Estudios de Doctorado *Avances en Informática*
Curso Tecnologías WEB
Universidad de Oviedo

Definición de una arquitectura software para el diseño de aplicaciones web basadas en tecnología Java-J2EE

Daniel Fernández Lanvin
20217190Y
dflanvin@hotmail.com

Tabla de Contenidos

Introducción ... 3
Aplicaciones WEB ... 3
  Evolución ... 3
    Modelo 1 ... 3
    Modelo 1.5 ... 3
    Modelo 2 ... 4
    Modelo 2X ... 4
Definición de la arquitectura software ... 5
  Aspectos generales ... 5
  Descomposición funcional del sistema ... 9
    Descomposición en microaplicaciones ... 9
  Separación Lógica en capas ... 11
    Capa de presentación ... 11
    Capa de negocio ... 16
    Capa de acceso a datos ... 17
    Capa de infraestructura ... 19
  Comunicación entre capas. Desacoplamiento. ... 21
Bibliografía ... 24

Introducción

El objeto de este documento es definir una arquitectura flexible y consistente, basándose en la utilización de patrones de diseño, para la construcción de aplicaciones web empleando tecnología java. Partiendo del popular modelo de aplicación de 3 capas, se pretende alcanzar un modelo más refinado que evolucione el actual, describiendo ciertas recomendaciones y pautas de diseño, y haciendo hincapié en los puntos más débiles que habitualmente aparecen en este tipo de proyectos, como la gestión de la sesión de usuario, escalabilidad, portabilidad a sistemas similares, etc. No se pretende abarcar todas las soluciones existentes para cada posible escenario en un proyecto de estas características, sino agrupar las opciones de diseño más recomendables para una aplicación web de corte común, en base a los problemas que habitualmente aparecen durante la elaboración de este tipo de productos.

Aplicaciones WEB

Lo que el mercado demanda actualmente son mayormente aplicaciones web, que permitan a las empresas tradicionales llegar al cliente de a pié sin necesidad de que éste se desplace hasta la ubicación física de la misma. Partiendo de este tipo de productos, además de la tecnología empleada para dar solución al problema, se han ido creando y perfeccionando distintas arquitecturas más o menos independientes de la tecnología aplicada. Dentro de esta familia de aplicaciones o herramientas, hay que destacar el papel que el lenguaje JAVA, junto con sus extensiones J2EE y el apoyo del mundo open-source están jugando.

Evolución

La evolución tecnológica que el sector ha sufrido durante los últimos años ha permitido otra evolución paralela, la de la arquitectura de las aplicaciones web. A medida que aparecían nuevos recursos técnicos, los patrones de diseño se amoldaban para aprovechar las nuevas característica que estás novedades ofrecían. De esta forma, el modelo arquitectónico de las aplicaciones de Internet ha sufrido dos grandes saltos desde la aparición de los primeros portales. Los distintos modelos de aplicación sobre los que ha ido desarrollando se clasifica en:

Modelo 1

Son las más primitivas. Se identifican con este modelo las clásicas aplicaciones web CGI, basadas en la ejecución de procesos externos al servidor web, cuya salida por pantalla era el html que el navegador recibía en respuesta a su petición. Presentación, negocio y acceso a datos se confundían en un mismo script perl.

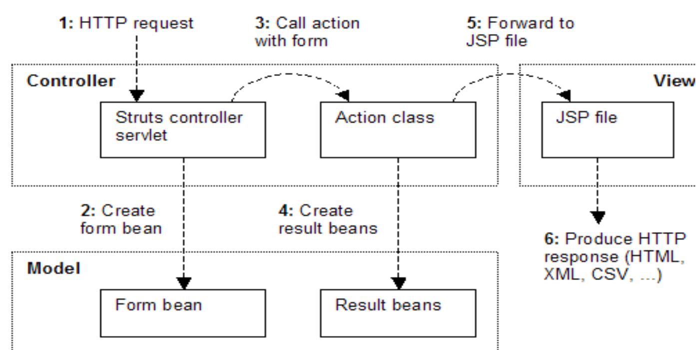
Modelo 1.5

Aplicado a la tecnología java, se da con la aparición de las jsps y los servlets. En este modelo, las responsabilidades de presentación (navegabilidad, visualización, etc)

recaen en las páginas jsps, mientras que los beans incrustados en las mismas son los responsables del modelo de negocio y acceso a datos.

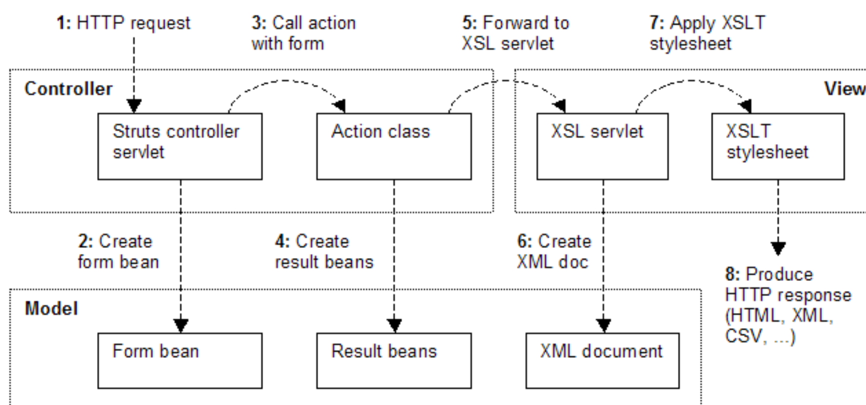
Modelo 2

Como evolución del modelo 1.5 con la incorporación del patrón MVC a este tipo de aplicaciones, se define lo que se conoce como *Model 2* de la arquitectura web. En el diagrama siguiente se aprecia la incorporación de un elemento controlador de la navegación de la aplicación. El modelo de negocio queda encapsulado en los javabeans que se incrustan en las jsps, aunque como se verá a lo largo del resto del documento, esta responsabilidad se explota cuando se alcanza el modelo de diseño *n*-capas[FOW02].



Modelo 2X

El modelo 2X aparece como evolución del modelo 2, y con objeto de dar respuesta a la necesidad, cada vez más habitual, de desarrollar aplicaciones multicanal, es decir, aplicaciones web pueden ser atacadas desde distintos tipos de clientes remotos. Así, una aplicación web multicanal podrá ejecutarse desde una PDA, desde un terminal de telefonía móvil, o desde cualquier navegador html estándar. El medio para lograr publicar la misma aplicación para distintos dispositivos es emplear plantillas XSL para transformar los datos XML y determinar la plantilla a emplear en base al parámetro ***user-agent*** recibido en la request. La aplicación de esta solución al modelo 2 de aplicaciones web define lo que se conoce como modelo 2X [STXX][MERC02].



Definición de la arquitectura software

Es este apartado se describe la arquitectura propuesta para la construcción de aplicaciones web basadas en tecnología java. El objetivo es recoger una serie de recomendaciones de diseño que, mediante la aplicación de patrones y soluciones ya experimentadas en proyectos reales, ayuden a conseguir productos sólidos, estructurados y, sobre todo, de fácil mantenimiento.

Aspectos generales

A la hora de abordar el desarrollo de un proyecto web, hay una serie de consideraciones acerca del mismo que hay que tener muy presentes, dado que son claves en que tipo de diseño y metodologías de desarrollo aplicar.

