Universitat Autònoma
de Barcelona

# Estudio del servidor de aplicaciones Glassfish y de las aplicaciones J2EE

Memòria del projecte

d'Enginyeria Tècnica en

Informàtica de Sistemes

realitzat per

David Serra Manchado

i dirigit per

Daniel Franco Puntes

**Escola d'Enginyeria**

Sabadell, Juny de 2010

Universitat Autònoma de Barcelona

*Model de certificat del tutor*

El sotasignat, Daniel Franco Puntes,

professor de l'Escola d'Enginyeria de la UAB,

**CERTIFICA**:

Que el treball al que correspon la present memòria
ha estat realitzat sota la seva direcció
per en David Serra Manchado

I per a que consti firma la present.
Sabadell, Juny de 2010

---

Signat: Daniel Franco Puntes

Universitat Autònoma de Barcelona

*Model de certificat del tutor de l'empresa*

El sotasignat, Joan Piedrafita Farràs,
de OTSA Maquinaria Eléctrica S.L.,

**CERTIFICA:**

Que el treball al que correspon la present memòria
ha estat realitzat sota la seva supervisió
per en David Serra Manchado

I per a que consti firma la present.
Sabadell, Juny de 2010

---

Signat: Joan Piedrafita Farràs

# ÍNDICE

# ÍNDICE DE FIGURAS

# CAPÍTULO 1

## Introducción

## *1.1 Previo*

Este proyecto se basa en un estudio del servidor de aplicaciones Glassfish así como un estudio general de las aplicaciones J2EE.

En el mundo empresarial se necesitan a diario multitud de aplicaciones tanto para uso interno como para ofrecer a personas externas. Inicialmente las aplicaciones se desarrollaban para un solo ordenador para más adelante realizarse las primeras aplicaciones cliente/servidor. Pero las necesidades de las empresas van evolucionando y se crearon los sistemas distribuidos. De este modo aparece Java con su estándar J2EE, que permite el desarrollo de aplicaciones multi-nivel.

Ante la complejidad y heterogeneidad de este tipo de sistemas, J2EE ofrece una arquitectura unificada y modular que facilita el desarrollo de aplicaciones distribuidas, proporcionándoles una serie de servicios que permiten acelerar el proceso de desarrollo dentro de las necesidades específicas de una empresa.

Las características más importantes de la arquitectura J2EE son la portabilidad, la escalabilidad, la simplicidad y la capacidad de integración. Su desarrollo se basa en especificaciones de de Enterprise JavaBeans, Servlets y JSP, que explicaremos en el transcurso de este proyecto.

Como servidor de las aplicaciones J2EE se ha optado por Glassfish, desarrolllado por Sun Microsystems, es un servidor Open Source que ofrece multitud de herramientas que encajan con la filosofía de J2EE y dispone su apoyo así como de todo el soporte de JSP, JSF, EJB, etc.

Todas estas aplicaciones supondrían un gasto muy elevado para las empresas, que gracias a estas herramientas open source se pueden llevar a cabo sin ninguna inversión inicial.

## 1.2 Objetivos

El objetivo principal del proyecto consiste en realizar un estudio sobre J2EE y Glassfish. Realizar un análisis de sus componentes así como de sus posibles alternativas tanto de aplicaciones como de servidores. Además realizar un estudio completo sobre EJB así como sus diferentes tipos. Y finalmente, realizar una comparativa entre las diferentes posibilidades de interfaces de presentación que se se pueden utilizar según las diferentes tecnologías existentes.

Finalmente el último objetivo es la realización de un prototipo de una aplicación empresarial utilizando las herramientas explicadas en el proyecto consistente en un administrador de los recursos humanos de la empresa para la gestión interna de los trabajadores.

## 1.3 Motivación

La temática de este proyecto fue propuesta al autor por la empresa OTSA Maquinaria Eléctrica S.L.

La principal motivación del proyecto siempre ha sido entrar en contacto con un proyecto real en el mundo laboral. Al surgir esa posibilidad, dentro del proyecto se ofrecía ampliar conocimientos de base de datos y java, además de introducirse en la arquitectura de sistemas multicapa basados en J2EE y servidor de aplicaciones.

Glassfish y J2EE son dos tecnologías que gracias a su distribución open source hace que haya una creciente comunidad de usuarios y las empresas se dedican cada vez más en trabajar sobre ellas, así que estudiar sobre ella supone una oportunidad para aprender sobre tecnologías en desarrollo actual.

## 1.4 Estructura de la memoria

Esta memoria se compone de 6 capítulos. El capítulo 1 es la Introducción al proyecto, el capítulo 2 es una introducción a J2EE así como un estudio a sus alternativas, el capítulo 3 se basa en el entorno de desarrollo J2EE detallando todas sus características, estudio de los tipos de arquitectura así como del servidor de aplicaciones Glassfish y sus competidores más importantes, para acabar con la instalación del entorno de desarrollo que se ha utilizado para el proyecto. El capítulo 4 se basa en un estudio completo de la tecnología EJB, todos los tipos que hay y cómo se utilizan, finalizando con un tutorial de creación de EJB. El capítulo 5 se basa en la implementación de diferentes tipos de lógica de usuario, intercambiando diferentes interfaces y realizando una comparación final de los tiempos de ejecución de cada uno. Finalmente, en el capítulo 6 podemos encontrar las conclusiones finales del proyecto.

## 1.5 Agradecimientos

En primer lugar me gustaría agradecer a Joan Piedrafita Farrás por el tiempo dedicado a ayudarme a realizar este proyecto y por ofrecerse a aceptar a estudiantes para integrarse en el mundo laboral.

En general agradezco a todo el equipo de OTSA Maquinaria Eléctrica S.L. la ayuda que me han ofrecido en todo este tiempo, junto con mis compañeros estudiantes de la UPC que gracias a ellos el transcurso de este proyecto se ha hecho mucho más ameno.

También agradecer a Daniel Franco por ofrecerme la oportunidad de realizar este proyecto en la empresa OTSA Maquinaria Eléctrica S.L.

Por último agradecer a mi familia todo el apoyo que me han ofrecido durante todos los años de la carrera y a mis amigos y novia por aguantarme y apoyarme.

# CAPÍTULO 2

## Introducción a J2EE

## *2.1 Plataforma Java*

La plataforma Java es el entorno de software basado en Java que se ejecuta sobre otras plataformas y su software puede ser usado sobre varios sistemas operativos y hardware. Está formada por tres componentes:

- **Lenguaje.** Es un lenguaje de propósito general, de alto nivel que utiliza el paradigma de orientación a objetos.

- **La Máquina Virtual.** Los programas escritos en Java son compilados como archivos ejecutables de una máquina virtual llamada Java Virtual Machine (JVM), esto permite que los programas ejecutables puedan ejecutarse en distintas arquitecturas.

- **Las Bibliotecas.** El conjunto de bibliotecas del lenguaje es conocido como la Java Aplication Programming Interface (Java API) y es un conjunto de componentes que proporcionan diferentes herramientas para el desarrollo.

Para la plataforma Java existen diferentes ediciones:

- **Java 2 Plataform, Micro Edition (J2M3).** Desarrollo para artículos móviles pequeños.

- **Java 2 Plataform, Standard Edition (J2SE).** Desarrollo para ordenadores personales y aplicaciones en general.

- **Java 2 Plataform Enterprise Edition (J2EE).** Desarrollo orientado a aplicaciones empresariales.

### *2.1.1 Java Specification Request*

El Java Community Process, es un proceso formalizado el cual permite a las partes interesadas a involucrarse en la definición de futuras versiones y características de la plataforma Java.

El Proceso JCP conlleva el uso de Java Specification Request, las cuales son documentos formales que describen las especificaciones y tecnologías propuestas para que sean añadidas a la plataforma Java. Una de estas especificaciones es la plataforma J2EE.

## *2.2 Java EE/J2EE*

### *2.2.1 Introducción*

Según Sun Java Web: “J2EE define el estándar para el desarrollo de aplicaciones multicapa basados en componentes de la empresa”.

Java J2EE es un conjunto de especificaciones para APIs, una arquitectura de computación distribuida, y las definiciones para el paquete de componentes distribuidos para el desarrollo.

Es una colección de componentes estandarizados, contenedores y servicios para crear y desarrollar aplicaciones distribuidas en una arquitectura bien definida.

J2EE está dirigido a sistemas empresariales a gran escala. El software que funciona a este nivel no se ejecuta en un solo PC por falta de recursos, por esa razón, el software tiene que ser dividido en partes y desplegados en las plataformas de hardware adecuado. Esa es la esencia de la computación distribuida. J2EE proporciona un conjunto de componentes estandarizados que facilitan la implementación de software, interfaces estándar que definen la interconexión de los distintos módulos de software, y los servicios estándar que define como se comunican los distintos módulos.

J2EE permite desarrollar y ejecutar software de aplicaciones en Lenguaje de programación Java con arquitectura de N niveles distribuida, basándose ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones.

La característica principal de esta plataforma es que en lugar de presentarse como un producto o solución software se presenta como una especificación. Esta especificación se considera informalmente como un estándar debido a que los productos que la implementen deben cumplir unos ciertos requisitos de conformidad con esta. De esta manera se consigue un grado de apertura bastante importante ya que la implementación particular de un producto conforme a J2EE no "ata" el desarrollo de aplicaciones a dicho producto y permite un alto grado de capacidad de despliegue en productos conformes con la especificación. Por otro lado el hecho de estar desarrollado sobre la plataforma Java permite la instalación de la solución en diferentes arquitecturas hardware y sistemas operativos.

J2EE proporciona un conjunto de API's para el desarrollo y coordinación de aplicaciones y componentes distribuidos tales como EJB, XML, JDBC, RMMI, JMS, Java Server Pages, Servlets, etc.

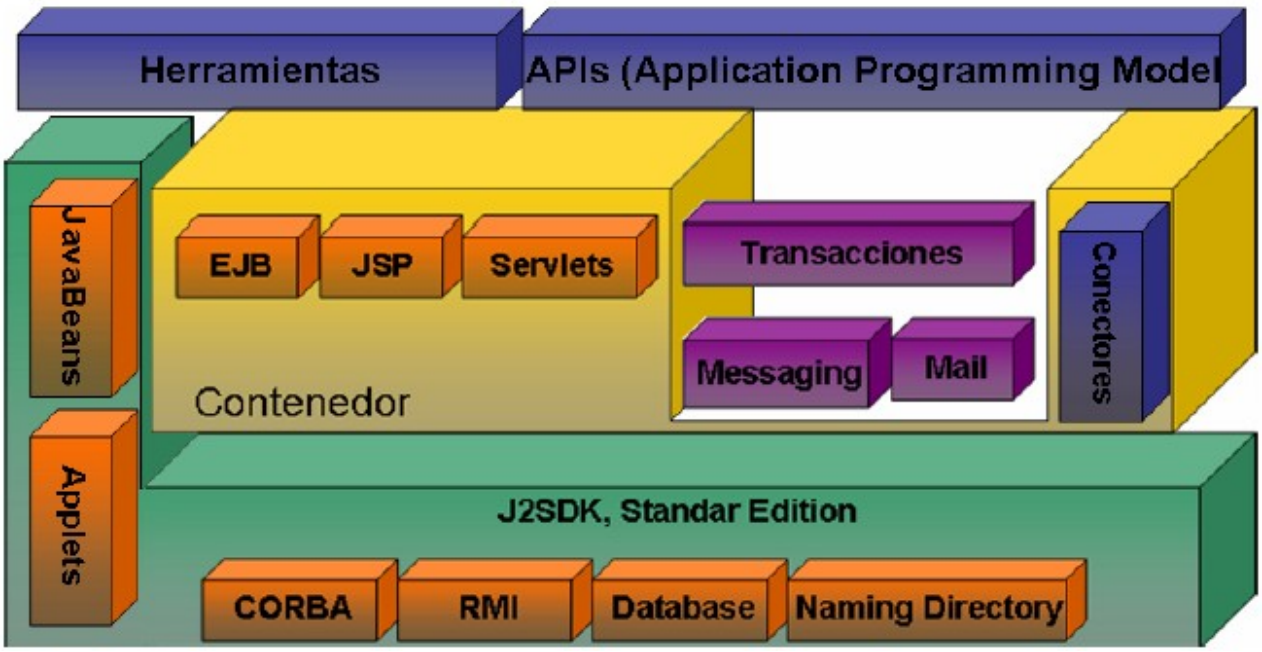



***Figura 2.1*** *Arquitectura J2EE*

Además J2EE también se encarga de proporcionar características para la gestión de:

- **Seguridad**
- **Control de transacciones**
- **Gestión de componentes desplegados**
- **Control de concurrencia**
- **Uso y asignación de recursos**

Estas características han sido clave para la elección de la plataforma utilizada ya que le confieren un nivel muy aceptable para el desarrollo de una aplicación distribuida multinivel.

### *2.2.2 Alternativas a J2EE*

Algunas alternativas a la tecnología J2EE puede ser .NET, PHP o C-C++.

#### *2.2.2.1 PHP*

La opción PHP permite crear un programa que se pueda ejecutar en cualquier servidor desde un programa visualizador de páginas web y dar respuestas en función de los datos que introduzca el usuario. PHP destaca en la sencillez, velocidad y facilidad de uso. Características que a su vez limitan su usabilidad y portabilidad con otros lenguajes y programas. PHP tampoco ofrece ningún tipo de separación entre aplicación y presentación.

Java Beans es más lento que PHP en servidores pequeños pero es totalmente portable y compatible, haciendo del lenguaje Java un lenguaje universal y no solo un lenguaje para programar páginas dinámicas como PHP.

#### *2.2.2.2 C-C++*

Los lenguajes de programación C-C++ se desarrollaron en los años 70 pensando para trabajar con el hardware extremadamente caro de la época. Las limitaciones que suponían los elevados costes de ampliación de velocidad del hardware de aquella época, suponían que la orientación de los lenguajes de programacion fuese hacia generar velocidad desde la integración del lenguaje con la máquina.

La evolución del hardware hasta la actualidad ha vencido las barreras del hardware y hoy en día la necesidad es la portabilidad y la escalabilidad, características que no posee C y en las que se basa Java y, por extensión, Java Beans. Además, C-C++ no poseen un estándar para grandes proyectos de tipo J2EE.

## *2.3 .NET*

.NET es un proyecto de Microsoft para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma de hardware y que permita un rápido desarrollo de aplicaciones. Basado en ella, la empresa intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el sistema operativo hasta las herramientas de mercado. .NET podría considerarse una respuesta de Microsoft al creciente mercado de los negocios en entornos Web, como competencia a la plataforma J2EE de Sun Microsystems.

Desde el punto de vista del programador, el entorno .NET ofrece un solo entorno de desarrollo para todos los lenguajes que soporta. Provee un extenso conjunto de soluciones predefinidas para necesidades generales de la programación de aplicaciones, y administra la ejecución de los programas escritos específicamente con la plataforma. Esta solución es el producto principal en la oferta de Microsoft, y pretende ser utilizada por la mayoría de las aplicaciones creadas para la plataforma Windows.

Para el desarrollo y ejecución de aplicaciones en este nuevo entorno tecnológico Microsoft proporciona el conjunto de herramientas conocido .NET Framework SDK, que es posible descargarlo gratuitamente de su sitio web e incluye compiladores de lenguajes como C#, Visual Basic.NET, Managed C++ y JScript.NET específicamente diseñados para crear aplicaciones para él.

### *2.3.1 Características de .NET*

Las principales características de .NET son:

- **Interoperabilidad con otros entornos.** Permite operar con distintos entornos,independientemente de la plataforma, usando MSIL (Microsoft Intermediate Language) que es un conjunto de instrucciones independientes del procesador.

- **Soporte para desarrollar aplicaciones independientes del lenguaje.** Debido a la publicación de la norma para la infraestructura común de lenguajes (CLI), el desarrollo de lenguajes se facilita. .NET soporta ya más de 20 lenguajes de programación como son: C#, Visual Basic, Delphi, C++, J#, Perl, Python, Fortran y Cobol.NET.

- **Soporte para aplicaciones Web y servicios XML**. La arquitectura .NET soporta estos dos tipos de aplicaciones mediante la tecnología ASP.NET.

- **Soporte para aplicaciones remotas y COM.** Proporciona servicios para interactuar con componentes COM (Component Object Model) y COM+.

### *2.3.2 Common Language Runtime (CLR)*

El corazón de la plataforma.NET es el CLR (Common Language Runtime), que es una aplicación similar a un máquina virtual que se encarga de gestionar la ejecución de las aplicaciones para ella escritas. A estas aplicaciones les ofrece numerosos servicios que facilita su desarrollo y mantenimiento y favorecen su fiabilidad y seguridad. Entre ellos los principales son:

- Modelo de programación consistente y sencillo, orientado a objetos.

- Eliminación del temido problema de compatibilidad entre DLLs

- Ejecución multiplataforma y multilenguaje.

- Recolector de basura: elimina de memoria objetos no utilizados.

- Soporte de multiproceso (hilos): permite ejecutar código en forma paralela.

- Gestión del acceso a objetos remotos que permite el desarrollo de aplicaciones distribuidas de manera transparente a la ubicación real de cada uno de los objetos utilizados en las mismas.

- Empaquetador de COM: coordina la comunicación con los componentes COM para que puedan ser usados por el .NET Framework.

### *2.3.3 .NET framework*

La arquitectura .NET es una plataforma de desarrollo de software creada por Microsoft con el objetivo de proporcionar a los programadores herramientas para el desarrollo de aplicaciones. El framework constituye la base de la plataforma .NET y denota la infraestructura sobre la cual se reúnen un conjunto de lenguajes, herramientas y servicios que simplifican el desarrollo de aplicaciones en entorno de ejecución distribuido. Los elementos principales de .NET Framework son básicamente:

- La **Common Language Specification** (CLS) describe un conjunto de características comunes a diferentes lenguajes.

- La **Base Class Library** (BCL), que contiene la funcionalidad más comúnmente utilizada para el desarrollo de todo tipo de aplicaciones.

- **ADO.NET**, que contiene un conjunto de clases que permiten interactuar con bases de datos relacionales y documentos XML como repositorios de información persistente.

- **ASP.NET**, que constituye la tecnología dentro del .NET Framework para construir aplicaciones con interfaz de usuario Web y servicios Web.

- **Windows Forms**, que constituye la tecnología dentro del .NET Framewok que permite crear aplicaciones con interfaz de usuario basada ventanas.

- El **Common Language Runtime** (CLR), comentado anteriormente.

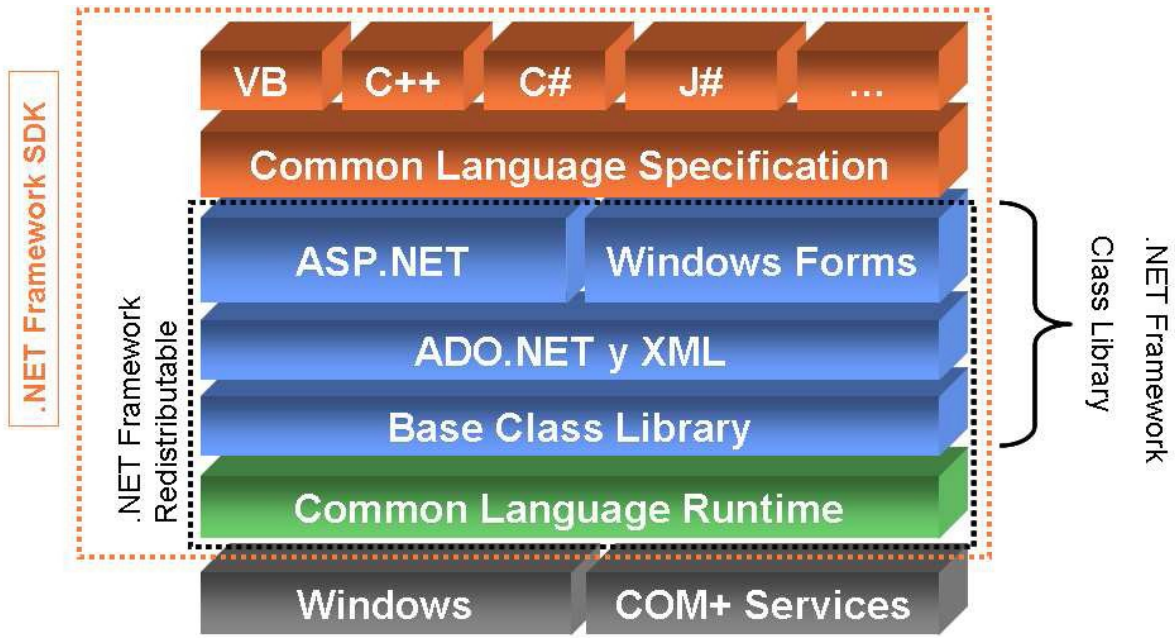



***Figura 2.2*** *Organización de la estructura de un Framework en .NET*

### *2.3.4 .NET Remoting*

La plataforma .NET es una apuesta de Microsoft para competir con la plataforma Java. .Net Remoting es parte del .Net Framework y permite crear fácilmente aplicaciones ampliamente distribuidas, tanto si los componentes de las aplicaciones están todos en un equipo como si están repartidos por el mundo. Se pueden crear aplicaciones de cliente que utilicen objetos en otros procesos del mismo equipo o en cualquier otro equipo disponible en la red. También se puede utilizar .NET Remoting para comunicarse con otros dominios de aplicación en el mismo proceso.

.NET Remoting proporciona un enfoque abstracto en la comunicación entre procesos que separa el objeto utilizado de forma remota de un dominio de aplicación de cliente o servidor específico y de un mecanismo específico de comunicación. Por lo tanto, se trata de un sistema flexible y fácilmente personalizable. Se puede reemplazar un protocolo de comunicación con otro o un formato de serialización con otro sin tener que recompilar el cliente ni el servidor. Además, el sistema de interacción remota no presupone ningún modelo de aplicación en particular. Se puede comunicar desde una aplicación Web, una aplicación de consola, un servicio de Windows, desde casi cualquier aplicación que se desee utilizar. Los servidores de interacción remota también pueden ser cualquier tipo de dominio de aplicación. Cualquier aplicación puede albergar objetos de interacción remota y proporcionar sus servicios a cualquier cliente en su equipo o red.

### *2.3.5 Arquitectura*

Un cliente se limita a crear una nueva instancia de la clase de servidor. El sistema de interacción remota crea un objeto proxy que representa a la clase y devuelve al objeto del cliente una referencia al objeto proxy. Cuando un cliente llama a un método, la infraestructura de interacción remota controla la llamada, comprueba el tipo de información y dirige la llamada por el canal hacia el proceso del servidor. Un canal a la escucha detecta la solicitud y la reenvía al sistema de interacción remota del servidor, que a su vez busca (o crea, si es necesario) y llama al objeto solicitado. A continuación el proceso se invierte: el sistema de interacción remota del servidor incluye la respuesta en un mensaje que el canal del servidor envía al canal del cliente. Por último, el sistema de interacción remota del cliente devuelve el resultado de la llamada al objeto del cliente a través del objeto proxy.

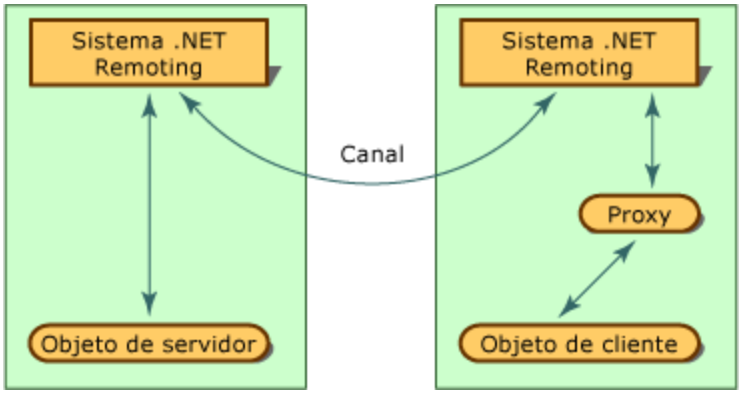



***Figura 2.3*** *Arquitectura de .NET Remoting*

## *2.4 Diferencias entre Java2EE y .NET*

Las tecnologías Java2EE y .NET pertenecen a Sun Microsystems y Microsoft respectivamente. Son dos empresas rivales que luchan por un mercado exclusivo.


***Figura 2.4*** *Java vs .NET*

### *2.4.1 Ventajas de .NET*

- .Net tiene la posibilidad de emplear múltiples lenguajes de programación, mientras que J2EE sólo trabaja con Java.

- Las herramientas de desarrollo incluidas por Microsoft en su Visual Studio .Net son mucho más simples, intuitivas y sencillas de manejar que las herramientas de desarrollo equivalentes en J2EE suministradas por otras empresas.

- C# es un lenguaje interesante y fácil de aprender por los programadores de Java y existe un conversor Java-C#.

- .Net se ha diseñado considerando los servicios Web siendo estos servicios propios de la plataforma.

### *2.4.2 Ventajas de J2EE*

- Las implementaciones de J2EE pueden adquirirse a distintas compañías, mientras que .Net solo puede comprarse a Microsoft.

- La seguridad frente a virus informáticos de los productos de Microsoft es menor que los basados en Java ya que desde un comienzo Java se fundamentó en un estricto modelo de seguridad.

- Las aplicaciones Java pueden correr en una amplia gama de sistemas operativos y de arquitecturas hardware.

