que procese tanto la petición web como la petición al EJB.

3.3 Beans de mensajería

Los beans de mensajería son la alternativa más eficaz para implementar un modelo de comunicación asíncrona dentro de la plataforma J2EE y otro de los componentes donde podemos incluir lógica de negocio. Se trata de un tipo de componentes capaz de recibir y procesar mensajes enviados por un cliente mediante JMS. Como se puede ver en la figura 6, el cliente se comunicará con un destino de mensajería, ya sea una cola o un tópico según el tipo de comunicación que desee. Estas colas y tópicos están alojadas dentro de un servidor de mensajería, también llamado MOM (*Messaging Oriented Middleware*) que puede estar dentro del propio servidor de aplicaciones o tratarse de un servidor externo.

Cada bean de mensajería estará asociado con una cola o tópico determinado (ver línea roja en la figura). Cuando el servidor de mensajería recibe un mensaje lo redirecciona a la cola o tópico correspondiente y desde ahí pasa a ser procesado por el bean de mensajería asociado. Ese bean puede contener en su interior la lógica de negocio o más apropiadamente redigirá la petición a los *Enterprise Java Beans* encargados de procesar la petición.

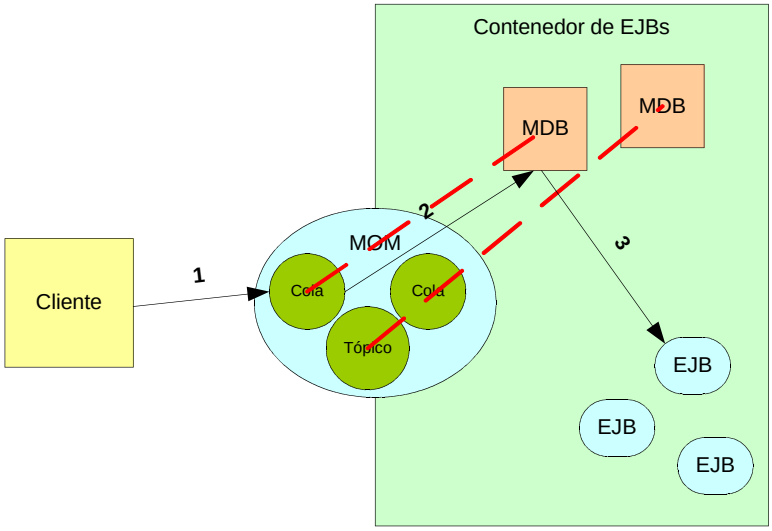

Figura 6 : Proceso de llamada a un bean de mensajería

Las ventajas que ofrece este tipo de procesado son múltiples respecto a los sistemas síncronos:

- Rendimiento. Una aplicación síncrona obligará al cliente a bloquearse hasta que el servidor sea capaz de

devolver el resultado. Si el proceso realizado en el servidor consume un tiempo excesivo el cliente se verá envuelto en molestas esperas.
- Fiabilidad. El contenedor de mensajería, MOM (Message Oriented Middleware), puede estar funcionando independientemente del servidor de aplicaciones de modo que los mensajes no se perderán aunque éste último no se encuentre disponible.
- Interoperabilidad. Los sistemas de mensajería permiten la interoperabilidad entre diferentes plataformas de modo que es factible enviar un mensaje desde una aplicación creada en una plataforma y tratar dicho mensaje desde otra aplicación creada en una plataforma completamente diferente.
- Soporte de múltiples clientes y receptores. Los beans de mensajería nos permiten de una manera sencilla implementar aplicaciones donde existan múltiples clientes que envíen mensajes a múltiples receptores. Esto es algo complejo de implementar mediante el protocolo RMI-IIOP tradicional.
- Acoplamiento débil. Esta separación explícita entre el emisor y el receptor de los mensajes redunda en sistemas con un acoplamiento débil que permiten una gran independencia entre los componentes de dichos sistemas.

3.3.1 Fachada de mensajería

La fachada de mensajería es un patrón común al utilizar beans de mensajería. Sus funcionamiento es el mismo que el de la fachada de sesión que se veía en el apartado 3.2.1, salvo que en este caso la lógica del proceso de negocio se implementará dentro de los beans de mensajería en lugar de dentro de los beans de sesión. Una alternativa cuando ya tenemos nuestra arquitectura diseñada previamente y queremos añadir algún tipo de comunicación asíncrona es que los beans de mensajería se comuniquen con la fachada de sesión que sería la que implementaría la lógica de negocio como es tradicional.

Además de evitar el problema de rendimiento que supone el realizar múltiples llamadas remotas, utilizando una fachada de mensajería evitamos la necesidad de lanzar múltiples peticiones JMS para completar un único proceso de negocio. Si utilizásemos múltiples peticiones de mensajería podríamos llegar a situaciones donde el procesado de las peticiones se realizase en diferente órden del que utilizamos para su envío, imponiendo este hecho la necesidad de tratamientos especiales e imposibilitando la implementación de un modo sencillo procesos que necesariamente tengan que seguir una secuencia temporal.

3.4 Beans de entidad

Los beans de entidad representan datos. Como tal no se encuadran demasiado bien dentro de la capa de lógica de negocio aunque en ciertos casos pueden contener ese tipo de código. Los beans de entidad se verán más en profundidad en el próximo apartado del cual se reserva el punto 4.1.2.1 para el análisis de la inclusión de lógica de negocio en dicho tipo de beans.

4. Capa de integración

La capa de integración[3] es la encargada de realizar el tratamiento y acceso a los datos que podrán provenir de sistemas y fuentes muy heterogéneas, como por ejemplo bases de datos, sistemas legacy, servidores de directorio, etc. Mientras la capa de acceso a datos obtiene los datos, pudiendo realizar mínimos cambios en los mismos (por ejemplo conversiones de formato), la capa de lógica de negocio accederá a estos datos y los integrará para la implementación de los ya famosos procesos de negocio.

Aunque en este apartado iremos recorriendo algunas de las alternativas para implementar esta capa de integración, es importante distinguir previamente entre arquitecturas orientadas al dominio y otro tipo de arquitecturas más orientadas al servicio. La base de las arquitecturas orientadas al dominio se fundamenta en que los objetos de dominio o entidades de nuestro sistema tienen absoluto conocimiento de sus responsabilidades. Una de las implicaciones de este hecho es que la lógica de acceso a datos se encuentra en el interior del objeto de dominio junto con la lógica de negocio. La alternativa contrapuesta y utilizada por otras arquitecturas es localizar la lógica de acceso a datos en el exterior del objeto. En este apartado analizaremos de forma separada las dos partes más importantes de la capa de integración, es decir, los datos y el proceso de acceso a los mismos.

4.1 Los datos

Los datos conforman el corazón de una buena arquitectura empresarial. Las entidades han de representar abstracciones del mundo real, abstracciones tales como un pedido, una línea de factura, una partida presupuestaria o un artículo de álmacen. Además de representar entidades del mundo real es conveniente que nuestros datos sean reutilizables. Por

3 Este es el nombre que recibe esta capa dentro de los blueprints de *SUN Microsystems*. Otros nombres que recibe comúnmente esta capa son los de capa de datos o capa de acceso a datos.

poner un ejemplo, serviría de poco el crear una entidad “linea de pedido” para nuestra aplicación de control de proveedores si después no podemos reutilizar dicha entidad para nuestra aplicación de gestión de costes.

En J2EE son dos las vertientes más habituales para la representación de entidades. Una es el utilizar clases Java normales y la otra es utilizar Enterprise Java Beans.

4.1.1 POJOs

Los POJOs, o clases Java normales descritas en el apartado 3.1, son una buena alternativa para la creación de entidades para nuestras aplicaciones empresariales. Normalmente los POJOs se implementan como JavaBeans, es decir, clases Java normales que siguen un convenio especial de nombrado. Esta implementación tan simple permite su reutilización en diferentes tipos de aplicaciones, incluso en aplicaciones cliente-servidor que no precisen de un servidor de aplicaciones. Esta independencia de un contenedor de EJBs hace que sea ésta una alternativa mucho más flexible que la que veremos en el apartado 4.1.2.

Al tratarse de clases Java normales, los POJO como entidades presentan un rendimiento bastante alto ya que su acceso no está sobrecargado con controles adicionales de seguridad, transacciones, etc. Obviamente, cuando este tipo de control es algo importante para una aplicación, el uso de POJOs puede suponer una carga de trabajo adicional importante ya que el desarrollador tendrá que implementar todos esos servicios manualmente.

El servicio implementado de manera automática en otros sistemas que puede echar más de menos una aplicación que use POJOs es sin duda el de persistencia. Utilizando POJOs, el desarrollador se verá obligado a implementar toda la lógica de acceso a datos, típicamente operaciones CRUD (Create, Read, Update, Delete), que en otros sistemas como en los beans de entidad ya viene implementado por defecto al utilizar persistencia con CMP. Por suerte, existen varias alternativas que permiten la creación de entidades con POJOs y la generación automática del código de persistencia. Estas alternativas son principalmente el uso de motores de persistencia que veremos en el apartado 4.2.2 y también el utilizar herramientas de generación de código de las que se hablará en el apartado 7.

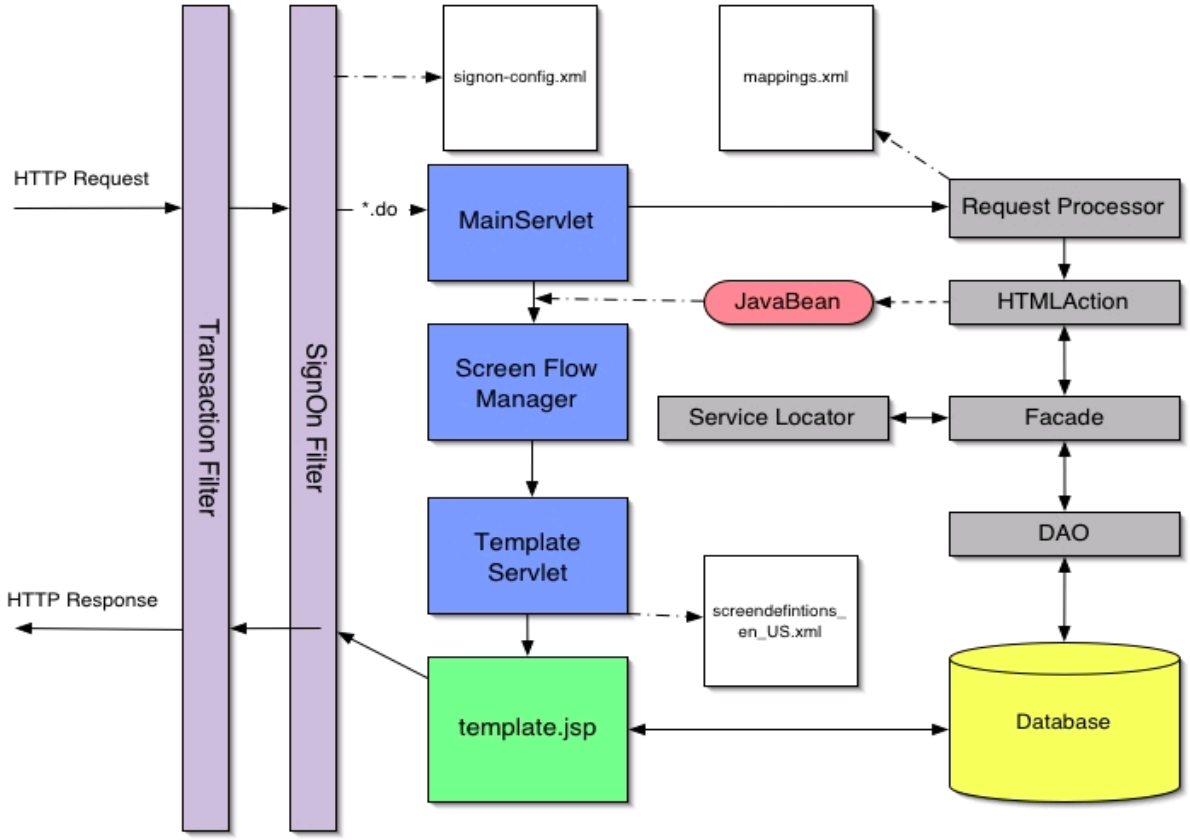

Figura 7 : Imagen de parte de la arquitectura de *Adventure Builder*

La figura 7 muestra un diagrama de la aplicación principal de los nuevos *blueprint* de *SUN Microsystems* que servirán como modelo de desarrollo de aplicaciones para la plataforma J2EE 1.4. Como se puede ver, dicha aplicación no utiliza para nada EJBs sino que se basa en clases normales Java. Aunque desde *SUN Microsystems* se afirma que se ha desarrollado este diseño para mostrar que existen otra forma de desarrollar aplicaciones web sin utilizar EJBs, lo cierto

es que no deja de ser un reconocimiento de que la alternativa de no utilizar EJBs está ahí y está cogiendo cada vez más adeptos.