- Escalabilidad

  Una de las características principales de las aplicaciones publicadas en la Web es que el número de usuarios de la misma puede verse incrementado de forma vertiginosa en un periodo de tiempo relativamente corto. El éxito o el fracaso de un sitio web orientado al usuario común determinarán, entre otros aspectos, el dimensionamiento del sistema sobre el que se instala y soporta el software que sustenta dicho sitio. En consecuencia, una de los requisitos fundamentales de una aplicación web es que sea completamente escalable sin que un aumento de los recursos dedicados a la misma suponga modificación alguna en su comportamiento o capacidades. La escalabilidad de un sistema web puede ser:

  - **Horizontal**: cuando literalmente clonamos el sistema en otra máquina (u otras máquinas) de características similares y balanceamos la carga de trabajo mediante un dispositivo externo. El balanceador de carga puede ser:

    - *Balanceador Software* – Por ejemplo, habitualmente encontramos un servidor web apache junto con el módulo mod_jk que permite redirección las peticiones http que a tal efecto sean configuradas entre las distintas máquinas que forman la granja de servidores. Este tipo de balanceadores examinan el paquete http e identifican la sesión[1] del usuario, guardando registro de cual de las máquinas de la granja se está encargado de servir a dicha sesión. Este aspecto es importante, dado que nos permite trabajar (de cara al diseño de la aplicación) apoyándonos en el objeto session propio del usuario y almacenando en ella información relativa a la sesión del mismo, puesto que tenemos la garantía de que todas las peticiones de una misma sesión http van a ser redireccionadas hacia la misma máquina.

    - *Balanceador hardware* – Se trata de dispositivos que, respondiendo únicamente a algoritmos de reparto de carga (*Round Robin*, LRU, etc.), redireccionan una petición http del usuario a la máquina que, según dicho algoritmo, convenga que

[1] Se define la sesión del usuario como la secuencia de eventos que ha provocado desde que el usuario comienza su ejecución de la aplicación.

se haga cargo de la petición. Son mucho más rápidos de los anteriores, dado que se basan en conmutación de circuitos y no examinan ni interpretan el paquete http. Sin embargo, el no garantizar el mantenimiento de la misma sesión de usuario en la misma máquina, condiciona seriamente el diseño, dado que fuerza a que la información relativa a la sesión del usuario sea almacenada por el implementador del mismo, bien en cookies o bien en base de datos[2].

- *Balanceador hardware http*- Se trata de dispositivos hardware pero que si que examinan el paquete http y mantienen la relación usuario – máquina servidora. Mucho más rápidos que los balanceadores software, pero algo menos que los hardware, suponen hoy en día una de las soluciones más aceptadas en el mercado.

- **Vertical:** Habitualmente, la separación lógica en capas se implementa de tal forma que se permita una separación física de las mismas. Interponiendo elementos conectores que actúen de middlewares (por ejemplo, EJBs [ROMA01]), es posible distribuir la aplicación de forma vertical (una máquina por cada capa del sistema), e incluso si esto no fuera suficiente, distribuyendo los elementos de una misma capa entre distintas máquinas servidoras.

- **Cluster**: Con la aparición de los servidores de aplicaciones en cluster se abre una nueva capacidad de escalabilidad que, dependiendo de cómo se aplique, podría clasificarse como vertical u horizontal. Un cluster de servidores de aplicaciones permite el despliegue de una aplicación web corriente de forma que su carga de trabajo vaya a ser distribuida entre la granja de servidores que forman el cluster, de modo transparente al usuario y al administrador. El cluster, mediante el mecanismo de replicación de sesión, garantiza que sea cual sea la máquina que sirva la petición http tendrá acceso a la sesión del usuario (objeto HttpSession en java). Este tipo de sistemas, debido precisamente a la replicación de sesión, suele presentar problemas de rendimiento.

Pese a que hace un tiempo la tendencia de diseño era contraria al empleo de la sesión (objeto session) como soporte de apoyo en el desarrollo del sistema, sobre todo debido a problemas de rendimiento y a la abundancia de balanceadores hardware comunes, hoy en día es habitual el empleo de la misma. No obstante, es un recurso delicado, dado que un abuso de está técnica acarrea problemas de rendimiento por un excesivo uso de memoria. Hay que tener en cuenta a la hora de hacer el diseño de una aplicación web java que, hasta que la sesión no caduque, todos los objetos contenidos en la misma que no hayan sido eliminados explícitamente persisten en la memoria del servidor, puesto que no están disponibles para el recolector de basura.

[2] Al mantener la información de sesión en base de datos, es accesible desde cualquiera de las instancias de la aplicación. No obstante, es necesario controlar la caducidad de la misma para evitar un crecimiento desmesurado del repositorio.

- Separación de responsabilidades o incumbencias entre los distintos elementos del sistema.

  Esta premisa supone la base de la separación en capas del sistema. Distintas responsabilidades no deben ser delegadas en la misma clase, y llevado esto algo más allá, en el mismo conjunto de clases. En la actualidad, la tendencia más aceptada es la aplicación de patrones de diseño de arquitectura que dividen la responsabilidad en distintas capas (separación de incumbencias) que interaccionan unas con otras a través de sus interfaces. Se trata de los sistemas denominados multicapa o *n*-capas [JURI00]. Aplicados a los proyectos web, el modelo más básico es el de aplicaciones 3 capas:

  1. *Presentación*

     Como su nombre indica, se limita a la navegabilidad y a gestionar todos aquellos aspectos relacionados con la lógica de presentación de la aplicación, como comprobación de datos de entrada, formatos de salida, internacionalización de la aplicación, etc.

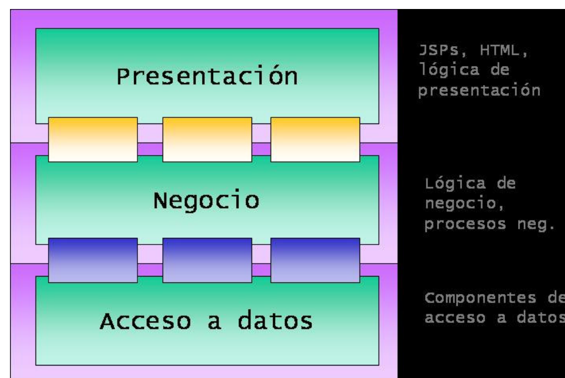
  2. *Negocio o dominio*

     El resultado del análisis funcional de la aplicación viene a ser la identificación del conjunto de reglas de negocio que abstraen el problema real a tratar. Estás son las que realmente suponen el motor del sistema, dado que se basan en el funcionamiento del modelo real. En un caso hipotético de un programa de gestión cualquiera, estaríamos hablando, por ejemplo, del conjunto de operaciones que dan el servicio de negocio "emisión de factura".

  3. *Acceso a datos*

     Esta capa es la encargada de persistir las entidades que se manejan en negocio, el acceso a los datos almacenados, la actualización, etc, aunque puede ofrecer servicios relacionados con la persistencia o recuperación de información más complejos.

Tomando como base este modelo, distintos patrones arquitectónicos han ido apareciendo como evolución del mismo (Modelo Brown [FOWL02], los patrones de Sun, etc.), casi todos insertando más capas que habitualmente están orientadas a conseguir una mayor independencia entre las tres anteriormente descritas.



- Portabilidad

En la medida de lo posible, una aplicación web debe poder adaptarse a las distintas posibles arquitecturas físicas susceptibles de ser empleadas para el despliegue del paquete, limitándose en la medida de lo posible el impacto de tal adaptación a tareas de configuración, y evitándose así la necesidad de modificar el código de la misma ante dichas situaciones. Un ejemplo que se da habitualmente, es el encontrar un cliente reacio al empleo de tecnologías J2EE por los consabidos costes de rendimiento (o simplemente económicos) que acarrean. Dado que el modelo de separación física de capas mediante Enterprise javabeans de tipo sesión implementando el patrón de diseño Façade [GOF94][JURI00] es algo habitual y recomendado desde muchos ámbitos académicos, surge la necesidad de modificar el código del producto para eliminar las capas intermedias, puesto que no se cuenta con un contenedor el EJBs para la implantación.