- Java es una tecnología Open Source y posibilita que los desarrolladores puedan conocer y entender completamente cómo hace las cosas Java y aprovecharlo para sus aplicaciones.

- J2EE es ahora el producto de la colaboración de más de 400 empresas y organizaciones de todo tipo. .Net es y será el producto de una sola compañía.

## *2.5 Interoperabilidad entre J2EE y .NET*

Las empresas actuales exigen cada vez más una mejora en la escalabilidad y portabilidad de las aplicaciones, así como la facilidad de integración de las aplicaciones. Dado que las plataformas J2EE de Sun y .NET de Microsoft son las plataformas más utilizadas actualmente en el mercado han surgido varios métodos para realizar la integración.

### *2.5.1 Interoperabilidad basada en Web Services*

Los Servicios Web se están convirtiendo cada vez más en la solución tecnológica más atractiva para temas de interoperatividad. Los servicios web se basan en XML. La forma de invocar funcionalidades en otra máquina se hace a través de texto plano usando protocolos estándar W3C. Los protocolosbinarios son mucho más rápidos que un servicio web en XML. Este inconveniente sin embargo se ve compensado por la facilidad con la que diferentesplataformas pueden comunicarse unas con otras. Esta ventaja implica que podemos tener aplicaciones en .NET ejecutándose en una máquina Windows y esta aplicación puede ser accedida por cualquier otra aplicación escrita en otro lenguaje corriendo en otra máquina con otro sistema operativo

Los servicios web son código ejecutándose en otra plataforma que tienen un punto de entrada que permite activar el código usando un documento especial XML. En la mayoría de los casos se envía un documento XML en un formato llamado SOAP (Simple Object Access Protocol) sobre HTTP. El servidor tiene un oyente a la espera de un paquete SOAP, cuando se recibe un paquete de este tipo, el servidor se pone en funcionamiento y ejecuta su código nativo. El servidor coge el resultado de ese código nativo, lo empaqueta en otro paquete SOAP y se lo devuelve al cliente.

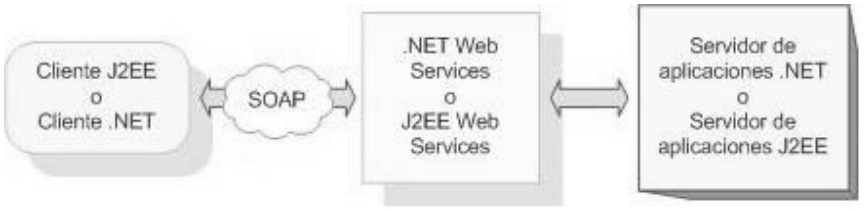


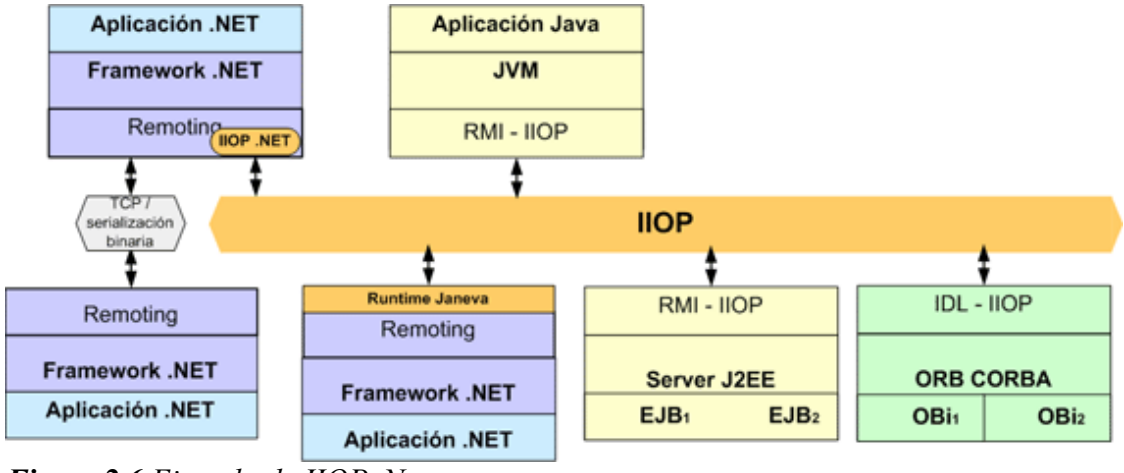

***Figura 2.5*** *Web Services*

### *2.5.2 Interoperabilidad basada en IIOP .NET*

IIOP.Net permite una interoperatividad entre .Net, CORBA y objetos distribuidos J2EE. Esto se hace gracias a la incorporación del soporte de CORBA/IIOP por parte de .Net sustentado por el framework Remoting.

Las características de IIOP.Net son las siguientes:

- Alto acoplamiento entre los objetos distribuidos en .Net, CORBA y J2EE.
- Transparencia. Los servidores existentes pueden ser usados sin ser modificados, sin código envolvente ni adaptadores.
- Gran cobertura de mapeo de tipos CORBA/.NET
- Integración nativa en el framework .Net. IIOP.Net está directamente basada en la infraestructura estándar de Remoting.




***Firura 2.6*** *Ejemplo de IIOP .Net*

# CAPÍTULO 3

## Entorno de desarrollo para J2EE

## *3.1 Desarrollo de aplicaciones*

Inicialmente las aplicaciones se desarrollaban en un solo ordenador. Eran aplicaciones centralizadas ya que el coste del hardware era muy elevado. Con la aparición de la LAN aparecieron las primeras aplicaciones cliente/servidor y con el tiempo el crecimiento de Internet así como el aumento de anchos de bandas aparecieron aplicaciones basadas en Cloud Computing o Web Services. En este proyecto nos centraremos el tema en las aplicaciones distribuidas.

### *3.1.1 Introducción de Sistemas distribuidos*

Un sistema distribuido se define como una colección de computadores autónomos conectados por una red, y con el software distribuido adecuado para que el sistema sea visto por los usuarios como una única entidad capaz de proporcionar facilidades de computación. [ *Colouris 1994* ]

El desarrollo de los sistemas distribuidos vino de la mano de las redes locales de alta velocidad a principios de 1970. El avance de la electrónica y de las tecnologías de la información han propiciado un aumento en su uso en detrimento de las aplicaciones diseñadas para ordenadores centralizados multiusuario. Las aplicaciones distribuidas varían dependiendo de la naturaleza de su uso, ya sean estos sistemas de cómputo masivo o aplicaciones bancarias, aunque en todas ellas podemos destacar una serie de puntos clave comunes a todas ellas, que son:

- **Compartición de recursos**
- **Apertura (openness)**
- **Concurrencia**
- **Escalabilidad**
- **Tolerancia a fallo**
- **Transparencia**

De las soluciones existentes para el desarrollo de sistemas distribuidos se ha optado en este proyecto por la utilización de la plataforma J2EE desarrollada por Sun Microsystems, ahora Oracle, mediante la implementación de ella por el producto Glassfish.

## *3.2 Arquitectura de las aplicaciones distribuidas*

### *3.2.1 Arquitectura basada en componentes*

Aproximan el diseño de sistemas como un conjunto de componentes. Un componente es una pieza de software que expone una interfaz bien definida y que puede colaborar con otros componentes para resolver un problema. Normalmente presenta características de herencia, polimorfismo y encapsulación.

Ventajas:

- La división en componentes reduce la complejidad, permite la reutilización y acelera el proceso de ensamblaje de software.

- Los creadores de componentes pueden especializarse creando objetos cada vez mas complejos y de mayor calidad.

- La interoperabilidad entre componentes de distintos fabricantes aumenta la competencia, reduce los costos y facilita la construcción de estándares.

- Los costes de mantenimiento del software se reducen.

Desventajas:

- Desarrollos más complejos.

- El coste de comunicaciones entre componentes es elevado.

- Dificultad en establecer los límites de los componentes y la relación entre ellos.

### *3.2.2 Arquitectura orientada a servicios*

La Arquitectura Orientada a Servicios o SOA es un concepto de arquitectura de software que define la utilización de servicios para dar soporte a los requisitos del negocio.

Permite la creación de sistemas altamente escalables que reflejan el negocio de la organización y ofrece una invocación de servicios mediante Web Services, lo que facilita la interacción entre diferentes sistemas.

SOA define las siguientes capas de software:

- **Aplicaciones básicas.** Sistemas desarrollados bajo cualquier arquitectura o tecnología, geográficamente separados.

- **De exposición de funcionalidades.** Donde las funcionalidades de la capa de aplicación son expuestas en forma de Web services.

- **De integración de servicios.** Facilitan el intercambio de datos entre elementos de la capa aplicativa orientada a procesos empresariales internos.

- **De composición de procesos.** Define el proceso en función del negocio y sus necesidades.

- **De entrega.** Donde los servicios son desplegados a los usuarios finales.

SOA proporciona una metodología y un marco de trabajo para documentar las capacidades de negocio y puede dar soporte a las actividades de integración y consolidación.

La arquitectura orientada a servicios está destinado a servicios que se encuentran en Internet o en una intranet usando servicios web. Existen diversos estándares relacionados con los servicios web como XML, HTTP, SOAP, etc.

Ventajas:

- Proporciona una gran integración y homogeneidad.
- Facilidad en la adaptación de nuevos servicios.
- Facilidad en la reestructuración de sistemas

Desventajas:

- La velocidad de intercambio entre sistemas es más lenta que una conexión directa.
- Intercambiar grandes cantidades de información puede afectar al rendimiento del bus.

### *3.2.3 Arquitectura orientada a capas*

El estilo de arquitectura de capas se basa en una distribución jerárquica de los roles de cada componente para proporcionar una división efectiva de los problemas que pueda surgir. Las capas de una aplicación pueden estar en la misma máquina o estar distribuidos entre varios equipos. La mayoría de interacciones entre los servicios solo ocurren entre capas vecinas que se comunican mediante alguna interfaz conocido por las dos partes.

En toda arquitectura de capa los elementos agrupados en una misma capa pueden comunicarse entre si; pero existen variantes en cuanto a las comunicaciones permitidas entre elementos de capas diferentes:

- **Arquitectura top-down.** Los elementos de una capa i+1 pueden enviar solicitudes de servicio a elementos de la capa inferior. Entonces se produce una cascada de solicitudes. Una arquitectura top-down puede ser no estricta si los elementos de una capa i+1 pueden enviar solicitudes a un elemento de cualquiera de las capas inferiores.

- **Arquitectura buttom-up.** Cada elemento de una capa i puede notificar a elementos de la capa superior de algún evento. Una

Ventajas:

- Se consigue una independencia entre capas que hace que se puedan realizar actualizaciones en cualquiera de las capas sin que afecte al resto del sistema.

- Mejora de rendimiento ya que distribuyendo las capas en distintos niveles físico se mejora la escalabilidad, la tolerancia a fallos y el rendimiento.

- Cada capa tiene una interfaz bien definida por lo que las pruebas se pueden realizar independientemente en cada capa sin afectar en el resto.

Desventajas:

- La dificultad para programar es mayor ya que hay diferentes dispositivos que tienen que comunicarse entre sí.

- Si se utilizan varios niveles de hardware, supone una gran carga en la red debido a la mayor cantidad de tráfico.

Es importante distinguir los conceptos de capas y niveles porque es común que se confundan o se denominen de forma incorrecta.

Las capas se ocupan de la división lógica de componentes y funcionalidad y no tienen en cuenta la localización física de componentes en diferentes servidores oen diferentes lugares. Por el contrario, los niveles se ocupan de la distribución física de componentes y funcionalidad en servidores separados, teniendo en cuenta la topología de redes y localizaciones remotas. Aunque tanto las capas como los niveles usan los mismos nombres (presentación, negocio, etc.), es importante no confundirlos y solo los niveles implican una separación física.

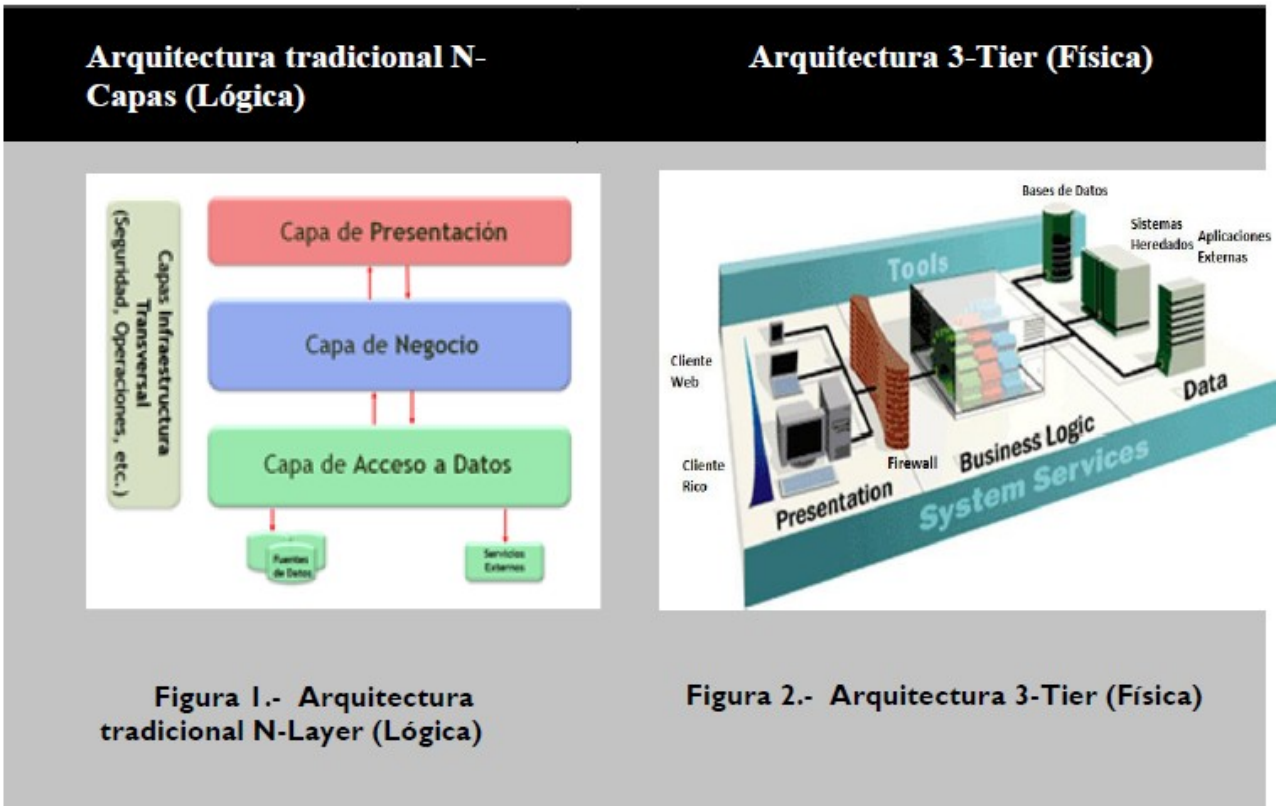



***Figura 3.1*** *Diferencia entre capas y niveles*

## *3.3 Software multinivel*

La arquitectura multinivel (o n-tier) es una evolución de las arquitecturas de los softwares antiguos. Primero, el cliente, los datos y el proceso estaban centralizados en el mismo lugar. Después evolucionó a una arquitectura cliente/servidor donde el proceso estaba dividido entre el cliente y el servidor y las solicitudes eran consultas a una base de datos. La lógica de negocios se desarrollaba en el cliente una vez recibidos los datos del servidor. Finalmente evolucionó a la arquitectura de tres capas donde se separó la capa de presentación de la lógica de negocio. Esta separación significa que la lógica de negocio no necesita saber que tipo de cliente muestra los datos. Las capas son mas escalables y pueden trabajar en diferentes tipos de plataformas. La seguridad es mas fácil de implementar ya que el software de la aplicación ya no está en la capa cliente.

### *3.3.1 Arquitecturas de 1-tier*

Son aplicaciones sencillas que han sido escritas para ejecutarse en un único equipo. Todos los servicios proporcionados por la aplicación, la interfaz de usuario, el acceso a datos, lógica de negocio, etc. existe en el mismo equipo físico y se agrupan en la misma aplicación. Esta arquitectura se llama de 1-tier ya que todo está localizado en la misma capa.

Los sistemas de un nivel son relativamente fáciles de administrar y la persistencia de datos es simple porque los datos se almacenan en un lugar único. Sin embargo, también tienen muchas desventajas. Estos sistemas no son escalables para gestionar múltiples usuarios, y no proporcionan un medio fácil de compartición de datos. Estos sistemas solo puede trabajar una persona a la vez.

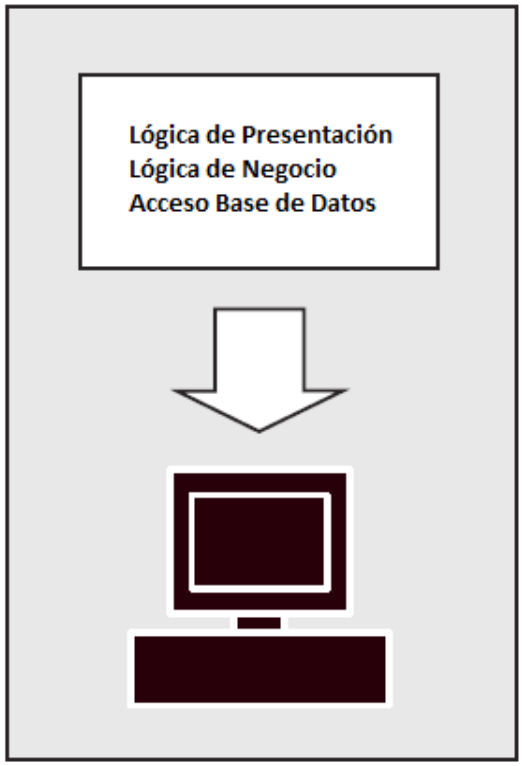



***Figura 3.2*** *Arquitectura 1-tier*

### *3.3.2 Arquitectura de 2-tier*

Muchas más aplicaciones constan de una arquitectura de 2 niveles. Esta se basa en un servidor de base de datos y acceso a datos persistentes mediante el envío de comandos SQL al servidor para guardar y recuperar datos. En este caso, la base de datos se ejecuta como un proceso independiente de la aplicación, o incluso en una máquina diferente a la que ejecuta el resto del programa. Los componentes de acceso a datos están separados del resto de la lógica de la aplicación. Esta arquitectura se basa en centralizar los datos para que varios usuarios trabajen simultáneamente con una base de datos común. Esta arquitectura también está denominada como cliente/servidor.

Una de las desventajas de la arquitectura de dos niveles es que la lógica que manipula los datos y aplica las normas específicas de los datos se agrupan en la propia aplicación. Esto plantea un problema en las aplicaciones de uso múltiple con una base de datos compartida ya que si se quiere hacer alguna modificación en las reglas de inserción de datos, por ejemplo, hay que asegurarse de realizar todos los cambios en todos los ordenadores que ejecuten la aplicación. Y además estos cambios deben realizarse al mismo tiempo y volverse a compilar.

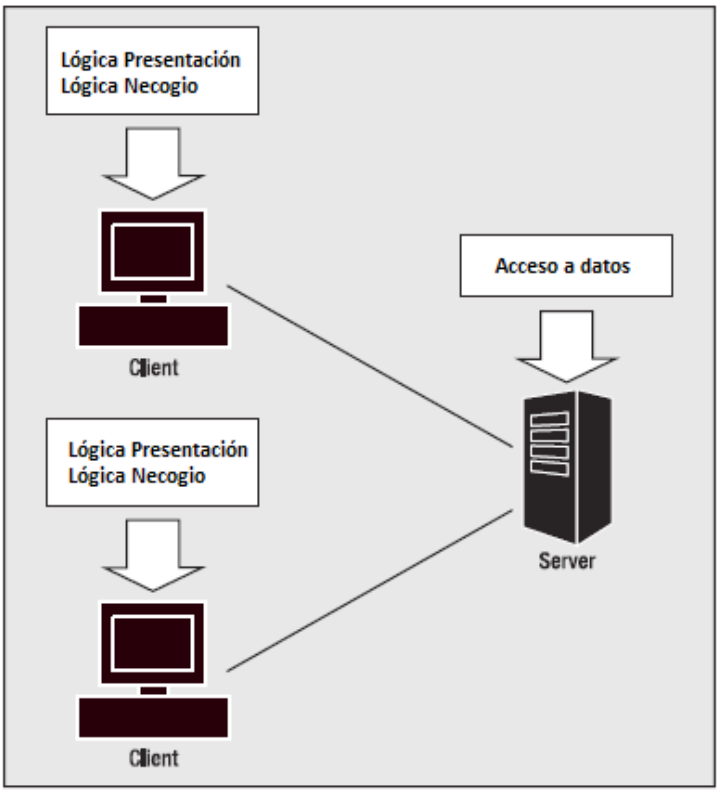



***Figura 3.3*** *Arquitectura de 2-tier*

Con el objetivo de solucionar el problema de tener que modificar todas las aplicaciones, surgió la idea de separar físicamente las reglas de negocio en un servidor independiente para que el software que ejecuta las reglas de negocio solo tenga que actualizarse una vez.

### *3.3.3 Arquitectura de 3-tier*

En la arquitectura de 3-tier, se añade una capa adicional entre el código de interfaz de usuario y la base de datos. Este nivel se llama lógica de negocio y representa la funcionalidad.

La arquitectura de tres niveles es la utilizada por servidores de aplicaciones. Las ventajas de esta arquitectura es que permite la modificación de nivel capa por separado facilitando la tarea al programador, simplifica la administración de los sistemas, facilita una disponibilidad inmediata en los cambios y además reparte la carga de trabajo entre los distintos ordenadores. Los niveles son las siguientes:

- **Lógica de usuario.** Es la parte de la aplicación que se ejecuta en los ordenadores de los usuarios y puede ser proporcionado por una aplicación independiente o usando un navegador de Internet.

- **Lógica de negocio.** Se ejecuta en otra máquina remota y es posible que esté repartida en varias máquinas. Utiliza una tecnología en la que se ve la funcionalidad como una colección de objetos en vez de llamadas SQL.

- **Lógica de datos.** Posiblemente implementada en otras máquinas, mediante uno o más gestores de Base de Datos. La funcionalidad solo debería ser accesible desde la Lógica de negocio.

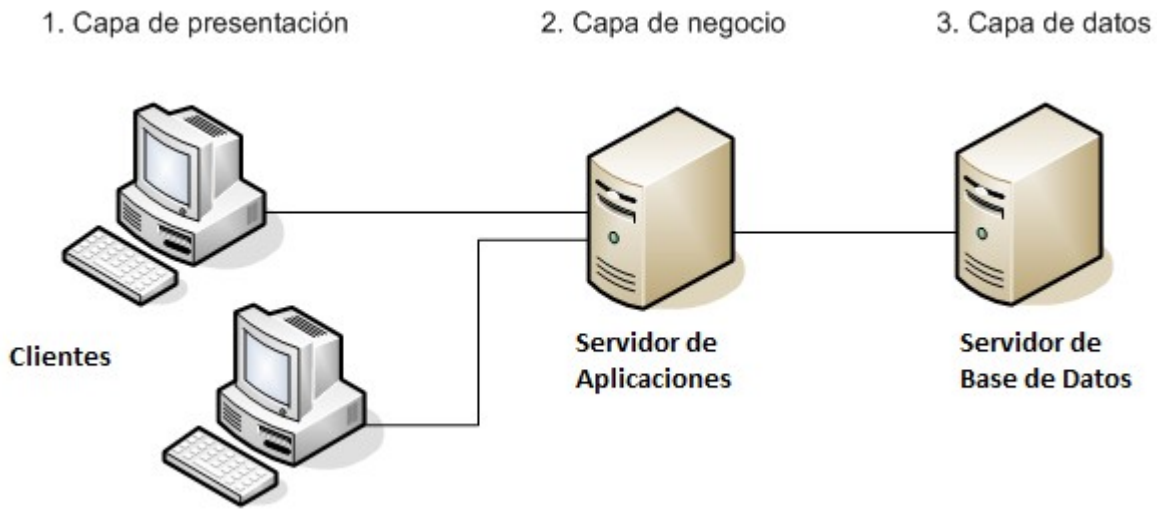



***Figura 3.4*** *Arquitectura de 3-tier*

### *3.3.4 Arquitectura de n-tier*

Una vez se sobrepasan los 3 niveles ya se pueden utilizar los niveles que se necesiten. Las arquitecturas de este número de niveles son mucho más complejas. Se pueden utilizar varios para enviar diferentes consultas SQL o por ejemplo utilizar un nivel más para seguridad en compras con tarjeta de crédito. También es común utilizar una arquitectura de 4-tier separando el servidor web de la lógica de negocio o utilizando la persistencia de datos como una capa separada.

