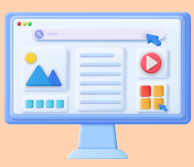
Arquitectura EE

ARQUITECTURA EE Y JAVA EE



Desarrollar aplicaciones con arquitecturas multinivel implica el uso de computación distribuida, lo que complica el proceso de desarrollo. Aunque la plataforma JavaEE simplifica en gran medida el desarrollo de sistemas en tres niveles, el diseño de componentes distribuidos es más complejo que si nos centráramos en uno o dos niveles.

- dificultar la comprensión del diseño y el código
- alargar los tiempos de desarrollo debido a la curva de aprendizaje
- encarecer el mantenimiento del software
- dificultar la evaluación y aplicación de los cambios
- provocar la realización de errores

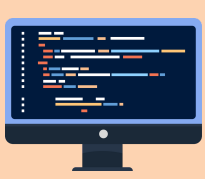
1. Capa de presentación: Esta capa se encarga de la interfaz de usuario y la presentación de datos. Puede estar compuesta por tecnologías como JavaServer Faces (JSF), JavaServer Pages (JSP), Servlets y JavaServer Faces (JSF), entre otras. Estas tecnologías permiten crear interfaces de usuario dinámicas y manejar las interacciones con el usuario.

1. Capa de acceso a datos: Esta capa se ocupa del acceso y la persistencia de los datos. Puede estar compuesta por tecnologías como Java Persistence API (JPA), que proporciona un conjunto de APIs para interactuar con bases de datos relacionales, y Java Database Connectivity (JDBC), que permite la conexión y manipulación de bases de datos.

Arquitecturas de 2 Capas vs Multicapa

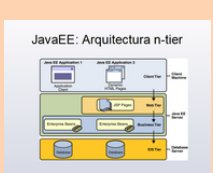
Antes de la entrada de las tecnologías distribuidas como JavaEE o CORBA, los sistemas de empresa normalmente utilizaban una arquitectura de 2 capas conocida como cliente/servidor, la cual se desplegaba a su vez en 2 niveles.
Bajo el modelo de cliente/servidor, la lógica de negocio se implementaba como parte integrada de un cliente rico y pesado que accedía al servidor para obtener los datos de una base de datos centralizada.
El modelo cliente/servidor disfrutó de un éxito considerable, ya que era fácil de comprender, fácil de probar y soportado por numerosas herramientas de desarrollo visual a alto nivel.

- Despliegue: en las grandes empresas donde existen miles de clientes, el hecho de realizar un despliegue supone un reto logístico
- Compartición de servicios: si la aplicación debe compartir servicios dentro de la empresa, y la lógica esta integrada con los clientes, tenemos una gran redundancia de código entre las diferentes aplicaciones.
- Pobre separación de la presentación y la lógica de negocio: este tipo de arquitectura provoca un alto acoplamiento, ya que es muy fácil mezclar ambas capas.



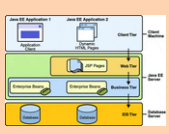
Arquitectura Web

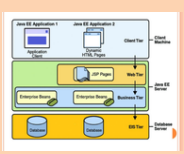
En los últimos tiempos, predominan los clientes web ligeros respecto a los interfaces de usuario pesados (Swing), aunque no hemos de descartarlos. Por lo tanto, las aplicaciones web son el estándar de facto de las aplicaciones empresariales.
Estas arquitecturas, fácilmente escalables a nivel de hardware (clustering), pueden acceder al mismo tipo de APIs que los EJBS, además, se benefician de las capacidades que ofrece el servidor J2EE, como pueden ser la gestión de transacciones y el pool de conexiones.



Finalmente, también pueden utilizar todos los servicios de empresas, tales como JMS, JDBC, JavaMail y JCA. La única tecnología a la que no puede acceder es a los Entity Beans. Por lo tanto, cubre las necesidades de la mayoría de las aplicaciones.

En este tipo de arquitectura, tanto la capa web como la de negocio se ejecutan en la misma JVM, normalmente desplegadas sobre un contenedor web (tipo Tomcat, Resin, etc...). Sin embargo, es muy importante, que a nivel lógico si que las separemos.





Un prerequisito de la mayoría de arquitecturas empresariales es la escalabilidad. Este tipo de arquitectura puede escalar mediante múltiples servidores con la ayuda de balanceadores de carga que repartan las peticiones entre los diferentes nodos de servidor.

Arquitectura Contenedor Ligero

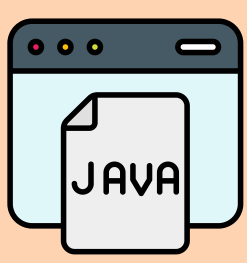
Por supuesto que las arquitecturas JavaEE pueden ser exitosas sin EJBs. Sin embargo, se puede utilizar algunas de las buenas ideas de los EJBs y llevarlas a una arquitectura sin EJBs. Para ello, se utiliza una arquitectura de Contenedor Ligero.
Del mismo modo que los EJBs, se plantea una arquitectura centrada a partir de una capa de servicios de negocio gestionados por el contenedor. Pero aquí se acaba la similitud. En vez de ejecutarse dentro de un contenedor EJB, los objetos de negocio corren dentro de un contenedor ligero.

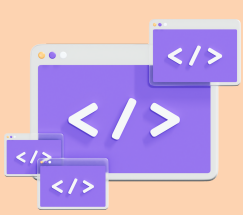
Un contenedor ligero no está atado a JavaEE, por lo que puede correr en un contenedor web, una aplicación "standalone", o incluso en un contenedor EJB (si fuese necesario). Además, tampoco esta atado al API de los Servlets, como los frameworks MVC, lo cual seria una pobre elección para gestionar los objetos de negocio.