Otra demostración de que las arquitecturas basadas en POJOs están teniendo cada vez una repercusión más importante en el mercado es su inclusión dentro de los escenarios de desarrollo incluidos en el libro de patrones referencia para la creación de aplicaciones J2EE, *Core J2EE Patterns 2nd Edition*. En este libro, una arquitectura basada en POJOs pasa a tomar un valor considerable y aparece en multitud de escenarios.

4.1.2 Beans de entidad

Los beans de entidad son el corazón de la especificación de Enterprise Java Beans. Este tipo de beans representa entidades de nuestro dominio. Los beans de entidad están asociados con datos que se encuentran en bases de datos relacionales y saben como realizar las operaciones de persistencia con la tabla o tablas con las que están asociados. Idealmente, es el propio bean de entidad el que se encarga de cargar, guardar, actualizar y eliminar los datos de la base de datos dejando en mano del usuario tan sólo el trabajo de decidir que objetos son los que se guardan, actualizan, eliminan o cargan.

Además de esa persistencia automática, los beans de entidad ofrecen otra serie de beneficios que hacen que esta tecnología se convierta en una opción tentadora para el desarrollador. A parte de las ventajas de rendimiento que podemos obtener con técnicas ofrecidas por la persistencia manejada por el contenedor, CMP, y que veremos en el apartado 4.2.4, los beans de entidad pueden ser fácilmente reutilizados por diferentes clientes por lo que suponen un importante ahorro de memoria para el servidor de aplicaciones. Por otra parte, cada servidor de aplicaciones se encarga de mantener una serie de pools para estos beans lo que permite que unas cuantas instancias sirvan las peticiones de decenas, cientos o miles de clientes, lo que nuevamente supone un ahorro considerable de memoria y un aprovechamiento mayor de los recursos.

Otros servicios ofrecidos automáticamente por este tipo de beans son las transacciones, la seguridad, o la gestión del ciclo de vida de los componentes. Por una parte, utilizando beans de entidad, el desarrollador tiene un control transaccional sobre el acceso al bean y que puede modificar de una manera declarativa y/o programática. El control de seguridad puede realizarse también de manera declarativa y/o programática. Cuando se aprovecha la posibilidad de gestionar estos servicios de manera declarativa el desarrollador puede controlar las transacciones y la seguridad a nivel de método mientras que de manera programática la granularidad llega incluso a nivel de bloques de código.

4.1.2.1 Problemas de rendimiento

Una de las críticas más habituales a los sistemas desarrollados con beans de entidad ha sido desde siempre la falta de rendimiento. Los beans de entidad son objetos considerablemente más pesados que las clases normales de Java y por lo tanto tienen una carga adicional de trabajo que hace que un mal uso de los mismos pueda influir negativamente en el rendimiento de una aplicación. Aún así, existen una serie de causas históricas que han influido en esta “mala fama” y que se ha demostrado que han sido superadas.

- **Uso de interfaces remotas**. Uno de los problemas más conocidos de los beans de entidad se debía a que hasta la especificación EJB 2.0 la única forma de acceder a estos beans era utilizando RMI-IIOP a través de su interfaz remota. Esto hacía que si se ejecutaban múltiples llamadas a beans de entidad el rendimiento de la aplicación se resintiese considerablemente. Esta fue la principal causa de la aparición del patrón de diseño fachada de sesión que vimos anteriormente.

  Afortunadamente, con la aparición de EJB 2.0 ya es posible acceder a los beans de entidad de manera local sin que sea necesario utilizar ningún protocolo de red de modo que se consigue un aumento de rendimiento importante.

- **EJBtitis**. Se trata de una enfermedad que hace que los desarrolladores se empeñen en utilizar constantemente EJBs para cada una de las entidades que aparezcan dentro del dominio de la aplicación empresarial. Habitualmente la causa de esta enfermedad es una ansia de utilizar siempre las últimas tecnologías aunque no sea necesario. La EJBtitis tiene como consecuencias una disminución alarmante del rendimiento de las aplicaciones y puede llegar a provocar la saturación de los servidores de aplicaciones.

  Sarcasmo aparte, lo cierto es que al aparecer la especificación de Enterprise Java Beans, los desarrolladores se lanzaron como locos a implementar sus EJBs sin seguir ningún criterio especial. Como ya se ha comentado, los EJB son objetos bastante pesados y hay que ser muy cuidadoso de no sobreutilizarlos, especialmente los beans de entidad.

  Los beans de entidad deben utilizarse para representar datos sobre los que vayamos a realizar accesos de

escritura y se recomienda utilizar otra técnica, por ejemplo accesos directos con JDBC implementando el patrón *JDBC for reading* para los accesos de solo lectura. También debemos tener en cuenta que deberíamos utilizar sólo beans de entidad cuando tengamos verdaderas necesidades en cuanto a transacciones, seguridad, o concurrencia, ya que cualquiera de esos servicios añaden cierta sobrecarga al procesado de los beans.

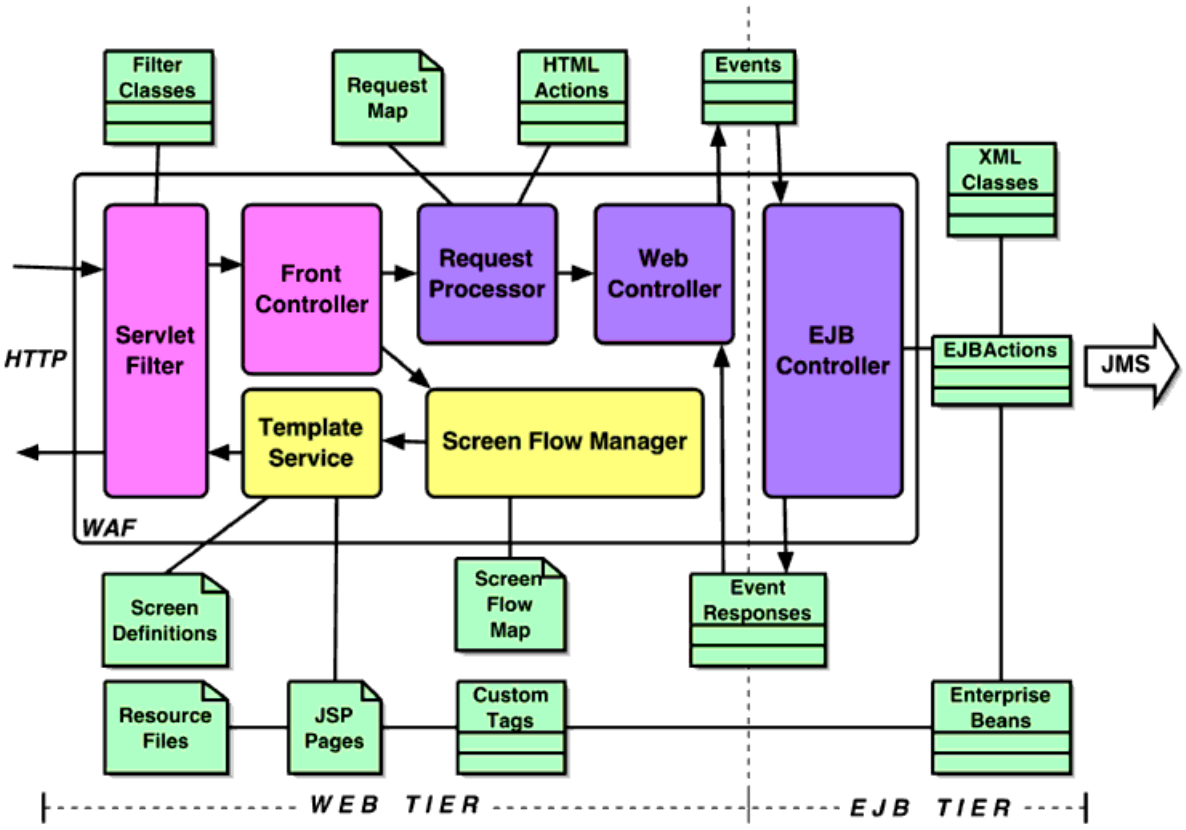

Figura 8 : Parte de la arquitectura de *Java Pet Store*

La figura 8 representa la arquitectura de la Java Pet Store, la aplicación que ha servido como modelo de arquitectura hasta la aparición de J2EE 1.4. Como se puede ver, en una aplicación web los Servlet, a menudo a través de acciones que implementan el patrón *Command*, acceden a la fachada de sesión para ejecutar procesos de negocio. Los beans de la fachada de sesión se encargan de acceder a los beans de entidad para obtener los datos necesarios para ejecutar el proceso.

4.1.2.2 Lógica de negocio en los beans de entidad

Como alternativa a las vistas en el apartado 3, los beans de entidad pueden contener en su interior lógica de negocio, siendo esta aproximación mucho más cercada a un diseño orientado al dominio que a un diseño orientado al servicio. Si los beans de entidad únicamente van a ser accedidos de modo local, entonces la decisión de colocar lógica de negocio dentro de los beans de entidad puede ser apropiada si el diseño concuerda con esta decisión.

Aún así, existen una serie de factores que pueden influir negativamente en el rendimiento de la aplicación cuando los beans de entidad contienen en su interior lógica de negocio. Entre algunos de los factores que podemos encontrar entre la bibliografía, destacan:

- Introducción de relaciones entre entidades
- Introducción de gestión del *workflow* dentro del bean de entidad
- Adquisición por parte del bean de entidad de responsabilidades que deberían ser reclamadas por otro componente de negocio

Cualquiera de estos síntomas puede provocar una caída de rendimiento en nuestra aplicación al introducir lógica de negocio dentro de los beans de entidad. En general, se recomienda que un bean de entidad contenga lógica de negocio propia, es decir, que afecte sólo a sus datos y a sus objetos dependientes sin influir en ningún modo en terceras entidades.