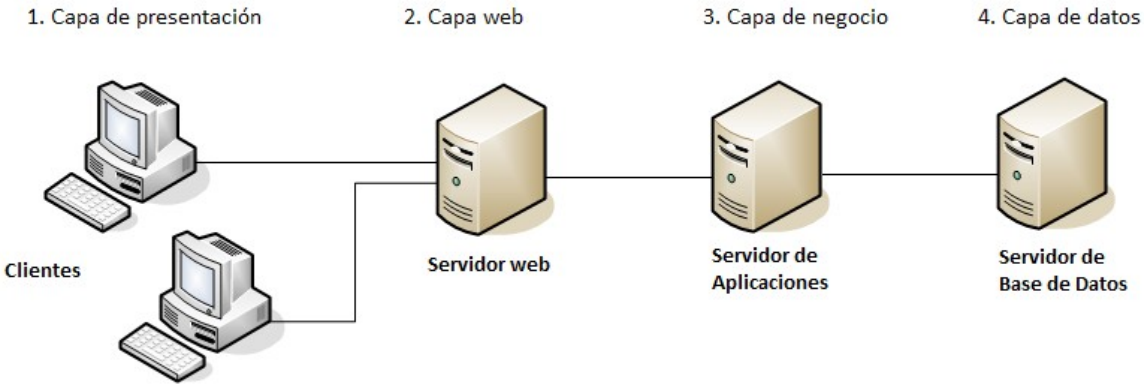



***Figura 3.5*** *Arquitectura de 4-tier*

## *3.4 J2EE*

### *3.4.1 Arquitectura Aplicaciones J2EE*

Las aplicaciones J2EE constan de 3 partes básicas, que son las siguientes:

- **Módulos EJB.** Encargado de tener la lógica de negocio y transacciones. Básicamente es el encargado de ejecutar programas y hacer consultas a la base de datos.

- **Módulo WAR.** Es la encargada de tener todos los elementos de interfaz como páginas web, servlets, etc.

- **Aplicación EAR.** Contiene toda la configuración de la aplicación J2EE, incluyendo el módulo WAR y EJB.

### *3.4.2 EJB*

Uno de los principales componentes de J2EE es la tecnología de Enterprise JavaBeans, que proporciona un estándar para el desarrollo de las clases que encapsulan la funcionalidad y reglas del negocio y que serán accedidas desde las aplicaciones cliente de modo casi idéntico a como lo serían si formasen parte de estas aplicaciones.

El esquema propuesto por J2EE para aplicaciones distribuidas es el de un servidor de aplicaciones que proporciona una gran cantidad de servicios, como acceso a la base de datos, servidores de correo, etc. Para implementar la funcionalidad del negocio se crean una seria de Enterprise Beans que serán cargados por el servidor de aplicaciones y pueden acceder a estos servicios. Estos beans serán administrados por el Módulo o Contenedor EJB.

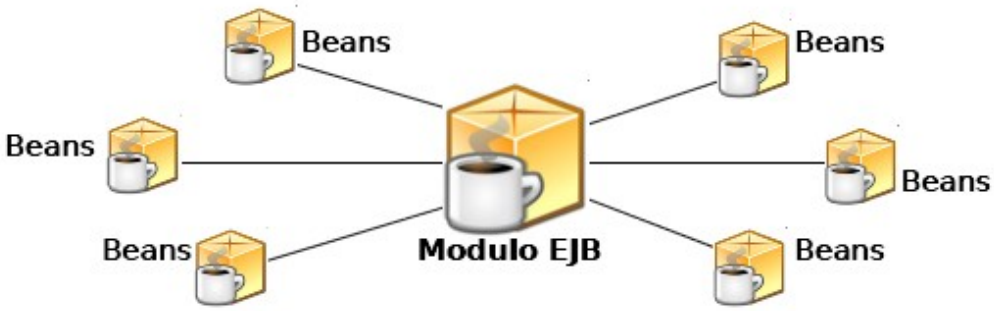



***Figura 3.6*** *Módulo EJB*

Principales características de EJB:

- Comunicación remota utilizando CORBA
- Transacciones
- Control de la concurrencia
- Eventos utilizando JMS (Java messaging service)
- Servicios de nombres y de directorio
- Seguridad
- Ubicación de componentes en un servidor de aplicaciones.

El objetivo de los EJB es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad, etc.) para centrarse en el desarrollo de la aplicación.

Es posible acceder a esta funcionalidad no solo desde aplicaciones Java, sino desde cualquier aplicación CORBA, lo que proporciona la posibilidad de trabajar con el lenguaje que queramos para crear las aplicaciones de cliente. J2EE también proporciona soporte para el acceso a la funcionalidad del negocio desde páginas HTML o XML, con lo que se puede utilizar un navegador para interactuar con las clases de negocio.

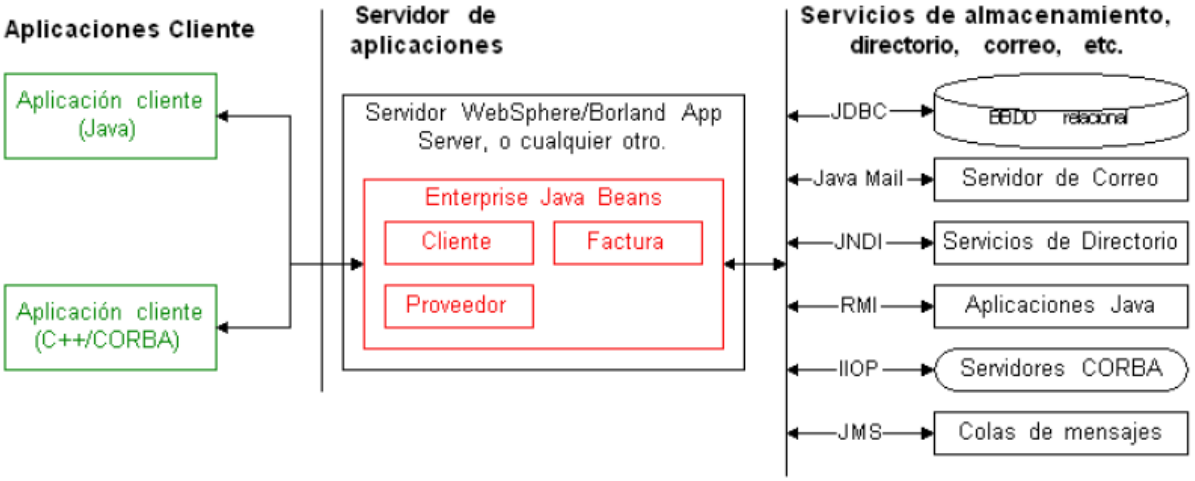



***Figura 3.7*** *Relación entre aplicaciones cliente, código de negocio, servidor de aplicación y otros elementos basado en J2EE*

### *3.4.2.1 Tipos de EJBs*

- **Beans de Sesión.** Representan las acciones que realizan los clientes. Pueden mantener un estado, pero solo durante el tiempo que el cliente interactua con el bean. Esto significa que los beans de sesión no almacenan sus datos en la Base de Datos después que el cliente termine el proceso. Hay dos tipos: con estado y sin estado.

- **Beans de sesión con Estado.** Almacenan datos específicos obtenidos durante la conexión con el cliente. Es decir, se almacena el estado conversacional de un cliente con el que interactua y se modifica conforma el cliente va realizando llamadas a los métodos de negocio del bean. Este estado no se guarda cuando el cliente termina la sesión.

- **Beans de sesión sin Estado.** No se modifican con las llamadas de los clientes. Sólo reciben datos y devuelven resultados pero sin modificar internamente el estado del bean. Son usados para ejecutar procesos de negocio o como puente de acceso a una Base de Datos o a un bean de entidad.

- **Beans de Mensajes.** Pueden escuchar mensajes de un servicio de mensajes JMS, éstos nunca se comunican directamente con el cliente.

- **Beans de Entidad.** Modelan conceptos o datos de negocio. Es la representación de la Base de Datos. El contenedor se encarga de sincronizar las variables de instancia del bean con la Base de Datos. Ya que los beans de entidad se guardan en un mecanismo de almacenamiento se dice que es persistente. Es decir, el estado del bean existe más tiempo que la duración de la aplicación.

### *3.4.3 Historia de J2EE*

La especificación original J2EE fue desarrollada por Sun Microsystems. Comenzando con J2EE 1.3, la especificación fue desarrollada bajo el Java Community Process. JSR 58 especifica J2EE 1.3 y JSR 151 especifica J2EE 1.4. El SDK de J2EE 1.3 fue liberado inicialmente como beta en abril de 2001. La beta del SDK de J2EE 1.4 fue liberada por Sun en diciembre de 2002. La especificación Java EE 5 fue desarrollada bajo el JSR 244 y la liberación final fue hecha el 11 de mayo de 2006. Finalmente la especificación Java EE 6 se lanzó en Diciembre de 2009.

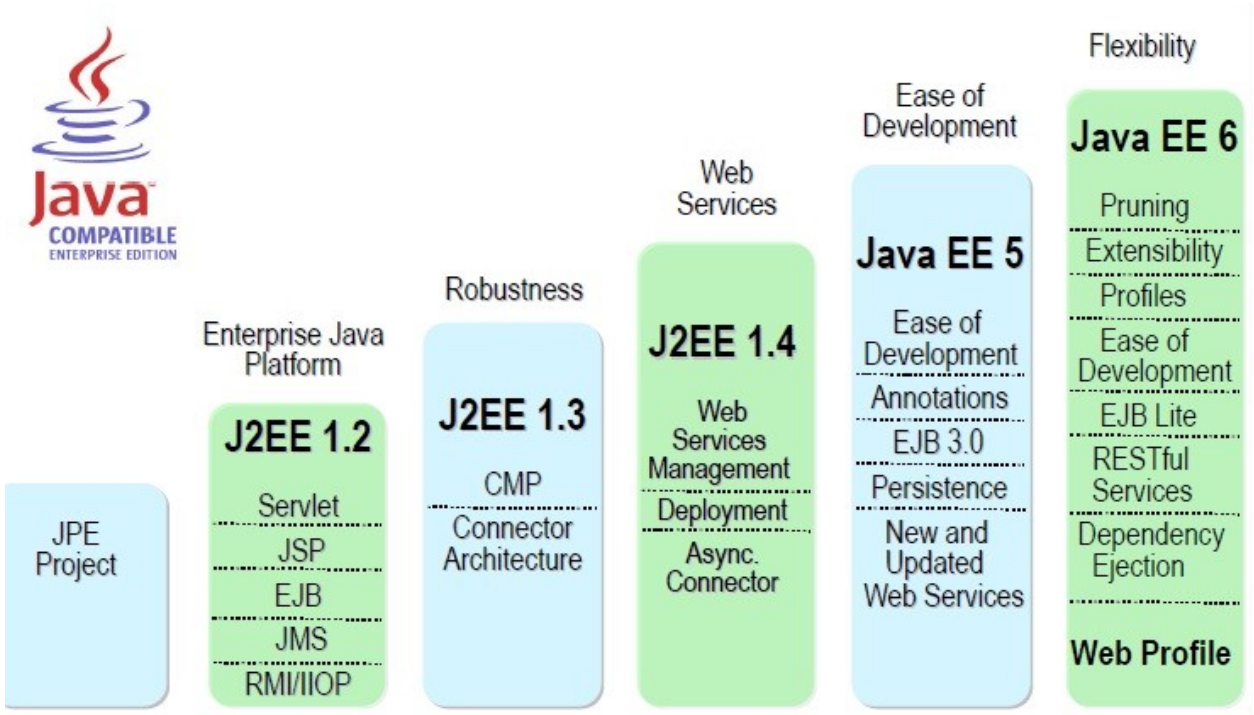



***Figura 3.8*** *Diferentes versiones de J2EE*

### *3.4.4 Soporte de J2EE a las diferentes capas de la aplicación*

A continuación describiremos las diferentes aplicaciones que podemos encontrar en los diferentes subsistemas de J2EE.

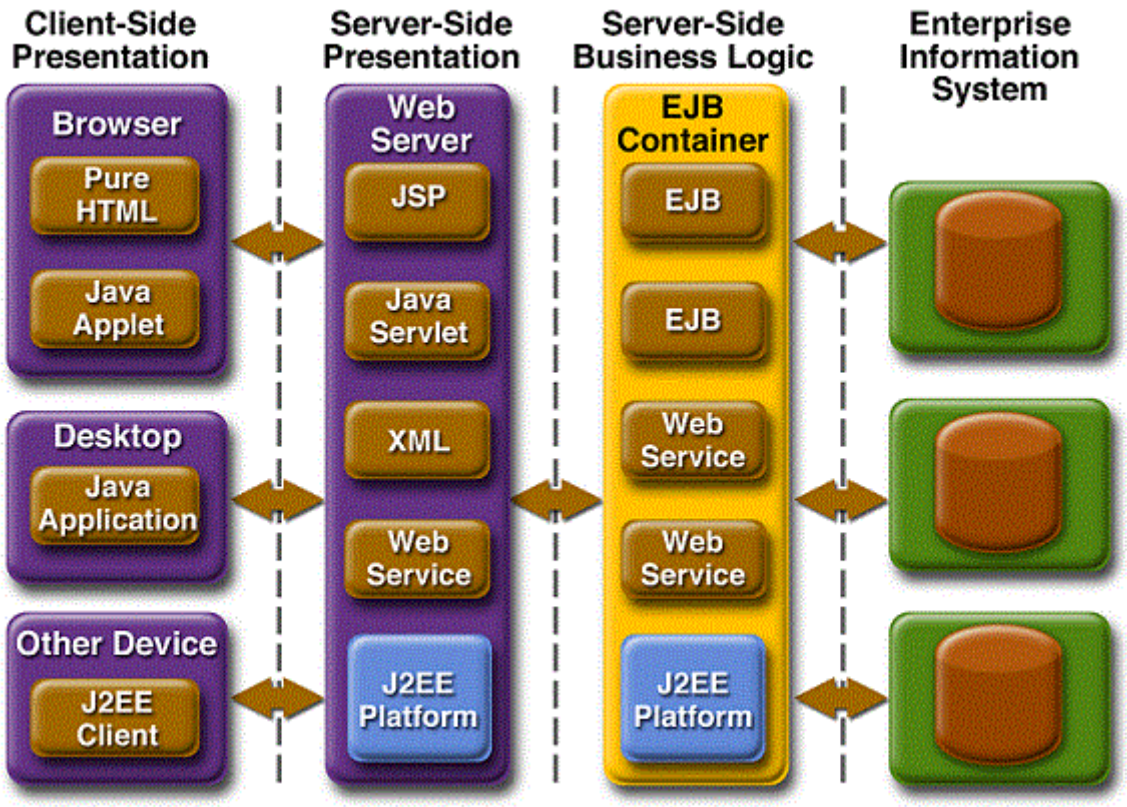



***Figura 3.9*** *Aplicaciones de J2EE*

### *3.4.4.1 Soporte a la capa de persistencia*

#### *3.4.4.1.1 Soporte para JPA*

Java Persistence API, es la API de persistencia desarrollada para la plataforma Java EE incluida en el estándar EJB 3.0. Anteriormente a esta versión se utilizaban beans de entidad. Se encarga del mapeo entre una tabla relacional y su objeto Java. Proporciona métodos para manejar la persistencia de un Bean de Entidad, permite añadir, eliminar, actualizar y consultar así como manejar su ciclo de vida.

El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos y utilizar los elementos de persistencia como objetos planos de Java (POJOs).

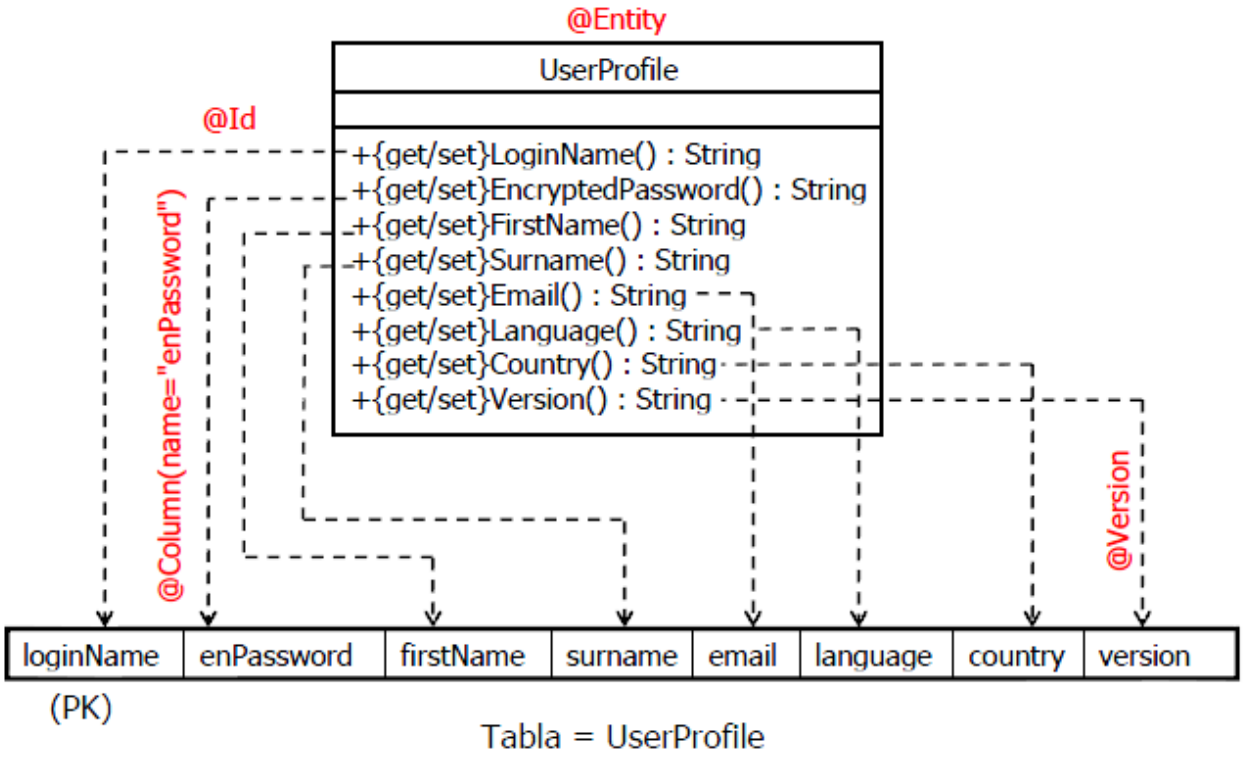



***Figura 3.10*** *Ejemplo de mapeo JPA*

### *3.4.4.1.2 Soporte para JDBC*

Java DataBase Connectivity API se presenta como una colección de interfaces Java y métodos de gestión de manejadores de conexión hacia cada modelo específico de base de datos.

Un manejador de conexiones hacia un modelo de base de datos en particular es un conjunto de clases que implementan las interfaces Java y que utilizan los métodos de registro para declarar los tipos de localizadores a base de datos (URL) que pueden manejar.

Para utilizar una base de datos particular, el usuario ejecuta su programa junto con la biblioteca de conexión apropiada al modelo de su base de datos, y accede a ella estableciendo una conexión, para ello provee el localizador a la base de datos y los parámetros de conexión específicos. A partir de allí puede realizar con cualquier tipo de tareas con la base de datos a las que tenga permiso: consulta, actualización, creación, modificación y borrado de tablas, ejecución de procedimientos almacenados en la base de datos, etc.

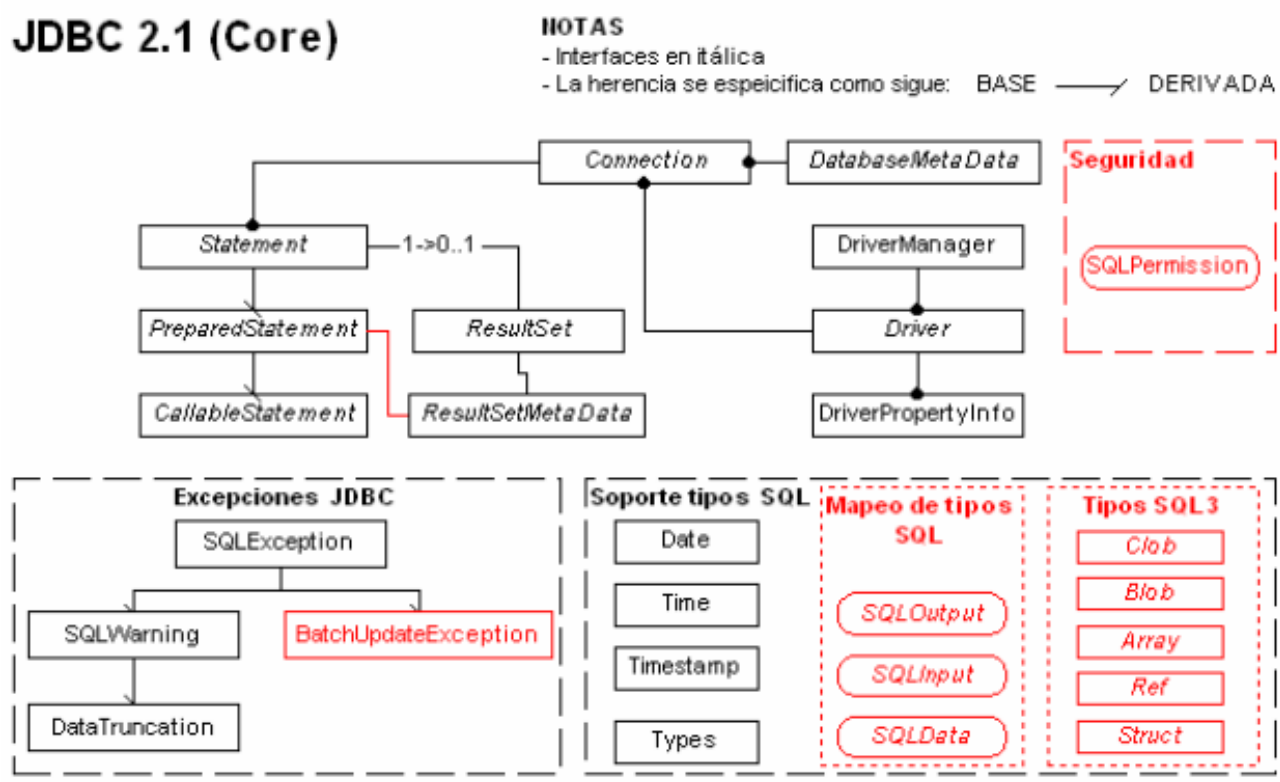



***Figura 3.11*** *Diagrama de clases de JDBC*

## *3.4.4.2 Soporte a la capa de lógica de negocio*

### *3.4.4.2.1 Control de transacciones JTA*

J2EE proporciona una API estándar que hace más fácil la coordinación de los distintos sistemas en lo que se refiere a operaciones de tipo transaccional, llamada JTA (*Java Transaction API),* en la cual no es necesario que se escriba código.

El JTA consta de tres elementos: una interfaz de aplicación de transacción de alto nivel, una interfaz de administración para transacciones de alto nivel para el servidor de aplicaciones, y un mapeo de Java del estándar X/Open XA que permite a un administrador de recursos de transacciones participar en una transacción global mediante un manager.

- La aplicación cliente puede ser sencillo como un objeto simple o completo como una aplicación entera. La aplicación de cliente usan datos mediante transacciones con el fin de garantizar que su interacción con los datos se controla y se pueda corregir fácilmente si algo sale mal.

- El manager de transacción es el responsable de coordinar la aplicación de cliente y el resto de componentes de procesamiento de transacciones.

- El administrador de recursos es el componente responsable de coordinar el acceso a los recursos afectados por una transacción. Puede ser tan simple como un controlador JDBC que gestiona el acceso a una base de datos y sus tablas, o la cola que controla la conexión de JMS.

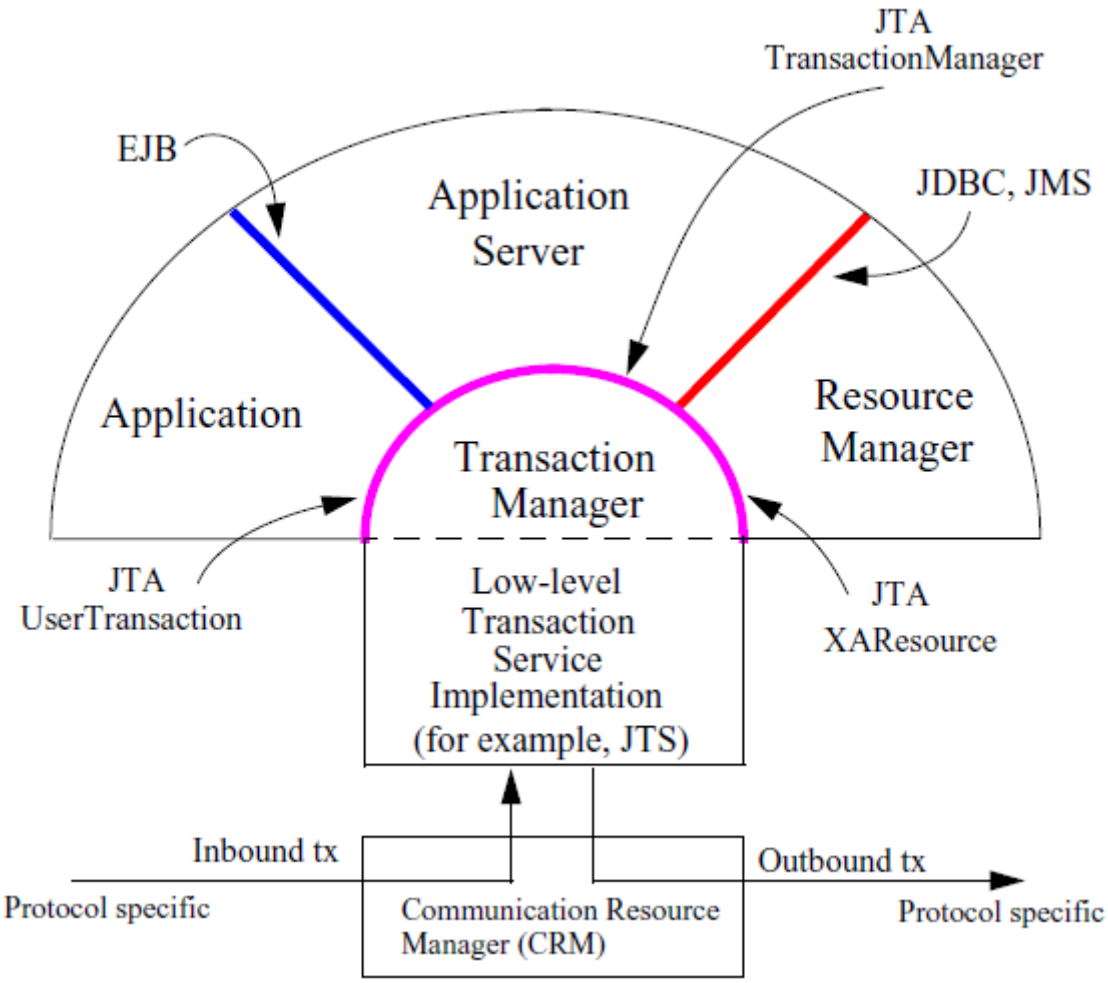



***Figura 3.12*** *Arquitectura JTA*

### *3.4.2.2 Servicios de mensajería JMS*

Los productos de mensajería en el software se están convirtiendo en un componente esencial para la integración de los componentes del programa.