- Componentización de servicios de infraestructura

En todas las aplicaciones, incluidas las aplicaciones web, aparecen una serie de servicios que podríamos denominar de infraestructura, y que han de estar disponibles desde distintas partes del sistema. Así, esta necesidad rompe aparentemente la separación vertical de capas, dando lugar a una capa de infraestructura que sirve funcionalmente a todas las demás. Casos habituales de servicios de esta naturaleza son:

- Servicio de log
- Pool de conexiones JDBC (o de cualquier otro sistema de persistencia)
- Sistema de configuración
- Gestor de accesos/permisos de usuario.
- Etc.

La arquitectura propuesta pretende tratar este conjunto de servicios como componentes, servicios de la capa de infraestructura, de forma que:

1. Puedan ser sustituidos por nuevas versiones sin necesidad de parada del sistema ni recompilación y/o repaquetización del mismo
2. Puedan ser reutilizados por futuros proyectos o distintas partes del mismo, evitando que en un mismo sistema coexistan n distintos gestores de permisos, n distintos proveedores de conexiones a bases de datos, gestores de logs, etc.
3. Sean accesibles desde todas las capas del sistema sin romper la independencia entre las mismas.

- Gestión de la sesión de usuario, cacheado de entidades

  Con objeto de limitar en la medida de lo posible los accesos innecesarios a memoria secundaria (bases de datos, ficheros externos de configuración, etc) se propone un sistema que se apoya en parte en el empleo de la sesión HTTP(s) para cachear ciertos datos referentes a la sesión del usuario, o bien comunes a todas las sesiones de usuario. Obviamente, la cantidad y naturaleza de las entidades susceptibles de ser cacheadas será determinada teniendo muy presentes aspectos de rendimiento del producto, dado que un empleo abusivo de esta técnica puede y suele llevar a un consumo excesivo de los recursos del sistema (memoria).

- Aplicación de patrones de diseño

  El empleo y aplicación de patrones de diseño [GOF94] facilita el entendimiento del código y, por tanto, reduce considerablemente el coste de mantenimiento, dado que además de aportar soluciones eficientes para problemas comunes, son muy interesantes como medio de entendimiento entre diseñadores e implementadores.

Descomposición funcional del sistema

Uno de los objetivos más claros e importantes en el proceso de desarrollo de software ha de ser la reutilización de código. Este principio hay que contemplarlo desde dos perspectivas distintas: no desarrollar elementos o componentes que ya existan y estén probados, y desarrollar pensando en que puede que mañana el código que se está implementando pueda ser reutilizado en distintos proyectos. El factor de reutilización de un paquete de código viene determinado por la independencia que exista entre los componentes del sistema. Si una clase realiza varias invocaciones directas a otra distinta, no será portable a otro proyecto a no ser que no llevemos la segunda también. A un nivel más alto de abstracción se debe también procurar desarrollar módulos lo más independientes posible del resto de la aplicación.

Descomposición en microaplicaciones

Si se aplica este principio a nivel de proyecto entero, y procuramos descomponer el producto identificando funcionalidades, obtendríamos una aplicación final formada por una agrupación de lo que podríamos denominar módulos o microaplicaciones. Pongámonos, por ejemplo, en el caso del desarrollo de una intranet web, proyecto muy común por otra parte. Servicios habituales del la intranet son la gestión de usuarios, la publicación de noticias, foros, etc. Afinando más, la gestión de usuarios, por ejemplo, se puede separar entre la de personas, la de perfiles, la de grupos, etc. Surge en este contexto el problema de la colaboración intrínseca entre estas microaplicaciones (dado

que en caso contrario, no serían tales, sino aplicaciones completas), puesto que procesos asociados a una microaplicación requerirán de otros procesos o datos localizados en microaplicaciones distintas. Un error de diseño frecuente en este tipo de contextos es permitir que una microaplicación recupere información directamente del repositorio de datos mantenido por otra. Esto genera productos fuertemente acoplados, y dificulta mucho las labores de mantenimiento, dado que un cambio leve en el modelo de datos de un servicio cualquiera puede conllevar efectos laterales importantes. Se pierde el control sobre la gestión de las entidades de la base de datos, además de dificultar la depuración de errores.

Colaboración entre microaplicaciones

Cuando surge la necesidad de colaborar con ciertos procesos desde una microaplicación a otra, la primera cuestión a resolver es a que nivel de la arquitectura se debe hacer esto, manteniendo siempre el principio de separación de responsabilidades y de máxima independencia entre los distintos módulos de la aplicación. Como se ha dicho antes, se ha de evitar el acceso directo al repositorio de información. Si por el contrario, se accediera a nivel de la capa de acceso a datos, a través de sus façades, se evitaría el problema antes descrito. Sin embargo, limita la posible evolución de la microaplicación fuente de datos, puesto que se esta forzando a que mantenga la misma interfaz en su capa de acceso a datos en un futuro. El punto adecuado para comunicar dos microaplicaciones es pues la capa de negocio, dado que además es en esta capa donde se comprueba la potestad del usuario para invocar a los procesos en ella implementados. Así, será necesario que la microaplicación que requiere del servicio se autentifique en la segunda, para evitar que terceros sistemas invoquen procesos de negocio inadecuadamente. Esto puede obligar a elevar servicios de persistencia al nivel de la capa de negocio, incluso cuando no sea necesario implementar reglas de negocio, pero así se mantiene centralizado el control de acceso en una misma capa.

Para mantener la independencia entre microaplicaciones, debe sustituirse el modelo habitual de colaboración entre módulos, es decir, la invocación directa de sus clases, por otro que permita un mayor desacoplamiento de los mismos. A este punto se presentan distintas alternativas, de donde se destacan estas dos:

1. Empleo de EJBs de sesión.

   La invocación de los servicios de negocio de una microaplicación desde otra mediante un EJB tiene la gran ventaja de que, al ser habitual el empleo de estos componentes a modo de façade, es posible que no tenga impacto alguno en la microaplicación fuente de datos. Además, puede llegar a ser imprescindible si la operación a realizar se viera implicada en una transacción, dado que el EJB es capas de gestionar transacciones distribuidas. Por otro lado, la invocación es costosa (RMI o IIOP), y sobre todo inviable si las microaplicaciones cuentan con un firewall entre ellas. Esto limita la distribución y/o sustitución del producto.

2. Servicios WEB

   Los servicios web cuentan con la ventaja de permitir invocaciones remotas en formato XML y sobre el protocolo (entro otros) http. Esto posibilita la distribución o colaboración de las microaplicaciones presentes en distintas redes, o a través de Internet, dado que no son interceptados por un fierwall. Se

debe publicar un servicio web por cada servicio de negocio que se desee compartir con otras microaplicaciones de la misma aplicación o, porqué no, de otras aplicaciones o sistemas distintos. Esta opción es más aconsejable que la anterior, aunque inviable en el caso de las transacciones distribuidas.

El empleo de esta alternativa de diseño permite amplias posibilidades de distribución de un mismo producto, pero además facilita enormemente algo muy demandado hoy en día en el mercado como la integración con otros sistemas de la misma compañía. Con esta arquitectura, es relativamente sencillo adaptar, por ejemplo, la gestión de permisos sobre servicios de la aplicación, a un nuevo entorno con un gestor propio, o centralizar el control de acceso para todas las compañías de un grupo empresarial, cada una con su propia instancia del producto, en un mismo punto de la red. El desarrollo de servicios web es sencillo y está disponible para prácticamente todos los entornos de desarrollo.

