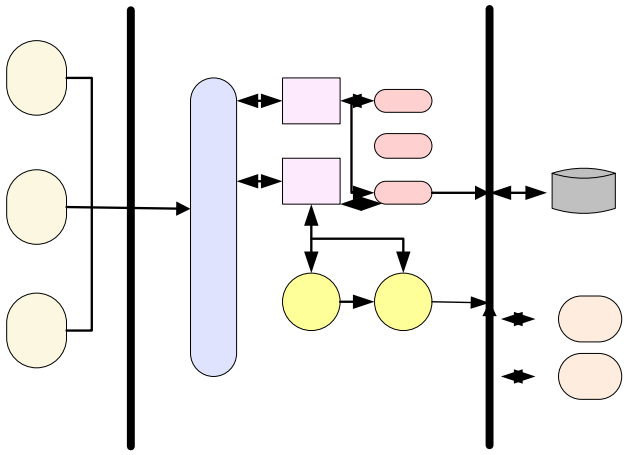
CAPITULO II

COMPONENTES J2EE

Este capítulo contiene los elementos de J2EE, con los cuales se construyen aplicaciones n-capas, se podrá tener una visión general de cómo utilizar estas herramientas, para la presentación con la utilización de páginas dinámicas, para desarrollar componentes con EJB, así también el acceso a base de datos son algunos de los elementos que se detallaran en este capitulo.



1. Servlet
2. JSP
3. EJB
4. JNDI
5. RMI
6. JDBC

2.1 SERVLET

2.1.1 Introducción

Los Servlets son módulos que extienden los servidores orientados a petición-respuesta, como los servidores Web compatibles con Java. Los Servlets son para los servidores lo que los ***applet***s son para los navegadores. Sin embargo, al contrario que los applets, los Servlets no tienen interfaz gráfico de usuario.

Los Servlets pueden ser incluidos en muchos servidores diferentes, porque el API Servlet el que se utiliza para escribir Servlets, no asume nada sobre el entorno o ***protocolo*** del servidor. Los Servlets se están utilizando ampliamente dentro de servidores HTTP; muchos servidores Web soportan el API Servlet. **[www010]**

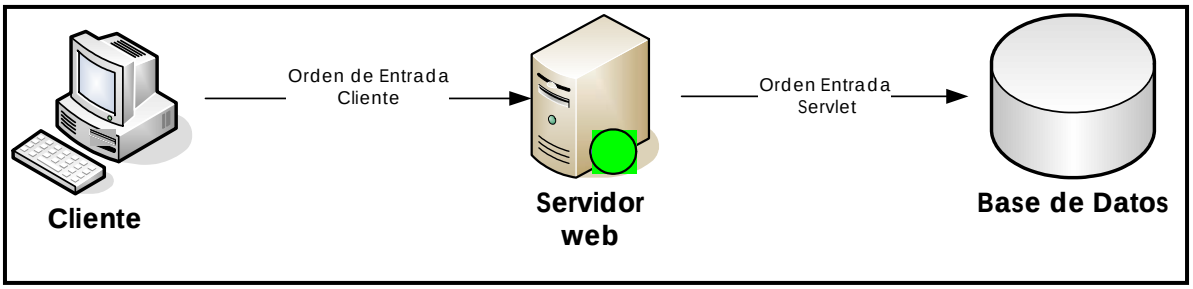


Figura 2.1: Arquitectura Web

2.1.2 Conociendo los Servlets

Qué es un Servlet

Los Servlets son clases de Java que amplían la funcionalidad de un servidor Web mediante la generación dinámica de páginas Web. Un entorno de ejecución denominado motor de Servlets administra la carga y descarga del Servlet, y trabaja con el servidor Web HTTP para dirigir las peticiones de los usuarios ***remotos*** (clientes) a los Servlets y enviar la respuesta a los clientes.

Entre las principales características de los Servlets tenemos:

- ✓ Son 100% puro Java, lo que los hacen multiplataforma
- ✓ Por ser un Lenguaje Orientado a Objetos y por lo tanto poseen todas las características que se derivan de esto.
- ✓ Puede utilizar todas las tecnologías de Java: clases de almacenamiento, hilos o ***threads***, acceso a bases de datos, flujos de E/S, RMI, acceso a la red, etc.
- ✓ Son mucho más rápidos ya que están precompilados
- ✓ La comunicación con otros Servlets es muy sencilla
- ✓ Mediante el manejo de excepciones los errores pueden manejarse sencillamente durante la ejecución del Servlet.

2.1.3 Arquitectura del Servlet

El principal componente Servlet API es la interfaz Servlet. Todos los Servlets implementan esta interfaz, por medio la de extensión de la clase que la implementa, HttpServlet. Esta interfaz está provista de métodos que manipulan a los Servlets y la comunicación con sus clientes. **[www010]**

El paquete **javax.servlet** proporciona ***clases*** e interfaces para escribir servlets. La arquitectura de este paquete se describe a continuación.

2.1.3.1 La Interfaz Servlet

La abstracción central en el API Servlet es la interfaz **Servlet**. Todos los Servlets implementan esta interfaz, bien directamente o, más comúnmente, extendiendo una clase que lo implemente como **HttpServlet**

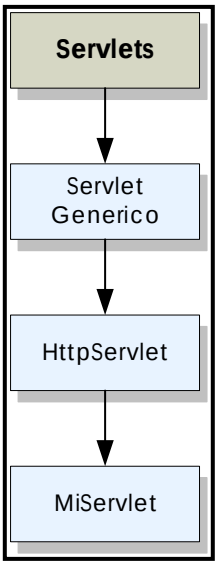


Figura 2.2: Arquitectura Servlet

La interfaz **Servlet** declara, pero no implementa, métodos que manejan el Servlet y su comunicación con los clientes. Los escritores de Servlets proporcionan algunos de esos métodos cuando desarrollan un servlet.

2.1.3.2 Interacción con el Cliente

Cuando un servlet acepta una llamada de un cliente, recibe dos objetos.

- Un ServletRequest, que encapsula la comunicación desde el cliente al servidor.
- Un ServletResponse, que encapsula la comunicación de vuelta desde el servlet hacia el cliente.

ServletRequest y **ServletResponse** son interfaces definidos en el paquete **javax.servlet**.

La Interfaz ServletRequest

La Interfaz **ServletRequest** permite al servlet acceder a:

- ✓ Información como los nombres de los parámetros pasados por el cliente, el protocolo que está siendo utilizado por el cliente, y los nombres del host remote que ha realizado la petición y la del Servidor que la ha recibido.
- ✓ El stream de entrada, ServletInputStream. Los Servlets utilizan este stream para obtener los datos desde los clientes que utilizan protocolos como los métodos POST y PUT del HTTP.

Las interfaces que extienden el interfaz **ServletRequest** permiten al servlet recibir más datos específicos del protocolo. Por ejemplo, la interfaz **HttpServletRequest** contiene métodos para acceder a información de cabecera específica HTTP.

La Interfaz ServletResponse

La Interfaz **ServletResponse** le da al servlet los métodos para responder al cliente.

- Permite al servlet seleccionar la longitud del contenido y el tipo ***MIME*** de la respuesta.
- Proporciona un stream de salida, ServletOutputStream, y un **Writer** a través del cual el servlet puede responder datos.

Las interfaces que extienden la interfaz **ServletResponse** le dan a los servlets más capacidades específicas del protocolo. Por ejemplo, la interfaz **HttpServletResponse** contiene métodos que permiten al servlet manipular información de cabecera específica HTTP.

2.1.4 El Ciclo de Vida de un Servlet

Cada servlet tiene el mismo ciclo de vida. **[www010]**

- ✓ Un servidor carga e inicializa el servlet.
- ✓ El servlet maneja cero o más peticiones de cliente.
- ✓ El servidor elimina el servlet. (Algunos servidores sólo cumplen este paso cuando se desconectan).