La API JMS de Java proporciona una interfaz para aplicaciones que requieren los servicios de un sistema de mensajería. Un sistema de mensajería permite que los mensajes que contienen texto, objetos u otro tipo de mensajes se envíen y reciban de forma asíncrona.

Una implementación de un sistema de mensajería que cumpla con la API de JMS se llama proveedor de JMS. Glassfish contiene un proveedor de JMS.

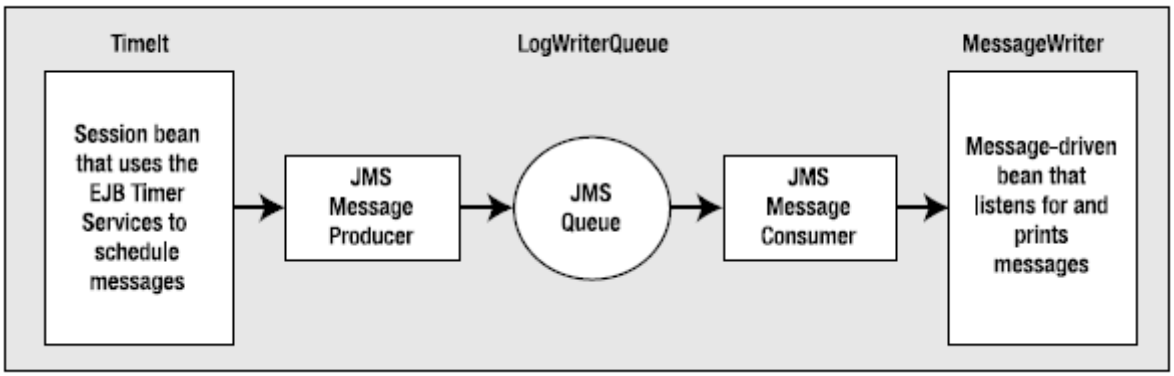



***Figura 3.13*** *Funcionamiento de JMS*

Como podemos observar en la figura anterior, el bean de Sesión del EJB son los encargados de enviar el mensaje. Estos se conocen como clientes JMS. El sistema de mensajes es el encargado de guardar los mensajes hasta que puedan ser entregados al destinatario. Hay dos tipos de destinos en JMS:

- **Point-to-point.** Es un tipo de mensajería en el que una aplicación envía un mensaje directamente a una cola de mensajes específicos. Sólo una aplicación recupera el mensaje de la cola. En este caso solo participan dos clientes, el que envía y el que recibe. Un ejemplo de este sistema de mensajería sería una cola de entrada de pedidos.

- **Publish/Subscribe.** En este tipo de mensajería, el remitente del mensaje publica un mensaje. Cualquiera que esté interesado en recibir mensajes sobre un tema determinado se subscribe al mensaje. Como resultado, cuando el emisor envía el mensaje con el tema, cada suscriptor recibe una copia del mensaje. En este caso el mensaje tiene un remitente y, muchos destinatarios.

Uno de los aspectos más importantes de este tipo de mensajes es que el remitente no sabe nada sobre los abonados. No sabe cuántos suscriptores hay, dónde están ubicados o lo que hacen con los mensajes. Un ejemplo de este tipo de mensajes sería cuando una empresa enviase un mensaje que indica que les ha llegado un pedido, las demás secciones obtienen el mensaje y saben lo que tienen que hacer a partir de ahí.

## *3.4.3 Servicios de comunicación*

### *3.4.3.1 Soporte para CORBA*

Es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. Permite realizar llamadas a objetos remotos, independiente del lenguaje de programación en que fueron programados. CORBA es una especificación, no es un software o aplicación. Hay muchas implementaciones de CORBA las cuales son conocidas como ORB (Object Request Broker).

En un sentido general, CORBA "envuelve" el código escrito en otro lenguaje, en un paquete que contiene información adicional sobre las capacidades del código que contiene y sobre cómo llamar a sus métodos. Los objetos que resultan, pueden entonces ser invocados desde otro programa (u objeto CORBA) desde la red. En este sentido CORBA se puede considerar como un formato de documentación legible por la máquina, similar a un archivo de cabeceras, pero con más información.

CORBA utiliza un lenguaje de definición de interfaces (IDL) para especificar las interfaces con los servicios que los objetos ofrecerán. CORBA puede especificar a partir de este IDL, la interfaz a un lenguaje determinado, describiendo cómo los tipos de dato CORBA deben ser utilizados en las implementaciones del cliente y del servidor. Implementaciones estándar existen para Ada, C, C++, Smalltalk, Java, Python, Perl y Tcl.

Al compilar una interfaz en IDL se genera código para el cliente y el servidor (el implementador del objeto). El código del cliente sirve para poder realizar las llamadas a métodos remotos. Es el conocido como *stub*, el cual incluye un *proxy* del objeto remoto en el lado del cliente. El código generado para el servidor consiste en unos *skeletons* que el desarrollador tiene que rellenar para implementar los métodos del objeto.

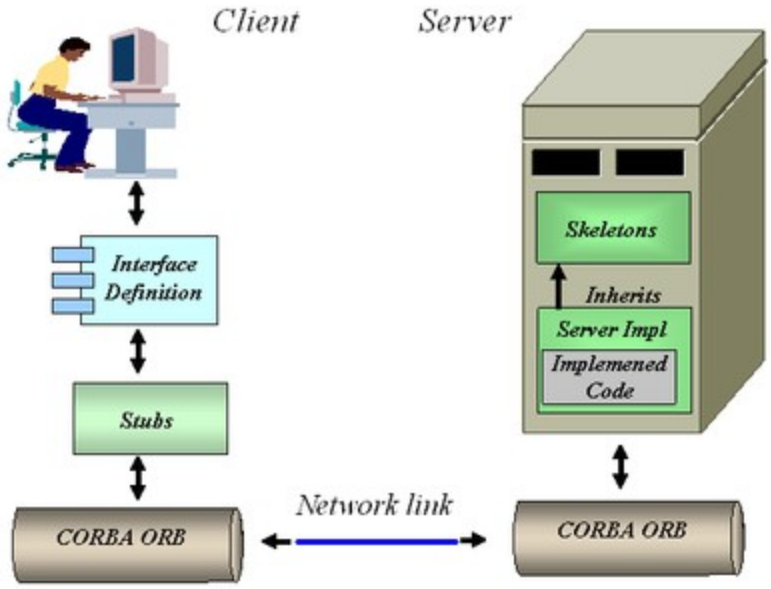



***Figura 3.14*** *Esquema de CORBA*

### *3.4.3.2 Soporte para Servicios Web*

Los Servicios Web o Web Services son una API para permitir exponer servicios a través de la Web. Permite que aplicaciones Web interactúen dinámicamente con otras aplicaciones, utilizando para ello estándares abiertos como XML (Extensible Markup Language), UDDI (Universal Description, Discovery and Integration) y SOAP (Simple Object Access Protocol). Las funciones que pueden ser realizadas por los web services pueden ir desde simples intercambios de información hasta complicados procesos de negocios. Se puede encapsular su lógica de negocio mediante web services y exponerlas para que los clientes las consuman a través de la web.

Los Web Services permiten realizar invocaciones a procedimientos remotos, tanto en redes pequeñas como una Intranet empresarial como en redes grandes como Internet. Esto es posible porque están basados en un protocolo simple para realizar las invocaciones. Este protoco estándar lo se denomina SOAP, y está creado por la W3C, basado en XML.

SOAP está compuesto por cuatro componentes: un envoltorio que define un framework para describir los mensajes y cómo estos deben ser procesador, un conjunto de reglas para codificar instancias de tipos de datos definidos por las aplicaciones, una convención de cómo representar invocaciones remotas y sus respuestas y una convención para vincular el intercambio de mensajes con un protocolo de transporte. El protocolo de transporte es http.

Utilizar http como protocolo de transporte facilita el uso de la infraestructura Web ya existente prácticamente en todo empresa, para el intercambio de información o publicación de servicios de la empresa.

Los Web Services también incorporan WSDL (Web Services Description Language) como un lenguaje también basado en XML que permite describir los contratos de cada servicio e incluye un protocolo para recibir y enviar documentas a través de una URL conocida, o utilizando mecanismos UDDI en el caso de no conocer la URL.

Las ventajas de utilizar Web Services son las siguientes:

- Aportan interoperabilidad entre aplicaciones de software independientemente de sus propiedades o de las plataformas sobre las que se instalen.

- Los servicios Web fomentan los estándares y protocolos basados en texto, que hacen más fácil acceder a su contenido y entender su funcionamiento.

- Al utilizar http, los servicios Web pueden aprovecharse de los sistemas de seguridad firewall sin necesidad de cambiar el filtrado.

- Permiten que servicios y software de diferentes compañías ubicadas en diferentes lugares puedan ser combinados fácilmente.

- Permiten la interoperabilidad entre plataformas de distintos fabricantes por medio de protocolos estándar y abiertos.

Los inconvenientes de utilizar Web Services son los siguientes:

- Para realizar transacciones no pueden compararse en su grado de desarrollo con los estándares abiertos de computación distribuida como CORBA.

- Su rendimiento es bajo si se compara con otros modelos de computación distribuida como RMI o CORBA.

- Al utilizar http, pueden esquivar medidas de seguridad basadas en firewall.

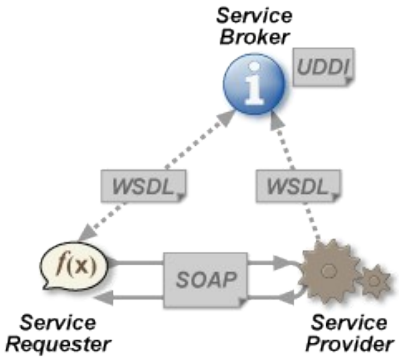



***Figura 3.15*** *Funcionamiento de Web Services*

### *3.4.3.3 Soporte para RMI*

RMI (Remote Method Invocation) es el mecanismo ofrecido en Java que permite a una aplicación poder ser invocado remotamente. RMI, al ser nativo de Java, es mucho más amigable y natural para un programador Java, permitiéndole además aprovechar las ventajas del entorno. En el RMI existen tres procesos fundamentales:

- El cliente: proceso que invoca un método en un objeto remoto.
- El servidor: proceso que posee el objeto remoto.
- Registro de objetos: obtiene acceso a objetos remotos utilizando su nombre.

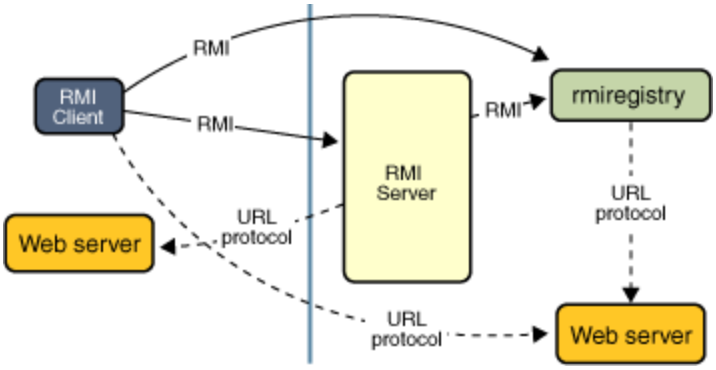



***Figura 3.16*** *Funcionamiento de RMI*

Por medio de RMI, un programa Java puede exportar un objeto. A partir de esa operación este objeto está disponible en la red, esperando conexiones en un puerto TCP. Un cliente puede entonces conectarse e invocar métodos. La invocación consiste en el "marshaling" de los parámetros, luego se sigue con la invocación del método. Mientras esto sucede el llamador se queda esperando por una respuesta. Una vez que termina la ejecución el valor de retorno es serializado y enviado al cliente. El código cliente recibe este valor como si la invocación hubiera sido local.

La arquitectura de RMI es la siguiente:

- **Capa de aplicación.** Se corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda java.rmi.Remote.

- **Capa de presentación.** Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

- **Capa de sesión.** Es la de referencia remota, y es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo stream (stream-oriented connection) desde la capa de transporte.

- **Capa de transporte.** Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP (Java Remote Method Protocol), que solamente es "comprendido" por programas Java.

| CLIENTE | OBJETO REMOTO | OSI |
|---|---|---|
| Cliente invocando método en el objeto remoto | Objeto remoto ofrece el servicio | Capa de aplicación |
| Stub | Skeleton | Capa de presentación |
| JRMP | JRMP | Capa de sesión |
| TCP | TCP | Capa de transporte |
| IP | IP | Capa de red |
| Interfaz de hardware | Interfaz de hardware | Capa de vínculos de datos |

***Figura 3.17*** *Comparación de RMI con el modelo OSI*

### *3.4.3.4 Servicios de nombrado JNDI*

La interfaz de Nombrado y Directorio Java (JNDI) es una API para servicios de directorio. Esto permite a los clientes buscar objetos y nombres a través de un nombre siendo independiente de la implementación.

La API JNDI se usa por RMI a las APIs de J2EE para buscar objetos en una red. La API suministra lo siguiente:

- Un mecanismo para asociar un objeto a un nombre.
- Una interfaz de búsqueda de directorio que permite consultas generales.
- Una interfaz de enventos que permite a los clientes determinar cuando las entradas de directorio han sido modificadas.

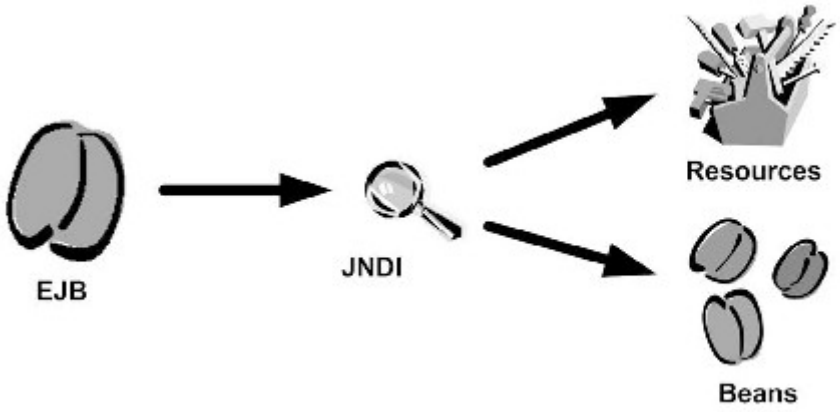



***Figura 3.18*** *JNDI puede ser usado para buscar beans en EJB*

## *3.4.4 Soporte a la capa de presentación*

### *3.4.4.1 Soporte para Servlet*

La palabra servlet deriva de otra anterior, applet, que se refería a pequeños programas que se ejecutan en el contexto de un navegador web. Por contraposición, un servlet es un programa que se ejecuta en un servidor y generan respuestas a solicitudes de clientes. En una aplicación J2EE, el cliente se pone en contacto con una página JSP que se comunida con el servlet. El servlet llama a un bean de sesión que interactúa con otros beans tanto de sesión como de entidad. Los beans de entidad utilizarán JDBC para comunicarse con la base de datos. Pero los servlets también pueden hacer una llamada directamente a la base de datos.

Los servlets son a menudo vistos como una alternativa a CGI. Un programa CGI ha sido una forma popular de añadir contenido dinámico a páginas Web en lenguaje Perl. Además de las limitaciones del lenguaje presenta varios problemas. Cada solicitud requiere un nuevo proceso para manejar la situación y tener que manejar este volumen de solicitudes presenta problemas para los servidores.

Por contra, los servlets se escriben en Java y uno solo puede manejar las solicitudes de todos los usuarios, con lo que no se genera la sobrecarga al no tener que crear un servlet cada vez que se solicita. Los servlets se inicializan una vez y luego persisten. Además, como utilizan lenguaje java pueden utilizar objetos java que trabajen en el servlet.

Un servlet se ejecuta dentro de una aplicación llamada contenedor de servlets dentro de una Java Virtual Machine en el servidor. El propio contenedor ya se encarga de funciones como el ciclo de vida de cada servlet.

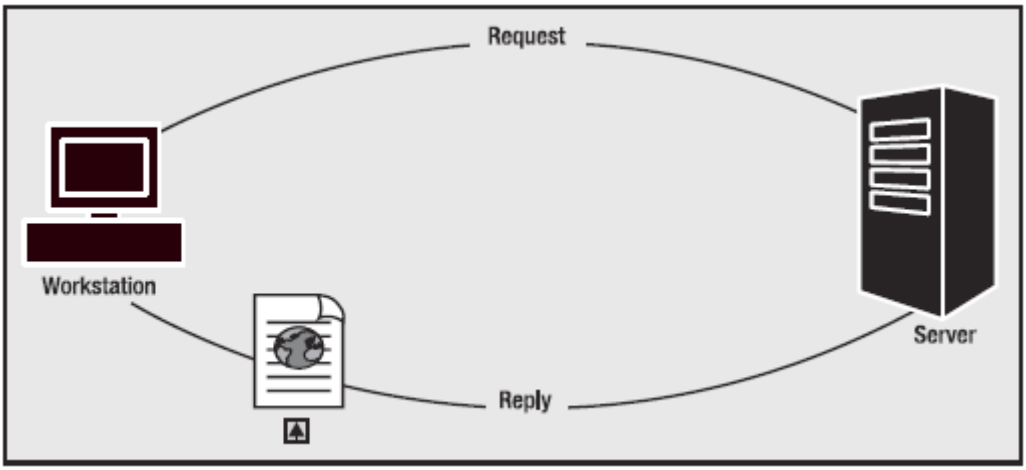



***Figura 3.19*** *El navegador envía una petición. El servidor identifica la petición y muestra la página web.*

### *3.4.4.2 Soporte para JSP*

Los servlets es una herramienta importante para responder a las solicitudes de los clientes. Sin embargo, no son las mejores herramientas para la generación de contenidos destinados a explorador web. JSP permite agregar muchas funcionalidades a una página HTML. Las páginas JSP son documentos de texto basado en HTML con trozos de código Java llamados scriplets que son incrustados en el documento HTML.

Cuando la página JSP se despliega, el contenido se ejecuta de adentro hacia fuera, un servlet se crea basándose en las etiquetas scriplets incrustados en el código Java. Todo esto sucede de manera transparente para el usuario.

No hay que confundir JSP con Javascript, este último también basado en código Java, puede ser incluido dentro de una página web y el código es ejecutado por el propio navegador. JSP es parecido pero el código se compila y se ejecuta en el servidor, y se envía el HTML resultante al navegador. Además las páginas JSP son ligeras y rápidas y proporciona una gran cantidad de escalabilidad de las aplicaciones.

JSP permite crear contenido tanto estático como dinámico. Ya que el contenido en que se basa una página JSP, no es necesario que sea creado por un programador, sino que puede ser desarrollado por un diseñador web.

Dado la ejecución de JSP se basa en servlets, JSP ofrece el mismo apoyo para la gestión de sesión como los servlets. Además también puede cargar y llamar componentes EJB, acceso a datos, realizar cálculos, etc.

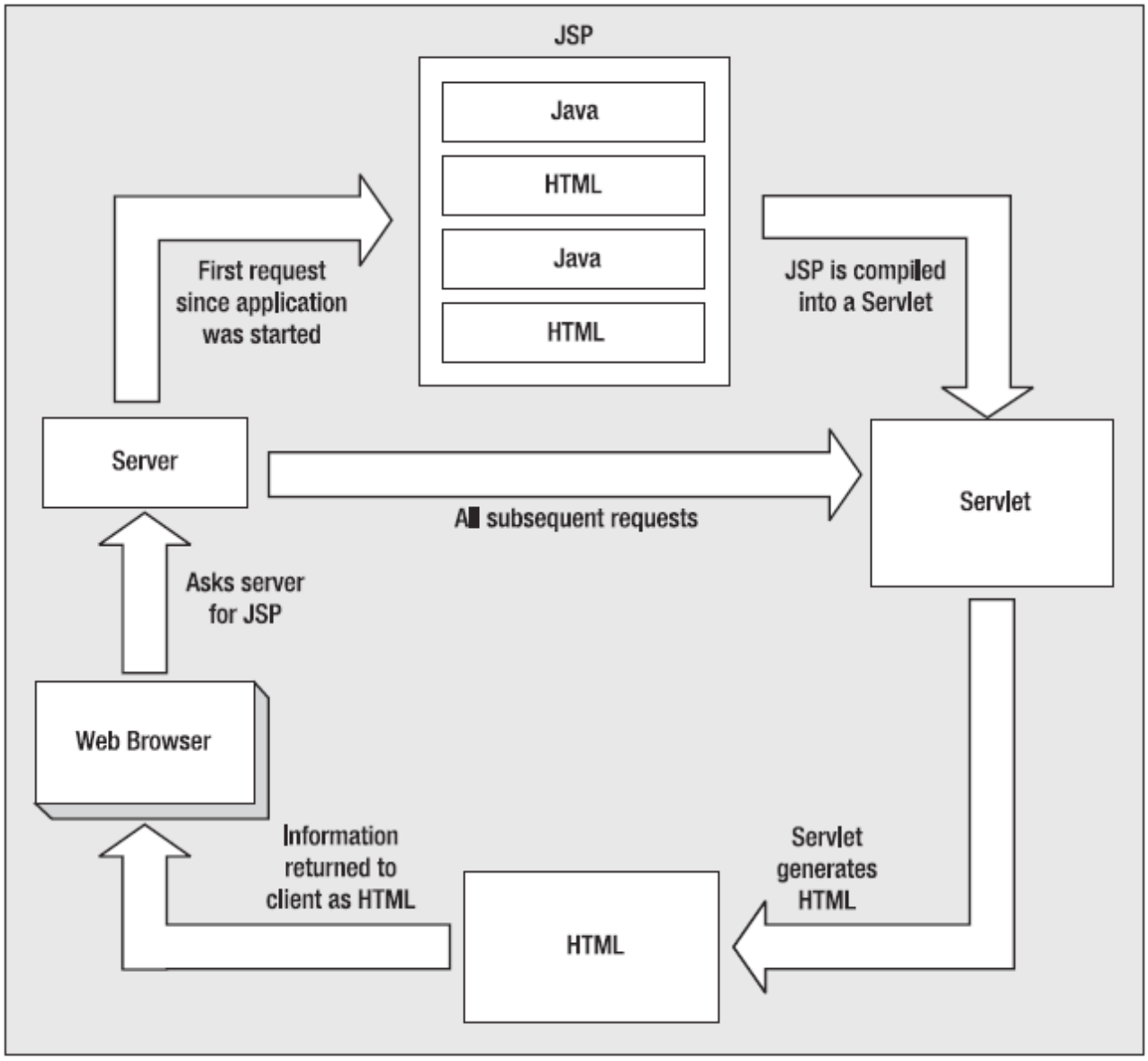



***Figura 3.20*** *Funcionamiento de JSP*

### *3.4.4.3 Soporte para JSF*

JSF es una tecnología que trata de proporcionar una interfaz robusta y variada para aplicaciones web. JSF se utiliza junto con servlets y JSP. Cuando se usa solo servlets o JSP para generar la presentación, la interfaz de usuario se limita a lo que se puede implementar en HTML con componentes como listas, casillas de verificación, botones, etc. JSF proporciona una API para la creación de interfaces de usuario.

Los componentes de JSF son componentes de interfaz de usuario que se pueden colocar fácilmente juntos para crear una interfaz de usuario del lado del servidor. La tecnología JSF también hace que sea fácil conectar los componentes de interfaz de usuario.

Los componentes de interfaz del mismo usuario pueden ser usados para generar código de presentación para cualquier dispositivo. Por lo tanto, si el dispositivo cliente cambia, solo es necesario cambiar la configuración del sistema, sin necesidad de cambiar nada de código.

La propia página de configuración de Glassfish está hecha con JSF junto con AJAX en un proyecto llamado woodstock, cuyos componentes se pueden utilizar con la API Visual Web JSF en el entorno Netbeans.