Separación Lógica en capas

A la hora de plantear el diseño de una aplicación web, el primer paso es conseguir separar conceptualmente las tareas que el sistema debe desempeñar entre las distintas capas lógicas y en base a la naturaleza de tales tareas. Se ha de partir de la separación inicial en tres capas, diferenciando que proceso de los que hay que modelar responde a tareas de presentación, cual a negocio y cual a acceso a datos. En caso de identificar algún proceso lógico que abarque responsabilidades adjudicadas a dos o más capas distintas, es probable que dicho proceso deba ser explotado en subprocesos iterativamente, hasta alcanzar el punto en el que no exista ninguno que abarque más de una capa lógica.

Capa de presentación

Como se dijo anteriormente, es la responsable de todos los aspectos relacionados con la interfaz de usuario de la aplicación. Así, en esta capa de resuelven cuestiones como:

- Navegabilidad del sistema, mapa de navegación, etc
- Formateo de los datos de salida: Resolución del formato más adecuado para la presentación de resultados. Está relacionado directamente con la internacionalización de la aplicación
- Internacionalización: Los textos, etiquetas, y datos en general a presentar se obtendrán de uno u otro fichero de recursos en base al idioma preferido del navegador del usuario. En base a esta condición se ven afectadas las representaciones numéricas, las validaciones sobre los datos de entrada (coma decimal o punto decimal) y otros aspectos relativos al idioma del usuario remoto.
- Validación de los datos de entrada, en cuanto a formatos, longitudes máximas, etc.
- Interfaz gráfica con el usuario.
- Multicanalidad de la aplicación: Una misma aplicación web puede contar con varias presentaciones distintas, determinándose el uso de la adecuada en base al

dispositivo visualizador desde el que trabaje el usuario. Así, no se representará la misma información con el mismo formato en un navegador web estándar que en un dispositivo móvil provisto de un navegador WAP.

El Patrón Model-View-Controller

El MVC es un patrón arquitectural aportado por SmallTalk y hoy en día muy difundido en uso en aplicaciones de entorno web. La evolución de lo que se conoce como modelo 2 de aplicaciones web (separación de responsabilidades de presentación, negocio y navegación) avanza un poco más en el reparto de tareas en la aplicación web. Pese a que hay distintos puntos de vista acerca de la forma de aplicar e implementar este patrón, en esencia las ideas principales sobre su estructura y funcionalidad son las mismas. El MVC tiene tres piezas claves que se reparten la responsabilidad de la aplicación:

- El modelo (model)

  Responsable de toda la lógica y estado del dominio de negocio.

- La vista (view)

  Responsable de la presentación del dominio de negocio.

- El controlador (controller)

  Responsable del flujo de control, la navegabilidad y el estado de la entrada del usuario.

1. **El modelo**

   En base al tipo de arquitectura sobre el que se está construyendo la aplicación, el modelo puede seguir distintos patrones en su diseño. En una aplicación dos capas, el modelo estaría compuesto por clases java corrientes que interactuarían directamente con la base de datos. Sin embargo, en una aplicación web basada en la arquitectura defendida en este documento, el modelo estará integrado por las capas inferiores a la de presentación, donde estará integrada la implementación del patrón MVC que se utilice.

2. **La vista**

   La vista en una aplicación web está compuesta por aquellos elementos que aporten algo a la presentación, como jsps, páginas html, imágenes, animaciones, componentes, etc. La mayoría del contenido dinámico de la presentación será generado en la capa superior de la aplicación, en el servidor de aplicaciones, aunque es posible que debido a requisitos o simplemente preferencias del implementador, parte se genere en el cliente por medio de algún lenguaje de script.

3. **El controlador**

   Habitualmente implementado por medio de un servlet (en proyectos java, lógicamente) es el corazón del funcionamiento del patrón. Es responsable de:

   - Interceptar y recoger las peticiones http del cliente. Así, el cliente no invocará directamente ninguna página jsp o html, sino que será redireccionado adecuadamente por el controlador.

- Traducir la petición en una operación de negocio específica.
- Invocar la operación o bien delegar en un manejador.
- Determinar la siguiente vista a mostrarle al cliente
- Retornar el control al cliente.

El hecho de que todas las peticiones http pasen por el controlador facilita el mantenimiento de la aplicación, sobre todo en lo referente al control de la navegabilidad, sustitución de páginas ,etc.

El Framework Struts

El proyecto Struts lo lanzó en Mayo del 2000 Craig R. McClanahan para proporcionar un marco de trabajo MVC estándard a la comunidad Java. En Julio del 2001, se liberó Struts 1.0. Este framework del proyecto Jakarta [JAKA03] es la implementación java orientada a aplicaciones web más difundida del patrón MVC. Provee su propio controlador (ActionServlet), y se integra con otras tecnologías para proveer el modelo y la vista. La navegación se configura en ficheros XML externos a modo de AFD con eventos. Struts organiza la lógica y responsabilidades siguiendo la distribución del MVC entre las siguientes clases y componentes:

- **ActionForms**

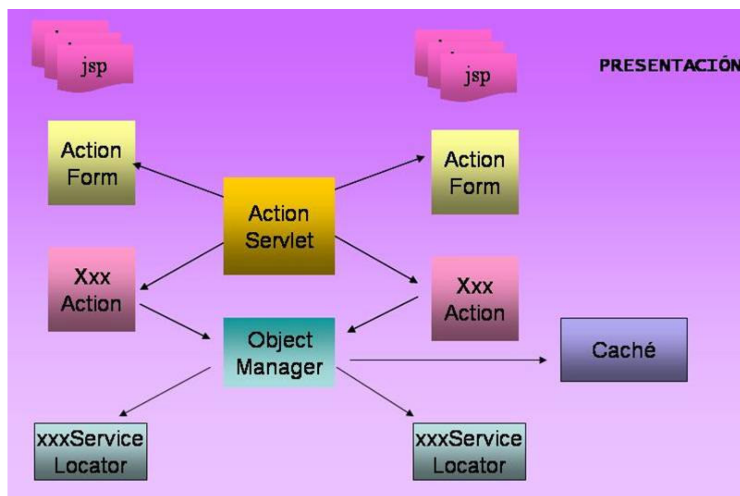
  Son parte de la vista. Representan formularios html y su uso facilita tareas como la validación de formatos, rellenado de formularios, etc.

- **ActionServlet**

  Es el elemento controlador, configurado por medio del fichero struts-config.xml.

- **Actions**

  Una de las características más interesantes de Struts es que induce a un diseño que identifica las acciones susceptibles de ser invocadas desde presentación y las mapea con un servlet (un action) a cada una. De esta forma, se fuerza un diseño muy modular y reutilizable, puesto que es común que distintas pantallas ejecuten la misma acción.



Persistencia de entidades en presentación

Uno de los aspectos a cuidar a la hora del elaborar el diseño de una aplicación web es como se han de gestionar las entidades en la capa de presentación, y más concretamente, hasta que punto y de que forma apoyarse en el empleo de la sesión del usuario para preservar ciertos objetos entre distintas invocaciones. El uso adecuado de esta técnica en sustitución del apoyo en cookies[3], o del mantenimiento de la sesión en base de datos, puede mejorar notablemente el rendimiento del sistema, al evitar accesos innecesarios a memoria secundaria, y facilitar la implementación del mismo. Sin embargo, se trata de un recurso delicado, dado que un crecimiento descontrolado del tamaño de la sesión de usuario como repositorio de información puede llevar a una aplicación web a colapsarse con un número de usuarios muy reducido. Por otro lado, la carencia de comprobación de tipos a la hora de guardar datos en sesión conlleva riesgo de errores de programación y dificulta el mantenimiento del producto. A continuación se propone una solución de diseño de la capa de presentación que facilita el desarrollo de este aspecto del proyecto, procurando aprovechar en la medida de lo posible la potencia que el empleo de la sesión de usuario aporta a la aplicación, pero tratando de minimizar los posibles problemas derivables de su uso.