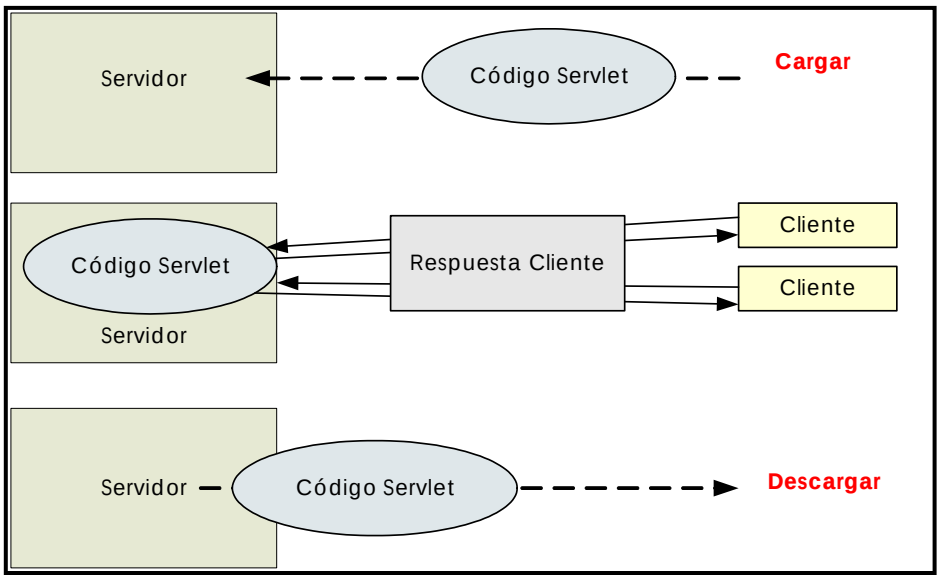


Figura 2.3: Ciclo de Vida de los Servlet

Inicializar un Servlet

Cuando un servidor carga un servlet, ejecuta el método **init** del servlet. La inicialización se completa antes de manejar peticiones de clientes y antes de que el servlet sea destruido.

Aunque muchos servlets se ejecutan en servidores multicapas, los servlets no tienen problemas de concurrencia durante su inicialización. El servidor llama sólo una vez al método **init**, cuando carga el servlet, y no lo llamará de nuevo a menos que vuelva a

ecargar el servlet. El servidor no puede recargar un servlet sin primero haber destruido el servlet llamando al método **destroy**.

Interactuar con Clientes

Después de la inicialización, el servlet puede manejar peticiones de clientes. Esta parte del ciclo de vida de un servlet.

Destruir un Servlet

Los servlets se ejecutan hasta que el servidor los destruye, por ejemplo, a petición del administrador del sistema. Cuando un servidor destruye un servlet, ejecuta el método **destroy** del propio servlet. Este método sólo se ejecuta una vez. El servidor no ejecutará de nuevo el servlet, hasta haberlo cargado e inicializado de nuevo. **[www010]**

2.1.5 Aplicaciones Servlets

Con los Servlets se puede implementar por ejemplo:

Típicos sistemas middleware para consultar las bases de datos, los Servlets pueden utilizar JDBC (Java Data Base Connection), lo que les permite extraer información de cualquier sistema de Base de Datos.

Automatización de un sistema de recepción y publicación de información. Por ejemplo podríamos montarnos una simple estación meteorológica que permita el acceso a su información mediante una página WEB. Por un lado tendríamos un servlet que recolectaría la información de los diversos tipos de sensores y la almacenaría en bases de datos, y por otro lado, un servlet que se encargaría de presentar esta información en función de las peticiones del cliente basándose en estas mismas bases de datos.

Control de la recepción de correo electrónico, y de sistemas de noticias, chats, etc. Conviene recordar que Java está especialmente indicado para la programación utilizando los protocolos ***TCP/IP***.

Dado que pueden manejar múltiples peticiones en forma concurrente, es posible implementar aplicaciones de colaboración, como por ejemplo una aplicación de videoconferencia.

2.2 PAGINA JAVA DE SERVIDOR (JSP)

2.2.1 Que es JSP

Un JSP (JavaServer Pages) es una página Java en Servidor y es una ***plantilla*** para una página Web que emplea código Java para generar un documento HTML dinámicamente. **[www009]**

Las JSP se ejecutan en un componente del servidor denominado ***contenedor de JSP***, y este las traduce o convierte a servlets de Java equivalentes y por lo tanto, lo que se puede hacer con un JSP, también se puede hacer con un servlet, así que tienen los JSP tienen las ventajas de un servlet.

Entorno de Software

Para ejecutar las páginas JSP, necesitamos un servidor Web con un ***contenedor Web*** que cumpla con las especificaciones de JSP y de Servlet.

El contenedor Web se ejecuta en el servidor Web y maneja la ejecución de todas las páginas JSP y de los Servlets que se ejecutan en ese servidor Web. Tomcat es una completa implementación de referencia para las especificaciones Java Servlet 2.2 y JSP

2.2.2 TECNOLOGÍA JSP

En la figura 2.4 se detalla el funcionamiento de las páginas JSP, desde que el usuario realiza la petición hasta que se le entrega.

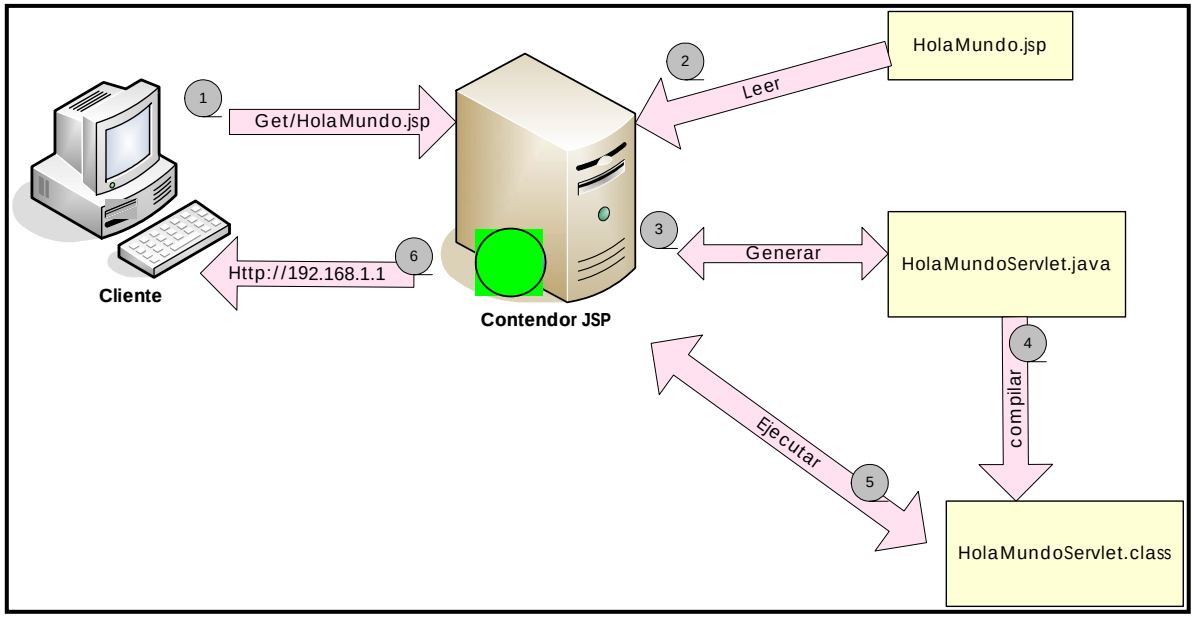


Figura 2.4: Funcionamiento de JSP

Como se puede ver en la figura 2.4, la página JSP se convierte en un servlet, esta conversión la realiza en la máquina servidora, el motor o contenedor JSP, la primera vez que se solicita la página JSP; este Servlet generado procesa cualquier petición para esa página JSP, y, si se modifica el código de la página JSP, entonces se regenera y recompila automáticamente el servlet y se recarga la próxima vez que sea solicitada

Un JSP atraviesa por etapas de evolución de tres pasos, en su código:

1. **Código fuente JSP.** Es escrito por el programador o desarrollador de JSP. Esta en un archivo de texto con extensión **.jsp** y se compone de una mezcla de código HTML, instrucciones en lenguaje Java, directivas JSP y acciones que describen cómo generar una pagina Web para responder a una solicitud por parte del cliente.