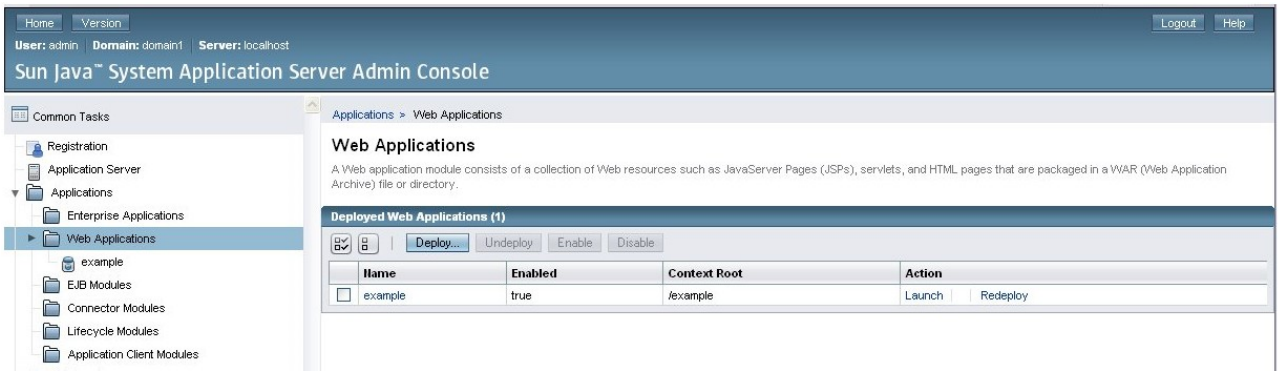



***Figura 3.21*** *Componentes JSF en la página de configuración de Glassfish*

Como alternativa a la interface de woodstock también encontramos otros de código libre como ICEfaces, que combina JSF con AJAX, o Oracle ADF.


***Figura 3.22*** *Interfaz de ICEfaces*

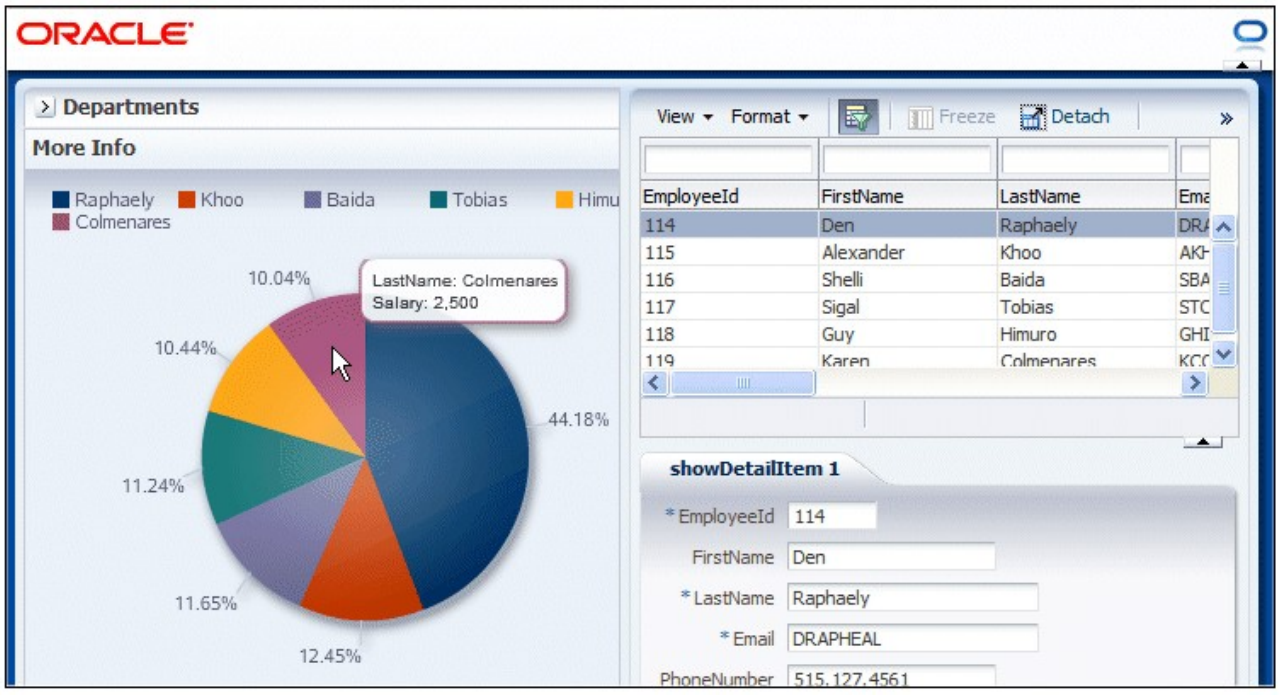



***Figura 3.23*** *Interfaz de Oracle ADF*

### *3.4.5 Otros niveles de soporte*

Una aplicación distribuida, además de dar respuesta a las necesidades concretas para la que ha sido diseñada, necesita resolver cuestiones técnicas que aumentan la dificultad del desarrollo.

#### *3.4.5.1 Seguridad*

La seguridad es un componente vital en las aplicaciones empresariales. J2EE ofrece mecanismos integrados de seguridad más seguros de los que se puedan añadir manualmente.

J2EE proporciona mecanismos de autentificación y autorización de acceso a los usuarios, así como recursos para el acceso anónimo si se necesita. Es posible especificar quien tiene acceso a cada método de un Enterprise Bean.

Otro aspecto importante de la seguridad es el de la transmisión segura de información, que se consigue mediante la encriptación. La mayor parte de los servidores de aplicación soportan comunicaciones seguras a través de SSL (Secure Sockets Layer).

#### *3.4.5.2 Soporte para Concurrencia*

El uso de un sistema distribuido implica que varios usuarios pueden estar accediendo a la misma información a la vez, problema de concurrencia. El servidor de aplicaciones se hace cargo del problema, por defecto el servidor no permite acceder a los objetos a mas de un thread, hasta que un cliente no haya terminado de ejecutar no puede ejecutar ningun otro cliente el mismo objeto.

### *3.4.5.3 Escalabilidad*

A veces pueden suceder sucesos como un inesperado aumento de usuarios en un momento dado, aumento de carga en el hardware, etc. La arquitectura J2EE proporciona una gran flexibilidad para adaptarse a cambios como el rencimiento o la capacidad de cambio.

La arquitectura de la aplicación de n capas permite aplicar potencia adicional cuando es necesario. También es posible dividir en más niveles los puntos específicos que presentan más problemas, sin afectar a otros componentes de la aplicación.

Además, algunos servidores de aplicaciones como Glassfish contienen sistemas para mejorar el rendimiento y la disponibilidad de las aplicaciones.

### *3.4.6 Soporte para SOA*

#### *3.4.6.1 JBI*

Java Business Integration es una especificación desarrollada bajo la JCP (Java Community Process) para el desarrollo de componentes de servicios. Esta especificación fue pensada para la crear una arquitectura interconectable para componentes que implementan servicios de tipo proveedor o componentes consumidores de servicio. OpenESB es una implementación open source basada en JBI.

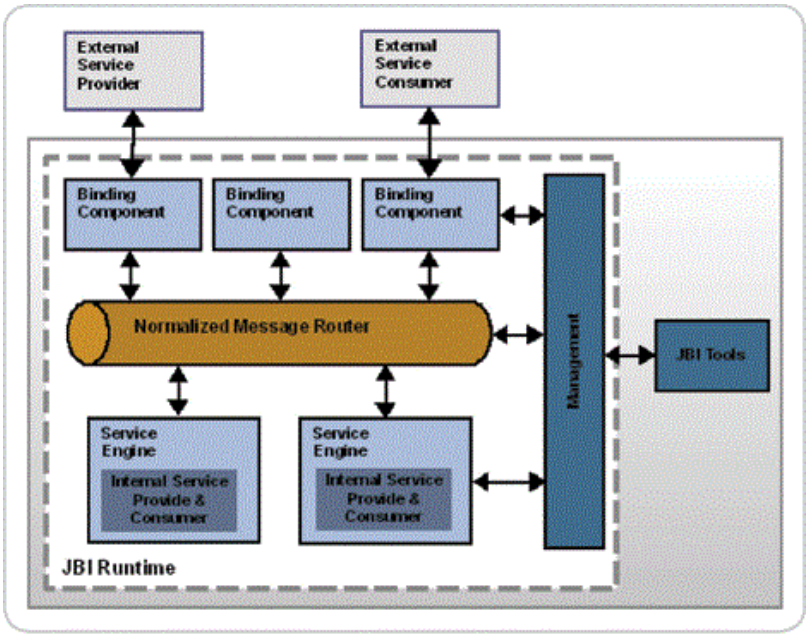



***Figura 3.24** JBI*

#### *3.4.6.2 Open ESB*

OpenESB es un implementación de un Enterprise Servicie Bus (ESB) basado en la especificación JBI, iniciada por Sun Microsystems. Permite integrar fácilmente aplicaciones empresariales y webservices como aplicaciones compuestas débilmente acopladas. Esto permite componer y recomponer de manera fluida y rápida aplicaciones compuestas, con todas las ventajas de una verdadera Arquitectura Orientada a Servicios.

OpenESB se ejecuta sobre el servidor de aplicaciones Glassfish/Sun Application Server e incluye una gran variedad de componentes JBI (Java Business Integration) y un motor de servicio WS-BPEL 2.0. Además incluye su propio motor BPEL (BPEL SE).

Open ESB amplía las capacidades de la implementación de JBI con services engines, binding components, herramientas y servicios de administración y monitorización adicionales. La integración con el entorno de desarrollo NetBeans permite el despliegue de aplicaciones de una manera rápida y eficiente, con una serie de facilidades como ayuda en el desarrollo, control de errores y pruebas.

## *3.5 Resumen general de J2EE*

J2EE proporciona una solución global, estandarizada y sencilla al desarrollo de aplicaciones corporativas. La funcionalidad del negocio se implementa mediante EJB y los clientes interactuan con una interface. El servidor de aplicaciones lleva a cabo tareas fundamentales como control de seguridad, concurrencia, etc. proporcionando además una serie de interfaces estándar para el acceso a bases de datos, correo electrónico, etc.

Respecto al acceso a los Enterprise Beans, J2EE proporciona numerosas alternativas como usar aplicaciones a medida, HTML, etc. Además, el protocolo utilizado hace posible que las aplicaciones cliente estén escritas en cualquier lenguaje que soporte CORBA.

Para soportar el desarrollo de HTML/XML, J2EE proporciona los APIs Servlets y Java Serves Pages, que hacen portable al código utilizado para generarlos entre diversas plataformas.

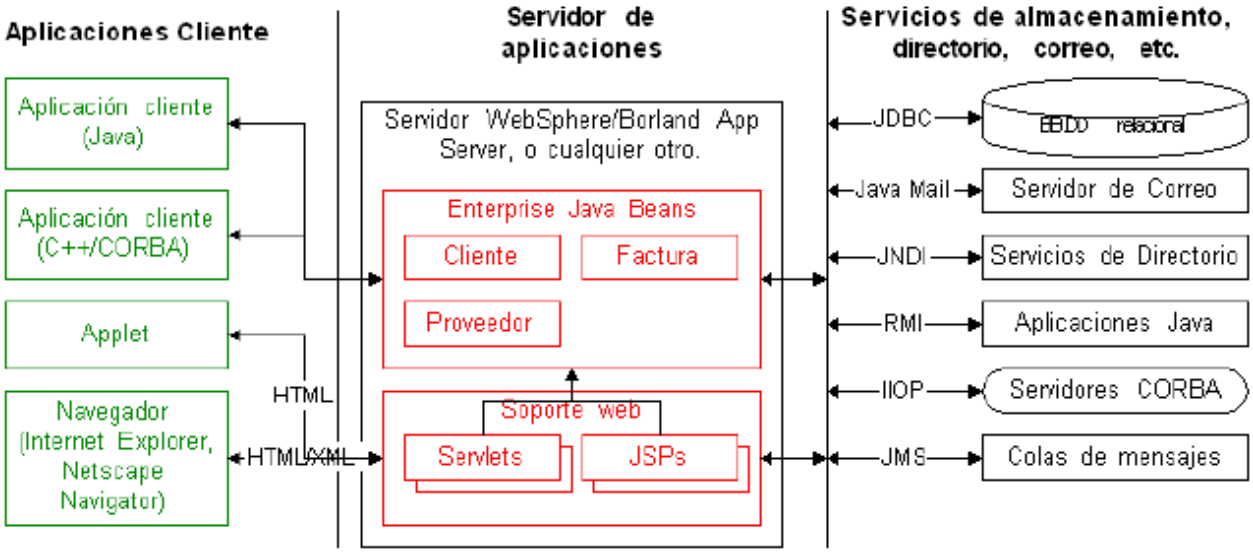



***Figura 3.25*** *Interacción entre diferentes interfaces, EJB y Servlets/JSPs, servidor de aplicaciones y servicios externos*

## *3.6 Servidores de Aplicaciones Java EE 5 certificados*

Un servidor de aplicaciones se trata de un dispositivo de software que proporciona servicios de aplicación a las computadoras cliente. Un servidor de aplicaciones gestiona las funciones de lógica de negocio y de acceso de datos a la aplicación. Los principales beneficios son la centralización y la disminución de la complejidad en el desarrollo de aplicaciones. Algunos servidores de J2EE son los siguientes:

- **JonAS.** Servidor de aplicaciones de código abierto de ObjectWeb.

- **WebLogic Application Server.** Servidor de aplicaciones desarrollado por BEA Systems posteriormente adquirida por Oracle Corporation

- **Jboss.** Desarrollado inicialmente por JBoss Inc y adquirido posteriormente por Red Hat. Existe una versión de código abierto soportada por la comunidad y otra empresarial.

- **Sun Java System Application Server Platform Edition 9.0.** Servidor de aplicaciones basado en GlassFish.

- **Apache Geronimo 2.0.** Servidor de aplicaciones de Apache Software Foundation.

- **GlassFish**, Servidor de aplicaciones de código abierto de Sun.

A continuación detallaremos los más importantes.

### *3.6.1 JBoss*

JBoss es un servidor de aplicaciones J2EE de código abierto implementado en Java puro. Al estar basado en Java, JBoss puede ser utilizado en cualquier sistema operativo que lo soporte. Los principales desarrolladores trabajan para una empresa de servicios, JBoss Inc., adquirida por Red Hat en Abril del 2006, fundada por Marc Fleury, el creador de la primera versión de JBoss. El proyecto está apoyado por una red mundial de colaboradores. Los ingresos de la empresa están basados en un modelo de negocio de servicios.

JBoss es una implementación Open-Source de un contenedor EJB; es mediante este tipo de productos que es posible llevar acabo un desarrollo con EJB's. La gran gamma de productos en J2EE han sido comercializados como Java Application Servers.

Como se observa en la siguiente imagen un Java Application Server se encuentra compuesto por dos partes: un Servlet Engine y un EJB Engine, dentro del Servlet Engine se ejecutan exclusivamente las clásicas aplicaciones de Servidor (JSP's y Servlets) , mientras el EJB Engine (Contenedor) es reservado para aplicaciones desarrolladas alrededor de EJB's.

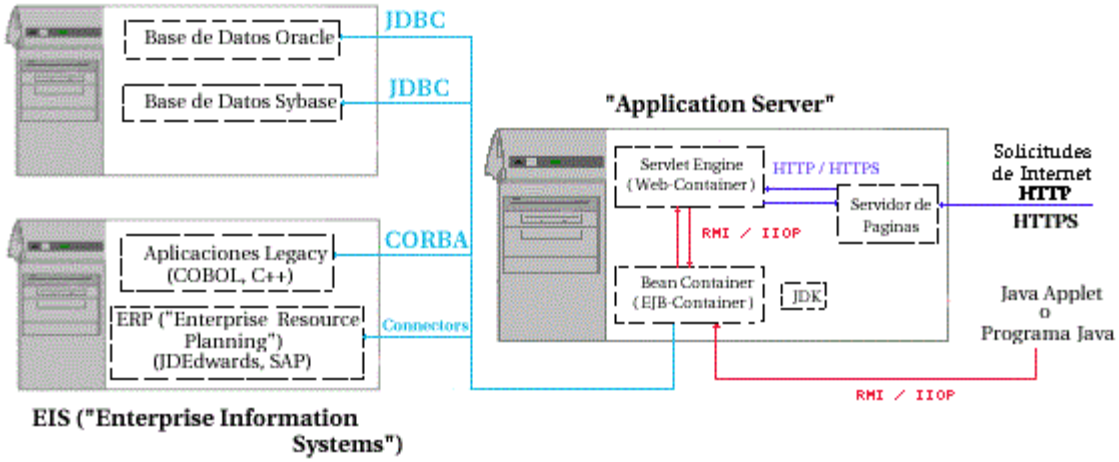



***Figura 3.26*** *Esquema de JBoss*

### *3.6.1.1 Servidor de aplicaciones JBoss*

JBoss AS es el primer servidor de aplicaciones de código abierto, preparado para la producción y certificado J2EE, ofreciendo una plataforma de alto rendimiento para aplicaciones de e-business. Combinando una arquitectura orientada a servicios revolucionaria con una licencia de código abierto, JBoss AS puede ser descargado, utilizado, incrustado y distribuido sin restricciones por la licencia.

### *3.6.1.2 Servicios de Jboss*

- **EJB 3.0.** Implementa la especificación inicial de Enterprise JavaBeans 3.0.

- **JBoss AOP.** Está orientado a trabajar con Programación Orientada a Aspectos. Esto permitirá añadir fácilmente servicios empresariales (transacciones, seguridad, persistencia) a clases Java simples.

- **Hibernate.** Es un servicio de persistencia objeto/relaciones y consultas para Java. Hibernate facilita a los desarrolladores crear las clases de persistencia utilizando el lenguaje Java - incluyendo la asociación,herencia, polimorfismo y composición y el entorno de colecciones Java.

- **JBoss Cache.** Es un producto diseñado para almacenar en caché los objetos Java más frecuentemente accedidos de manera que aumente de forma notable el rendimiento de aplicaciones e-bussines. Eliminando accesos innecesarios a la base de datos, JBoss Cache reduce el tráfico de red e incrementa la escalabilidad de las aplicaciones.

- **JBoss IDE.** Brinda una IDE Eclipse para el JBoss AS. De esta forma la depuración y otras tareas asociadas al desarrollo de aplicaciones puede ser realizadas desde el entorno de Eclipse.

- **JBoss jBPM.** PM es una plataforma para lenguajes de procesos ejecutables, cubriendo desde gestion de procesos de negocio (BPM) bajo workflow hasta orquestacion de servicios. Actualmente jBPM soporta tres lenguajes de procesos sobre una sola tecnologia: Maquina Virtual de Procesos(PVM).

- **JBoss Portal.** Es una plataforma de código abierto para albergar y servir un interfaz de portales Web, publicando y gestionando el contenido así como adaptando el aspecto de la presentación.

- **Tomcat.** Es un contenedor de servlets utilizado como la implementación de referencia oficial para las tecnologías de JavaServer Pages y Java Servlet.

- **JBoss Mail Server, MQ, Messaging.** Servicios de mail, mensajería en Java Message Service y un servicio de mensajería robusto y de alto rendimiento que soporta esquemas de integración que van desde simples mecanismos entre aplicaciones hasta grandes Arquitecturas de Servicios (SOAs) y Canales de Servicios Empresariales (ESBs).

- **JBoss Forum (Jforum).** Es un foro de discusión en Java similar en prestaciones y aspecto a phpBB. Tiene licencia BSD, soporte para bases de datos MySQL, PostgreSQL y HSQLDB, una interfaz altamente configurable, soporte para un número ilimitado de grupos de usuarios con permisos distintos, notificaciones por email de actividad en los posts, soporte para internacionalización, etc.

### *3.6.1.3 Ventajas de JBoss*

Las ventajas de JBoss son múltiples.

- El producto está siendo constantemente actualizado y cuenta con buena documentación.
- Producto de licencia de código abierto.
- Cumple los estándares.
- Confiable a nivel de empresa.
- Orientado a arquitectura de serviccios.
- Soporte completo para Java Management eXtensions.
- Ayuda profesional 24 horas.

### *3.6.1.4 Desventajas de JBoss*

La principal y muy importante desventaja de JBoss es el precio, ya que no todas las empresas pueden permitirse gastarse el muy elevado precio de la licencia Jboss. También existe una versión gratuita de menor rendimiento y prestaciones.

Comparándolo con Glassfish:

- Glassfish tiene una mejor consola de administración.
- Despliegue en caliente es más fiable en Glassfish.
- La nueva versión de Glassfish soporta Java EE 6 mientras que Jboss utiliza Java EE 5.
- WebServices funcionan mejor sobre Glassfish.
- Glassfish tiene un entorno más amigable para desarrolladores. Además si se usa NetBeans es más sencillo.

### *3.6.2 Apache Geronimo*

Apache Geronimo es un servidor de aplicaciones de código abierto desarrollado por la Apache Software Foundation y distribuido bajo la licencia Apache. Geronimo es actualmente compatible con la especificación Java Enterprise Edition (Java EE) 5.0. IBM ha proporcionado un apoyo considerable al proyecto a través de la comercialización, las contribuciones de código, y la financiación de varios proyectos.

El núcleo de Geronimo es Java EE agnostic. Su único objetivo es la gestión de bloques de construcción de Geronimo. Se caracteriza por un diseño arquitectónico que se basa en el concepto de Inversión of Control (COI), lo que significa que el núcleo no tiene dependencia directa de cualquiera de sus componentes. El núcleo es un framework para los servicios que controla el ciclo de vida de servicio y el registro y está basado en Java EE.

La mayoría de los servicios de Geronimo se agregan y configuran a través de GBeans para convertirse en una parte del global de las aplicaciones de servicio. GBean es la interfaz que conecta los componentes al núcleo. La interfaz de GBeans permite cambiar entre dos contenedores de servlets, por ejemplo, Jetty o Tomcat sin afectar a toda la arquitectura utilizando una interfaz de Gbeans. Esta arquitectura flexible permite a los desarrolladores de Geronimo para integrar varias existentes, probados en los proyectos de software de código abierto.

#### *3.6.2.1 Servicios de Apache Geronimo*

- **Apache Tomcat.** Servidor de HTTP y contenedor de Servlet que soporta Java Servlet 2.5 y JavaServer Pages (JSP) 2.1.

- **Jetty**. Tiene las mismas funcionalidades que Tomcat, es una alternativa.

- **Apache ActiveMQ.** Servicio de mensajería open source basado en Java Message Service (JMS) 1.1.

- Apache OpenEJB. Open source Enterprise JavaBeans (EJB) Container System y EJB Server que soporta Enterprise JavaBeans 3.0, incluye Container Managed Persistence 2 (CMP2) y EJB Query Language (EJBQL).

- **Apache OpenJPA.** Implementación de open source Java Persistence API (JPA) 1.0

- **Apache ServiceMix.** Open source Enterprise Service Bus (ESB) y componentes basados en el Java Business Integration (JBI) standard JSR 208.

- **Apache Axis y Apache Scout.** Axis es una implementación Simple Object Access Protocol (SOAP), Scout es una implementación JSR 93 (JAXR).

- **Apache CXF.** Estructura de servicios web con variedad de protocolos como SOAP, XML/HTTP, RESTfull HTTP o COBRA.

- **Apache Derby.** Relational database management system (RDBMS) con soporte para Java Database Connectivity (JDBC)

- **Apache WADI.** Clustering, balanceo de carga y solución de error para la aplicación web.

- **MX4J**. Java Management Extensions, suministra las herramientas para la gestión y seguimiento de las aplicaciones, los objetos del sistema, los dispositivos y las redes de servicios orientados.

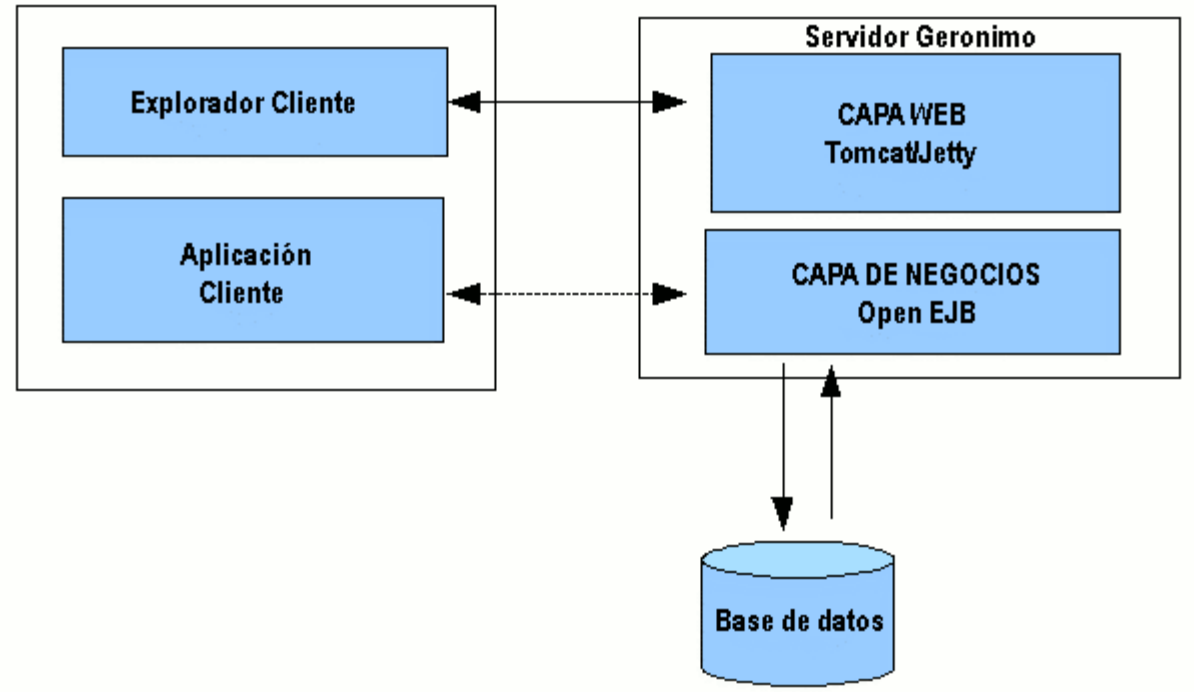



***Figura 3.27*** *Ejemplo de Geronimo utilizando Tomcat/Jetty y OpenEJB*

### *3.6.2.2 Ventajas de Apache Geronimo*

Las principales ventajas de este servidor de aplicaciones son las siguientes.