4.2 Acceso a datos

El acceso a datos es quizás donde más variantes podemos encontrar para implementar nuestra arquitectura. El fracaso que ya comentamos de los beans de entidad en los años anteriores han hecho que muchos desarrolladores consideren otras opciones antes de utilizar BMP o CMP como servicios de persistencia para sus entidades, buscando otras alternativas mucho más ligeras. En los siguientes puntos iremos viendo las alternativas más comunes a la hora de implementar sistemas de acceso a datos en J2EE.

4.2.1 JDBC

JDBC es la opción más sencilla para la implementación de la persistencia en aplicaciones J2EE. Se utiliza JDBC cuando se tiene una estructura de POJOs o beans de entidad con persistencia manejada por el propio bean, es decir, BMP.

La principal ventaja de utilizar JDBC es que es con creces la tecnología más ligera ya que no añade ningún tipo de funcionalidad adicional. Otra de las ventajas de utilizar JDBC es que es muy sencillo encontrar desarrolladores familiarizados con este API mientras que encontrar desarrolladores familiarizados con APIs como JDO o con tecnologías como CMP, por poner algunos ejemplos, puede ser más complejo.

La mayor desventaja del uso de JDBC es que la creación manual de todas las sentencias de acceso a datos requiere un gran esfuerzo por parte de los desarrolladores. Por si fuera poco, el acceso a jerarquías complejas de objetos puede resultar en sentencias demasiado complicadas y con seguridad dependientes del motor de base de datos. El problema de la variación de base de datos no sólo afecta cuando se están creando sentencias complejas de SQL con múltiples JOINs sino que puede ser muy habitual. Si se piensa que una aplicación puede cambiar frecuentemente de base de datos o puede ser utilizada para acceder a bases de datos heterogéneas entonces el patrón DAO se vuelve indispensable.

4.2.1.1 Data Access Objects (DAO)

Este patrón es uno de los más implementados en las diferentes tecnologías para el desarrollo de aplicaciones empresariales ya que la problemática del cambio frecuente de fuente de datos es independiente totalmente de la plataforma de desarrollo. Aunque JDBC permite un acceso homogéneo a diferentes bases de datos, lo cierto es que a la hora de la verdad la sintaxis SQL suele variar entre los diferentes gestores, lo que nos obliga a implementar algún tipo de estrategia que nos permita una rápida transición entre diferentes sistemas. Ahí es donde entra en juego el patrón DAO.

El patrón DAO propone la creación de una serie de interfaces para abstraer el acceso a datos. Una vez creadas esas interfaces que permitirán la creación, lectura, actualización y eliminación de entidades, se crean las diferentes implementaciones, una por cada base de datos a la que se vaya acceder. Si en un momento dado la base de datos cambia, entonces se utilizará la implementación más adecuada del interfaz DAO.

Si se utiliza una aproximación únicamente con JDBC para el control de la persistencia de las aplicaciones será necesario implementar manualmente el patrón DAO. Lo mismo ocurre cuando se utiliza persistencia manejada por el bean, BMP. Por el contrario, tanto los motores de mapeo objetos/relacional como la persistencia manejada por el contenedor (CMP) como JDO implementan automáticamente el patrón DAO, con las ventajas que esto supone.

4.2.2 Motores de mapeo objetos/relacional

Los motores de mapeo objetos/relacional, también llamados comúnmente *frameworks* de persistencia, son una de las herramientas más útiles y más utilizadas dentro de las aplicaciones empresariales. Se tratan de sistemas que automatizan todo el proceso de creación, lectura, actualización y eliminación de entidades de bases de datos relacionales. Por si fuera poco, no se limitan a la implementación de estas funciones sino que la mayor parte de motores son auténticas herramientas de mapeo relacional-objetos.

La mayoría de motores de persistencia permiten la definición de nuestras entidades a través de ficheros XML o cualquier otra forma. A partir de esa definición estos motores son capaces de extraer la definición de esquema de la base de datos necesaria para representar ese modelo de objetos. Estos motores, permiten el mapeo de estos esquemas de base de datos a objetos de modo que a partir de las tablas de base de datos se pueden generar las clases Java necesarias para modelar dicho esquema con todas las relaciones de jerarquía que esten presentes en el mismo.

Obviamente esto supone una ventaja grandísima ya que la cantidad de código necesaria para realizar todas esas funciones es realmente considerable. Además, de este modo, se puede cambiar de manera sencilla el esquema de base de datos modificando el fichero XML y en un momento tener toda la estructura de clases Java y la base de datos

actualizada con el mínimo coste de desarrollo para el programador que tan sólo se tendrá que preocupar por añadir lógica de negocio para soportar los cambios.

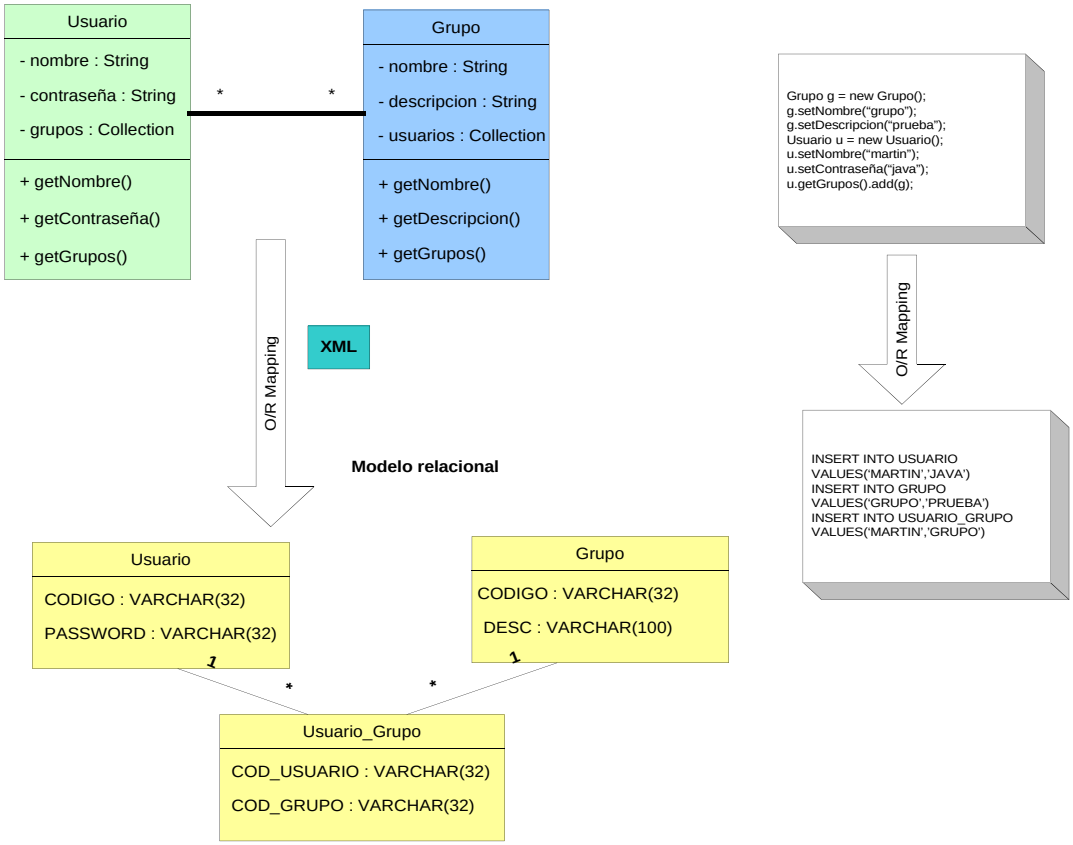

Figura 9 : Ejemplo de mapeo Objetos - Relacional

Otra de las ventajas de estos *frameworks* es que permiten persistir jerarquías completas de objetos con tan sólo ejecutar una única orden. En la figura 9 se puede apreciar como se mapearía una estructura simple de usuarios-grupos a un modelo relacional. Ese mapeo se realiza de manera automática y las tablas y clases se generan a partir de un fichero XML de configuración. Posteriormente, cambios en el modelo de clases como inserciones, eliminaciones o actualizaciones provocan que se ejecuten una serie de sentencias de manera transparente en la base de datos que se esté utilizando.

En el ejemplo anterior, sería necesario escribir tres sentencias de insercción en *JDBC* para crear un usuario y un grupo. Utilizando un motor de persistencia no es necesario realizar ninguna llamada; se maneja el modelo de objetos y posteriormente se llama a alguna sentencia del motor de persistencia que vuelca el estado de la jerarquía de objetos en la base de datos.

Por si fueran pocas las ventajas, la mayoría de motores de persistencia ofrecen al desarrollador decenas de optimizaciones y funcionalidad añadida que pueden mejorar sus aplicaciones. Carga perezosa o carga agresiva de jerarquías, cachés, objetos dinámicos, funciones agregadas, claves primarias compuestas, versionado, transacciones, etc., son solo algunos ejemplos de funcionalidad implementada por los motores de persistencia.

La principal desventaja de estos sistemas es que no son estándares. Al utilizar un motor de persistencia se corre el importante riesgo de quedarse ligado a él y habitualmente la migración entre este tipo de sistemas suele ser compleja ya que no todos ofrecen las mismas funcionalidades. Al no tratarse de herramientas estándar ofrecen mayor funcionalidad pero también se corre el riesgo de que en cualquier momento cese el desarrollo y mantenimiento de la librería lo que desembocaría en herramientas desfasadas y sin soporte. Tecnologías como BMP o CMP y JDO no presentan este problema ya que son especificaciones estándar.

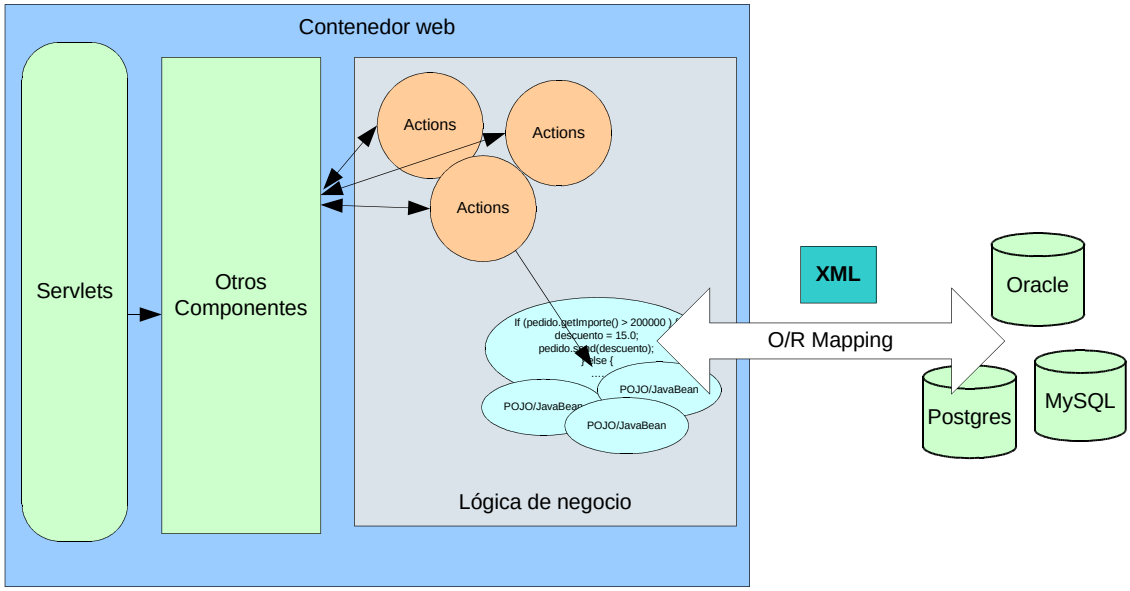

Figura 10 : Arquitectura web con un sistema de mapeo O/R

Se puede utilizar un motor de persistencia en cualquiera de los modelos de desarrollo vistos hasta ahora. Normalmente se suele utilizar este modelo con sistemas que no utilizan *Enterprise Java Beans*, siendo ideal para trabajar con arquitecturas de POJOs ya que estos dos sistemas unidos nos proporcionan un gran rendimiento y una arquitectura simple y que resulta rápida de implementar. Aún así, también se puede usar un motor de persistencia al utilizar beans de entidad con persistencia manejada por el propio bean, BMP. Algunos de los motores de persistencia más conocidos son Hibernate, Castor, Cayenne, Jakarta OJB, TopLink o FrontierSuite, estos dos últimos con licencia propietaria.