El ObjetManager

Esta clase se instancia y coloca en la sesión de cada usuario la primera vez que se solicita. Su tarea es centralizar la lógica de recuperación de datos, de forma que cuando alguno de los elementos de la capa de presentación (jsps, servlets, etc) requiere un dato, un bean o cualquier otra cosa, el implementador del mismo no se ha de preocupar de sí ha de buscarlo en sesión, en el ámbito (*scope*) de aplicación, o recuperarlo del servicio de configuración o de negocio; simplemente, se lo pide al objetManager. Es esta clase la que sabrá, bien por reglas implementadas o bien por configuración externa, donde colocar, cachear y recuperar los datos que maneja en base a las reglas referidas. Nuevamente se separan responsabilidades, incluso dentro de la misma capa de presentación, lo cual facilita el futuro mantenimiento del sistema. Lógicamente, esta clase deberá contar con un método por cada objeto susceptible de ser recuperado a través suyo, debido a lo cual es completamente dependiente del modelo, y no portable a otras aplicaciones.

Caché de tres niveles

La caché será manejada directa y únicamente por el ObjectManager. En una aplicación web común, hay ciertos datos susceptibles de ser retenidos en memoria, puesto que debido a la alta frecuencia de solicitud de los mismos, el coste que supone mantenerlos en las sesión del usuario compensa frente al coste de recuperarlos de memoria secundaria en cada petición, sobre todo se trabajamos con un modelo de aplicación cuyas capas se encuentran separadas físicamente en distintas máquinas[4]. Así, por ejemplo, los elementos contenidos dentro de un combo-box presente en la pantalla

[3] El empleo de cookies para mantener el estado de la sesión de usuario puede presentar problemas en tanto en cuanto ciertos navegadores web (clientes) no las aceptan, además de resultar tediosas a la hora de serializar según que estructuras de datos y limitadas en capacidad.

[4] Referido a la distribución vertical de la aplicación, cuando las capas se separan físicamente, hay que tener en cuenta que por cada petición que se lance de una capa a otra, se está ejecutando una invocación RMI o IIOP, con el consecuente coste que ello conlleva.

principal de la aplicación, susceptible esta de ser cargada varias veces durante la sesión del usuario, son candidatos a ser cacheados en la sesión del usuario (siempre y cuando estemos hablando de un tamaño moderado de estructuras). Si los elementos contenidos en el combo-box no fueran dependientes del usuario, sino comunes a todos ellos (por ejemplo, la lista de países, idiomas, etc.), el lugar adecuado para almacenarlos sería en el objeto Application, común a todas las sesiones del servidor de aplicaciones. Es importante implementar ciertos mecanismos de control sobre este tipo de técnicas, dado que un uso abusivo de los objetos session o application puede generar serios problemas de rendimiento cuando el número de usuarios concurrentes alcanza ciertas cotas. Así mismo, es necesario un control sobre la caducidad de los datos almacenados, tanto en sesión como en la aplicación, puesto que por ejemplo, en el caso de la segunda, los datos no se refrescarían hasta que se reiniciase la aplicación, algo que en un sistema de producción puede no ocurrir en varios meses o años.

El empleo de esta técnica implica, lógicamente, que el uso del objeto session quede centralizado a través de la caché. Es conveniente que la caché ejerza el control sobre los tipos de los objetos que se almacenan a través suyo. Así, por ejemplo, si el diseñador, jefe de proyecto o cualquier otro responsable configura la caché para que el objeto que se almacene en sesión bajo la etiqueta ***user*** sea del tipo UserBean (es decir, un bean que represente la entidad usuario), la caché deberá evitar, controlando el tipo de los objetos que se le pasen, que se trate de colocar un objeto de tipo String bajo esa misma etiqueta, aunque contenga en esencia la misma información (el login de usuario). En proyectos donde distintos equipos desarrollan distintas partes del sistema de forma paralela, el forzar este modo de trabajo evitará errores de coordinación entre los distintos desarrolladores implicados.

Aunque lo que mas comúnmente se almacena en memoria (sesión o aplicación) son objetos de la naturaleza del perfil del usuario o similares, hay ciertos casos en los que, sobre todo debido al alto coste que implica la obtención de los resultados, o a la alta frecuencia de invocación, lo que conviene retener en memoria son los parámetros de devolución de ciertos métodos. Así, pongámonos en el caso de que un informe de cierre de caja conlleve la ejecución de una sentencia SQL pesada, de varios minutos de cálculo en el motor de base de datos, y que dicho informe sea susceptible de ser consultado por todos los gerentes de la empresa durante los primeros días del mes. Interesaría en este caso almacenar el resultado del informe (que no cambiará hasta el mes que siguiente), con una caducidad preventiva de una semana, para tampoco almacenar durante tres semanas datos que no van a ser consultados por nadie, y con un tamaño máximo de objetos de este tipo en caché de uno, de forma que tan pronto se invocara el cierre del mes siguiente (sólo cambia el parámetro), este dato se retirara de la caché. Otro caso parecido sería la lista de provincias de un país. Si el publico objetivo de una aplicación web es en un 95% español, nos interesa cachear el resultado de la consulta que retorna las provincias españolas o, por ejemplo, el resultado de la consulta para los tres países más seleccionados en el ámbito de la aplicación. En conclusión, la caché deberá almacenar los datos a tres niveles: objeto, método y parámetros.

Parametrización de la caché

Con objeto de que la caché se pueda manejar como un componente, y ser reutilizado en otros proyectos sin necesidad de modificar su código, deberá permitir ser parametrizada mediante ficheros XML externos. Entre los parámetros a establecer, por cada uno de los objetos o métodos a cachear tendrá:

- identificador del objeto a cachear
- nº máximo de objetos de este tipo a cachear (por ejemplo, cinco listas de provincias)
- caducidad del dato
- Política de reemplazo de elementos a emplear (Round Robin, LRU, etc).
- scope del dato: Será sesión o aplicación. Le decimos así a la caché donde almacenar y de donde recuperar el dato que se le solicita.
- Tipo del dato y sus parámetros si corresponde. La caché comprobará que el objeto que se intenta almacenar es del tipo adecuado.

Capa de negocio

En esta capa es donde se deben implementar todas aquellas reglas obtenidas a partir del análisis funcional del proyecto. Así mismo, debe ser completamente independiente de cualquiera de los aspectos relacionados con la presentación de la misma. De esta forma, la misma capa de negocio debe poder ser empleada para una aplicación web común, una aplicación WAP, o una standalone. Por otro lado, la capa de negocio ha de ser también completamente independiente de los mecanismos de persistencia empleados en la capa de acceso a datos. Cuando la capa de negocio requiera recuperar o persistir entidades o cualquier conjunto de información, lo hará siempre apoyándose en los servicios que ofrezca la capa de acceso a datos para ello. De esta forma, la sustitución del motor de persistencia no afecta lo más mínimo a esta parte del sistema. Debería poder reemplazarse el gestor de bases de datos por un conjunto de ficheros de texto sin necesitar tomar ni una línea de código de presentación o negocio. Las responsabilidades que conviene abordar en esta capa son:

- Implementación de los procesos de negocio identificados en el análisis del proyecto.