- Fácil de usar.
- Open Source.
- El tiempo de ejecución es bueno para satisfacer las necesidades de desarrolladores, administradores e integradores de sistemas.

- Integración completa con Eclipse.

- Actualizaciones frecuentes con nuevas características y corrección de errores.

- Existe una comunidad que desarrolla nuevas herramientas constantemente.

### *3.6.2.3 Desventajas de Apache Geronimo*

- Comparándolo con Glassfish, Apache Geronimo es un servidor de aplicaciones más sencillo y con menos aplicaciones que el primero. Además el rendimiento de Geronimo es menor.

- Glassfish aprovecha todas las funcionalidades de JEE5 (o 6 en la última versión) y es más sencillo de utilizar gracias a la compatibilidad con Netbeans.

- Geronimo es mucho menos utilizado que Glassfish o Jboss, lo cual significa que el apoyo de la comunidad será menor y es más dificil encontrar expertos que trabajen en él.

### *3.6.3 Oracle WebLogic*

Oracle WebLogic es un servidor de aplicaciones J2EE y también un servidor web HTTP desarrollado por BEA Systems posteriormente adquirida por Oracle Corporation. Puede ejecutarse en distintos sistemas operativos. La última versión de WebLogic forma parte de Oracle Fusion Middleware, que consiste en un conjunto de software de Oracle.

WebLogic server permite desarrollar y desplegar aplicaciones fiables, seguras, escalables y manejables. Dirige los detalles a nivel de sistema para que el desarrollador sólo se tenga que preocupar por la lógica de negocio y la presentación.

WebLogic puede utilizar distintas bases de datos como DB2, Microsoft SQL Server u otras bases de datos que se ajusten al estándar JDBC. Es compatible con WS-Security y cumple con los estándares de J2EE, incluyendo JEE 5 en su última versión 10.

Oracle WebLogic Server es parte de Oracle WebLogic Platform, de la cual forman parte los siguientes componentes.

- **Portal.** Servidor de comercio y personalización.

- **WebLogic Integration.** Aplicación basada en java para la integración de sistemas y conectividad.

- **WebLogic Workshop.** IDE para Java.

- **JRockit.** Máquina Virtual de Java para CPUs de Intel.

WebLogic Server incluye interoperabilidad .NET y admite las siguientes capacidades de integracióno nativa:

- Mensajería nativa JMS.

- J2EE Connector Architecture.
- Tuxedo Connector.
- Conectividad COM+ y CORBA.
- Conectividad IBM WebSphere MQ.

El modelo de seguridad de WebLogic Server incluye Separar la lógica de negocio del código de seguridad y rango completo de cobertura de seguridad para todos los componentes, sean o no J2EE.

WebLogic implementa las siguientes tecnologías en sus distintas capas.

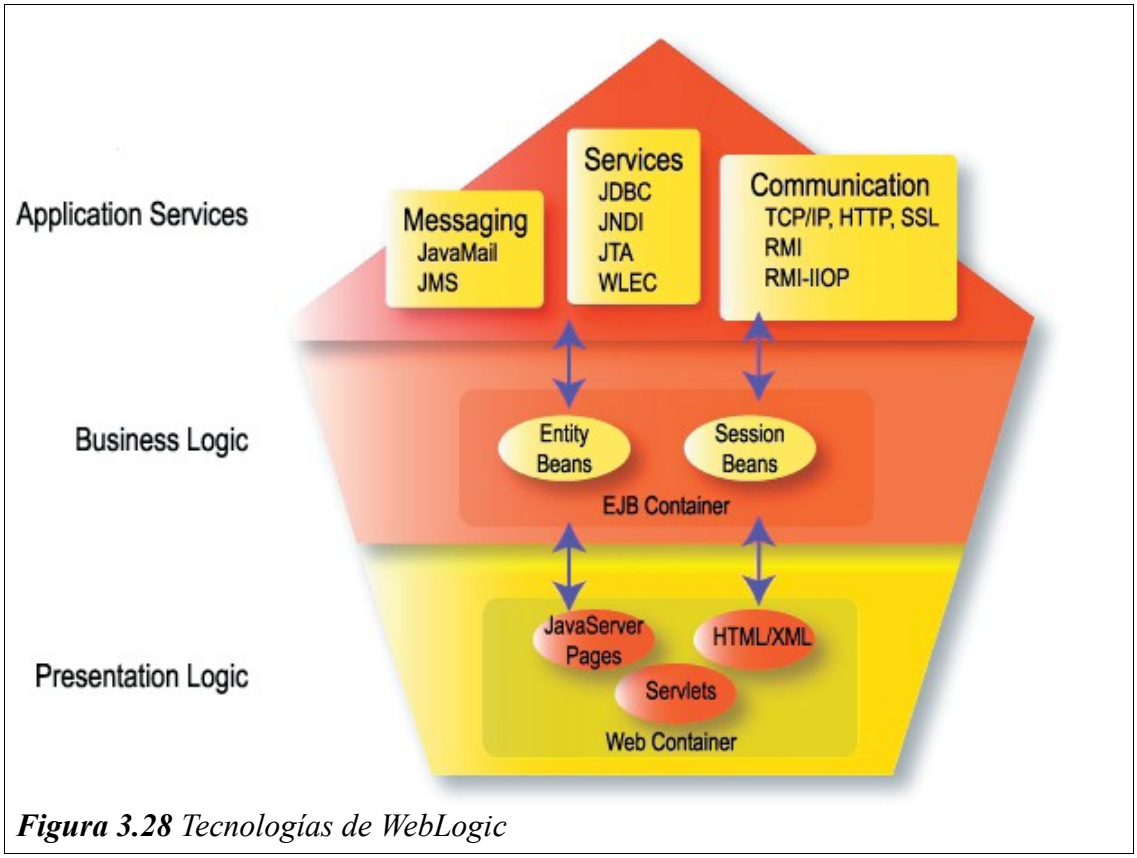



***Figura 3.28*** *Tecnologías de WebLogic*

### *3.6.3.1 Ventajas de Oracle WebLogic*

Las principales ventajas de Oracle WebLogic son las siguientes.

- Soporte de Java Enterprise para facilitar la implementación y despliegue de componentes de aplicación.

- Diversas opciones de cliente. WebLogic soporta navegadores Web y otros clientes que usen HTTP, clientes Java que usan RMI o IIOP o dispositivos móviles que usan WAP. Los conectores permiten virtualmente a cualquier cliente o aplicación trabajar con WebLogic.

- Escalabilidad. Está asegurada a través del uso de componentes EJB y otros mecanismos.

- Ofrece una Consola de Administración basada en Web para configurar y monitorizar los servicios.

- Seguridad lista para el comercio electrónico. WebLogic proporciona soporte de Secure Sockets Layer (SSL) para encriptar datos. La seguridad permite la autentificación y autorización del usuario para todos los servicios.

- Flexibilidad en el desarrollo y despliegue. WebLogic proporciona integración y soporte con las bases de datos más importantes, herramientas de desarrollo y otros entornos.

### *3.6.3.2 Desventajas de Oracle WebLogic*

Los inconvenientes más importantes que tiene WebLogic respecto a Glassfish son los siguientes.

- Glassfish continuará con la implementación Java EE como referencia y como proyecto Open Source. Esta es la desventaja más importante ya que el gasto para las empresas es muy elevado

- Glassfish cuenta con Netbeans como IDE, ya incorporado en el propio Netbeans.

- Glassfish soporta las mas recientes versiones de JSP, Java Servlets y la versión JEE 6.

## *3.7 Glassfish*

### *3.7.1 Qué es Glassfish*

El término Glassfish, traducido al español sería algo parecido como “Pez de Cristal”, es el nombre de un pez que realmente existe y vive en el agua dulce; su cuerpo es transparente, por lo que sus huesos son visibles. El nombre fue elegido debido a la transparencia que los creadores querían darle al proyecto, que utiliza una licencia *Open Source*, concretamente la licencia *Common Development and Distribution License (CDDL) v1.0 y la GNU Public License (GPL) v2.*

### *3.7.2 Para qué sirve Glassfish*

GlassFish es un servidor de aplicaciones desarrollado por Sun Microsystems que implementa las tecnologías definidas en la plataforma Java EE y permite ejecutar aplicaciones que siguen esta especificación. La versión comercial es denominada Sun GlassFish Enterprise Server. Soporta las últimas versiones de tecnologías como: JSP, Servlets, EJBs, Java API para Servicios Web (JAX-WS), Arquitectura Java para Enlaces XML (JAXB), Metadatos de Servicios Web para la Plataforma Java 1.0, y muchas otras tecnologías.

Glassfish además de ser un servidor de aplicaciones, es una comunidad de usuarios, que descargan y utilizan libremente Glassfish, también existen partners que contribuyen agregándole más características importantes a Glassfish. Además ingenieros y beta testers que desarrollan código y prueban las versiones liberadas para eliminar todo fallo que se encuentre, y muchos otros miembros. La comunidad fue lanzada en el año 2005 en java.net. Al igual que el pez original, la Comunidad Glassfish es transparente en cuanto a términos de entrega de código fuente, discusiones de ingeniería, agendas, datos de descarga, etc. Tú puedes tener acceso a todo ésto, además puedes formar parte de todo el proceso detrás de la comunidad Glassfish. Un ejemplo de esto es la comunidad FishCAT.

### *3.7.3 Cómo funciona un servidor de aplicaciones*

Un servidor de aplicaciones proporciona generalmente gran cantidad de funcionalidades built in de forma transparente al usuario de manera que no sea necesario escribir código fuente. Estas funcionalidades son posibles ya que los componentes se ejecutan dentro del contenedor en un espacio de ejecución virtual llamado dominio de ejecución. Su función principal es la de interponerse entre las llamadas que se hacen a los métodos de los beans y las implementaciones de los mismos, de modo que entre otras cosas puede hacer las comprobaciones para verificar si el usuario que llama al método tiene los permisos adecuados, antes de llamarlo.

### *3.7.4 Modular, Integrable y Extendible*

Glassfish dispone de una arquitectura Modular, se puede descargar e instalar solamente los módulos que se necesiten para las apps, con lo cual se minimiza el tiempo de inicio, consumo de memoria y espacio en disco.

Basándose en el modelo de componentes dinámico y completo para Java OSGi (Open Services Gateway Initiative), las aplicaciones y/o componentes de Glassfish pueden ser remotamente instalados, iniciados, actualizados, etc. sin necesidad de reiniciar el servidor.

Es posible ejecutar Glassfish dentro de una máquina virtual sin necesidad de disponer de instalar un servidor de aplicaciones. Es posible usar Glassfish como una librería más en la JVM, seleccionando solo lo que se necesita y probando pequeñas aplicaciones webs sin necesidad de correr todo el AppServer, teniendo en cuenta las limitaciones de no tener el AppServer instalado.

### *3.7.5 Herramientas de programación*

- **AJAX**. Glassfish dispone de una tecnología y framework para Java basadas en web (Java Server Faces) llamado Woodstock, que simplifica el desarrollo de interfaces de usuario en aplicaciones J2EE en el cual se pueden incluir componentes AJAX.

- **Ruby Rails.** Se pueden ejecutar aplicaciones basadas en Ruby Rails de dos formas diferentes. La primera es mediante jRuby que está incluido en la Java Platform y la segunda sería ejecutar Rails en un interprete nativo de Ruby comunicándose con Gassfish mediante CGI.

- **PHP.** Puede utilizarse PHP con la implementación Quercus PHP 5 desenvolupada por Caucho en Java.

### *3.7.6 Tecnologías de Integración*

- **TopLink Essentials.** Es la implementación de JPA (Java Persistence API) para la comunidad Glassfish. La API se proporciona un modelo de programación sencillo para las entidades persistencia de EJB y además incluye la herramienta para conectar diferentes proveedores de persistencia.

- **CORBA.** Glassfish incluye una implementación completa de CORBA. Esta aplicación ha ido mejorando con las diferentes versiones de Glassfish.

- **OpenMQ Messaging.** Glassfish incorpora una herramienta de mensajería que proporciona:

  - Mensajes entre los componentes del sistema

  - Distribución escalable de servidores de mensajería

  - Integracion de mensajes SOAP / HTTP

  - Java y C Cliente API

- **Java Business Integration.** Glassfish incluye soporte para la API JBI. Se encarga de la integración de bus y componentes de arquitectura. La implementación incluida en Glassfish proviene del proyecto OpenESB.

### *3.7.7 Historia*

- **Junio de 2005.** Primer lanzamiento del proyecto.

- **Mayo de 2006.** Primera versión que soporta la especificación Java EE 5.

- **Mayo de 2007.** El proyecto Project SailFin se anuncia en JavaOne como un subproyecto bajo GlassfFish. SailFin es un proyecto que añade nuevas funcionalidades, como el servlet de Session Initiation Protocol (SIP).

- **Septiembre de 2007.** Aparece la versión 2 (también conocida como Sun Java System Application Server 9.1) con capacidades de cluster y nuevas características de interconexión entre servicios web.

- **Diciembre de 2008.** Sun Microsystems y la comunidad lanza GlassFish 2.1 (Sun GlassFish Enterprise Server 2.1), el que sirve como la base para el proyecto Sailfin SIP AppServer project (también conocido como Sun Communication Application Server).

- **Diciembre de 2009.** Aparece la versión 3 que soporta la especificación Java EE 6.

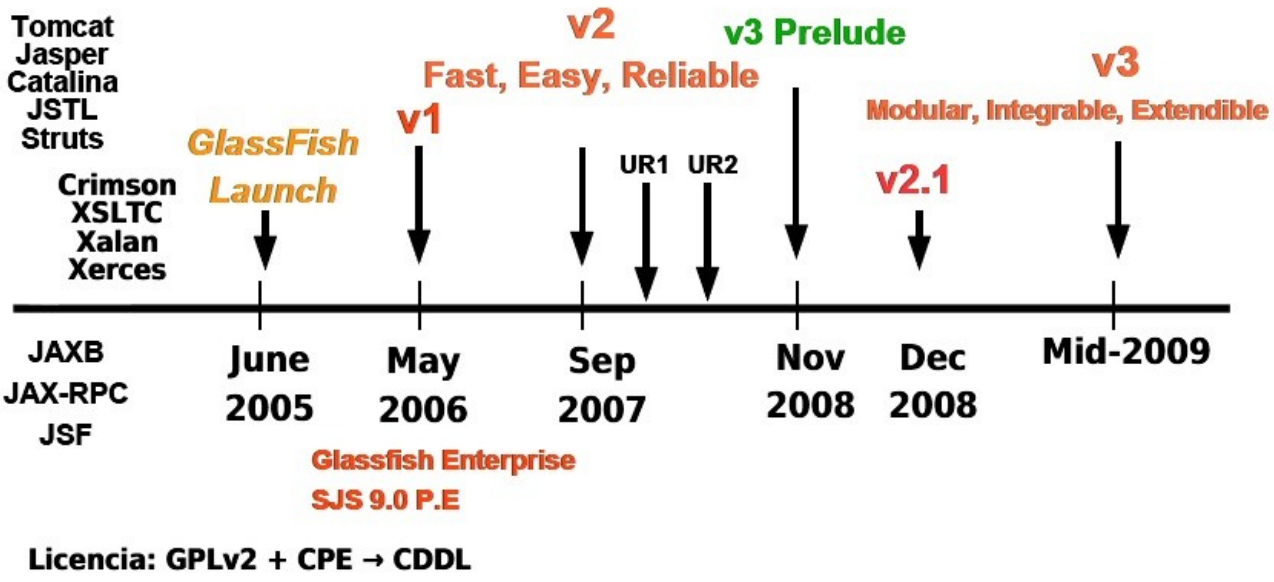



***Figura 3.29*** *Evolución de Glassfish*

### *3.7.8 Diferencia entre versiones de Glassfish*

- **Glassfish v1**: Después de un año, ésta fue la primera versión que fue liberada, conjuntamente con la liberación de Java EE 5. El principal objetivo de ésta versión, fue desarrollar un servidor de aplicaciones totalmente compatible con Java EE 5, y lo lograron, recibiendo excelentes reseñas de analistas. A la vez que se liberaba la primera versión de Glassfish, también se lanzaba un producto correspondiente de Glassfish bajo el Sun Java System 9.0 Platform Edition. La principal diferencia entre la versión Open Source y el producto de Sun fue: marca de Sun, mejor instalador, drivers de DataDirect JDBC e indemnización limitada. Todo lo demás era exactamente lo mismo.

- **Glassfish v2**: ésta versión fue liberada en Septiembre del 2007, junto con algunas actualizaciones, éstas incluían reparación de bugs y algunos parches. El principal enfoque de la versión v2 fue agregar varias características empresariales. Las tres palabras clave que resumen ésta versión son: Rápido, Fácil y Fiable.

- **Glassfish v2.1**: versión liberada en Diciembre del 2008, dónde se repararon más de 500 problemas. Permite el uso de SailFin 1.0 e incluye muchísimas mejoras de calidad. Las características principales de esta versión son:

  - Java EE5

  - Java Web Technologies (Servlet 2.5, JSP 2.1, JSF 1.2)

  - Metro Web Services Stack

  - .NET 3.0 Web Services Interoperability

  - EJB 3.0

  - JPA 1.0 (TopLink)

  - Grizzly (Java NIO)

  - CORBA

- **Glassfish v3**: Esta versión tiene como principales características: altamente modular, integrable y extendible. Además de que es totalmente compatible con Java EE 6. Características de esta versión:

  - Java Web Technologies (Servlet 3.0, JSP 2.2, JSF 2.0)

  - Metro Web Services Stack

  - .NET 3.5 Web Services Interoperability

  - EJB 3.1

  - JPA 2.0 (EclipseLink)

  - Grizzly (Java NIO)

  - CORBA

  - Arquitectura Modular Basada en OSGi

# CAPÍTULO 4

## Desarrollo de EJB's

## *4.1 Introducción de EJB*

Con EJB es posible desarrollar componentes que luego podemos utilizar y ensamblar en distintas aplicaciones. El desarrollo basado en *enterprise beans* supone un paso más en la escalabilidad que la programación orientada a objetos. Con componentes es posible reutilizar un mayor número de funcionalidades, incluso modificar éstas y adaptarlas a cada entorno de trabajo sin tocar el código del componente desarrollado. Un componente o *enterprise bean* es una especie de objeto tradicional con un conjunto de servicios adicionales soportados en tiempo de ejecución por el contenedor de componentes, o *EJB*. El EJB es una especie de sistema operativo donde residen los componentes. Podemos ver un componente como un objeto remoto RMI que reside en un contenedor EJB que proporciona un conjunto de servicios adicionales.

## *4.2 Servicios proporcionados por el contenedor EJB*

El contenedor EJB ya incorpora unos servicios concretos sin necesidad de programar ninguna clase que los implemente. Los servicios mas importantes que proporcionan los enterprise beans 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 diferentes máquinas.

- Gestión de recursos. Gestión automática de recursos como las colas de mensajes, bases de datos, etc.

- Persistencia. Sincronización entre los datos del bean y tablas de una base de datos.

- Gestión de mensajes. Utilización de Java Message Service.

- Escalabilidad. Posibilidad de añadir clusters de servidores de aplicaciones con múltiples hosts para poder dar respuesta a aumentos de carga de la aplicación sólo con añadir hosts adicionales.

- Adaptación en tiempo de despliegue. Posibilidad de modificación de todas estas características en el momento del despliegue del bean.

## *4.3 Funcionamiento de los componentes EJB*

El contenedor EJB es un programa Java que se ejecuta en el servidor y contiene todas las clases y objetos necesarios para el correcto funcionamiento de los enterprise beans.

El funcionamiento básico es el siguiente. El cliente que realiza peticiones al bean y el servidor que contiene el bean están ejecutándose en máquinas virtuales Java distintas. El contenedor EJB proporciona un EJBObject al cliente, que hace de interfaz. Cualquier petición del cliente se hace 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 a la base de datos. El propio contenedor EJB ya se encarga de comprobar las cuestiones de permisos, abrir y cerrar transacciones, etc.

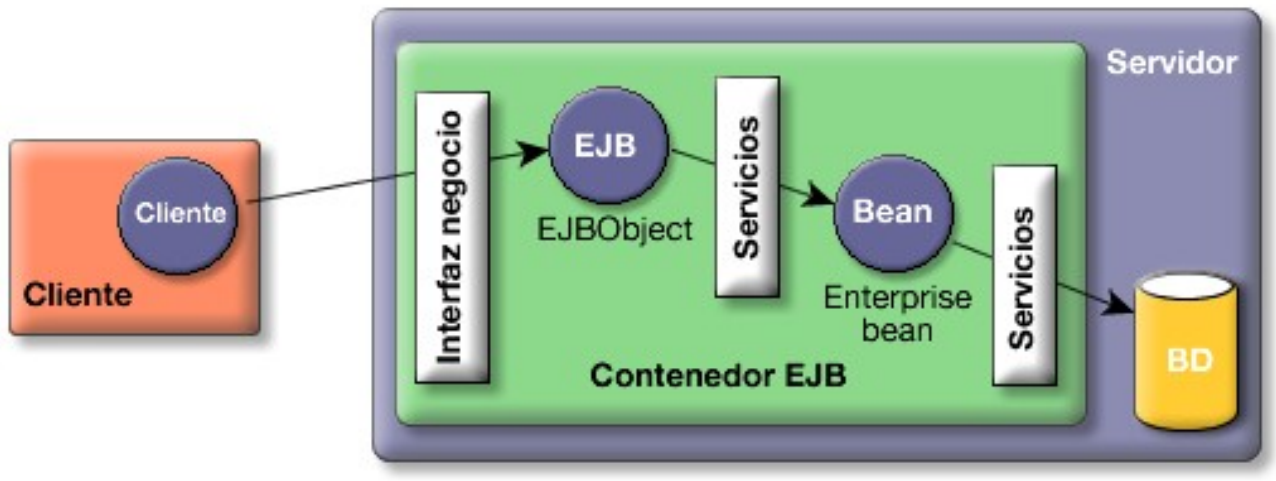



***Figura 4.1*** *Representación del funcionamiento de los enterprise beans*

## *4.4 Tipos de EJB*

- **EJB de Entidad**. Representan un objeto concreto que existe en la base de datos. Una instancia de un bean de entidad representa una fila en una tabla de la base de datos.

- **EJB de Sesión**. Gestionan el flujo de la información en el servidor. Representa un proceso o una acción de negocio. Cualquier llamada a un servicio del servidor debe comenzar con una llamada a un bean de sesión.

- **EJB dirigidos por mensajes.** Son los únicos beans con funcionamiento asíncrono. Usando el Java Messaging System, se suscriben a un tema o a una cola y se activan al recibir un mensaje dirigido a dicho tema o cola. No necesitan objetos EJBObject porque los clientes no se comunican con ellos directamente.

## Uso de anotaciones

Una anotación proporciona un recurso adicional al elemento de código al que va asociado en el momento de su compilación. Cuando se ejecuta, la clase busca estas anotaciones y determina el comportamiento a seguir con el código al que va unido.

### *4.4.1 Beans de Sesión*

Los beans de sesión representan sesiones interactivas con uno o más clientes. Pueden mantener un estado, pero sólo durante el tiempo que el cliente interactúa con el bean. Cuando termina el proceso los beans de sesión no almacenan sus datos en la base de datos. Los beans de sesión no son persistentes.

A diferencia de los beans de entidad, los beans de sesión no se comparten entre los clientes, sino que hay un bean de sesión por cada cliente. Por eso el contenedor EJB no necesita implementar mecanismos de manejo de concurrencia en el acceso a estos beans.

Dentro de los beans de Sesión encontramos dos tipos: sin estado (stateless) y con estado (stateful).

#### *4.4.1.1 Beans de sesión sin estado*

Los beans de sesión sin estado no se modifican con las llamadas de los clientes. Los métodos que están implementados en las aplicaciones cliente son llamadas que reciben datos y devuelven resultados, pero no modifican el estado del bean. Esto permite que el contenedor de EJB cree una reserva de instancias del mismo bean de sesión sin estado y pueda asignar cualquier instancia a cualquier cliente, o incluso el mismo bean a múltiples clientes.
Una de las ventajas del uso de beans de sesión frente al uso de clases Java u objetos RMI es que no es necesario escribir los métodos de los beans de sesión de forma segura para threads ya que el contenedor EJB se encarga que solo haya un thread accediendo al objeto. Para eso utiliza múltiples instancias del bean.