2. **Código fuente Java.** El contenedor de JSP traduce el código fuente JSP a código fuente de un servlet Java equivalente. Este código fuente se guarda en un área de trabajo y puede ser útil en el proceso de depuración de errores.

3. **Clase Java compilada.** Como sucede con cualquier otro programa de Java, el servlet generado se compila en bytecode (código de bytes) resultando en un archivo **.class** que esta listo para ser cargado y ejecutado por el servidor.

2.2.3 Sintaxis y Estructura JSP

Una página JSP es básicamente una página Web con HTML tradicional y código Java. La extensión de fichero de una página JSP es ".jsp" en vez de ".html" o ".htm", y eso le dice al servidor que esta página requiere un manejo especial que se conseguirá con una extensión del servidor o un ***plugin***. **[www009]**

Elementos de una página JSP

Como se explico anteriormente las páginas JSP contiene:

1. Código HTML
2. Código JAVA

Además de código HTML la página JSP puede incluir marcadores que se agrupan en tres tipos de elementos:

- ✓ Elementos de ***Scripting*** (guiones)._ Permiten insertar código Java en la página JSP que incluyen expresiones, scriplets y declaraciones.
- ✓ Directivas._ afectan a toda la estructura del servlet generado
- ✓ Acciones._ Afectan al comportamiento en tiempo de ejecución del

2.2.3.1 Scripting

Declaraciones.

Una declaración permite notificar al intérprete de Java que se van a definir nuevas variables o métodos en el archivo de la clase generada. Las declaraciones contienen instrucciones o sentencias en lenguaje Java con la sintaxis siguiente:

<%! sentencia; [sentencias; ...] %>

Las secciones de la declaración se pueden usar para declarar clases o variables de instancia, métodos o clases internas. Para declarar tanto una variable como un método se utiliza el símbolo "!". Las declaraciones son creadas e inicializadas cuando el usuario accede a la página JSP y su ámbito es de tipo "class", es decir que están disponibles en toda la clase que genera después de la solicitar una página JSP. Cualquier declaración hecha al comienzo de una página puede ser utilizada al final de la misma.

Expresiones.

Una expresión en una página JSP es un pequeño fragmento de código (scriptlet) que devuelve un resultado (salida) por pantalla. Su sintaxis es:

<%= expr %>

Donde **expr** es cualquier expresión de Java válida. La expresión puede tomar cualquier valor como un dato, mientras éste se pueda convertir en una cadena. Esta conversión

se efectúa generalmente con una instrucción out.print() o puede ser evaluada como un java.util.String.

Scriptlets.

Un scriptlet es un conjunto de instrucciones o sentencias de Java incluidas en una página HTML. Estas instrucciones se distinguen del HTML porque están colocadas entre los marcadores <% y %> para que el intérprete de JSP sepa que debe procesar todo el código que se encuentre dentro de esas etiquetas. Su sintaxis es:

<% sentencias; [sentencias; ...] %>

2.2.3.2 Directivas JSP

Se utilizadas para definir y manipular una serie de atributos dependientes de la página que afectan a todo el JSP.

Las directivas existentes son las siguientes:

- ✓ Page
- ✓ Include
- ✓ Taglib

Son etiquetas que se utilizan en una página JSP cuya principal característica es incluir el símbolo @ en su sintaxis, que tiene la siguiente forma:

<%@ nombre_directiva [atributo_i = "valor_i"] %>

Directiva Page.

Se utiliza para definir atributos globales que deben ser aplicados a la página JSP completa, y a cualquier archivo, excepto los de contenido dinámico, que se haya incluido en ésa página con la directiva include o la acción jsp:include. **[www009]**

Directiva include.

Permite la inclusión en una página JSP en tiempo de ***compilación***, de una página HTML, archivo de código Java, archivo de texto u otra página JSP. La página final que va a procesar el motor JSP esta conformada por la página base más el contenido del recurso que se haya incluido. Esta directiva puede ir situada en cualquier lugar de la página JSP. Al usar esta directiva se recomienda especial cuidado en la colocación de las etiquetas HTML <html></html>, <body></body>, etc, porque el archivo incluido podría entrar en conflicto con las etiquetas similares del archivo JSP original, por lo que hay que chequear que estén correctamente anidadas y no duplicadas.

OBJETOS IMPLÍCITOS

JSP utiliza los objetos implícitos, basados en la API de servlets.

Estos objetos están disponibles para su uso en páginas JSP y son los siguientes:

Objeto request.- Representa la petición lanzada en la invocación de service(). Proporciona entre otras cosas los parámetros recibidos del cliente, el tipo de petición (GET/POST)

Objeto response.- – Instancia de HttpServletResponse que representa la respuesta del servidor a la petición. Ámbito de página

Objeto sesión.

Es una instancia de la clase *javax.servlet.http.HttpSesion*. El cometido de este objeto es manejar todas las acciones relacionadas a la sesión del usuario, por lo tanto su ámbito de utilización es de tipo sesión. Una sesión es creada automáticamente cuando un usuario solicita una página JSP, y de esta manera podemos almacenar información relativa a ese usuario.

```
<% HttpSession  unaSesion = request.getSession();
unaSesion.setAttribute(usuario,"pancho_lopez");
%>
```

2.2.4 Comparación entre JSP vs Servlets

- ✓ Tienen un mejor desempeño y capacidad de adaptación, debido a que se conservan en la memoria y manejan múltiples subprocesos.
- ✓ No se requiere una configuración especial por parte del cliente.
- ✓ Soportan sesiones HTTP, lo que hace posible la programación de aplicaciones.
- ✓ Pueden acceder a la tecnología disponible en Java para manejar hilos o threads, sockets o trabajo en red, conectividad con bases de datos y todo esto sin las limitaciones de los applets del cliente.
- ✓ Se compilan automáticamente cuando sea necesario
- ✓ Su ubicación en el espacio común de documentos del servidor Web permiten ubicarlas más fácilmente que a los servlets
- ✓ Son similares a las de HTML, por lo tanto son mas compatibles con las herramientas de desarrollo de Web (DreamWeaver , FrontPage, etc.)

2.3 COMPONENTES JAVA EMPRESARIALES (EJB)

2.3.1 Introducción

El desarrollo basado en componentes promete un paso más en el camino de la programación orientada a objetos. Con la programación orientada a objetos puedes reutilizar clases, pero con componentes es posible reutilizar a mayor nivel de funcionalidades e incluso es posible modificar estas funcionalidades y adaptarlas a cada entorno de trabajo particular sin tocar el código del componente desarrollado.

Un componente es como un objeto tradicional que tiene un conjunto de servicios adicionales soportados en tiempo de ejecución por el contenedor de componentes. El contenedor de componentes se denomina ***contenedor EJB*** y es algo así como el sistema operativo en el que éstos residen.

Con la tecnología J2EE Enterprise JavaBeans es posible desarrollar componentes (enterprise beans) que luego puedes reutilizar y ensamblar en distintas aplicaciones.

2.3.2 Que son Componentes Java Empresariales

La ***arquitectura*** Enterprise JavaBeans es una arquitectura de componentes para el desarrollo y despliegue de aplicaciones de empresa distribuidas y ***orientadas a objetos***. Las aplicaciones escritas usando la arquitectura Enterprise JavaBeans son ***escalables***, transaccionales y seguras para ***multiusuarios***. Estas aplicaciones pueden escribirse una

vez, y luego desplegarse en cualquier servidor que soporte la especificación Enterprise JavaBeans. **[www018]**

Por ejemplo, podrías desarrollar un bean Cliente que represente un cliente en una base de datos. Podrías usar después ese bean Cliente en un programa de contabilidad o en una aplicación de comercio electrónico o virtualmente en cualquier programa en el que se necesite representar un cliente. De hecho, incluso sería posible que el desarrollador del bean y el ensamblador de la aplicación no fueran la misma persona, o ni siquiera trabajaran en la misma empresa.