4.2.3 Persistencia manejada por el bean, BMP

Volviendo a los beans de entidad, existen dos sistemas para controlar la persistencia de este tipo de entidades: BMP y CMP. En BMP, todo el código de persistencia es implementado por parte del desarrollador. El desarrollador tiene que implementar los métodos de carga, actualización, creación y borrado que el contenedor se encargará de llamar automáticamente para persistir los datos de las entidades.

El desarrollador puede elegir la tecnología que desee para implementar la persistencia aunque habitualmente elegirá JDBC básico ya que el resto de alternativas, por ejemplo los motores de persistencia, no tienen demasiado sentido dentro BMP al ofrecer un modelo de objetos propio.

4.2.4 Persistencia manejada por el contenedor, CMP

La persistencia manejada por el contenedor, o CMP, ha sido uno de los aspectos más mejorados con la introducción de la especificación EJB 2.0. En este tipo de persistencia el contenedor es el que implementa y ejecuta el código necesario para las operaciones CRUD de acceso a base de datos. Este código además es compatible para todas las bases de datos soportadas por el servidor de aplicaciones.

Con EJB 2.0, el rendimiento de CMP se ha visto incrementado considerablemente. CMP ofrece automáticamente muchas de las funcionalidades presentes dentro de los motores de persistencia, como por ejemplo la carga perezosa, la carga agresiva, cachés, etc. La desventaja es que la mayoría de estas optimizaciones no son estándar y son añadidos que ofrecen los servidores de aplicaciones por lo que si se depende demasiado de los mismos se corre el riesgo de quedarse ligado a un fabricante en concreto.

Para la ejecución de sentencias SQL, CMP utiliza un lenguaje especial denominado EJB-QL. Este lenguaje declarativo permite utilizar un subconjunto de las instrucciones definidas por el estándar SQL-92, aunque tiene muchas limitaciones. EJB-QL permite manejar el esquema abstracto de datos definido al utilizar CMP de un modo muy sencillo y facilitando especialmente la navegación a través de las relaciones entre entidades. Entre las carencias de EJB-QL 1.0, que es la versión disponible en los servidores compatibles con J2EE 1.3, destacan la imposibilidad de ejecutar sentencias *ORDER BY*, el carecer de soporte para tipos *DATE*, y la imposibilidad de ejecutar funciones agregadas como *COUNT*, *MAX* o *SUM* entre otras.

Hasta ahora han sido las diferentes implementaciones concretas de los diferentes servidores de aplicaciones las que han ido solucionando este problema a través de extensiones propietarias.Por suerte, con EJB-QL 1.1, que estará presente en los servidores compatibles con J2EE 1.4, estas carencias se han solucionado.

4.2.4.1 BMP frente a CMP

Una vez que se ha decidido que en un proyecto se van a utilizar beans de entidad, normalmente la siguiente pregunta a formularse es ¿ cómo se va a gestionar la persistencia ?. ¿La va a gestionar el propio bean o la va a gestionar el contenedor ?. Como en casi todas las disyuntivas, cada alternativa tiene sus ventajas y sus desventajas.

BMP es la opción que más flexibilidad ofrece ya que es el propio desarrollador el que se encarga de crear a mano todas las sentencias de acceso a base de datos. Por contra, en CMP el desarrollador no tiene control sobre gran parte de las operaciones que se realizan; el desarrollador puede controlar las sentencias *SELECT* utilizando el lenguaje *EJB-QL* pero aún así las posibilidades son limitadas. Si las consultas que queremos realizar son muy complejas seguramente nos veremos obligados a utilizar BMP. Por último comentar que en BMP, además de tener la opción de escribir a mano todo el código JDBC podemos utilizar un motor de persistencia con el que obtendremos todas sus ventajas.

La gran flexibilidad de los beans con BMP hace sean más fáciles de depurar ya que si utilizamos beans con CMP habrá gran cantidad de código al que no podremos acceder con el depurador.Esa flexibilidad nos permite acceder a múltiples bases de datos dese el mismo bean, algo que con CMP es absolutamente imposible. También si desde un mismo bean tenemos que acceder a diferentes bases de datos, o a otro tipo de fuentes de datos como sistemas legacy, entonces la mejor opción es utilizar BMP.

La flexibilidad que hemos mencionado acarrea sin embargo un importante coste de desarrollo. En CMP todas las operaciones de acceso a base de datos se realizan de manera automática por lo que el desarrollador se ve liberado de una importante carga de trabajo y al mismo tiempo se minimiza la posibilidad de error. En CMP el código de persistencia se genera automáticamente, esto hace que sea una tecnología que se integra mucho mejor con herramientas RAD (*Rapid Application Development*) de modo que cualquier IDE moderno será capaz de generar nuestros beans de entidad directamente a partir de un esquema de base de datos y viceversa. Esto redunda nuevamente en un importante ahorro de trabajo para el desarrollador.

Una ventaja importante de CMP es que es una tecnología independiente de la base de datos. Al utilizar BMP, a no ser que utilicemos un motor de persistencia o realicemos una implementación del patrón DAO, se adquiere una importante ligazón con una base de datos en concreto. Cualquier cambio en la estructura de la base de datos o cualquier cambio en la base de datos puede provocar importantes cambios en el código de persistencia que se encuentre en el interior de los beans BMP. En este aspecto, el punto débil de CMP es que normalmente los servidores no soportan todas las bases de datos que serían deseables.

La comparativa que más polémica causa siempre al confrontar BMP frente a CMP es la que analiza el rendimiento de ambas alternativas. Nuevamente esta polémica suele ser a causa del desconocimiento de las novedades de CMP 2.0. Con CMP 2.0 se ha cambiado el modelo abstracto de persistencia de modo que el servidor de aplicaciones puede ejercer mucho más control sobre la implementación concreta de los beans de entidad. Esto hace que normalmente estos beans tengan un rendimiento mucho mayor que si se utiliza BMP.

El contenedor, con CMP, puede interponer entre la definición abstracta del bean y su implementación gran cantidad de optimizaciones como carga perezosa, carga agresiva, cachés, etc. Un ejemplo de optimización típica es la que soluciona el problema de los *n+1* accesos. Este problema se produce al intentar acceder a un conjunto de beans, ya que con BMP se realiza un acceso para ejecutar el método *find* y *n* accesos para ejecutar el método *ejbLoad()* de cada uno de los beans. Con CMP sin embargo es necesario un único acceso ya que el servidor de aplicaciones se encarga automáticamente de cargar todos los beans con una única sentencia *SELECT* enorme.Esta y otras optimizaciones no existían en CMP 1.0 y por lo tanto durante mucho tiempo hubo gran cantidad de desarrolladores que continuaban afirmando que BMP era más rápido que CMP.

Una novedad de CMP 2.0 y que también hace que su rendimiento sea mayor que el de utilizar BMP son las relaciones. En CMP se pueden definir diferentes tipos de relaciones (1..n, m..n, 1..1) entre los beans de entidad. El servidor de aplicaciones puede optimizar las consultas para cargar estas jerarquías completas de golpe de modo que se ahorran gran cantidad de accesos a base de datos. Ni que decir tiene que este control automático de las relaciones entre beans ahorra gran cantidad de código al desarrollador, del mismo modo que lo hacen los motores de persistencia.

La tabla 2 sintetiza todo lo que hemos visto en este apartado:

	BMP	*CMP*
Ventajas	• Mayor control sobre las sentencias SQL • Permite el acceso a múltiples fuentes de datos • Permite el acceso a otros sistemas que no sean bases de datos, por ejemplo sistemas legacy, repositorios, etc. • Más sencillo de depurar	• Mejor rendimiento, el servidor tiene la posibilidad de aplicar grandes optimizaciones • Se necesita menos esfuerzo por parte del desarrollador • Es una tecnología más mantenible. Soporta mejor los cambios de estructura y base de datos • Persiste jerarquías de objetos fácilmente • Mejor integración con herramientas RAD • Portabilidad. Suele ser un aspecto muy probado en los tests de compatibilidad
Desventajas	• Farragoso para el desarollador • Sin DAO o un motor de persistencia es una tecnología poco mantenible • Peor rendimiento que CMP (a no ser que se use un motor de persistencia)	• Poco control de las sentencias SQL • Un bean sólo puede estar asociado con una base de datos • Sólo soporta bases de datos relacionales • Difícil de depurar

Tabla 2 : Comparativa entre BMP y CMP

Si se analiza fríamente la tabla anterior, es fácil llegar a la conclusión de que una arquitectura con BMP y una buena implementación DAO, o preferiblemente un buen motor de persistencia no tendrá nada que envidiar a CMP y en muchos casos presentará más ventajas que desventajas.

4.2.5 JDO

JDO, o Java Data Objects, es una especificación estándar definida por el JSR-12. Se trata de una especificación que ha creado un sistema de persistencia estándar que permite al desarrollador olvidarse por completo de las sentencias SQL y de la tediosa tarea del mapeo entre un modelo de objetos y un esquema relacional. JDO podría definirse como un motor de persistencia estándar y por lo tanto todo lo que hemos comentado en el apartado 4.2.2 se aplica a *Java Data Objects*.

Pero JDO va mucho más allá de ser una herramienta de mapeo objetos/relacional. Sería mucho más correcto definir a JDO como una especificaicón que define un modelo estándar de persistencia de objetos. La especificación JDO permite persistir estos modelos de objetos tanto en bases de datos relacionales como en bases de datos orientadas a objetos, sistemas de ficheros o cualquier otro mecanismo de persistencia que se nos pueda ocurrir y que soporten las diferentes implementaciones.

La mayor parte de la literatura que se puede encontrar sobre JDO se empeña en prsentar a esta tecnología como principal rival de CMP. La realidad es que no es así. JDO es una alternativa a CMP, y cada tecnología presenta sus ventajas y desventajas que espero que el lector pueda haber comprendido mejor después de la lectura de este artículo. El verdadero rival de JDO son los motores de mapeos objetos/relacional.