  Como se deduce del párrafo anterior, los procesos de negocio implementados en esta capa son totalmente independientes de cualquier aspecto relativo a la presentación de los mismos. Pongamos por ejemplo la generación de un informe que conste de varias filas las cuales deberán sombrearse con un color determinado por la superación o no de ciertos umbrales para ciertos valores. La aplicación de ciertos umbrales, y la consecuente determinación de la situación (correcta, incorrecta, etc) de cada valor de la fila es un cálculo de negocio que debe afrontarse en esta capa. Sin embargo, sería en un error completar un campo adicional en el que, como resultado del cálculo, se determinara directamente el color de la fila. Lo adecuado es codificar la situación de la misma y, una vez en presentación, determinar el color adecuado en base al código recibido. De esta forma, si el usuario requiere que a partir de cierta fecha, el color rojo sea sustituido por el naranja, la modificación de este aspecto de presentación de información se limitaría en la aplicación a una modificación de la capa de presentación.

- Control de acceso a los servicios de negocio.

Dado que una misma aplicación puede contar con más de una capa de presentación al mismo tiempo, es aconsejable que la responsable última de ejecutar tareas sobre el control de acceso a los servicios del sistema no sea la capa de presentación, sino la de negocio. Los implementadores de la capa de negocio ni pueden ni deben confiar en que las futuras implementaciones de nuevas capas de presentación gestionen adecuadamente el acceso a los servicios de negocio. De esta forma, en cada invocación a cualquier método restringido de negocio se deberá comprobar por medio del sistema de autentificación adecuado, los derechos del usuario actual a realizar tal operación.

- Publicación de servicios de negocio

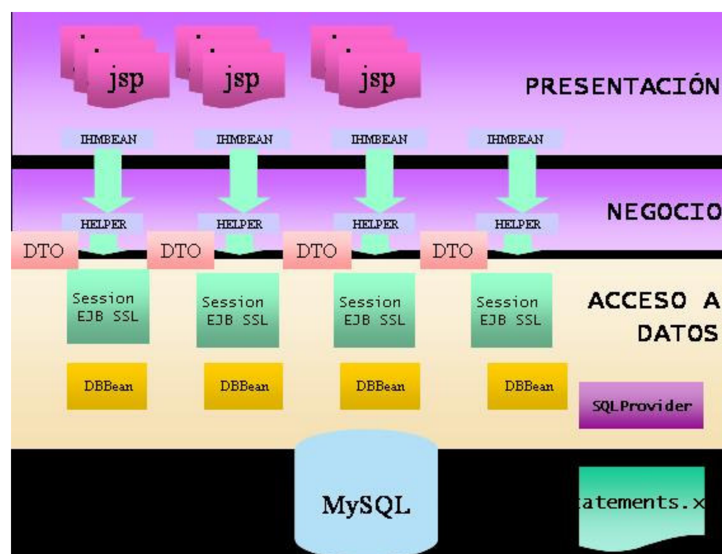
  Tal y como se explicó en el apartado relativo a la descomposición funcional del sistema al hablar de las microaplicaciones, el lugar adecuado para que dos microaplicaciones o aplicaciones completas interactúen es a nivel de la capa de negocio. Así mismo, el modelo de colaboración recomendado en esta definición de arquitectura es el que se basa en el empleo de servicios web. De esta forma, la capa de negocio ofrecerá dos vistas alternativas, dado que por un lado el conjunto de façades con presentación ofrecerá a esta los servicios que se requieran para el funcionamiento de la microaplicación en sí, y por otro, otro conjunto de servicios serán publicados por medio (recomendablemente) de servicios web. Estos últimos estarán orientados a la colaboración con otros sistemas distintos u otras microaplicaciones pertenecientes al mismo proyecto, y deberán por tanto llevar un control más férreo sobre el acceso al servicio, dado que ahora, además de hacerlo a nivel de usuario, puede que sea necesario hacerlo a nivel de aplicación, por lo que cada aplicación remota debería estar explícitamente autorizada a invocar el servicio de negocio publicado.

- Invocación a la capa de persistencia

  Los procesos de negocio son los que determinan que, como y cuando se debe persistir en el repositorio de información. Los servicios ofertados por la interfaz de la capa de acceso a datos son invocados desde la capa de negocio en base a los requerimientos de los procesos en ella implementados.

Capa de acceso a datos

La capa de acceso a datos es la responsable de la gestión de la persistencia de la información manejada en las capas superiores. En un modelo académicamente purista, la interfaz de esta capa estaría compuesta por vistas de las entidades a persistir (una vista de “factura”, otra de “cliente”), pero a efectos prácticos, y con objeto de aprovechar la habitual potencia de los gestores de bases de datos, la interfaz muestra una serie de servicios que pueden agrupar operaciones en lo que se puede denominar “lógica de persistencia”, como **insertar cliente** o **inserción factura**, en la que podrían darse de alta al mismo tiempo una factura y todos las entidades que dependan de dicha factura (porque no, el mismo cliente).



Si atendemos a una aplicación web común implementada en java sobre una base de datos relacional, lo más probable es encontrarse con un modelo arquitectónico de acceso a datos consistente en un conjunto de servicios verticales, uno por cada una de las entidades presentes en el modelo de datos, que ofrecen a las capas superiores las operaciones de persistencia. Las entidades del dominio están representadas cada una por un javabean, una clase que contiene un atributo y sus correspondientes métodos get y set de acceso por cada miembro de la entidad, y que tienen la interesante característica de poder ser introspeccionados, de forma que se pueda conocer lo que en última instancia sería la estructura de una tabla en un modelo relacional.

El conjunto de interfaces remotas de los Ejbs componen la interfaz de la capa de acceso a datos con las capas superiores, a priori la de negocio o dominio. Las clases invocadas por el Façade, los aquí denominados DBBeans, son los que realmente implementan la lógica de persistencia e invocan al gestor de bases de datos para satisfacer los servicios solicitados desde negocio. Sus métodos ejecutarán las sentencias SQL necesarias (suponiendo un gestor de bases de datos relacionales). Esta es la arquitectura mayormente recomendada por los fabricantes de servidores de aplicaciones (Sun, Resine CMP, Bea Systems, etc) y por los arquitectos más célebres (Martin Fowler, Beck, etc).

El SQLProvider

Uno de los aspectos más importantes de cara al mantenimiento de una aplicación es saber donde se encuentran las cosas dentro de ella. Así, se torna aconsejable cierta centralización de los recursos, datos, u objetos en general que compartan un cometido común dentro del diseño de la aplicación. Este principio de diseño, llevada al ámbito de la interacción con el repositorio de datos, lleva a la definición del SQLProvider. Este elemento del diseño agrupará todas la sentencias SQL[5] susceptibles de ser empleadas en la capa de acceso a datos. De esta forma, se facilita la depuración y mantenimiento de la aplicación, dado que ante un fallo en una sentencia SQL, o un cambio en una de las

[5] Lógicamente, al igual que muchas de las otras recomendaciones presentes en este documento, esto se llevará a cabo siempre que sea posible, dado que en ciertos casos (generación dinámica de SQLs para la generación de informes, por ejemplo), esto no es factible al 100%, o su imposición puede complicar mucho la capa de acceso a datos.

tablas (por ejemplo, el nombre de un campo), el desarrollador sabe donde debe tocar el código. Llevando esta máxima un poco más allá, las sentencias SQL pueden ir externalizadas en un fichero XML[6], de forma que el mantenimiento, o el impacto de una posible futura migración de gestor de bases de datos[7], quedan reducidos al mínimo. Así, el SQLProvider puede implementarse como una clase que, siguiendo el patrón de diseño *singleton*, se instancie en la primera petición y en ese momento cargue a memoria el contenido del fichero XML.