2.3.3 Servicios proporcionados por el contenedor EJB

Los servicios más importantes del contenedor EJB son los siguientes:

- ✓ Manejo de transacciones: apertura y cierre de transacciones asociadas a las llamadas a los métodos del bean.
- ✓ Seguridad: comprobación de permisos de acceso a los métodos del bean.
- ✓ Concurrencia: llamada simultánea a un mismo bean desde múltiples clientes.
- ✓ Servicios de red: comunicación entre el cliente y el bean en máquinas distintas.
- ✓ Gestión de recursos: gestión automática de múltiples recursos, como colas de mensajes, bases de datos o fuentes de datos en aplicaciones heredadas que no han sido traducidas a nuevos ***lenguajes***/entornos y siguen usándose en la empresa.
- ✓ Persistencia: sincronización entre los datos del bean y tablas de una base de datos.
- ✓ Gestión de mensajes: manejo de Java Message Service (JMS).

- ✓ Escalabilidad: posibilidad de constituir ***clusters*** de servidores de aplicaciones con múltiples ***hosts*** para poder dar respuesta a aumentos repentinos de carga de la aplicación con sólo añadir hosts adicionales.
- ✓ Adaptación en tiempo de despliegue: posibilidad de modificación de todas estas características en el momento del despliegue del bean.

2.3.4 Funcionamiento de los componentes EJB

El funcionamiento de los componentes EJB se basa fundamentalmente en el trabajo del contenedor EJB. El contenedor EJB es un programa Java que corre en el servidor y que contiene todas las clases y objetos necesarios para el correcto funcionamiento de los Enterprise Beans.

En la siguiente figura se puede ver una representación de muy alto nivel del funcionamiento básico de los Enterprise Beans. En primer lugar, se ve que el cliente que realiza peticiones al Bean y el servidor que contiene al Bean están ejecutándose en máquinas virtuales Java distintas. Incluso pueden estar en distintos hosts. Otra cosa a resaltar es que el cliente nunca se comunica directamente con el Enterprise Bean, sino que el contenedor EJB proporciona un EJBObject que hace de interfaz. Cualquier petición del cliente (una llamada a un método de negocio del Enterprise Bean) se debe hacer a través del objeto EJB, el cual solicita al contenedor EJB una serie de servicios y se comunica con el Enterprise Bean. Por último, el Bean realiza las peticiones correspondientes a la base de datos.

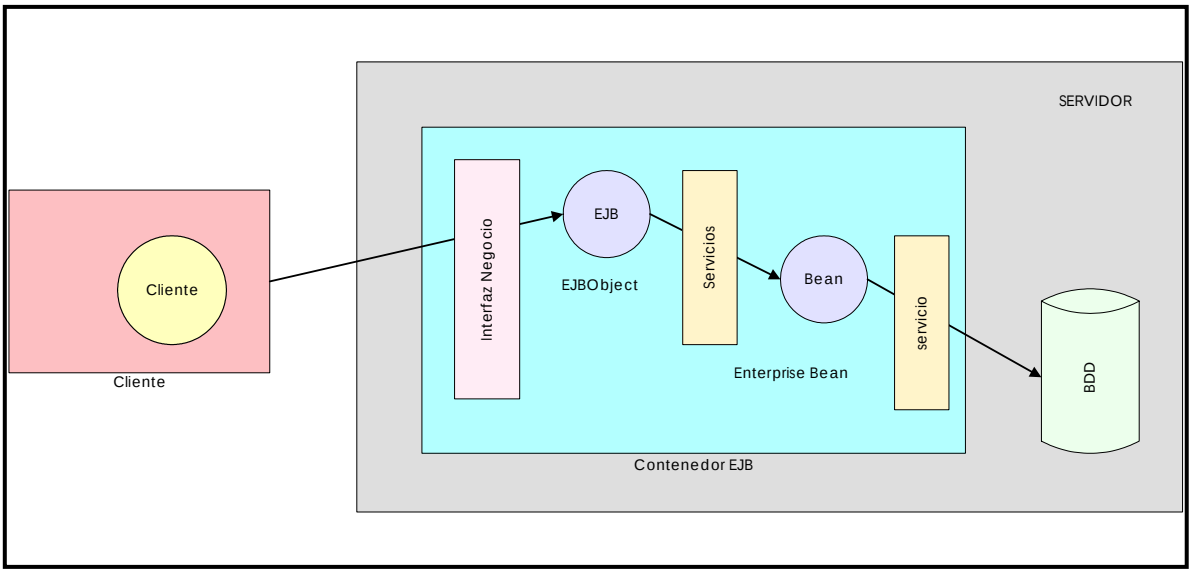


Figura 2.5: Representación de alto nivel del funcionamiento de los Enterprise Beans.

Vamos a ver un ejemplo para que puedas entender mejor el flujo de llamadas. Supongamos que tenemos una aplicación de bolsa y el Bean proporciona una implementación de un Broker. La interfaz de negocio del Broker está compuesta de varios métodos, entre ellos, por ejemplo, los métodos compra o venta. Supongamos que desde el objeto cliente queremos llamar al método compra. Esto va a provocar la siguiente secuencia de llamadas:

1. Cliente: "Necesito realizar una petición de compra al Bean Broker."
2. EJBObject: "Espera un momento, necesito comprobar tus permisos."
3. Contenedor EJB: "Sí, el cliente tiene permisos suficientes para llamar al método compra."
4. Contenedor EJB: "Necesito un Bean Broker para realizar una operación de compra. Y no olvides comenzar la transacción en el momento de instanciarnos."
5. Pila de Beans: "A ver... ¿a quién de nosotros le toca esta vez?".
6. Contenedor EJB: "Ya tengo un Bean Broker. Pásale la petición del cliente."

2.3.5 Tipos de Beans

La tecnología EJB define tres tipos de Beans: Beans de sesión, Beans de entidad y Beans dirigidos por mensajes. **[www018]**

Los Beans de entidad representan un objeto concreto que tiene existencia en alguna base de datos de la empresa. Una instancia de un Bean de entidad representa una fila en una tabla de la base de datos. Por ejemplo, podríamos considerar el Bean Cliente, con una instancia del Bean siendo Eva Martínez (ID# 342) y otra instancia Francisco Gómez (ID# 120).

Los Beans dirigidos por mensajes pueden escuchar mensajes de un servicio de mensajes JMS. Los clientes de estos Beans nunca los llaman directamente, sino que es necesario enviar un mensaje JMS para comunicarse con ellos. Los Beans dirigidos por mensajes no necesitan objetos EJBObject porque los clientes no se comunican nunca con ellos directamente. Un ejemplo de Bean dirigido por mensajes podría ser un Bean EscucharNuevoCliente que se activara cada vez que se envía un mensaje comunicando que se ha dado de alta a un nuevo cliente.

Un Bean de sesión representa un proceso o una acción de negocio. Normalmente, cualquier llamada a un servicio del servidor debería comenzar con una llamada a un Bean de sesión. Mientras que un Bean de entidad representa una cosa que se puede representar con un nombre, al pensar en un Bean de sesión se debería pensar en un verbo. Ejemplos de Beans de sesión podrían ser un carrito de la compra de una aplicación de negocio electrónico o un sistema verificador de tarjetas de crédito.

2.3.6 Arquitectura

La arquitectura EJB define seis papeles principales. Brevemente, son:

- ✓ **Desarrollador de Beans**: desarrolla los componentes Enterprise Beans.
- ✓ **Ensamblador de aplicaciones**: compone los Enterprise Beans y las aplicaciones cliente para conformar una aplicación completa
- ✓ **Desplegador**: despliega la aplicación en un entorno operacional particular (servidor de aplicaciones)
- ✓ **Administrador del sistema**: configura y administra la infraestructura de computación y de red del negocio
- ✓ **Proporcionador del Contenedor EJB y Proporcionador del Servidor EJB**: un fabricante (o fabricantes) especializado en manejo de transacciones y de aplicaciones y otros servicios de bajo nivel. Desarrollan el servidor de aplicaciones.