Esta rivalidad era de esperar cuando el objetivo que persigue JDO es exactamente el mismo que el que persiguen dichos motores de persistencia aunque JDO parte con la ventaja de ser mucho más flexible. El caso es que la rivalidad existe y es clara. Por una parte, los motores orientados a bases de datos relacionales llevan mucho más tiempo en el mercado y por tanto tienen ya muchos fieles seguidores. Por el otro lado, JDO es una especificación estándar y por lo tanto se supone que tendrá un mayor soporte y que presenta muchos menos riesgos de caer en un *vendor lock-in*.

Los detractores de JDO alegan que se trata de una especificación orientada específicamente a bases de datos orientadas a objetos que todavía no están demasiado en uso. Esta afirmación se basa en el hecho de que la mayoría de participantes en el desarrollo de la especificación de JDO son fabricantes de sistemas de acceso a este tipo de base de datos. Lo cierto es que la flexibilidad de JDO redunda en que se trata de una especificación no orientada en concreto a bases de datos relacionales lo que supone un menor rendimiento. Sea como sea, los diferentes fabricantes van sacando cada vez más implementaciones mucho más orientadas a este tipo de bases de datos y es cuestión de tiempo que el rendimiento se iguale.

JDO ofrece más o menos las mismas funcionalidades que un motor de persistencia, incluida la generación del código de los modelos de objetos que queramos persistir. JDO ofrece también las optimizaciones que ofrecen los motores de

persistencia, en algunos casos con un mejor rendimiento siempre que utilicemos una base de datos orientada a objetos ya que suelen estar mejor diseñadas para persistir modelos de clases. Por último, la principal baza con la que cuenta JDO es que se trata de una especificación estándar y por lo tanto el usuario no corre el riesgo de quedar atado a una de las implementaciones concretas siempre y cuando no utilice características especiales de las mismas.

Motores de persistencia como Hibernate y compañía juegan con la baza de ser soluciones experimentadas y ya probadas y de estar específicamente diseñadas para ser utilizadas con bases de datos relacionales, muy diferentes de las bases de datos orientadas a objetos.

Si no se han explicado las características concretas de JDO por su similitud con los motores de persistencia, lo mismo sucede con su arquitectura. Las figuras 9 y 10 sirven perfectamente como ejemplo de lo que podría ser el modelo de funcionamiento de JDO y de una arquitectura para una aplicación web que utilice alguna implementación de JDO.

Entre algunas de las implementaciones más conocidas de JDO están Kodo JDO, LiDO de LIBeLIS o ObjectFrontier FrontierSuite Existen también implementaciones libres bastante utilizadas como OJB o XORM.

4.4 JCA

JCA son las siglas de *J2EE Connector Arquitecture*. Se trata de una especificación (JSR-115) que define una arquitectura estándar para conectar la plataforma J2EE a diferentes sistemas de información heterogéneos. Estos sistemas pueden ser de muy diversa índole: ERPs, bases de datos o aplicaciones de terceros. JCA por una parte define una arquitectura escalable, segura y transaccional que permite la comunicación entre EIS y servidores de aplicaciones, mientras que por otra parte define una interfaz de acceso común que permite a los clientes acceder a sistemas de información muy diferentes entre si, con el mismo API estándar.

JCA se basa en el concepto de adaptador de recursos. Un adaptador de recursos es un driver que el cliente utiliza para acceder al sistema empresarial para el que ha sido desarrollado. El adaptador puede ser desplegado (aunque no es obligatorio) dentro de un servidor de aplicaciones para que de este modo podamos utilizar todos los servicios que éste último nos ofrece. Normalmente, un adaptador será específico para un EIS determinado y permitirá únicamente el acceso a ese tipo de recursos. No existe un límite en el número de adaptadores instalables dentro de un servidor de aplicaciones. Ejemplos de adaptadores de recursos pueden ser un driver JDBC que tenga una interfaz compatible con JCA, un adaptador para conectarse a SAP, un adaptador de Siebel, un adaptador de conexión a algún sistema de mensajería, etc.

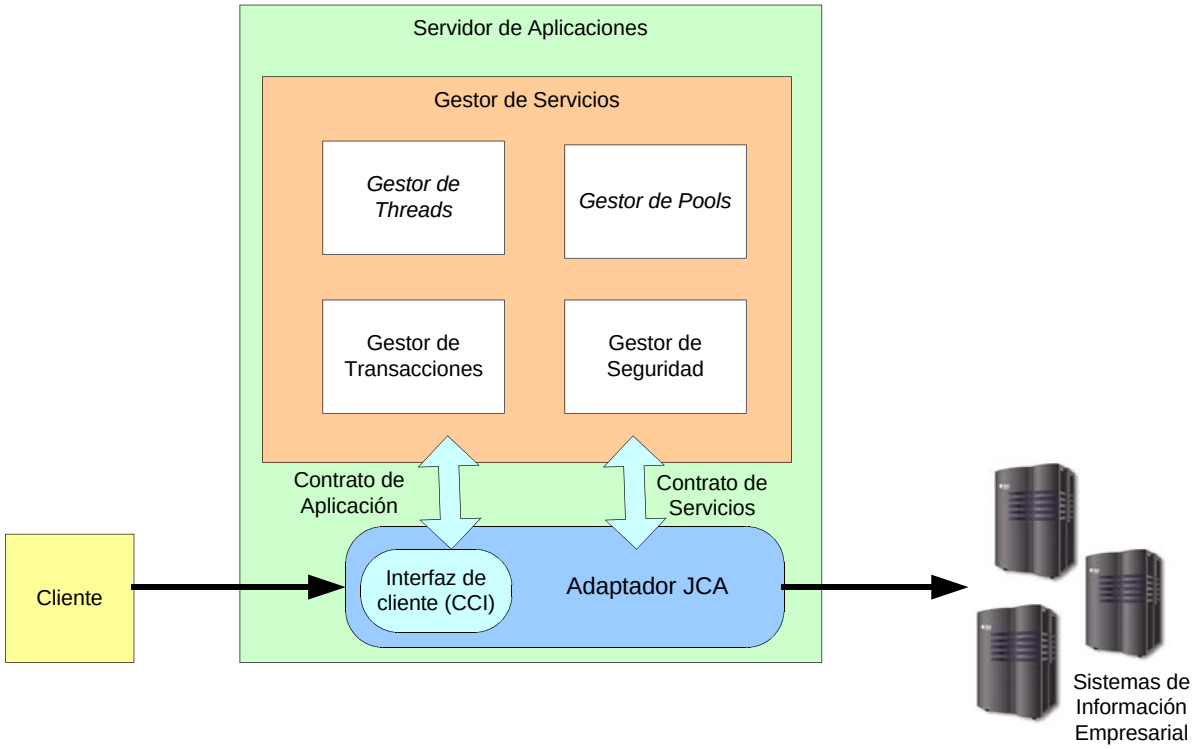

Figura 11 : Arquitectura de JCA

La figura anterior muestra la arquitectura típica de una aplicación que utilice JCA. JCA es una tecnología que no interfiere para nada en el modelo de aplicación que estemos utilizando por lo que es fácilmente usable tanto en

aplicaciones web, como en aplicaciones de escritorios, tanto con *Enterprise Java Beans*, como con *POJO*s, etc.

JCA ha sido una de las especificaciones que mejor acogida ha tenido dentro del mundo empresarial, sobre todo para la integración de sistemas que tradicionalmente han sido bastante cerrados, como por ejemplo SAP, Tuxedo, Navision, etc. La última versión de la especificación JCA es la 1.5. Esta especificación nos permite, entre otras cosas:

- Gestión de las conexiones de acceso a JCA
- Gestión de la seguridad y transacciones
- Gestión de los *threads* de acceso al JCA (nuevo JCA 1.5)
- Enviar mensajes asíncronamente al sistema legacy
- Recibir mensajes asíncronamente del sistema legacy (*inbound communication*, nuevo JCA 1.5)
- Portabilidad a través de los diferentes servidores de aplicaciones

JCA es una especificación destinada fundamentalmente a la integración de sistemas empresariales. Como tal, compite principalmente con otras dos tecnologías: JMS y servicios web. En el apartado 6.4 se puede ver una pequeña guía para escoger la tecnología adecuada a cada proyecto.

5. Capa de recursos

La capa de recursos contiene todos los sistemas de información a los que se accederá desde la capa de integración. Estos sistemas pueden ser muy diversos, desde servidores de bases de datos relacionales o de bases de datos orientadas a objetos, hasta sistemas de ficheros COBOL, pasando por diferentes ERPs, sistemas SAP o de otros fabricantes como Siebel o Navision, etc.

6. Servicios web

Si hay una tecnología que este de moda, esa es sin duda la de servicios web. Los servicios web son aplicaciones software accesibles en la web a través de una URL. La comunicación se realiza utilizando protocolos basados en XML como SOAP (*Simple Object Access Protocol*) y enviando los mensajes a través de protocolos de Internet como HTTP. Los clientes utilizan un servicio web accediendo directamente a su interfaz, definida mediante WSDL (*Web Service Definition Language*) o buscando el servicio web en algún repositorio o registro de servicios web.

La principal utilidad de los servicios web es que promueven la interoperabilidad entre diferentes plataformas, sistemas y lenguajes. Con servicios web, por ejemplo, sería posible integrar una aplicación *Windows* desarrollada con Microsoft .NET con una aplicación desarrollada en J2EE desplegada en un servidor de aplicaciones bajo un sistema Linux.

Otra ventaja importante de los servicios web son la facilidad de integración con sistemas existentes. Por ejemplo, si tenemos una aplicación desarollada en J2EE es muy sencillo crear una capa de servicios web que expongan una interfaz de acceso a dicha aplicación sin necesidad de modificar para nada la lógica de negocio ya desarrollada. Automáticamente podremos acceder a través de la web a nuestra aplicación y permitir el acceso desde otro tipo de sistemas a la misma.

Por último, cabe destacar que los servicios web facilitan el acceso a las aplicaciones a través de diferentes tipos de clientes. Al utilizar protocolos estándar como HTTP y XML, el número de clientes que pueden acceder a los servicios web se ve multiplicado. Una vez que tenemos nuestra lógica de negocio expuesta a través de un servicio web, podremos acceder al mismo utilizando un teléfono móvil, un PDA, una aplicación de escritorio, una aplicación desarrollada en otro lenguaje, un navegador web, etc. Los servicios web por lo tanto son una forma fácil de exponer nuestras aplicaciones hacia el mundo exterior sin tener que preocuparnos en recodificar las aplicaciones para soportar plataformas tan heterogéneas.

El principal problema de los servicios web es su inmadurez. Por ahora los servicios web son una tecnología que se encuentra en constante evolución y se basan en una serie de tecnologías y estándares de los cuales muchos de ellos están todavía siendo definidos y mejorados. En la actualidad existen todavía una gran cantidad de estándares aún en definición y que potenciarán todavía más el uso de servicios web. Ejemplos de esto son la gestión de la seguridad a través de servicios web, el control de transacciones, la creación de *workflows*, etc.

El objetivo final de los servicios web es permitir la integración de múltiples sistemas empresariales y de negocio, heterogéneos, a través de complejos flujos de negocio y con óptimas garantías de seguridad, transaccionalidad, etc. Así, se presentan como una alternativa óptima a los sistemas de intercambio electrónico de datos (EDI) ya que presentan una solución mucho menos costosa al estar basados en estándares abiertos que permiten el despliegue tanto de soluciones muy sencillas y baratas como de soluciones extremadamente costosas y complejas.