Cuando un cliente invoca un método de un bean de sesión sin estado, el contenedor EJB obtiene una instancia de la reserva cualquiera. Recordamos que cualquier instancia sirve ya que el bean no guarda información referida al cliente. Cuando el método termina la ejecución la instancia del bean está disponible de nuevo. Gracias a esto los beans de sesión sin estado son muy escalables y pueden ser utilizados por un gran número de clientes. Si el contenedor EJB necesita recursos y memoria en un momento dado solo tiene que destruir algunas instancias.

Los beans de sesión sin estado se usan en general para encapsular procesos de negocio, más que datos de negocio. Proporcionan un conjunto de procesos relacionados con un dominio específico del negocio. Por eso se utilizan cuando una tarea no está ligada a un cliente específico. Un ejemplo sería usarlo para enviar un e-mail que confirme un pedido on-line.

### *4.4.1.2 Beans de sesión con estado*

En un bean de sesión con estado, las variables de la instancia del bean almacenan datos específicos obtenidos durante la conexión con el cliente. Cada bean de sesión con estado almacena el estado conversacional de un cliente que interactúa con el bean. Este estado se modifica conforme el cliente va realizando llamadas a los métodos de negocio del bean y se elimina cuando el cliente termina la sesión.

La interacción del bean se divide en un conjunto de pasos en el que cada paso se añade nueva información al estado del bean. Un ejemplo específico sería por ejemplo un carrito de la compra, donde el cliente va guardando uno a uno cada cosa que compra.

El estado del bean se mantiene mientras existe el bean. A diferencia de los beans de entidad, no existe ningún recurso exterior al contenedor EJB en el que se almacene este estado.

Debido a que el bean guarda el estado conversacional con un cliente específico, no es posible crear un almacén de beans y compartirlos entre muchos clientes así como sucede con los beans de sesión sin estado. Por eso el manejo de este tipo de beans es más pesado que el de beans de sesión sin estado.

### *4.4.1.3 Anotaciones de un Bean se Sesión*

**@Stateful**: Indica que el Bean de Sesión es con estado. Atributos:

- **name**. Por defecto el nombre de la clase pero se puede especificar otro.
- **mappedName.** Si se quiere que el contenedor maneje el objeto de manera específica. Si incluimos esta opción nuestra aplicación puede que no sea portable y no funcione en otro servidor de aplicaciones.
- **description.** Descripción de la anotación.

**@Stateless:** Indica que el Bean de Sesión es sin estado y contiene los mismos atributos que Stateful.

**@Init:** Especifica que el método se corresponde con un método *create* de un EJBHome o EJBLocalHome de EJB 2.1. Sólo se puede llamar una vez a este método.
**@Remove:** Indica que el contenedor debe llamar al método cuando quiera destruir la instancia del Bean.

**@Local:** Indica que la interfaz es local.

**@Remote:** Indica que la interfaz es remota.

**@PostActivate:** Invocado después de que el Bean sea activado por el contenedor.

**@PrePassivate:** Invocado antes de que el Bean esté en estado passivate.

```
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.ejb.entidad.Productos;

@Stateless
public class ProductosFacade implements ProductosFacadeRemote {
    @PersistenceContext
    private EntityManager em;
```

***Figura 4.2** Anotación Stateless*

### *4.4.2 Beans de entidad*

Los beans de entidad utilizan conceptos o datos de negocio que pueden expresarse como nombres o 'cosas'. Es decir, los beans de entidad representan objetos reales como estudiantes o habitaciones y cosas abstractas como una reserva. Estos beans describen tanto el estado como la conducta de objetos del mundo real y permiten encapsular las reglas de datos y de negocio asociadas con un concepto específico. Por ejemplo un bean de entidad 'estudiante' encapsula los datos y reglas de negocio asociadas a un estudiante. Gracias a esto es posible manejar de forma consistente y segura los datos asociados a un concepto.

Los beans de entidad se corresponden con datos en un almacenamiento persistente como una base de datos. Las variables de instancia del bean representan los datos en las columnas de la base de datos. El contenedor EJB debe sincronizar las variables de instancia del bean con la base de datos. Los beans de entidad se diferencian de los beans de sesión en que las variables de instancia se almacenan de forma persistente.

A continuación podemos observar los pasos que realiza un bean de entidad:

- El cliente debe obtener una referencia a la instancia concreta del bean de entidad que se está buscando (una id concreta) mediante un método *finder*, los cuales están definidos en la interfaz home e implementados en la clase bean.

- El cliente interactúa con la instancia del bean usando los métodos *get* y *set*. El estado del bean se carga en la base de datos antes de procesar las llamadas a los métodos. Esto se realiza de forma automática por parte del contenedor.

- Finalmente, cuando el cliente termina la interacción con la instancia del bean sus contenidos se vuelcan en la base de datos.

Las ventajas de usar beans de entidad en lugar de acceder a la base de datos directamente son múltiples. El uso de beans de entidad nos da una perspectiva orientada a objetos de los datos y nos proporciona un mecanismo mucho más simple para acceder y modificar los datos. Por poner un ejemplo, es mucho mas sencillo cambiar el nombre de un estudiante llamando a una función que ejecutando un comando SQL contra la base de datos. Además el uso de objetos favorece la reutilización del software. Una vez se ha definido el bean de entidad, su definición puede usarse a lo largo de todo el sistema de forma consistente y simple. Esto hace que el desarrollo sea más sencillo y menos costoso.

En la versión EJB 3.0 los beans de entidad han pasado a ser substituidos por la entidad de Java Persistence API.

### *4.4.2.1 Anotaciones de un Bean de Entidad*

**@Entity**: Indica que es un Bean de Entidad.

Métodos del ciclo de vida de una entidad

**@EntityListeners:** Se pueden definir clases oyentes con métodos de ciclo de vida de una entidad.

**@ExcludeSuperclassListeners:** Indica que ningún listener de la superclase será; invocado por la entidad ni por ninguna de sus subclases.

**@ExcludeDefaultListeners:** Indica que ningún listener por defecto será; invocado por esta clase ni por ninguna de sus subclases.

**@PrePersist:** El método se llamará justo antes de la persistencia del objeto. Podría ser necesario para asignarle la clave primaria a la entidad a persistir en base de datos.

**@PostPersist:** El método se llamará después de la persistencia del objeto.

**@PreRemove:** El método se llamará antes de que la entidad sea eliminada.

**@PostRemove:** El método se llamará después de eliminar la entidad de la base de datos.

**@PreUpdate:** El método se llamará antes de que una entidad sea actualizada en base de datos.

**@PostUpdate:** El método se llamará después de que la entidad sea actualizada.

**@PostLoad:** El método se llamará después de que los campos de la entidad sean cargados con los valores de su entidad correspondiente de la base de datos. Se suele utilizar para inicializar valores no persistidos.

**@NamedQuery**: Especifica el nombre del objeto query utilizado junto a *EntityManager*.

- **name.** Nombre del objeto query.
- **query.** Especifica la consulta a la base de datos mediante lenguaje *Java Persistence Query Language.*

**@NamedQueries**: Especifica varias queries como la anterior.

**@NamedNativeQuery:** Especifica el nombre de una query SQL normal.

- **name.** Nombre del objeto query.
- **query.** Especifica la consulta a la base de datos.
- **resultClass.** Clase del objeto resultado de la ejecución de la consulta.

- **resultSetMapping.** Nombre del *SQLResultSetMapping* definido.

**@NamedNaviteQueries:** Especifica varias queries SQL.

**@SQLResultSetMapping:** Permite recoger el resultado de una query SQL.

- **name.** Nombre del objeto asignado al mapeo.
- **EntityResult[] entities().** Entidades especificadas para el mapeo de los datos.
- **ColumnResult[] columns().** columnas de la tabla para el mapeo de los datos.

**@PersistenceContext:** Objeto de la clase *EntityManager* que nos proporciona todo lo que necesitamos para manejar la persistencia.

- **name.** Nombre del objeto utilizado para la persistencia en caso de ser diferente al de la clase donde se incluye la anotación.
- **unitName.** Identifica la unidad de la persistencia usada en el bean en caso de que hubiera más de una.
- **type.** Tipo de persistencia.

**@PersistenceContexts:** Define varios contextos de persistencia.

**@PersistenceUnit:** Indica la dependencia de una *EntityManagerFactory* definida en el archivo *persistence.xml*

- name. nombre del objeto utilizado para la persistencia en caso de ser diferente al de la clase donde se incluye la anotación.
- **unitName.** identifica la unidad de la persistencia usada en el bean en caso de que hubiera más de una.

```
@Entity
@Table(name = "productos")
@NamedQueries({@NamedQuery(name = "Productos.findAll", query
= "SELECT p FROM Productos p"),@NamedQuery(name = "Productos.findByIdPro",
        query = "SELECT p FROM Productos p WHERE p.idPro = :idPro"),
        @NamedQuery(name ="Productos.findByDescPro", query = "SELECT p " +
        "FROM Productos p WHERE p.descPro = :descPro")})
```

***Figura 4.3*** *Anotaciones de Entidad*

Mapeos objeto-relacional

**@Table:** Especifica la tabla principal relacionada con la entidad.

- **name.** Nombre de la tabla, por defecto el de la entidad si no se especifica.
- **catalog.** Nombre del catálogo.
- **schema.** Nombre del esquema

**@SecondaryTable:** Especifica una tabla secundaria relacionada con el Bean de entidad si éste englobara a más de una. Tiene los mismos atributos que @Table

**@SecondaryTables:** Indica otras tablas asociadas a la entidad.

**@UniqueConstraints:** Especifica que una única restricción se incluya para la tabla principal y la secundaria.

**@Column:** Especifica una columna de la tabla a mapear con un campo de la entidad.

- **name.** Nombre de la columna.
- **unique. S**i el campo tiene un único valor.
- **nullable. S**i permite valores nulos.
- **insertable. S**i la columna se incluirá en la sentencia INSERT generada.
- **updatable. S**i la columna se incluirá en la sentencia UPDATE generada.
- **table.** Nombre de la tabla que contiene la columna.

- **length.** longitud de la columna.
- **precision.** Número de dígitos decimales.
- **scale.** Escala decimal.

**@JoinColumn:** Especifica una campo de la tabla que es foreign key de otra tabla definiendo la relación del lado propietario.

- **name.** Nombre de la columna de la FK.
- **referenced.** Nombre de la columna referencia.
- **unique.** Si el campo tiene un único valor.
- **nullable.** Si permite valores nulos.
- **insertable. S**i la columna se incluirá en la sentencia INSERT generada.
- **updatable. S**i la columna se incluirá en la sentencia UPDATE generada.
- **table.** Nombre de la tabla que contiene la columna.

**@JoinColumns:** Anotación para agrupar varias JoinColumn.

**@Id:** Indica la clave primaria de la tabla.

**@GeneratedValue:** Asociado con la clave primaria, indica que ésta se debe generar por ejemplo con una secuencia de la base de datos.

**@SequenceGenerator:** Define un generador de claves primarias utilizado junto con la anotación *@GeneratedValue*.

**@TableGenerator:** Define una tabla de claves primarias generadas.

**@AttributeOverride:** Indica que sobrescriba el campo con el de la base de datos asociado.

**@AttributeOverrides:** Mapeo de varios campos.

**@EmbeddedId:** Se utiliza para formar la clave primaria con múltiples campos.

**@IdClass:** Se aplica en la clase entidad para especificar una composición de la clave primaria mapeada a varios campos o propiedades de la entidad.

**@Transient:** Indica que el campo no se debe persistir.

**@Version:** Se utiliza a la hora de persistir la entidad en base de datos para identificar las entidades según su versión.

**@Basic:** Mapeo por defecto para tipos básicos.

**@OneToOne:** Indica que un campo está en relación con otro.

**@ManyToOne:** Indica que un campo está asociado con varios campos de otra entidad.

**@OneToMany:** Asocia varios campos con uno.

**@ManyToMany:** Asociación de varios campos con otros con multiplicidad muchos-a-muchos.

**@Lob:** Se utiliza junto con la anotación *@Basic* para indicar que un campo se debe persistir como un campo de texto largo si la base de datos soporta este tipo.

**@Temporal:** Se utiliza junto con la anotación *@Basic* para especificar que un campo fecha debe guardarse con el tipo java.util.Date o java.util.Calendar.

**@Enumerated:** Se utiliza junto con la anotación *@Basic* e indica que el campo es un tipo String.

**@JoinTable:** Se utiliza en el mapeo de una relación ManyToMany o en una relación unidireccional OneToMany.

**@MapKey:** Especifica la clave de una clase de tipo java.util.Map.

**@OrderBy:** Indica el orden de los elementos de una colección por un ítem específico de forma ascendente o descendente.

**@Inheritance:** Define la forma de herencia de una jerarquía de clases entidad, es decir la relación entre las tablas relacionales con los Beans de entidad.

**@PrimaryKeyJoinColumn:** Especifica la clave primaria de la columna que es clave extranjera de otra entidad.

```
@Entity
@Table(name = "productos")
@NamedQueries({@NamedQuery(name = "Productos.findAll", query =
"SELECT p FROM Productos p"),@NamedQuery(name = "Productos.findByIdPro",
        query = "SELECT p FROM Productos p WHERE p.idPro = :idPro"),
        @NamedQuery(name ="Productos.findByDescPro", query = "SELECT p FROM " +
        "Productos p WHERE p.descPro = :descPro")})
public class Productos implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "id_pro")
    private String idPro;
    @Column(name = "desc_pro")
```

***Figura 4.4*** *Anotaciones de Entidad*

### *4.4.2.2 Diferencias entre beans de sesión y de entidad*

Los beans de entidad se diferencian de los beans de sesión, básicamente, en que son persistentes, permiten el acceso compartido, tienen clave primaria y pueden participar en relaciones con otros beans de entidad:

- **Persistencia.** Debido a que un bean de entidad se guarda en una base de datos se dice que es persistente, el estado del bean de entidad existe más allá de la duración de la aplicación o del proceso del servidor J2EE.

  Los beans de entidad tienen dos tipos de persistencia: Persistencia Gestionada por el Bean (BMP) y Persistencia Gestionada por el Contenedor (CMP). En el primer caso el bean de entidad contiene el código que accede a la base de datos. En el segundo, la relación entre las entidades de la base de datos y el bean se describe en el dichero de propiedades del bean, y el contenedor EJB se ocupa de la implementación.

- **Acceso compartido.** Los clientes pueden compartir beans de entidad, con lo que el contenedor EJB debe gestionar el acceso concurrente a los mismos.

- **Clave primaria.** Cada bean de entidad tiene un identificador único. Este identificador único, o clave primaria, permite al cliente localizar a un bean de entidad particular.

- **Relaciones.** De la misma forma que una table en una base de datos, un bean de entidad puede estar relacionado con otros EJB. Las relaciones se implementan según si la persistencia está siendo manejada por el bean o por el contenedor. En el primer caso debemos programar y gestionar nosotros las relaciones, en el segundo caso es el propio contenedor el que se hace cargo.

### *4.4.3 Beans dirigidos por mensajes*

Estos beans permiten que las aplicaciones J2EE reciban mensajes JMS de forma asíncrona. Así el hilo de ejecución de un cliente no se bloquea cuando está esperando que se complete algún método de negocio de otro enterprise bean. Los mensajes pueden enviarse desde cualquier componente J2EE o por una aplicación o sistema JMS que no use la tecnología J2EE.

#### *4.4.3.1 Anotaciones de Beans dirigidos por mensajes*

**@Timeout:** Asigna un tiempo de ejecución a un método.

**@ApplicationException:** Excepción a enviar al cliente cuando se produzca.

### *4.4.3.2 Diferencias con los beans de sesión y de entidad*

La diferencia más importante de los beans de sesión o entidad con los beans dirigidos por mensajes es que en estos últimos los clientes no acceden a los beans mediante interfaces, sino que un bean dirigido por mensajes sólo tienen una clase bean.

En algunos aspectos, un bean dirigido por mensajes es parecido a un bean de sesión sin estado:

- Las instancias de un bean dirigido por mensajes no almacenan ningún estado conversacional ni datos de clientes.

- Todas las instancias de los beans dirigidos por mensajes son equivalentes, lo que permite al contenedor EJB asignar un mensaje a cualquier instancia. El contenedor puede almacenar estas instancias para permitir que los streams de mensajes sean procesador de forma concurrente.

- Un único bean dirigido por mensajes puede procesar mensajes de múltiples clientes.

Las variables de instancia de estos beans pueden contener algún estado referido al manejo de los mensajes de los clientes. Por ejemplo, pueden contener una conexión JMS, una conexión de base de datos, etc.

Cuando llega un mensaje, el contenedor llama al método *onMessage* del bean. Este método suele realizar un casting del mensaje a uno de los cinco tipos de mensajes de JMS y manejarlo de forma acorde con la lógica de negocio de la aplicación. El método puede llamar a métodos auxiliares, o puede invocar a un bean de sesión o de entidad para procesar la información del mensaje o para almacenarlo en una base de datos.

### *4.4.4 Anotaciones de dependencias*

**@EJB:** Mediante esta anotación el contenedor asignará la referencia del EJB indicado.

- **name.** Nombre del recurso.
- **BeanInterface.** Nombre del Bean especificado con el atributo *name* en caso de que varios Beans implementen la misma interfaz.
- **mappedName.** Si se quiere que el contenedor maneje el objeto indicado de manera específica.
- **description.** Descripción de la anotación para la inyección de dependencia.

**@Resource:** Referencia de un recurso específico.

- **name.** Nombre del recurso.
- **type.** Tipo del objeto.
- **authenticationType.** Especifica dónde se debe realizar el proceso de autenticación.
- **shareable.** Indica si el objeto se comparte.

## *4.5 Diferencias con la version EJB 2.x*

La especificación EJB 3.0 permite una fácil creación de EJBs con un desarrollo mas simple, facilitanto el desarrollo basado en pruebas y centrándose más en el modelo de persistencia basado en POJO. El API de Persistencia Java simplifica el uso de la persistencia transparente mediante anotaciones.

### *4.5.1 Stateless Session Beans*

En EJB 2.x y anteriores especificaciones, los beans de sesión requieren dos interfaces: la remota (o local) para definir los métodos de negocio, y la interface home para definir los métodos de ciclo de vida. Un bean de sesión también puede implementar varias interfaces.

Un interface remoto debe extenderse desde javax.ejb.EJBObject. La interfaz remota defina los métodos de negocio y debe seguir las reglas de RMI-IIOP. Un interfaz home debe extenderse desde javax.ejb.EJBHome y define los métodos de ciclo de vida. Debe contener el método *create()* sin parámetros para crear una instancia del contenedor de EJB. La clase de implementación del bean debe ser pública y tiene que implementar la interface javax.ejb.SessionBean.

```
// Remote Interface in EJB 2.x - StockQuote.java
package stockquote;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
public interface StockQuote extends EJBObject {
    public double getStockQuote(String Symbol) throws
        RemoteException;
}

// Home Interface in EJB 2.x - StockQuoteHome.java
package stockquote;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import javax.ejb.CreateException;
public interface StockQuoteHome extends EJBHome {
   public StockQuote create() throws RemoteException,
      CreateException;
}
```

***Figura 4.5*** *Remote y Home Interface en EJB 2.x*

Como podemos observar las diferencias son notables con la versión 3.0, la cual facilita la labor de programación. En EJB 3.0 un bean de sesión es un POJO (Plain Old Java Object) manejado por el contenedor. Además, como hemos explicado anteriormente, hace uso de anotaciones para especificar el tipo de bean. No existe el concepto de control remoto de interface home ya que la interfaz se define solo en la lógica de negocio y se indica si es local o remoto por medio de las anotaciones comentadas anteriormente.

### *4.5.2 Stateful Session Beans*

Las diferencias en los beans de sesión con estado son muy parecidas a las de beans de sesión con estado. La principal diferencia es que en el bean de sesión con estado puede tener una sobrecarga de métodos *create().* Para cada método *crear* en el interface home tiene que haber un métod *ejbCreate* in la implementación de la clase del bean.

**Session Bean – Interfaz de cliente**

En EJB 2.x el cliente obtiene un objeto de sesión mediante un nombre JNDI y luego llama al método *create()*. La diferencia con 3.0 es que en éste solo es necesario usar la anotación @EJB.

### *4.5.3 Message-Driven Beans*

En EJB 2.x implementan las interfaces javax.ejbMessageDrivenBean y javax.jms.MessageListener. Cuando el destino del mensaje lo recibe, el contenedor EJB invoca el método *OnMessage.* Igual que los beans de sesión, los beans dirigidos por mensajes también tienen un archivo de desarrollo de implementación (ejb-jar.xml), que contiene la información del nombre del bean, la clase, el destino del mensaje y su tipo. Las diferencias con EJB 3.0, como en la mayoría de casos, se basa en que el bean dirigido por mensaje es un POJO manejado por el contenedor EJB, el cual se implementa mediante anotaciones.

### *4.5.4 Entity Beans*

El bean de Entidad está manejado por un CMP (Container-Managed Persistence). En EJB 2.x los beans son objetos locales. La interfaz local del EJB es la encargada de habilitar el código para localizar y manejar los beans de entidad. La clase bean debe implementar ttodos los métodos de la interfaz javax.ejb.EntityBean. La clase de implementación del bean deben utilizar métodos de acceso para acceder al campo persistente. El contenedor maneja la persistencia y los métodos de ciclo de vida de los beans de entidad, que se declaran vacios en la clase de implementación del bean.

El contenedor sincroniza automáticamente el estado del bean de entidad con la base de datos llamando a los métodos del ciclo de vida. Los descriptores de despliegue de los beans de entidad necesitan ser escritos o generados usando herramientas.

En EJB 2.x, la configuración de los descriptores de despliegue XML para el bean de entidad CMP era un obstáculo importante. Por lo tanto, una de las ventajas importantes de la especificación EJB 3.0 es la de proteger al programador de tener que trabajar con archivos XML. Además en EJB 3.0 la clase entidad es un POJO que están marcados por la anotación @Entity y todas las propiedades en la clase entidad que no están marcadas con la anotación @Transient son consideradas persistentes.

En EJB 3.0, el API de persistencia define las anotaciones para definir los criterios de persistencia y la relación en las líneas de los conceptos de mapeo objeto-relacional. Las clases de entidad no necesitan las interfaces home ni local. Los métodos de búsqueda están especificados con la anotación @NamedQuery.

## *4.6 Ventajas de la tecnología EJB*

Las ventajas que ofrece la arquitectura Enterprise JavaBeans a los desarrolladores de las aplicaciones son las siguientes:

- **Simplicidad.** El contenedor de aplicaciones libera al programador de realizar las tareas del nivel del sistema, gracias a eso la escritura de un enterprise bean es casi tan sencilla como la escritura de una clase Java. No nos tenemos que preocupar de la seguridad, transacciones, concurrencia o programación distribuida. Como consecuencia solo debemos concentrarnos en la lógica de negocio y el dominio específico de aplicación.

- **Portabilidad de la aplicación.** Una aplicación EJB puede ser desplegada en cualquier servidor de aplicaciones que soporte J2EE.

- **Reusabilidad de componentes.** Una aplicación EJB está formada por componentes enterprise beans, cada uno de los cuales puede ser reusado a nivel de desarrollo y de aplicación cliente. Un bean desarrollado puede utilizarse en distintas aplicaciones adaptando sus características a las necesidades de cada momento. También un mismo bean puede ser usado por múltiples aplicaciones cliente.

- **Posibilidad de construcción de aplicaciones complejas.** La arquitectura EJB simplifica la construcción de aplicaciones complejas. Al estar basada en componentes y en un conjunto de interfaces, se facilita el desarrollo en equipo de la aplicación.

- **Separación de la lógica de presentación y la lógica de negocio.** Un enterprise bean encapsula típicamente un proceso de negocio, lo cual está independiente de la lógica de presentación. El bean proporciona unos datos de salida que pueden ser utilizados en distintos interfaces. Esta separación hace posible desarrollar varias lógicas de presentación para la misma lógica de negocio o cambiar los interfaces sin modificar el código de la lógica de negocio.

- **Despliegue en varios entornos.** La escalabilidad de los EJB permite el despliegue de las aplicaciones en distintos sistemas operativos, bases de datos o aplicaciones ya en marcha.

- **Despliegue distribuido.** La arquitectura EJB hace posible que las aplicaciones se desplieguen de forma distribuida entre distintos servidores de una red. El código es el mismo independientemente de si el bean se va a desplegar en una máquina o en otra.

- **Interoperabilidad entre aplicaciones.** La arquitectura EJB hace más fácil la integración de múltiples aplicaciones de diferentes orígenes.

- **Integración con otros sistemas.** Las APIs como Java Message Service hacen posible la integración de los enterprise beans con otros sistemas que no son Java.

- **Herramientas de desarrollo.** El hecho de que la especificación EJB sea un estándar hace que exista una oferta creciente de herramientas y formación para facilitar el trabajo.

A continuación vemos las ventajas que ofrece la arquitectura EJB a un posible cliente final:

- **Elección de servidor.** Debido a que las aplicaciones EJB pueden ser ejecutadas en cualquier servidor J2EE, no queda ligado solo a un tipo de servidores. Un cliente puede dejar de estar atado a un tipo de servidor y cambiarlo cuando sus necesidades lo requieran. La misma aplicación puede ser ejecutada en Jboss o Glassfish, por ejemplo.

- **Gestión de las aplicaciones.** Las aplicaciones son más sencillas de manejar debido a que existen herramientas de control más elaboradas.

- **Integración con aplicaciones y datos ya existentes.** La arquitectura EJB y otras APIs de J2EE simplifican y estandarizan la integración de aplicaciones EJB con aplicaciones no Java y sistemas en el entorno operativo del cliente que lo utilice. Por ejemplo, no hace falta cambiar un esquema de base de datos para encajar una aplicación.