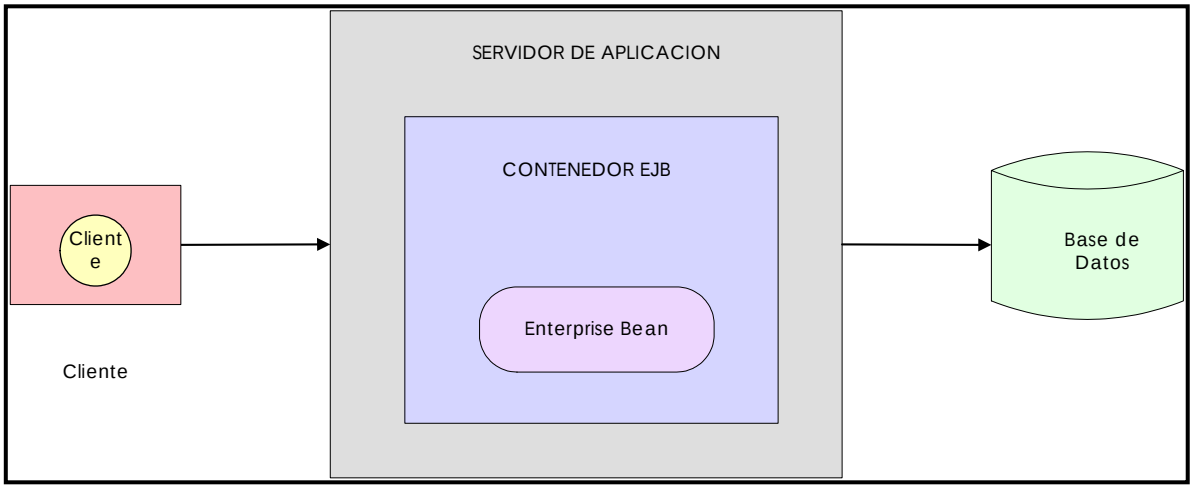


Figura 2.6: Arquitectura Enterprise Java Bean

2.4 INVOCACIÓN DE MÉTODOS REMOTOS (RMI)

2.4.1 Introducción

Las aplicaciones RMI normalmente comprenden dos programas separados: un servidor y un cliente. Una aplicación servidor típica crea un montón de objetos remotos, hace accesibles unas referencias a dichos objetos remotos, y espera a que los clientes llamen a estos métodos u objetos remotos. Una aplicación cliente típica obtiene una referencia remota de uno o más objetos remotos en el servidor y llama a sus métodos. RMI proporciona el mecanismo por el que se comunican y se pasan información del cliente al servidor y viceversa. Cuando es una aplicación algunas veces nos referimos a ella como Aplicación de Objetos Distribuidos.

2.4.2 Que es RMI

RMI significa sistema de Invocación Remota de Métodos (RMI) de Java, permite a un objeto que se está ejecutando en una Máquina Virtual Java (***JVM***) llamar a métodos de otro objeto que está en otra VM diferente. **[www015]**

RMI proporciona comunicación remota entre programas escritos en Java. Si unos de nuestros programas están escritos en otro lenguaje, deberemos considerar la utilización de ***IDL*** en su lugar.

2.4.3 Arquitectura Java RMI-IIOP

La interfaz Remota, objetos Remotos, stubs y skeletons

RMI-IIOP hace uso extensivo del principio de orientación a objetos que consiste en la separación entre la interfaz y la implementación de los objetos.

La interfaz define la información que expone el objeto, los nombres de los métodos y los parámetros de entrada. Es lo que necesita el cliente. La interfaz enmascara la implementación del objeto a los clientes.

La implementación es la lógica del objeto.

La separación entre interfaz e implementación permite modificar la lógica del objeto sin modificar los clientes que lo usan. En RMI-IIOP, es necesario escribir la interfaz remota.

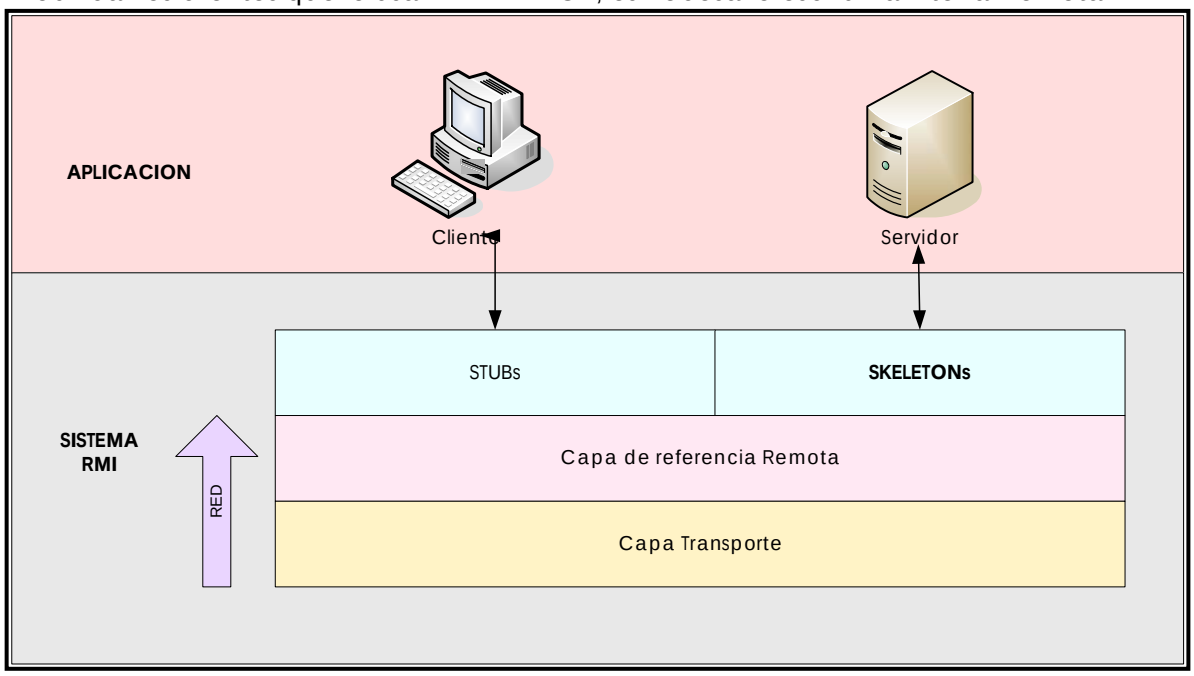


Figura 2.7: Arquitectura RMI

Los objetos remotos son objetos que pueden ser invocados por objetos que residen en otra JVM. Implementan la interfaz remota.

La ubicación física de los objetos remotos y la de los clientes que los invocan, no es importante. Pueden residir en el mismo equipo o estar distribuidos en Internet.

Transparencia local/remota es poder invocar a un método de un objeto remoto de la misma manera que si fuese un objeto Java cualquiera. RMI-IIOP provee el mecanismo para ***enmascarar*** si el objeto que se está invocando es local o remoto.

El Stub es un objeto local (reside en el cliente), acepta invocaciones locales y las delega a la implementación real del objeto (ubicada en cualquier lugar de la red).
De esta manera, las invocaciones remotas aparentan ser locales. El stub se ocupa de todos los aspectos de conectividad de la invocación.

El Skeleton es un objeto local para el objeto remoto (reside en el servidor). De la misma manera que el stub oculta los aspectos de conectividad de la invocación al cliente, el skeleton lo hace al objeto remoto. Recibe invocaciones a través de la red, del stub, y las delega a la implementación del objeto remoto.
Serialización de objetos consiste en convertir un objeto en un stream de datos que puede enviarse a cualquier lugar. Para reconstituir el objeto se deserealiza el stream de datos y se convierte en un objeto. Para indicar que un objeto es serializable. **[www016]**

2.4.4 Elementos para el Funcionamiento de RMI

2.4.4.1 Localizar Objetos Remotos

Las aplicaciones pueden utilizar uno de los dos mecanismos para obtener referencias a objetos remotos. Puede registrar sus objetos remotos con la facilidad de nombrado de RMI rmiregistry. O puede pasar y devolver referencias de objetos remotos como parte de su operación normal.