Un ejemplo de este tipo integración se puede encontrar en la aplicación modelo de los *blueprints* de *SUN Microsystems* para J2EE 1.4, aplicación claramente centrada en los servicios web y en su integración con la especificación en J2EE. En la figura 12, se puede observar como *Adventure Builder* propone la integración de instituciones financieras, operadoras de vuelo, hoteles y proveedores de aplicaciones de ocio a través de un servicio web de procesado de pedidos y como esta información una vez integrada puede ser accedida a través de aplicaciones *stand-alone*, navegadores web, PDAs o clientes creados en cualquier lenguaje y ejecutados en cualquier sistema operativo.

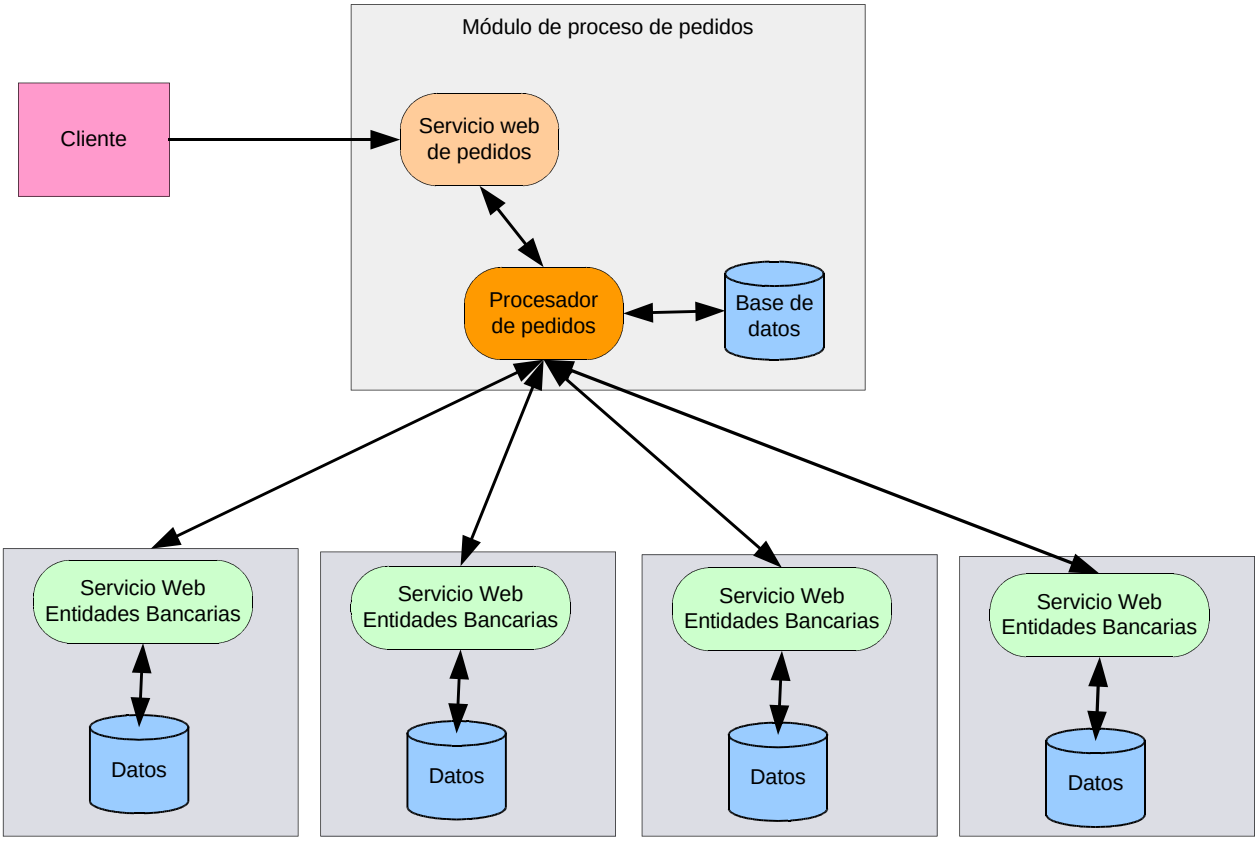

Figura 12 : Integración de múltiples *partners* en *Adventure Builder*

6.1 Servicios web en J2EE 1.3

En la actualidad, en la especificación J2EE 1.3, el modo más típico de desplegar servicios web es exponiendo nuestra lógica de negocio a través de Servlets que serán accedidos a través de alguna tecnología de acceso a servicios web como JAX-RPC (*Java API for XML-based RPC*) o JAXM (*Java API for XML Messaging*). Algunos servidores de aplicaciones como BEA WebLogic o IBM WebSphere ofrecen sus propios *frameworks* de servicios web en los que no entraremos.

JAX-RPC y JAXM son los sistemas estándar de acceso a servicios web en Java. JAXM es una tecnología más compleja que JAX-RPC y ofrece funcionalidades que esta última no puede ofrecer como son la mensajería asíncrona, el envío a múltiples destinatarios o la garantía de envío de los mensajes. Además, siempre que sea posible se recomienda utilizar JAX-RPC ya que es una espcificación mucho más simple y que requiere muy poco esfuerzo por parte del desarrollador para la realización de servicios web.

JAX-RPC es la especificación principal para la creación de servicios web tanto para aplicaciones cliente como para servidor. Esta especificación se centra en la ejecución de procesos RPC, punto a punto, utilizando mensajería SOAP. A pesar de que puede ser extendido para soportar mensajería asíncrona, ese no es el objetivo de esta API, sino que su objetivo es resultar de fácil uso para las tareas más comunes. Esa es la razón de que sea el API más recomendada.

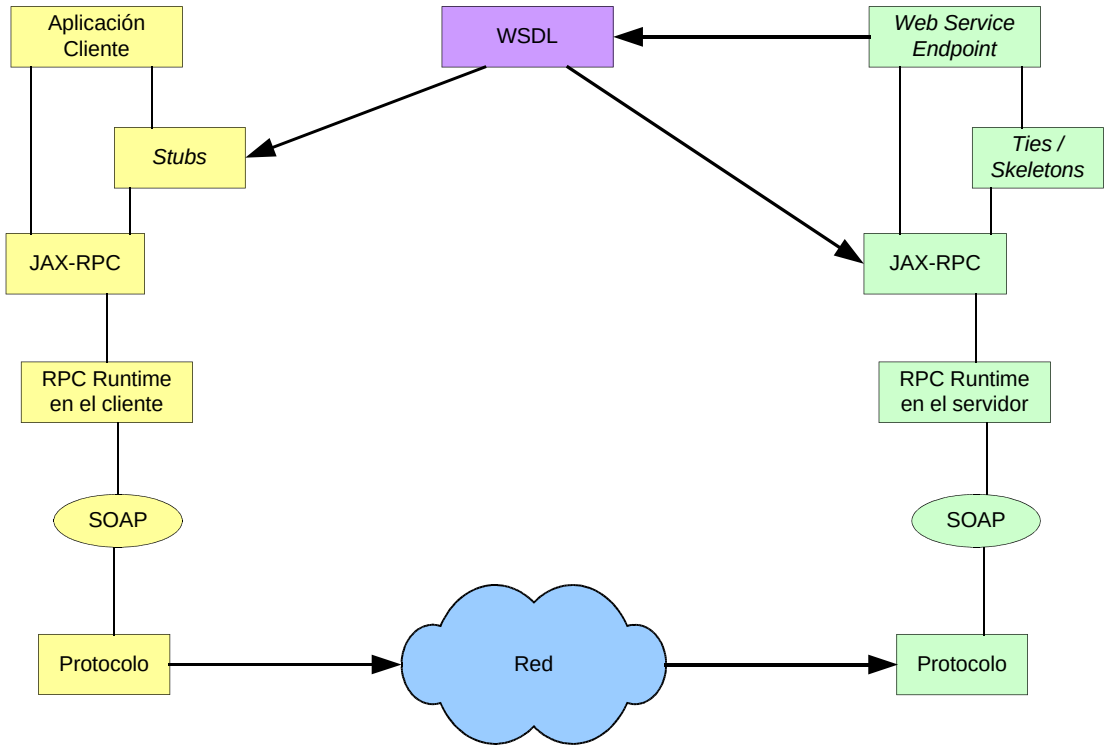

Figura 13 : Funcionamiento típico del API JAX-RPC

En la figura 13 se puede ver el funcionamiento de una implementación de JAX-RPC. El desarollador tan sólo tiene que crear una interfaz para su servicio web y una implementación de dicha interfaz. JAX-RPC automáticamente se encarga de crear el fichero de descripción WSDL, las clases proxy (*stubs* y *ties*) y de publicar el servicio web en un registro UDDI. Todo el proceso de traducción del contenido de los mensajes SOAP a clases y objetos Java se realiza de modo automático sin que el desarrollador deba preocuparse absolutamente de nada.

Podríamos enfrascarnos en una descripción mucho más profunda, tanto de JAX-RPC como del resto de tecnologías que conforman la pila de servicios web en J2EE, pero no es el objetivo de este artículo, para obtener más información puede consultar las referencias.

6.2 Servicios web en J2EE 1.4

El objetivo principal de la especificación J2EE 1.4 es añadir soporte completo de servicios web dentro de los servidores de aplicaciones J2EE. Para ello, todos los servidores compatibles con la especificación ofrecerán una implementación del *WS-I Basic Profile* definido por la *WS-I (Web Services Interoperability Organization*), organización que busca promover la interoperabilidad de los servicios web a través de las diferentes plataformas de desarrollo, los diferentes sistemas operativos y los diferentes lenguajes.

Así, la especificación J2EE 1.4 incluye especificaciones y tecnologías para soportar SOAP, WSDL, UDDI (*Universal Discovery, Description and Integration*), y ebXML (*Electronic Business using eXtensible Markup Language*). La plataforma incluye las APIs JAX-RPC, JAXM, SAAJ (*SOAP with Attachments API for Java*), JAXR (*Java API for XML Registries*), JAXP (*Java API for XML Processing*) y JAXB (*Java API for XML Binding*). La figura 14 muestra esta pila de APIs y protocolos y su integración en J2EE 1.4.