En definitiva, lo que se persigue es una capa de persistencia que goce de las siguientes cualidades:

- La capa de acceso a datos encapsula toda la lógica de almacenamiento, independizando al resto del sistema del mecanismo de persistencia. Si en un momento dado, tuviéramos que sustituir la habitual base de datos relacional por una base de datos orientada a objetos, o por simples ficheros XML, el resto de la aplicación no sufriría impacto alguno.
- El empleo de un objeto servidor de sentencias SQL que las tome de un fichero XML externo facilita la migración de una base de datos a otra, dado que el impacto se limitaría a configuración, sin necesidad de codificación, compilación, reempaquetamiento o despliegue.

Capa de infraestructura

Esta capa, adyacente a todas las demás de la aplicación, comprende todos aquellos servicios susceptibles de ser requeridos desde cualquiera de las capas lógicas de la aplicación web. La gestión de un servicio y de las clases que lo implementan se realizará desde la concepción de un componente, es decir, una clase totalmente independiente de la aplicación que lo utiliza. La capa de infraestructura estará formada entonces por componentes, y por las clases gestoras necesarias para su configuración y gestión. De esta forma, cuando una clase de cualquiera de las capas lógicas de la aplicación requiera el uso de alguno de los servicios ofrecidos por la capa de infraestructura (por ejemplo, el servicio de log), no tratará directamente con la clase que implemente tal servicio, sino que lo hará por medio del interfaz que cumpla la misma. Así mismo, la instanciación del servicio es responsabilidad de las clases gestoras de la capa de infraestructura. La clase cliente del servicio le pedirá a la capa de infraestructura que le facilite una instancia de la clase que implementa el servicio que necesita. La relación entre una interfaz que defina un servicio de infraestructura y la clase que implementa el servicio se establece en un fichero externo XML. De esta forma:

- La sustitución de un componente que implemente un servicio de la capa de infraestructura por otro distinto que cumpla la misma interfaz sólo requiere modificar el fichero de configuración.
- La configuración de cada uno de los componentes irá asimismo externalizada en ficheros XML.

[6] En el diagrama, *statements.xml*
[7] Al no existir un estándar de facto del lenguaje SQL, sino que cada fabricante añade sus propias innovaciones o parámetros propietarios, el cambio de gestor implica habitualmente la adaptación de las sentencias.

- Se consigue desacoplar completamente la aplicación de su entorno de despliegue. En caso de que en un futuro tuviera que ser integrada con otros sistemas, dicha tarea podría llegar a requerir sólo el desarrollo de los componentes adecuados o en encapsulamiento de los ya presentes en el sistema anfitrión. Volviendo al ya citado ejemplo del sistema de log, es habitual que una compañía tenga normalizado el formato de salida del mismo para todos sus sistemas. Si se necesitará instalar el producto en otra compañía, que probablemente cuente también con su propio formato de trazas de log, sólo se necesitaría encapsular las clases aportadas por la nueva compañía para la generación de trazas para adaptar su interfaz al que el servicio de la aplicación impone. Si además se tendiera al empleo de interfaces estándar (aunque esto no siempre es posible), esta tarea puede quedar reducida a una simple reconfiguración del sistema.
- Las clases gestoras, en caso de que el comportamiento del componente lo permitiera, pueden trabajar con pools de componentes para aquellos cuyo uso no implique un mantenimiento de estado, y sean susceptibles de invocarse con una frecuencia elevada.
- Las clases gestoras de la capa de infraestructura deben permitir establecer períodos de reconfiguración, de forma que la alteración del comportamiento del sistema a través de sus componentes (sustitución y configuración de los mismos) se pueda hace en caliente[8], evitando una parada en el sistema de producción. La modificación del comportamiento de ciertos componentes, como por ejemplo un pool de conexiones, facilita el tunning y dimensionamiento del sistema una vez entre en producción.

Casos habituales de servicios que deben según esta filosofía pertenecer a la capa de infraestructura son:

- Servicio de log
- Pool de conexiones JDBC (o de cualquier otro sistema de persistencia).
- Sistema de configuración de la aplicación.
- Gestor de accesos/permisos de usuario a los distintos servicios de la aplicación.
- El SQLProvider descrito en la capa de acceso a datos.
- Otros más específicos del entorno del proyecto pero independientes del modelo.

Esta concepción de la capa de infraestructura es la que se sugiere con el subproyecto Ávalon de Jakarta[JAKA03]. Este conjunto de clases aportado por Apache Group facilitan la gestión de componentes a través de interfaces, así como su configuración centralizada por medio de ficheros XML.

Básicamente, el funcionamiento se reduce a que las clases del sistema le piden a Avalon el componente que implementa un servicio determinado, identificado este por

[8] *Hotdeploy*

su interfaz, y Avalon, en base a sus ficheros de configuración, instancia la clase que implementa dicha interfaz (y así determinada en estos ficheros), la configura adecuadamente con la sección del árbol XML del fichero de configuración que le corresponda, y la devuelve, haciendo una gestión óptima de los componentes y cacheando la información que considera adecuada para limitar los accesos a memoria secundaria en la medida de lo posible

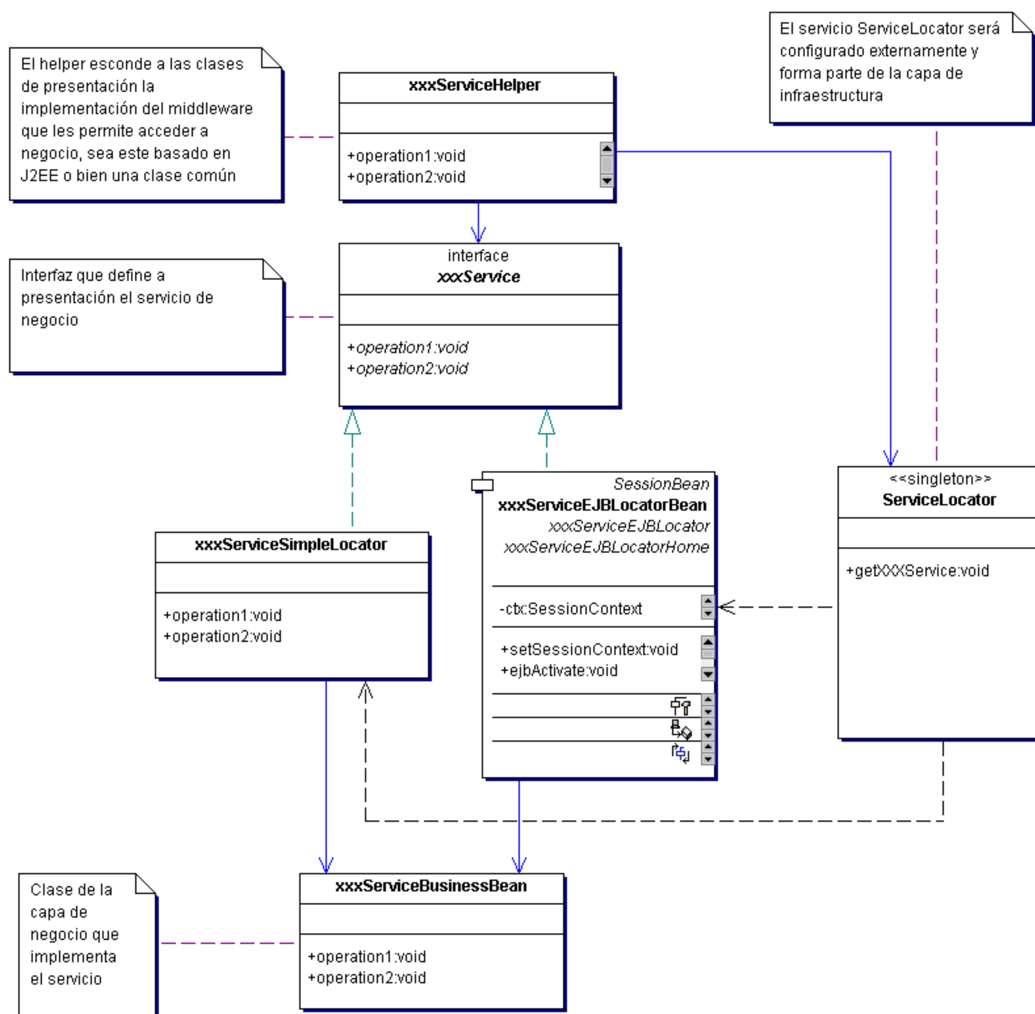
Comunicación entre capas. Desacoplamiento.