2.4.4.2 Comunicar con Objetos Remotos

Los detalles de la comunicación entre objetos remotos son manejados por el RMI; para el programador, la comunicación remota se parecerá a una llamada estándar a un método Java.

Como RMI permite al llamador pasar objetos Java a objetos remotos, RMI proporciona el mecanismo necesario para cargar el código del objeto, así como la transmisión de sus datos.

La siguiente ilustración muestra una aplicación RMI distribuida que utiliza el registro para obtener referencias a objetos remotos. El servidor llama al registro para asociar un nombre con un objeto remoto. El cliente busca el objeto remoto por su nombre en el registro del servidor y luego llama a un método. Esta ilustración también muestra que el sistema RMI utiliza una servidor Web existente para cargar los bytecodes de la clase Java, desde el servidor al cliente y desde el cliente al servidor, para los objetos que necesita.

El sistema RMI utiliza un servidor Web para cargar los bytecodes de la clase Java, desde el servidor al cliente y desde el cliente al servidor.

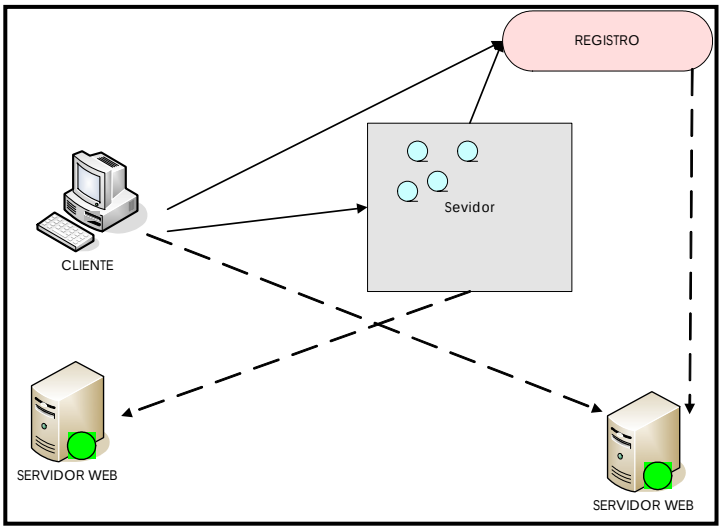


Figura 2.8: Requisitos para funcionamiento RMI

2.5 INTERFAZ DE NOMBRAMIENTO DIRECTORIO JAVA (JNDI)

2.5.1 Introducción

El directorio de servicios juega un papel vital en los ***Intranet*** e Internet proporcionando el acceso a una variedad de información sobre los usuarios, máquinas, redes, servicios, y aplicaciones.

El ambiente de la informática de una empresa consiste típicamente en varios medios de la denominación que representan a menudo partes diferentes de un ***namespace*** compuesto. Por ejemplo: el Internet. Sistemas de Nombre de Dominio (***DNS***) podría usarse como la facilidad de la denominación cima-nivelada por las organizaciones diferentes dentro de una empresa. Las organizaciones podrían usar un servicio de directorio como ***LDAP*** o ***NDS*** o ***NIS***. Desde la perspectiva de un usuario, hay un namespace que consiste en nombres compuestos. ***URLs*** son ejemplos de nombres compuestos porque ellos miden por palmos namespaces de medios de la denominación múltiples. Aplicaciones que usan los servicios del directorio deben apoyar esta perspectiva del usuario. **[www013]**

2.5.2 Que es JNDI

JNDI ("Java Naming Directory Interface") es una especificación que permite localizar información en distintos directorios distribuidos (como NDS de Novell), directorios LDAP (como OpenLDAP) o servicios CORBA "(***COS***)Corba Object Service".Debido a la importancia que tienen los servicios antes mencionados en sistemas empresariales, JNDI es una herramienta que cobrará gran importancia en este tipo de desarrollos.

El Java Naming and Directory Interface (JNDI) es una interfaz de programación (API) que proporciona funcionalidades de nombrado y directorio a las aplicaciones escritas usando

Java. Está definido para ser independiente de cualquier implementación de servicio de directorio. Así se puede acceder a una gran variedad de directorios,-- nuevos, emergentes, y ya desarrollados de una forma común.

Servicio de Nombres y de Directorio

Un servicio de nombres realiza dos tareas fundamentales:

- ✓ Asociar nombres con objetos
- ✓ Permitir la búsqueda de un objeto por su nombre

Un servicio de directorios es un servicio de nombres extendido que permite operar con objetos de tipo directorio. Los objetos directorio tienen atributos asociados.
Un directorio es un sistema de objetos directorio conectados, usualmente estructurado en forma jerárquica (un árbol).

En el mercado existen diferentes servicios de directorios:

- ✓ LDAP Lightweight Directory Access Protocol
- ✓ NDS Novell Directory Service
- ✓ NIS Network Information Service

Cada servicio tiene una forma de acceso diferente, lo que implica que si se cambia de fabricante hay que reescribir los clientes.

Es una API Java que permite escribir clientes que interactúan con sistemas de servicio de nombres y directorio.

Provee una interfaz común para acceder a diferentes servicios de directorio.

Los clientes Java que necesitan acceder a un directorio LDAP usan la misma

API que para acceder a un directorio NIS o NDS.

2.5.3 Arquitectura

La arquitectura JNDI consiste en un API y un "service provider interface (SPI)". Las aplicaciones Java usan el API JNDI para acceder a una gran variedad de servicios de nombres y directorios. El SPI permite conectar de forma transparente una gran variedad de servicios de nombres y directorios, por lo tanto permite a las aplicaciones Java usar el API JNDI para acceder a sus servicios. **[www013]**

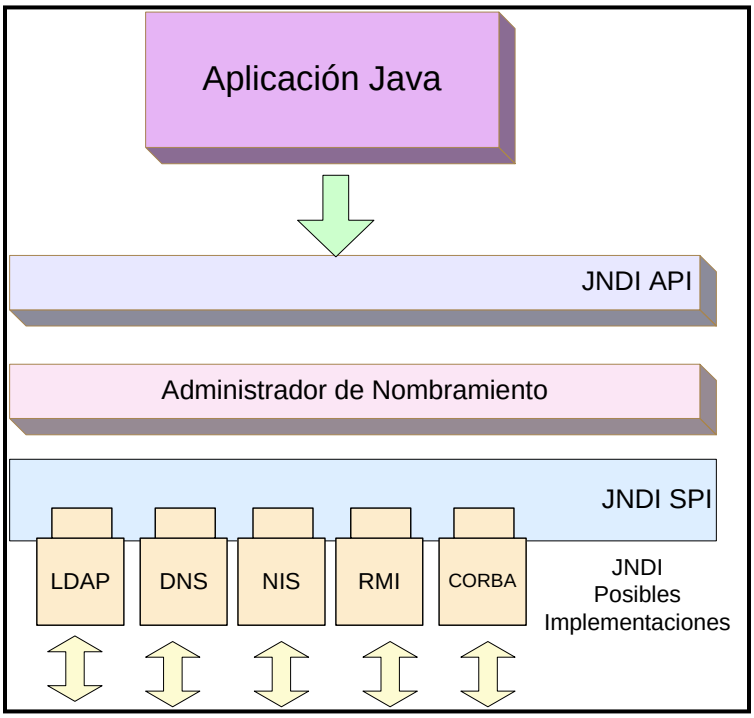


Figura 2.9: Arquitectura JNDI

En la figura 2.9, describe la arquitectura de JNDI, a continuación se describen los elementos de la arquitectura JNDI necesarios para cuando una aplicación Java realiza una petición:

JNDI

En la sección de LDAP se mencionó el diseño de un cliente LDAP el cual es utilizado para autorizar usuarios en un sistema de cómputo. La principal deficiencia de este cliente LDAP es la inhabilidad de realizar búsquedas o autorización en otros sistemas que puedan contener información de usuarios o contraseñas (Como NDS de Novell o ADS).