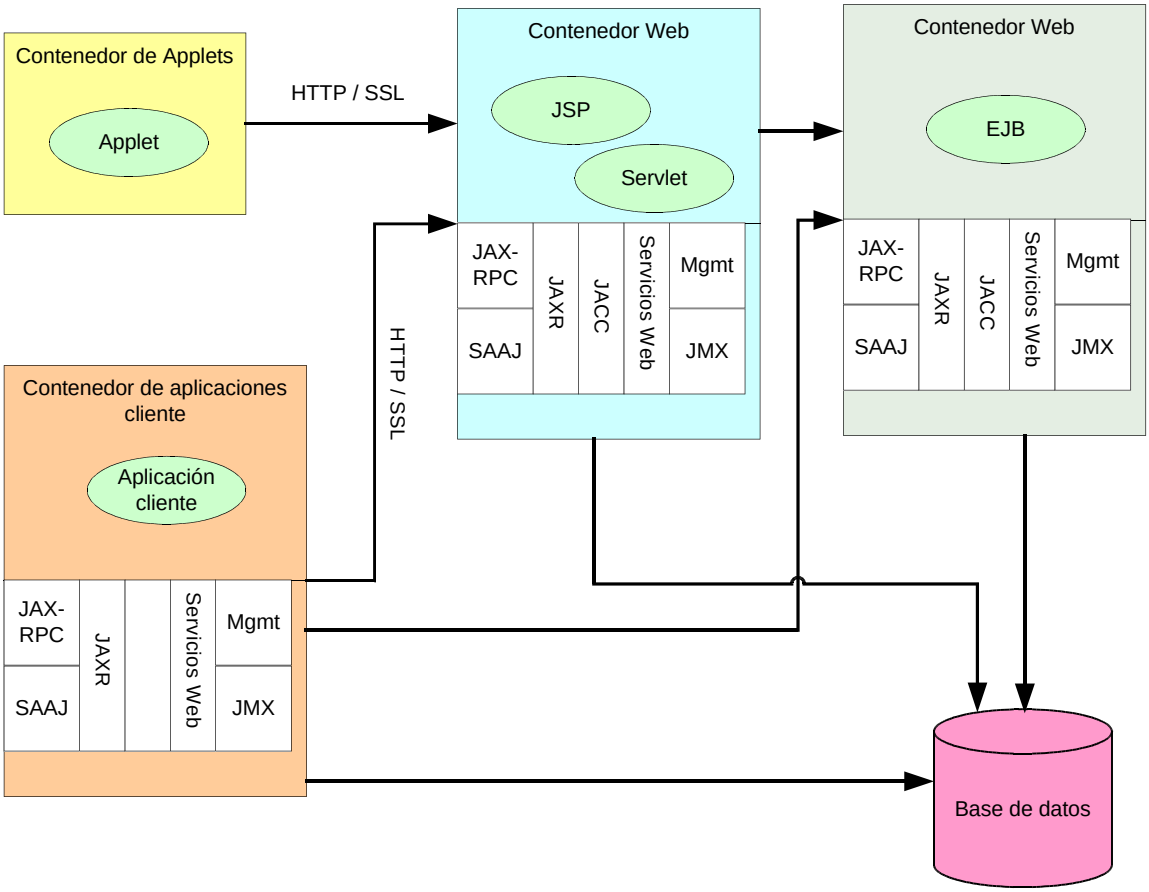

Figura 14 : Arquitectura de J2EE 1.4

En J2EE 1.4 aparece un nuevo concepto denominado *web service endpoint.* Un *endpoint* es el lugar donde se recibe la petición de ejecución de un servicio web por parte del cliente. El *endpoint* se encargará automáticamente de extraer la información sobre el método que desea ejecutar el cliente y sus parámetros así como de realizar automáticamente todo el mapeo entre XML y Java necesario para por último enviar todo esto a la implementación del método expuesto por el servicio web para su ejecución. Un proceso similar es ejecutado automáticamente por el *endpoint* para enviar la respuesta al cliente.

Toda la generación de *stubs* y/o *proxies* dinámicos se realiza de modo automático por parte del servidor de aplicaciones. Los tipos de clientes para un servicio web de J2EE 1.4 no están limitados. Pueden ser tanto otro servicio web, como un PDA, un teléfono móvil, una apliacción *stand-alone*, un componente J2EE o cualquier otro componente realizado en otra plataforma, otro lenguaje u otro sistema operativo.

Tal como se puede ver en la figura 15, J2EE 1.4 define dos tipos diferentes de *web service endpoints*:

- JAX-RPC *service endpoint* : Se implementa como un Servlet
- EJB *service endpoint* : Se implementa como un bean de sesión sin estado

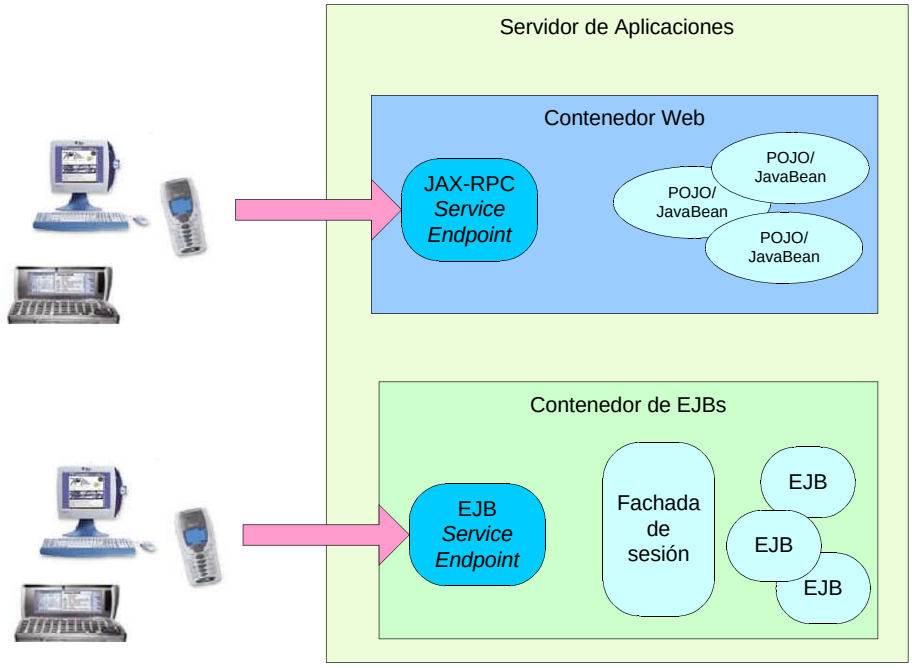

Figura 15 : Tipos de *web service endpoints* definidos en J2EE 1.4

La elección del tipo de *endpoint* a utilizar se basará en gran parte en el modo en que se haya implementado la lógica de negocio. Si se ha creado una aplicación web y la lógica de negocio se localiza en POJOs, entonces es preferible utilizar un *endpoint* basado en un servicio JAX-RPC, es decir, un Servlet. Por el contrario, si la lógica de negocio está localizada en *Enterprise Java Beans* se recomienda utilizar un *EJB service endpoint.*

Aún así, los *blueprints* de *SUN Microsystems* para servicios web recomiendan tener en cuenta otra serie de factores adicionales:

- **Tener en cuenta a la lógica de pre-procesado de la aplicación**. Si la lógica de pre-procesado ocurre en la capa de presentación, es decir, la capa web, entonces se debe utilizarn un *endpoint* de JAX-RPC. Sin embargo, si la lógica de pre-procesado se encuentra en la capa de *Enterprise Java Beans* se debe utilizar un EJB service *endpoint.*
- **Consideraciones de concurrencia**. UN EJB *service endpoint* no se tiene que preocupar por la gestión de la concurrencia ya que se implementa como un bean de sesión sin estado y se encarga el propio contenedor de EJBs de realizar la gestión. Un JAX-RPC *service endpoint* se implementa como un Servlet por lo que tiene que encargarse de gestionar la gestión a no ser que implemente la interfaz *SingleThreadModel* algo que no está recomendado ya que obliga a secuencializar el acceso al Servlet.
- **Consideraciones de transaccionalidad**.Un EJB *service endpoint* no tiene que preocuparse por la gestión de transacciones ya que es el contenedor de EJBs el que se encarga de este trabajo. Un JAX-RPC *service endpoint*, al implementarse como un Servlet, necesita controlar manualmente las transacciones a través de una API como JTA (*Java Transaction Service*)
- **Consideraciones de seguridad**. Cuando sea necesario controlar la seguridad a nivel de método se recomienda utilizar un EJB *service endpoint* ya que la especificación EJB permite el control de seguridad a dicho nivel. Con un JAX-RPC *service endpoint* el desarrollador necesita crear su propio sistema de seguridad.
- **Acceso a la sesión HTTP**. Si es necesario acceder a la sesión HTTP entonces obligatoriamente hemos de utilizar un JAX-RPC *service endpoint*. De todos modos, los servicios web no deberían depender nunca de este tipo de sesión ya que idealmente representan servicios sin estado.

Al diseñar servicios web, conviene crear servicios con una granularidad bastante gruesa. Con servicios web ocurre algo muy similar a lo que ocurre con los beans de entidad; al ser necesario realizar llamadas a través de la red para poder acceder al servicio web, es reponsabilidad del desarrollador el hacer la menor cantidad posible de llamadas remotas para minimizar el impacto en el rendimiento. Por si fuera poco, además del coste de realizar la llamada remota hay que añadir el coste asociado a realizar la traducción entre XML y objetos Java. También es importante que en servicios web, XML es un formato de texto que ocupa mucho más ancho de banda que un formato binario como RMI-IIOP. Realizar múltiples llamadas de granularidad fina en lugar de una única llamada de granularidad gruesa provocaría grandes problemas en el rendimiento de las aplicaciones.

6.3 Mapeo entre clases y XML

Las librerías de mapeo entre XML y Java o librerías de XML *data binding* hacen una labor muy parecida a la que hacían los motores de persistencia. Se encargan de transformar documentos en formato XML a clases Java y viceversa. Estas librerías tradicionalmente han sido bastante utilizadas para generación de informes, manejo de ficheros de configuración, almacenamiento de información estructurada, etc. Con la llegada de servicios web y el auge del uso de XML su uso se ha visto más extendido si cabe, conviertiéndose en herramientas fundamentales.

La mayor parte de este tipo de herramientas se basan en el concepto de *binding compiler*. Este compilador, que suelen incluir las librerías, se encarga de crear las clases Java necesarias para representar el esquema XML de los documentos que queremos convertir. Una vez creadas las clases, la librería de mapeo está en disposición de transformar documentos XML conformes a dicho esquema en jerarquías de clases utilizando para ello esas clases que había compilado previamente.

JAXB es la librería de *data binding* incluida en J2EE 1.4 y que se utiliza en JAX-RPC para transformar los mensajes SOAP en clases y parámetros que recibirán los servicios web para su ejecución. Cuando no se utiliza una librería como JAX-RPC para implementar servicios web, o cuando enviamos *attachments* en formato XML, JAXB toma un papel fundamental ya que nos permite transformar esos documentos XML que hemos enviado en clases Java que podamos manejar mucho más fácilmente desde nuestros programas.

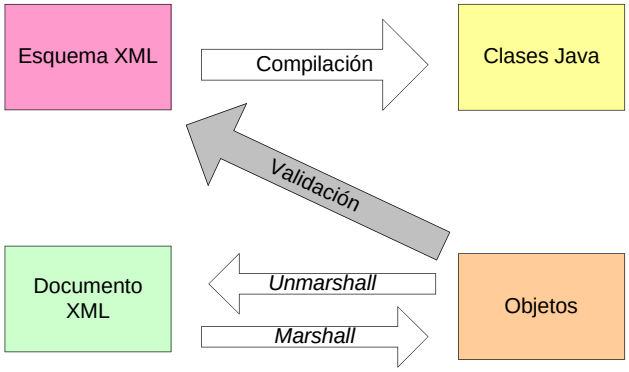

Figura 16 : Funcionamiento típico de una herramienta de *XML binding*

Normalmente estos sistemas de *data binding* consumen menos memoria de la que consumiría la representación del mismo documento XML utilizando un API como DOM. Esto se debe a que estos sistemas suelen estar basados en *parsers* ligeros de XML con lo que el consumo y rendimiento es mucho mayor. Por ejemplo, JAXB está basado en SAX 2.0.