La evolución del modelo 3-capas al modelo n-capas pasa por la incorporación de las capas intermedias que permiten desacoplar las primitivas y distribuirlas por medio de algún tipo de middleware. El modelo recomendado y en boga a día de hoy es el que se sirve de EJBs de tipo sesión que, implementando el patrón de diseño Façade[GOF94], muestra a la capa superior el conjunto de servicios que ofrece la capa inferior, al mismo tiempo que posibilita la invocación remota de los mismos de forma transparente al usuario del servicio. De esta forma la separación lógica entre capas pasa a ser también física, y se consigue dotar a la aplicación de escalabilidad vertical. No obstante, este tipo de arquitectura presenta un problema importante. Cuando la aplicación no está distribuida verticalmente, porque quizá no sea necesario aún, y se encuentra desplegada en una sola máquina, se están realizando invocaciones RMI al localhost por cada solicitud de servicio que se realice de una capa a otra. Esto, lógicamente, presenta un coste innecesario que puede provocar un descenso de rendimiento importante en el sistema, puesto que el proceso de realizar una invocación remota por RMI (o IIOP[9]) resulta bastante pesado.

En respuesta a esta problemática, *Sun Microsystems* decidió extender la especificación de EJBs aprovechando la publicación de la versión 2.0 de la misma con la incorporación de los interfaces locales. Con ellos, el desarrollador conserva la separación de responsabilidades, dado que sigue trabajando con el mismo interfaz del servicio, pero con la salvedad de que lo que se está realizado por debajo no es una llamada remota, sino una simple invocación local de la clase que implementa el interfaz. Se soluciona de este modo el problema del rendimiento, pero la aplicación sigue dependiendo de la presencia de un contenedor de EJBs, pese a que su aportación al sistema no es necesaria en absoluto, dado que, al menos en lo que a la distribución de las capas se refiere[10], no se realizan invocaciones remotas. Esto incide negativamente en la portabilidad del producto, dado que es posible que se imponga la implantación del mismo sobre una arquitectura física basada en un servidor web y un motor de jsps y servlets. En el caso habitual de que la aplicación sólo empleara EJBs a modo de middleware entre capas, estaríamos limitando la portabilidad de la misma por un aspecto ajeno a su funcionalidad principal, o imponiendo al cliente la incorporación a su sistema de un contenedor de EJBs. Dado que esto no es factible, se ha de optar por una solución alternativa que, limitándose a tareas de configuración del producto, permita ser desplegado tanto en sistemas no escalables verticalmente (es decir, careciendo de la separación física de las capas por medio de EJBs) como en los que si cuentan con un contenedor de EJBs y permiten dicha escalabilidad. El siguiente patrón ofrece una solución acorde con estas premisas, apoyándose en la capa de infraestructura concebida

[9] Protocolo de comunicación de CORBA que pueden emplear los Enterprise javabeans.
[10] Es importante tener en cuenta que estamos partiendo de la suposición de que no se están empleando EJBs de tipo entidad para el acceso a base de datos o cualquier otra función, sino simplemente EJBs de tipo sesión para la separación entre capas.

tal y como se describe en este documento. En el diagrama se muestra un ejemplo de cómo aplicar dicho patrón a la separación entre la capa de presentación y la capa de negocio. En él se muestran las relaciones existentes entre las clase necesarias para un escenario bien sin distribución vertical, o bien con ella empleando EJBs de tipo sesión sin estado a modo de fachadas.



El patrón de diseño que define este tipo de solución es el *Business Delegate*[GOF94]. La función de los distintos elementos del diagrama será:

- ***xxxService*Helper**

  El service helper funciona como elemento encapsulador del puente entre el servlet de presentación y el business bean en negocio. Simplifica la resolución del medio para llegar a la clase de negocio. Por medio del ServiceLocator (servicio de la capa de infraestructura), recupera la clase que implementa la interfaz del servicio de negocio, en base a la configuración actual de la capa de infraestructura. Esta puede ser bien una clase normal que simplemente invoque lo métodos del bean

de negocio (modelo sin distribuir), bien un Enterprise JavaBean de tipo Session (modelo distribuido), o incluso un cliente RMI, cliente SOAP, etc. En el caso de un modelo sin distribución vertical, se tratará de una clase normal, permitiéndose así el despliegue del producto en una arquitectura física desprovista de contenedores de EJBs.

- **ServiceLocator**

  A esta clase se le pide que resuelva cual es el componente que, en base a un fichero XML de configuración de la capa de infraestructura, debe instanciar en un momento para la interfaz que se le solicita. Será esta la que le sirva al helper el bridge a la clase de negocio. Habitualmente, se tratará de un singleton. Al estar configurada por medio de un fichero externo, el administrador del sistema puede establecer el bridge a usar (es decir, la clase que debe implementar la interfaz del servicio en cada ejecución) editando un fichero xml plano, y permitiendo así adaptar el sistema a distintos entornos sin necesidad de recodificar o repaquetizar el producto.

- ***xxxServiceLocator***

  Cuando el puente sea una clase simple, el ServiceLocator retornará una instancia de esta clase, que se limita a llamar a los métodos del bean de negocio cuando se invoquen los correspondientes a través del interfaz. En el diagrama esta clase aparece como *xxxSimpleService*Locator. En caso de una distribución basada en EJBs, se retornaría el stub de cliente, el cual invocaría a bean de implementación del EJB, que a su vez invocaría el bean de negocio. De esta forma, es transparente al implementador de la capa de presentación la forma con la que se invoca la capa de negocio.

Bibliografía

[FOWL02]

http://martinfowler.com/apsupp/appfacades.pdf

[GOF94]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (The Gang of Four). *Design Patterns*. Addison Wesley Professional Computing Series, 1994. ISBN 0-201-63361-2

[JAKA03]

http://jakarta.apache.org

[JURI00]

Matjaz Juric, Nadia Nashi, Craig Berry, Meeraj Kunnumpurath, John Carnell, Sasha Romanosky, *J2EE Design Patterns Applied.* Wrox Press Inc, 2002. ISBN 1861005288

[MARIN01]

Floyd Marinescu, *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, 2001. ISBN 0471208310

[ROA01]

E Roman, Scott W. Ambler, Tyler Jewell. *Mastering Enterprise JavaBeans, second edition.* John Wiley & Sons, 2001. ISBN 0471417114

[STXX]

http://stxx.sourceforge.net/

[MERC02]

Julien Mercay and Gilbert Bouzeid. *Boost Struts With XSLT and XML.* Java World, Febrero 2002.