En la mayoría de las empresas este suele ser el caso: la existencia de diversos depósitos de información sin ninguna uniformidad para accesarlos, obviamente el escribir diversos clientes para realizar búsquedas en los distintos sistemas empresariales no seria imposible pero si una labor ardua. Debido a esto surgió JNDI.

JNDI y SPI

Mediante JNDI se definen varios estándares para realizar búsquedas en diversos sistemas de información como LDAP Servers, NDS, COS. Observe el diagrama:

Los diversos vendedores de directorios distribuidos y "LDAP Servers" deben definir un SPI ("Service Provider Interface") para su producto, este SPI ofrecerá las funcionalidades del producto vía un ambiente en Java, esto es conocido como ganchos ("Hooks") del producto. (Varios vendedores ya han definido sus SPI SPI's de Java)

Mediante el API de JNDI (el cual se encuentra en los diversos ***JDK's***) es posible escribir cualquier tipo de programa para acceder información en directorios distribuidos o "LDAP

Servers", con la garantía de utilizar solo ciertas funciones para acceder a diversos directorios distribuidos o "LDAP Servers".

Administrador de Nombramiento

Esta escrito en programa en "Java" y que éste busque información de cualquier tipo en un directorio distribuido o "LDAP" debe indicarse dentro del programa la ubicación del directorio "LDAP Server", esto es, el "LDAP Server" o "NDS Novell se encuentra en el nodo Es mediante el "Naming Manager" que se logra insular la ubicación física del sistema en cuestión.

Lo anterior es de suma importancia al desarrollar aplicaciones que utilicen RMI/CORBA ya que les permite cambiar de "servidor físico" sin la necesidad de modificar el código fuente de la aplicación. **[www013]**

2.6 CONECTOR DE BASE DE DATOS JAVA (JDBC)

2.6.1 Introducción

Java Database Connectivity (JDBC) es una interfaz de acceso a bases de datos estándar SQL que proporciona un acceso uniforme a una gran variedad de bases de datos relacionales. JDBC también proporciona una base común para la construcción de herramientas y utilidades de alto nivel.

El paquete actual de JDK incluye JDBC y el puente JDBC-***ODBC***. Estos paquetes son para su uso con JDK 1.0

Para usar JDBC con un sistema gestor de base de datos en particular, es necesario disponer del controlador JDBC apropiado que haga de intermediario entre ésta y JDBC. Dependiendo de varios factores, este controlador puede estar escrito en Java puro, o ser una mezcla de Java y métodos nativos ***JNI*** (Java Native Interface).

2.6.2 Que es JDBC

JDBC es el API para la ejecución de sentencias SQL. (Como punto de interés JDBC es una marca registrada y no un acrónimo, no obstante a menudo es conocido como “Java Database Connectivity”). Consiste en un conjunto de clases e interfaces escritas en el lenguaje de programación Java. JDBC suministra un API estándar para los desarrolladores y hace posible escribir aplicaciones de base de datos usando un API puro Java.

Usando JDBC es fácil enviar sentencias SQL virtualmente a cualquier sistema de base de datos. En otras palabras, con el API JDBC, no es necesario escribir un programa que acceda a una base de datos Sybase, otro para acceder a Oracle y otro para acceder a ***Informix***. Un único programa escrito usando el API JDBC y el programa será capaz de enviar sentencias SQL a la base de datos apropiada. Y, con una aplicación escrita en el lenguaje de programación Java, tampoco es necesario escribir diferentes aplicaciones para ejecutar en diferentes plataformas. La combinación de Java y JDBC permite al programador escribir una sola vez y ejecutarlo en cualquier entorno. **[www019]**

Java, siendo robusto, seguro, fácil de usar, fácil de entender, y descargable automáticamente desde la red, es un lenguaje base excelente para aplicaciones de base de datos.

JDBC expande las posibilidades de Java. Por ejemplo, con Java y JDBC API, es posible publicar una página Web que contenga un applet que usa información obtenida de una base de datos remota. O una empresa puede usar JDBC para conectar a todos sus empleados (incluso si usan un conglomerado de máquinas ***Windows***, Macintosh y ***UNIX***) a una base de datos interna vía intranet. Con cada vez más y más programadores desarrollando en lenguaje Java, la necesidad de acceso fácil a base de datos desde Java continúa creciendo.

2.6.3 Tipos de Manejadores

2.6.3.1 Puente entre JDBC-ODBC

ODBC es un API estándar semejante a JDBC, que permite que lenguajes como C++ accedan de un modo estándar a distintos sistemas de BD. Un manejador tipo puente

JDBC-ODBC delega todo el trabajo sobre un manejador ODBC, que es quien realmente se comunica con la base de datos. El puente JDBC-ODBC permite la conexión desde Java a BD que no proveen manejadores JDBC. Fue muy útil cuando se creó el API JDBC, ya que muchas BD no disponían de manejadores JDBC, pero si de manejadores ODBC. Empleado el puente JDBC-ODBC podía accederse a estas bases de datos empleando el API JDBC . Este tipo de manejador tiene dos desventajas: por un lado depende de código nativo, ya que el manejador ODBC no ha sido desarrollado en Java. Esto compromete la portabilidad de nuestro desarrollo. Por otro lado al emplear este tipo de manejador nuestra aplicación llama al gestor de manejadores JDBC, quien a su vez llama al manejador JDBC (puente JDBC-ODBC), quien llama al manejador ODBC, que es el que finalmente llama a la base de datos. Hay muchos puntos donde potencialmente puede producirse un fallo, lo cual hace que estos manejadores muchas veces sean bastante inestables.

- ✓ Las llamadas JDBC son enviadas a una librería ODBC.

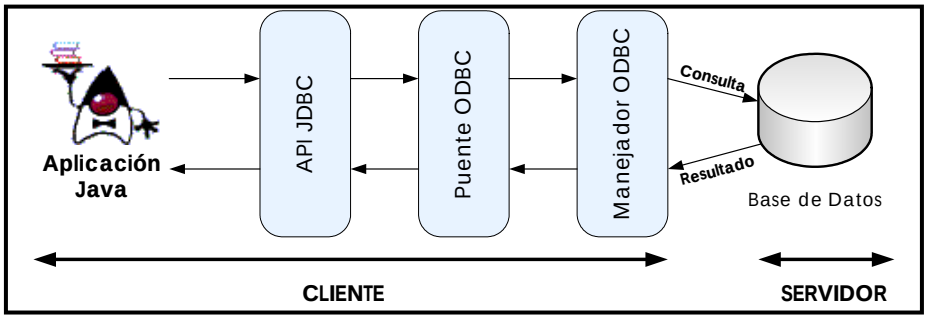


Figura 2.10: Puente entre JDBC-ODBC

2.6.3.2 Librería Nativa

Usa librerías nativas en C (Propias del RDBMS), para trasladar instrucciones JDBC al cliente nativo del RBDMS.

Se basa en una librería escrita en código nativo para acceder a la base de datos. El manejador traduce las llamadas JDBC a llamadas al código de la librería nativa, siendo el código nativo el que se comunica con las bases de datos. La librería nativa es proporcionada por los desarrolladores de la BD. Estos manejadores son más eficientes y tienen menos puntos de fallo que el puente JDBC-ODBC ya que hay menos capas entre el código de la aplicación y la base de datos. Sin embargo siguen teniendo el problema de pérdida de ***portabilidad*** por emplear código nativo.