Como ya se ha dicho, JAXB no es la única herramienta de mapeo entre XML y clases Java. Otras herramientas bastante utilizadas con Castor XML, XMLBeans o JaxMe (una implementación libre de JAXB)

6.4 Servicios web en comparación con JCA y JMS

JCA y JMS son dos tecnologías muy ligadas al sistema operativo Java y por lo tanto se recomiendan cuando sea esta nuestra principal plataforma de desarrollo. En sistemas más heterogéneos en los cuales se vaya a acceder a las aplicaciones utilizando aplicaciones escritas en otros lenguajes entonces se recomienda utilizar Servicios Web.

JCA es una tecnología en la que el cliente y el servidor están fuertemente ligados mientras que JMS y Servicios Web son sistemas con una ligadura floja en la que el cliente es independiente siempre del servidor. JMS es una interfaz de mensajería asíncrona. JCA y servicios web permiten tanto comunicación síncrona como asíncrona.

Si tenemos aplicaciones Java y planeamos continuar creando aplicaciones Java que queremos que se integren entre ellas entonces nos conviene utilizar o JMS o JCA. Por otra parte, si lo que queremos es interactuar con sistemas ofrecidos por nuestros *partners* nos conviene utilizar Servicios Web ya que no podemos ofrecer ningún tipo de control sobre dichos sistemas. Si necesitamos acceder a aplicaciones escritas en diferentes lenguajes entonces nuestra elección se limita a JMS o Servicios Web. Por otra parte, si lo que queremos es acceder a un sistema legacy que ofrece un driver JCA entonces la elección es muy clara.

7. Generación de código

Para terminar con este recorrido por el desarrollo de aplicaciones en J2EE conviene hablar de una serie de herramientas que se están imponiendo para la creación de arquitecturas empresariales.

La creación de aplicaciones en J2EE es algo complejo, no cabe duda. Existen gran cantidad de tecnologías y algunas de ellas como los *Enterprise Java Beans* esconden una complejidad que puede derrumbar los desarrollos más sólidos a primera vista. Crear aplicaciones bajo J2EE se puede convertir para el desarrollador en una tarea muy tediosa ya que está obligado a crear gran cantidad de clases e interfaces y a tener en cuenta múltiples factores. Por poner un ejemplo, para crear un simple EJB que sea accesible local y remotamente, el desarrollador se ve obligado a implementar cuatro interfaces y dos clases además de tener que crear un descriptor de despliegue estándar y varios descriptores cuyo número y complejidad dependerá del servidor de aplicaciones. Por último todo eso tiene que ser empaquetado siguiendo una estructura estándar definida por la especificación.

Por si fuera poco, si una vez que está creado el EJB el desarrollador quiere portarlo a un nuevo servidor de aplicaciones se verá obligado a crear de nuevo los descriptores de despliegue y si por casualidad había decidido utilizar alguna optimización propia de un servidor de aplicaciones en concreto tendrá que modificar también el código de dicho EJB.

Los IDEs modernos automatizan en gran medida esta generación de código ahorrándole gran cantidad de trabajo al desarrollador. Aún así, los IDEs no suelen contemplar todas las posibilidades. Si en el ejemplo anterior tenemos que el usuario quiere crear un bean BMP, que utilizará Hibernate como motor de persistencia, que será accedido mediante una fachada de sesión y que utilizará *Value Objects* como parámetros y en el que además esta fachada de sesión será accedida mediante una aplicación web implementada con *Struts* nos encontramos ante un caso donde la mayor parte de IDEs sucumbirían.

Ahí es donde herramientas como XDoclet muestran todo su potencial. XDoclet, tal como se define en su página web, es un motor de generación de código que permite un modelo de pogramación orientado a atributos dentro de Java. En pocas palabras, esto significa que el desarrollador puede añadir funcionalidad adicional a su código añadiendo metadatos (atributos) en sus programas Java. Estos metadatos se describen a través de una serie de etiquetas JavaDoc especiales.

Una vez creados los metadatos, XDoclet analiza el código fuente de las clases Java y genera toda la información especificada en los atributos que puede variar enormemente desde pequeños ficheros XML de configuración hasta gigantescas clases de mapeo entre Java y una base de datos relacional. Este modelo tan flexible propuesto por XDoclet permite al desarrollador centrarse exclusivamente en la lógica de negocio de las aplicaciones en lugar de tener que preocuparse por aspectos más específicos del sistema como pueda ser el tipo de tecnología utilizada para crear la persistencia, la gestión de la configuración o la marca del servidor de aplicaciones que se está utilizando.

Xdoclet posee multitud de etiquetas que simplifican la creación de aplicaciones empresariales: Etiquetas para *Struts*, para motores de persistencia como Hibernate, OJB o Castor JDO, para creación de Servicios Web con Axis, para creación de *Enterprise Java Beans*, Servlets, JSPs, creación de tests unitarios, etc.

Otra de las grandes ventajas de XDoclet es que posee etiquetas para multitud de servidores de aplicaciones. Estas etiquetas no solo permiten desplegar la aplicación en dichos servidores sino que también permiten aprovechar las características específicas de cada uno de ellos simultáneamente. Utilizando XDoclet es posible implementar una aplicación y obtener automáticamente un paquete desplegable para varios servidores de aplicaciones sin apenas esfuerzo adicional.

El ejemplo más característico del potencial de XDoclet es la XPetStore. XPetStore es una implementación de la Pet Store de *SUN Microsystems* utilizando XDoclet. XPetStore no sólo ocupa muchas menos líneas que la implementación original sino que su rendimiento es mucho mayor. Por si fuera poco, se han permitido el lujo de realizar dos implementaciones, una utilizando EJBs y otra utilizando únicamente Servlets y JSPs.

Uno de los exponentes más claros de la flexibilidad del modelo orientado a atributos es el último entorno de desarrollo de BEA Systems en el cual está basado completamente en este tipo de etiquetas. Tanto los controles simples (integración con base de datos, JCA, JMS, etc.), como los controles Java, los servicios web, las aplicaciones web (con soporte para Struts) y los *workflows* entre aplicaciones son configurables a través de etiquetas de anotación y todo parametrizable mediante un interfaz visual basado en propiedades.

Otras herramientas de generación de código menos conocidas pero también muy a tener en cuenta son Jenerator, EJBGen, UML2EJB o MiddleGen.

8. Conclusiones

Durante estos últimos años, con el aumento de popularidad de la plataforma J2EE han ido apareciendo una serie de patrones de desarrollo adicionales a los expuestos oficialmente por *SUN Microsystems*. En este artículo, hemos visto las ventajas e inconvenientes tanto de los patrones oficiales como de estos nuevos modelos de desarrollo *de facto* impuestos como resultado del *feedback* proveniente del trabajo diario del desarrollador con esta plataforma.

Como se ha visto en este artículo, existen una serie de herramientas que no han sido incluidas dentro de los *blueprints* de *SUN Microsystems* que se presentan como piezas fundamentales para el desarrollo de aplicaciones. *Frameworks* de persistencia o las herramientas de generación de código ofrecen al desarollador un importante ahorro de trabajo incrementando considerablemente su productividad.

La plataforma J2EE es una plataforma compleja, y como hemos visto, repleta de alternativas y posibilidades para implementar cada una de las capas típicas de una aplicación distribuida. Esta gran cantidad de alternativas puede desembocar en una saturación del desarrollador al llegar a un punto donde la grandeza de esta sopa de letras que son las diferentes APIs de J2EE le impide ser capaz de realizar la elección adecuada. Esta guía ha pretendido ofrecer un poco más de luz a esta dificil elección.

Ante todo, la principal conclusión que se debe sacar de este artículo es que se debe intentar encontrar la arquitectura más simple que permita satisfacer los requerimientos no funcionales de nuestras aplicaciones. Por ejemplo, si tenemos una aplicación muy simple sin requerimientos transaccionales, de seguridad o de concurrencia entonces no tiene sentido que la sobrecarguemos con EJBs. Si tenemos que soportar múltiples bases de datos diferentes para nuestra persistencia, a lo mejor nos conviene evaluar algún tipo de motor de persistencia.

Es también muy importante resaltar como los modelos oficiales de desarrollo, es decir, los *blueprints*, han evolucionado para adaptarse a los nuevos de desarrollo ligeros impuestos por las limitaciones de implementaciones como los *Enterprise Java Beans*. Escenarios de aplicaciones *solo-web* no soportados hasta ahora, pasan a tomar un rol fundamental dentro de la arquitectura de J2EE y de lo que serán los patrones de diseño de J2EE 1.4

Sería demasiado ambicioso el pretender que este trabajo supusiese una guía completa para el desarrollo de aplicaciones empresariales. Este tema ya ha sido, y lo seguirá siendo, fruto de un análisis profundo por parte de gran cantidad de libros y publicaciones. En las referencias puede encontrar algunos libros que le pueden ser de utilidad en esta lucha constante por sobrevivir a la complejidad de la plataforma J2EE.

9. Enlaces de interés

1. *Alur, Crupi y Malks. Core J2EE Patterns 2nd Edition, 2003, ...*
2. *Floyd Marinescu. EJB Design Patterns, 2002, Wiley*
3. *Bret McLaughlin. Building Java Enterprise Applications, 2002, O'Reilly*
4. *Designing Web Services with J2EE 1.4, http://java.sun.com/blueprints/guidelines/designing_webservices/*
5. *Varios autores. Designing Enterprise Applications with the J2EE Platform 2nd Edition, 2003, SUN Microsystems*
6. *Roman, Ambler y Jewel. Mastering EJBs 2nd Edition, 2002, Willey*
7. *Martin Fowler. Patterns of Enterprise Application Architecture, 2003, Addison Wesley*
8. *Gamma, Helm, Johnson y Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software, 1995, Addison Wesley*
9. *Eric Evans. Domain Driven Design, Tackling Complexity in the Heart of Business Software. Eric Evans, http://www.domainlanguage.com*
10. *J2EE Blueprints, http://java.sun.com/blueprints/enterprise/*
11. *J2ME Blueprints, http://java.sun.com/blueprints/wireless/*
12. *Blueprints para servicios web, http://java.sun.com/blueprints/webservices/*
13. *Java Pet Store, http://java.sun.com/blueprints/code/index.html#java_pet_store_demo*
14. *Adventure Builder, http://java.sun.com/blueprints/code/index.html#java_adventure*

10. Acerca del autor

Martín Pérez Mariñán es Ingeniero Técnico en Informática de Sistemas por la universidad de A Coruña. Posee las certificaciones de SUN Certified Java Programmer, SUN Certified Java Developer y SUN Certified Business Component Developer. Actualmente trabaja para la empresa Dinsa Soluciones como arquitecto J2EE desarrollando aplicaciones en dicha plataforma para el ámbito sanitario dentro del Hospital Juan Canalejo donde se ha especializado en la creación de aplicaciones de gestión económica de este tipo de entidades. Entre otras cosas Martín Pérez es miembro del equipo del portal hispano sobre tecnología Java javaHispano. También ha publicado artículos técnicos en revistas online y escritas, ha realizado presentaciones en varios eventos sobre desarrollo de software en la diferentes universidades de la geografía española y también suele impartir cursos sobre Java y J2EE para empresas.