Los contenedores ligeros tienen un coste de arranque insignificante y elimina el proceso de despliegue necesario en los EJBs. Los contenedores ligeros ofrecen un modo de gestionar y localizar los objetos de negocio; ya no necesitan búsquedas JNDI, localizadores de servicio o singletons, ya que ofrece un registro de los objetos de aplicación. Este enfoque es menos invasivo y más poderoso que los EJBs, siempre y cuando todo resida en la misma JVM.

Arquitectura JavaEE Local

Para solucionar algunos de los problemas en la arquitectura vista previamente, la siguiente solución modifica la arquitectura para incorporar los EJBs que ofrecen una vista local de sus interfaces a los clientes.
Por un lado, la especificación de servlets 2.3 garantiza que los objetos de la capa web puedan acceder a los EJBs vía sus interfaces locales si la aplicación se despliega en un servidor de aplicaciones JavaEE integrado en un única JVM.





Esto nos permite beneficiarnos de los servicios ofrecidos por el contenedor EJB, pero sin incurrir en la excesiva complejidad ni la necesidad de hacer nuestra aplicación distribuida.
Por el otro, la especificación de EJB 2.0, introdujo los interfaces locales como respuesta a la presión de la industria respecto al rendimiento de las arquitecturas EJB.

El uso de arquitecturas con EJBs locales ofrece las siguientes ventajas:

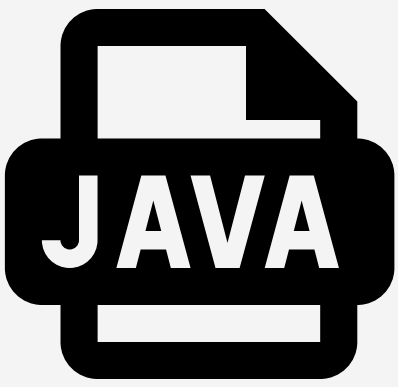
- Es menos compleja que una aplicación EJB distribuida
- El uso de los EJBs no altera el diseño básico de la aplicación. En este tipo de arquitectura, podemos hacer EJBs sólo aquellos objetos que necesitan servicios del contenedor EJB.
- Este tipo de uso de los EJBs implica una penalización de rendimiento muy pequeña (casi despreciable), ya que no realiza llamadas remotas ni serialización de los objetos.
- Ofrece los beneficios del contenedor EJB tales como la gestión de las transacciones, seguridad declarativa y gestión de la concurrencia.
- En el caso de quererlo, podemos utilizar beans de entidad

Arquitectura JavaEE Distribuida

Se trata de la arquitectura clásica JavaEE. Ofrece la capacidad de partir la capa de middleware de forma física y lógica, usando diferentes JVMs para los EJbs, de modo que los componentes web puedan utilizarlos. Se trata de una arquitectura compleja, con una sobrecarga de rendimiento significativa.

Ventajas
El uso de arquitecturas distribuidas ofrece las siguientes ventajas:

- Soporta todos los tipos de cliente JavaEE mediante una capa de middleware compartida.
- Permite la distribución de componentes de aplicación a través de los diferentes servidores físicos. Esto funciona particularmente bien si la capa de EJB es "sin estado". Las aplicaciones JavaEE con capa de presentación con estado (mediante el uso del objeto Session), pero con la capa de middleware sin estado (mediante Stateless Session Beans) se benefician de este tipo de despliegue y obtienen la máxima escalabilidad posible.



Limitaciones
Las limitaciones de esta arquitectura son:

- Se trata del enfoque más complejo de los que hemos considerado. Si los requisitos de la aplicación no obligan a esta complejidad, estaremos malgastando recursos a lo largo del ciclo de vida del proyecto, y sembrando un campo fértil para la aparición de errores.
- Afecta al rendimiento. Las llamadas a métodos remotos pueden ser cientos de veces más lentas que las llamadas locales por referencia. El efecto de esta sobrecarga de rendimiento depende del número de llamadas remotas necesarias.
- Las aplicaciones distribuidas son difíciles de probar y depurar.
- Todos los componentes de negocio deben ejecutarse en el contenedor EJB. Mientras esto ofrece un interfaz comprensible para los clientes remotos, es un aspecto problemático si el EJB no puede utilizar para resolver todos los problemas impuestos por los requisitos de negocio. Por ejemplo, si el patrón de diseño Singleton es una buena elección, será muy difícil de implementar satisfactoriamente mediante EJBs.
- El diseño OO se dificulta por el uso centralizado de RMI.
- La gestión de las excepciones es más compleja en sistemas distribuidos. Se deben controlar tanto los errores de transporte como los de aplicación.

Implementación
Aunque el diagrama muestre una aplicación web, esta arquitectura soporta cualquier tipo de cliente JavaEE. Por lo tanto, se trata de una arquitectura preparada para las necesidades de las aplicaciones cliente.
Esta arquitectura utiliza RMI entre las capas de presentación (o con otros cliente remotos) y los objetos de negocio, los cuales están expuestos como EJBs (los detalles de la comunicación RMI los abstrae el contenedor de EJBs, pero hemos de tratar con las implicaciones de su uso). Esto hace las invocaciones remotas un factor determinante de rendimiento y una consideración central a la hora de realizar el diseño. Como vimos anteriormente, hemos de minimizar el número de llamadas remotas (evitando las llamadas "chatty"). Además, todos los objetos enviados deben ser serializables, y debemos tratar con requisitos de gestión de errores más complejos.