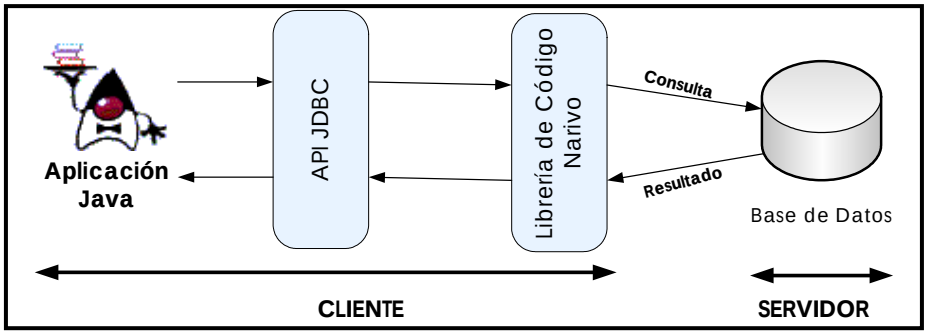


Figura 2.11: Librería Nativa

2.6.3.3 Protocolo de red

Las llamadas JDBC son trasladadas a un protocolo independiente de RDBMS y enviadas a un servidor intermedio (AppServer) sobre un socket TCP/IP.

El manejador se comunica con un servidor intermedio que se encuentra entre el cliente y la base de datos. El servidor intermediario se encarga de traducir las al API JDBC al protocolo específico de la base de datos. No se requiere ningún tipo de código nativo en el cliente, por lo que la portabilidad de la aplicación está garantizada: el manejador es tecnología 100% Java. Además si el intermediario es capaz de traducir las llamadas JDBC a protocolos específicos de diferentes sistemas de BD podremos emplear un mismo

manejador para comunicarnos con diferentes bases de datos. Esto lo convierte en el manejador más flexible de todos. No obstante se trata de un controlador complejo, ya que requiere la presencia de un middleware, una capa intermedia entre el cliente y la base de datos, por lo que no es muy común emplearlo en arquitecturas simples, sino más bien en arquitecturas sofisticadas donde muchas veces entran en juego varias bases de datos distintas. **[www019]**

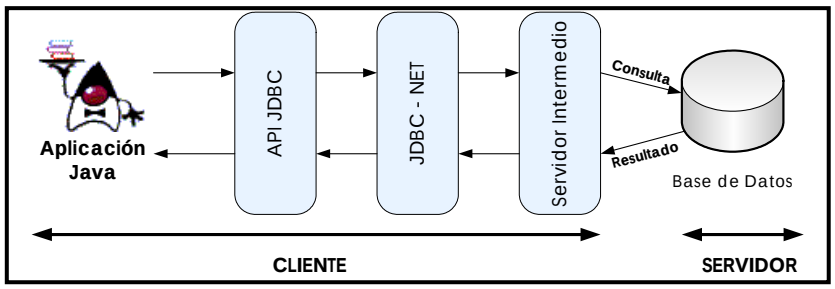


Figura 2.12: Protocolo de Red

2.6.3.4 Protocolo/Librería Nativa 100% java (4)

Las llamadas JDBC son convertidas directamente al protocolo del RDBMS.

El manejador traduce directamente las llamadas al API JDBC al protocolo nativo de la base de datos. Es el manejador que tiene mejor rendimiento, pero está más ligado a la base de datos que empleemos que el manejador tipo JDBC-Net, donde el uso del servidor intermedio nos da una gran flexibilidad a la hora de cambiar de base de datos. Este tipo de manejadores también emplea tecnología 100% Java.

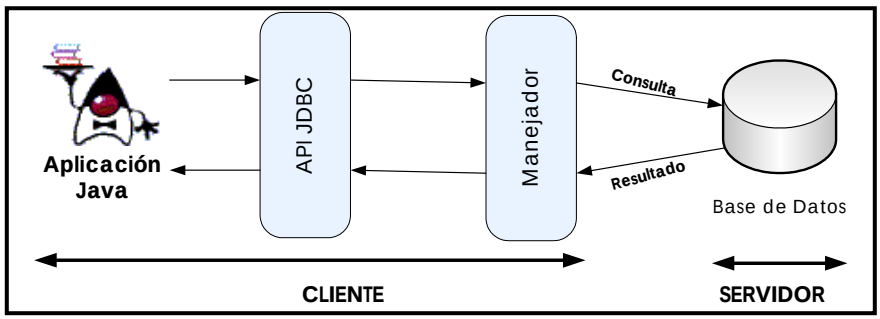


Figura 2.13: Librería 100% java

2.6.4 Funcionamiento de JDBC

Simplemente JDBC hace posible estas tres cosas:

- ✓ Establece una conexión con la base de datos.
- ✓ Envía sentencias ***SQL***
- ✓ Procesa los resultados.

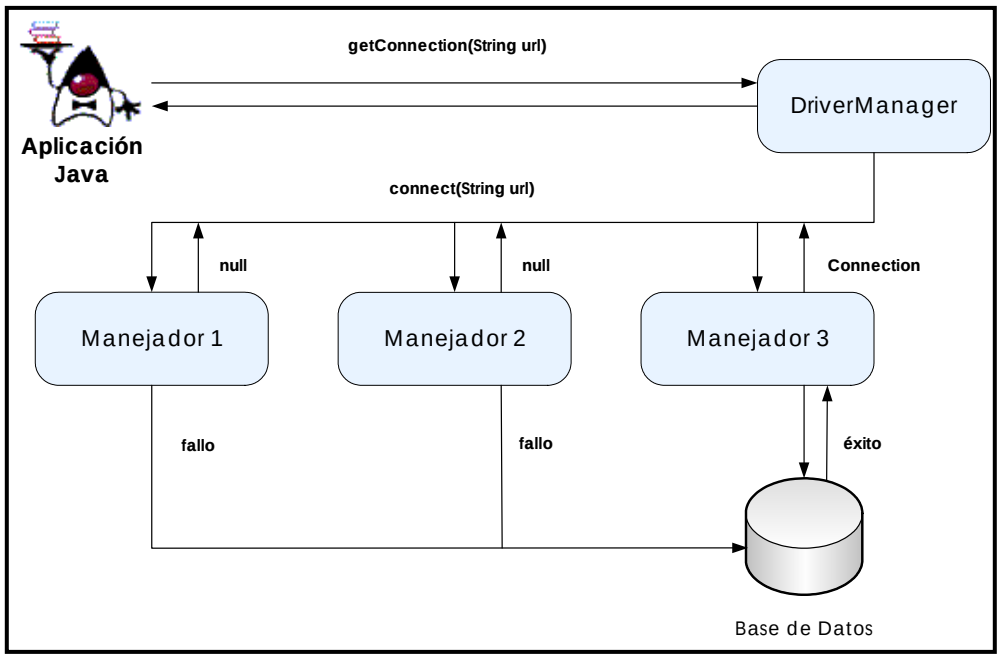


Figura 2.14: Funcionamiento JDBC

Representa una conexión con la base de datos. Permite crear objetos que representan consultas que se ejecutarán en la base de datos, y permite acceder a información sobre la base de datos y las posibilidades del manejador JDBC. En esta tabla recogemos los principales métodos de la interfaz Connection. Sobre muchos de ellos volveremos a hablar más adelante. **[www019]**

Métodos de la Interfaz Connection	
void close()	Libera lo los recursos de esta conexión.
void commit()	Hace permanentes los cambios que se realizaron desde el último commit o rollback.
Statement createStatement()	Crea un objeto de tipo Statement.
bolean getAutoCommit()	Indica si está en modo auto-commit.
DatabaseMetaData getMetaData()	Devuelve un objeto tipo DatabaseMetaData con meta información a cerca de la base de datos contra la cual se ha abierto esta conexión..
CallableStatement	Crea un objeto CallableStatement para ejecutar
prepareCall(String sql) PreparedStatement prepareStatement (String sql)	procedimientos almacenados SQL en la BD. Crea un objeto PreparedStatement para ejecutar consultas SQL parametrizadas en la BD.
void rollback()	Deshace todos los cambios realizados desde la última vez que se ejecutó comit o rollback
void setAutoCommit (boolean autoCommit)	Indica el modo de commit en el cual se opera.

Tabla 2.1: Métodos de ODBC