- **Seguridad.** La arquitectura EJB traslada la mayor parte de la responsabilidad de la seguridad de una aplicación de el desarrollador de aplicaciones al vendedor del servidor, los cuales están más cualificados que el desarrollador para hacer segura la aplicación.

## *4.7 Inconvenientes de EJB*

A continuación enumeraremos los inconvenientes de la tecnología EJB:

- **Prueba de componentes.** La mayor parte de los componentes pueden ser comprobados fuera del contenedor, pero el servicio de de contenedor de objetos sólo puede ser comprobado dentro del contenedor.

- **Conocimiento completo de Java.** EJB es uno de los principales componentes de J2EE, por lo cual para desarrollarlo se debe tener conocimiento de otras partes de J2EE como RMI o JDBC.

- **Tiempo de Desarrollo.** Desarrollar un sistema con EJB es complejo en lo referido a tiempo de desarrollo. Puede no ser ideal para todas las empresas.

## *4.8 Tutorial de creación de EJB*

A continuación describiremos los pasos de la creación de un módulo EJB junto con los distintos beans en el entorno de desarrollo Netbeans y utilizando una aplicación web basada en Servlets, ejecutándose en el servidor de aplicaciones Glassfish. En este caso utilizamos beans de sesión de tipo Stateless.

### 1. Creación del módulo EJB

Desde Netbeans, creamos un nuevo proyecto y seleccionamos *Java EE → EJB Module*. Escribimos el nombre del módulo y seleccionamos Glassfish como servidor.

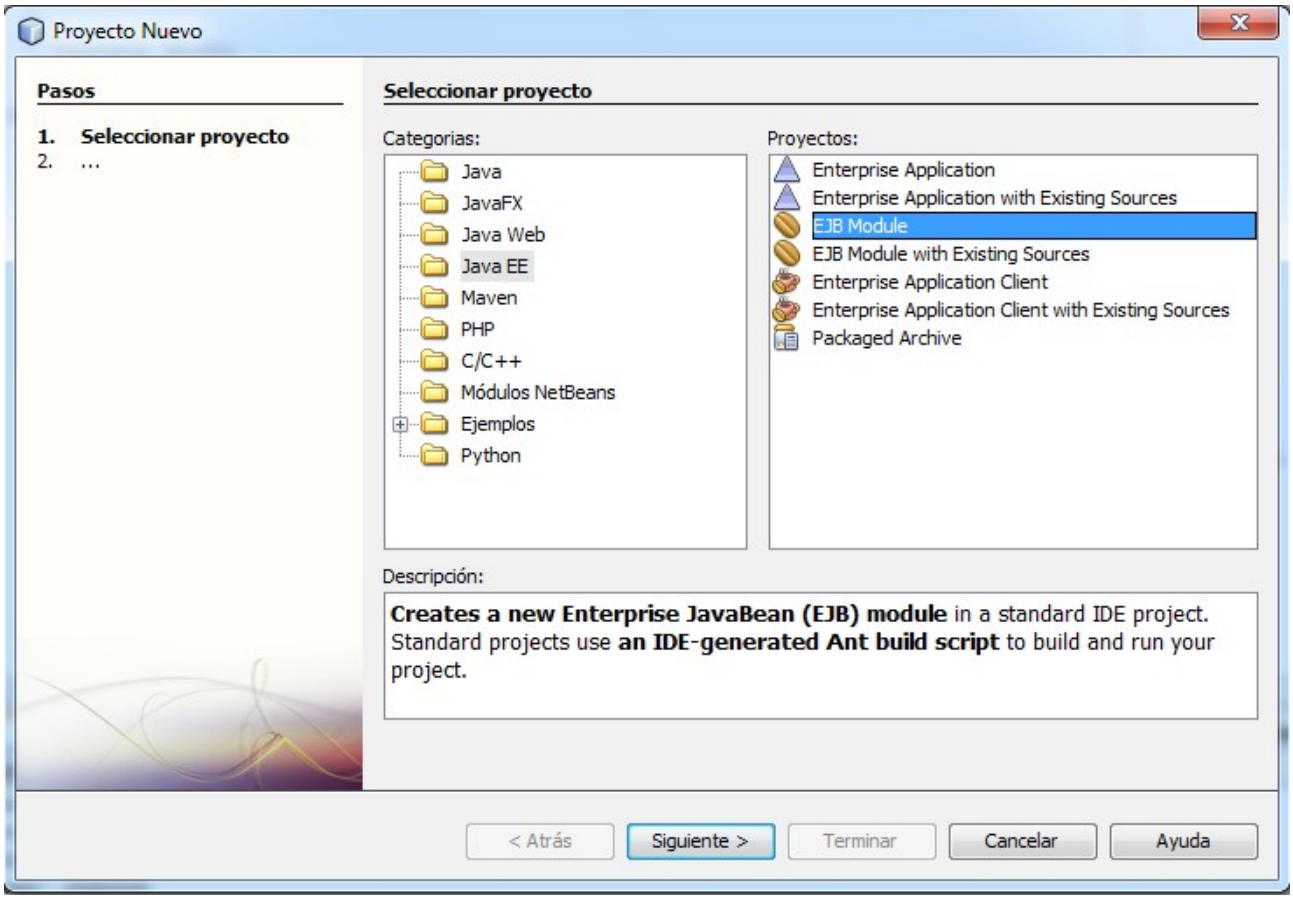



***Figura 4.6*** *Selección de Módulo EJB*

## 2. Creación de lógica de negocio

Con el módulo EJB creado procedemos a crear toda la lógica de negocio necesaria para obtener la información de las tablas de la base de datos.

## 3. Creación de bean de entidad

A continuación creamos las clases entidad que representaran a nuestras tablas de la base de datos. Para ello hacemos click derecho en el módulo EJB y seleccionamos un archivo nuevo de tipo *Persistencia → Clase entidad a partir de base de datos*.

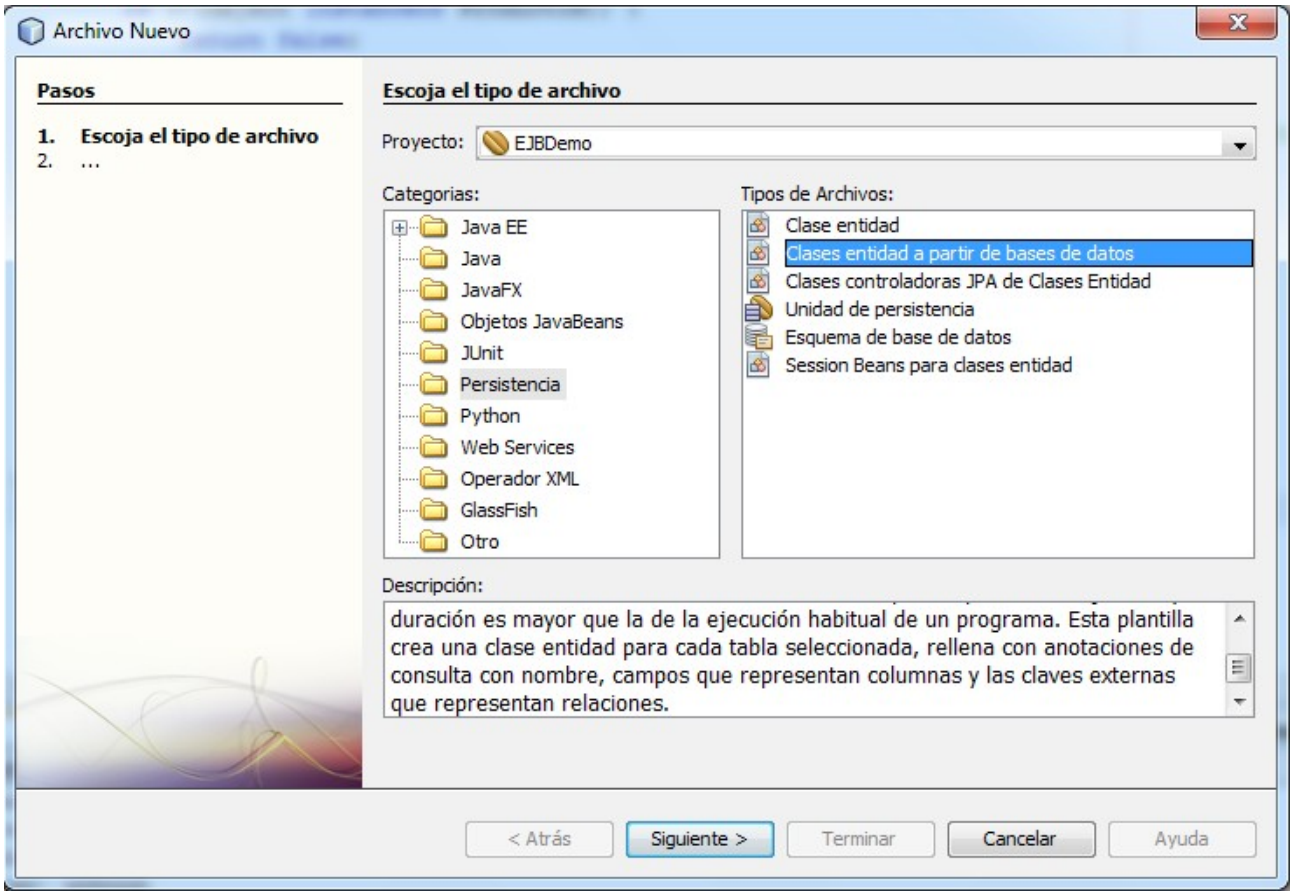



***Figura 4.7*** *Selección de bean entidad*

Hacemos click en siguiente y aparece una ventana con la opción *Nueva fuente de Datos*. Se nos muestra una ventana donde introducimos el nombre que representará nuestra fuente de datos y seleccionamos la conexión a la base de datos que elijamos.

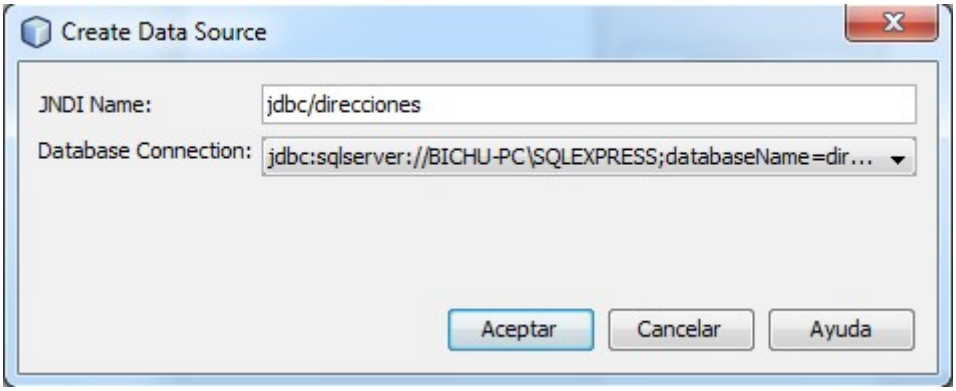



***Figura 4.8*** *Selección de Base de Datos*

Una vez realizado este paso se nos cargaran automáticamente las tablas creadas en la base da datos. Seleccionamos la tabla con la que queremos trabajar en el bean de entidad.


***Figura 4.9*** *Selección de tablas*

Hacemos click en siguiente y se nos muestra una ventana donde tenemos el nombre de la tabla y el nombre de la clase que representa a esta tabla. Ingresamos el nombre del paquete donde se guardará la clase. A continuación hacemos click en *Crear unidad de Persistencia* y se nos muestra la siguiente ventana.

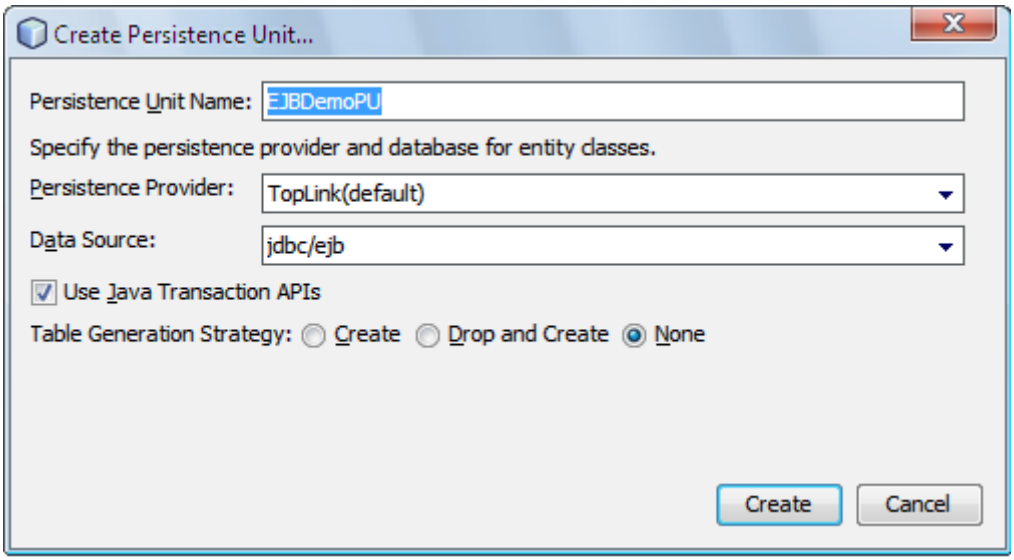



***Figura 4.10*** *Creando la unidad de persistencia*

Como podemos observar se nos crea la unidad de persistencia con el proveedor *Toplink* que se encarga de convertir los objetos Java en documentos XML.

Con está acción se nos ha creado la clase de la tabla de base de datos además del archivo persistence.xml que contiene la referencia a la base de datos.

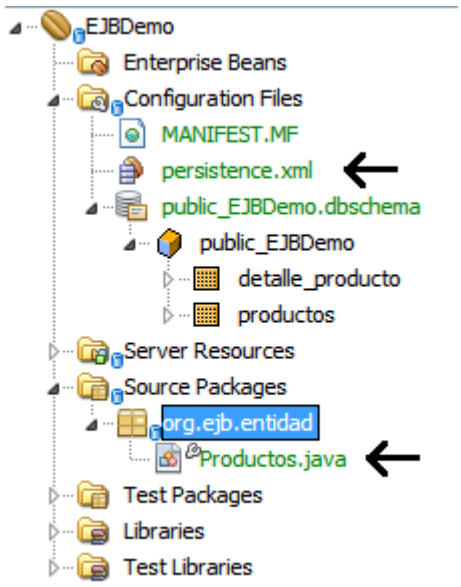



***Figura 4.11*** *Clase entidad*

## 4. Creación de beans de Sesión

A continuación crearemos las clases que implementarán los métodos para toda la lógica, que serán de tipo *SessionBean*. Hacemos click derecho al proyecto y añadimos un archivo *Persistencia → Session Beans para clases entidad.* Como podemos observar esto creará las classes en base a la clase que representa la entidad. Seleccionamos la clase que queremos añadir y activamos la opcion *Remote*.

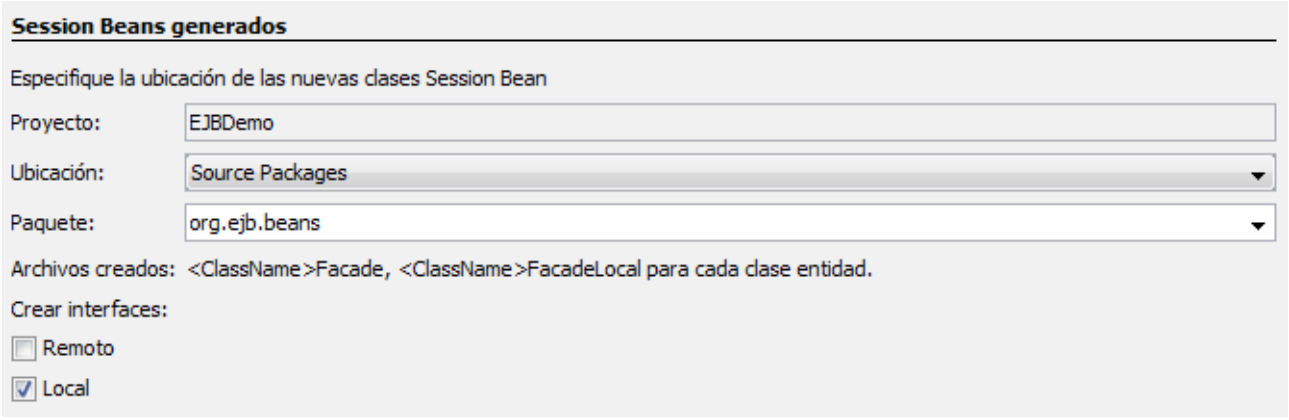



***Figura 4.12*** *Creación de Session Beans*

Automáticamente se nos ha creado las clases del tipo SessionBean creadas y los métodos de *create, edit, remove, find* y *findAll* implementados.

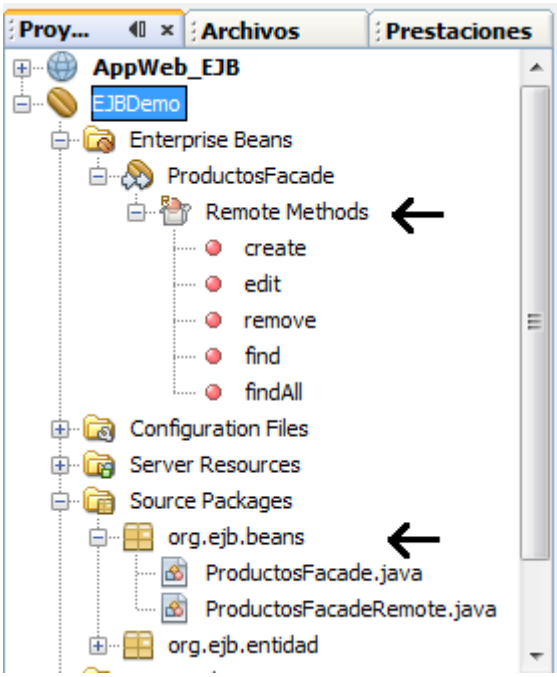



***Figura 4.13*** *SessionBean*

## 5. Añadir EJB a servidor Glassfish

A continuación hacemos click derecho en el proyecto y seleccionamos *deploy* para agregar el módulo EJB en el servidor Glassfish.

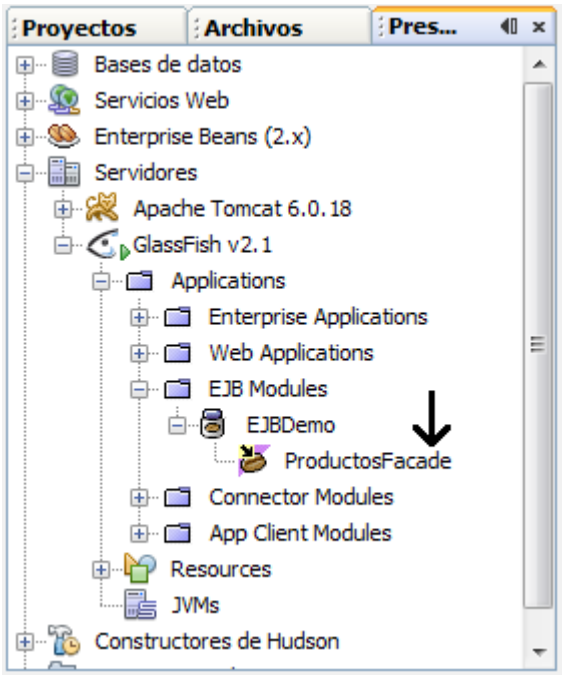



***Figura 4.14** Deploy EJB*

Con todo esto ya tenemos nuestro EJB creado y solo nos faltaría crear la interface que lo utilice.

## *4.9 Explicación de la creación de EJB*

Anteriormente hemos comentado cual es el proceso de creación de un EJB. A continuación explicaremos más detalladamente el proceso comentando el código.

### *4.9.1 API de Persistencia: Entity Manager*

Básicamente se encarga del mapeo entre una tabla relacional y su objeto Java. Proporciona métodos para manejar la persistencia de un Bean de Entidad, permite añadir, eliminar, actualizar y consultar así como manejar su ciclo de vida. Sus métodos más importantes son:

- **persist (Object entity).** Almacena el objeto entity en la base de datos.

- **merge (T entity).** Actualiza las modificaciones en la entidad devolviendo la lista resultante.

- **remove (Object entity).** Elimina la entidad.

- **find (Class <T> entity, Object primaryKey).** Busca la entidad a través de su clave primaria.

- **flush ().** Sincroniza las entidades con el contenido de la base de datos.

- **refresh (Object entity).** Actualiza el estado de la entidad con su contenido en la base de datos.

- **createQuery(string query).** Crea una consulta utilizando el lenguaje JPQL.

- **createNativeQuery ().** Crea una consulta utilizando el lenguaje SQL.

- **isOpen ().** Comprueba si está abierto el EntityManager.

- **close ().** Cierra el EntityManager.

Las funciones que hemos implementado son las siguientes:

```
public void create(Productos productos) {
    em.persist(productos);
}

public void edit(Productos productos) {
    em.merge(productos);
}

public void remove(Productos productos) {
    em.remove(em.merge(productos));
}

public Productos find(Object id) {
    return em.find(Productos.class, id);
}

public List<Productos> findAll() {
    return em.createQuery("select object(o) from Productos as o").getResultList();
}
```

***Figura 4.15*** *Funciones del EntityManager*

Podemos obtener una referencia al EntityManager a través de la anotación @PersistenceContext. El contenedor de EJB nos proporciona el contexto de persistencia mediante inyección por lo que no tendremos que preocuparnos de su creación y destrucción.

```
@Stateless
public class ProductosFacade implements ProductosFacadeRemote {
    @PersistenceContext
    private EntityManager em;
```

***Figura 4.16*** *Implementación del EntityManager*

### *4.9.2 Unidad de Persistencia*

Una unidad de persistencia defina un conjunto de todas las clases de entidad que están administrados por instancias EntityManager en una aplicación. Este conjunto de clases de entidad representan los datos contenidos en una única base de datos.

Las unidades de persistencia son definidas en el archivo de configuración persistence.xml. Cada unidad de persistencia debe tener un identificador único.

El contenido del archivo persistence.xml es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="EJBDemoPU" transaction-type="JTA">
        <jta-data-source>jbdc/ejb</jta-data-source>
        <properties/>
    </persistence-unit>
</persistence>
```

***Figura 4.17*** *persistence.xml*

### *4.9.3 Ciclo de vida de una Entidad*

Engloba dos aspectos: la relación entre el objeto Entidad y su contexto a persistir y por otro lado la sincronización de su estado con la base de datos. Para realizar estas operaciones la Entidad puede encontrarse en cualquiera de estos cuatro estados:

- **new.** Nueva instancia de la Entidad en memoria sin que aún le sea asignado su contexto persistente almacenado en la tabla de la base de datos.

- **managed.** Entidad dispone de contenido asociado con el de la tabla de la base de datos debido a que se utilizó el método *persist*(). Los cambios que se produzcan en la Entidad se podrán sincronizar con los de la base de datos llamando al método *flush*().

- **detached.** La Entidad se ha quedado sin su contenido persistente. Es necesario utilizar el método *merge*() para actualizarla.

- **removed.** Estado después de llamarse al método *remove*() y el contenido de la Entidad será eliminado de la base de datos.

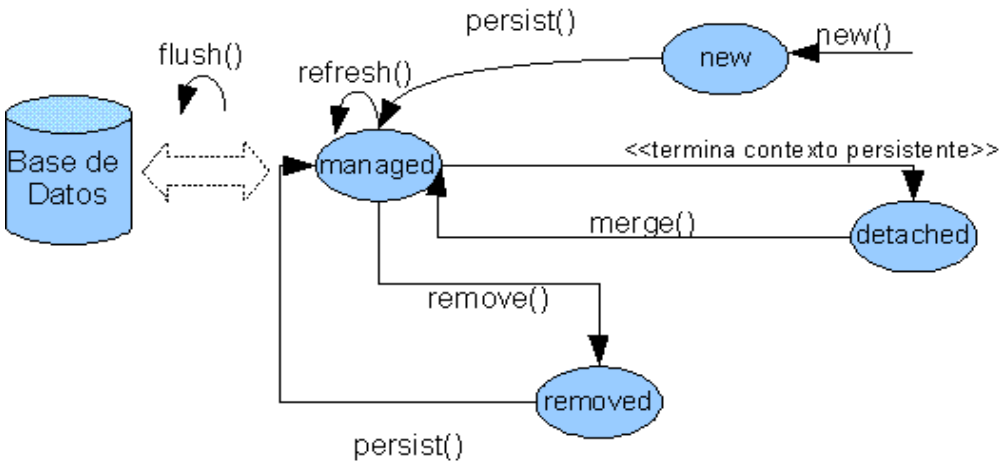



***Figura 4.18*** *Ciclo de vida de una Entidad*

## *4.10 Ejemplo*

A continuación desarrollaremos un ejemplo que utilice el EJB que hemos creado anteriormente. Para ello creamos una interface basada en JSP, dentro del cual creamos un *servlet*, que será el encargado de llamar al EJB.

Para esto dentro del método *processRequest* del servlet hacemos click derecho y seleccionamos *Insertar Código → Call Enterprise Bean.* Aquí escogemos nuestro bean de sesión creado anteriormente.

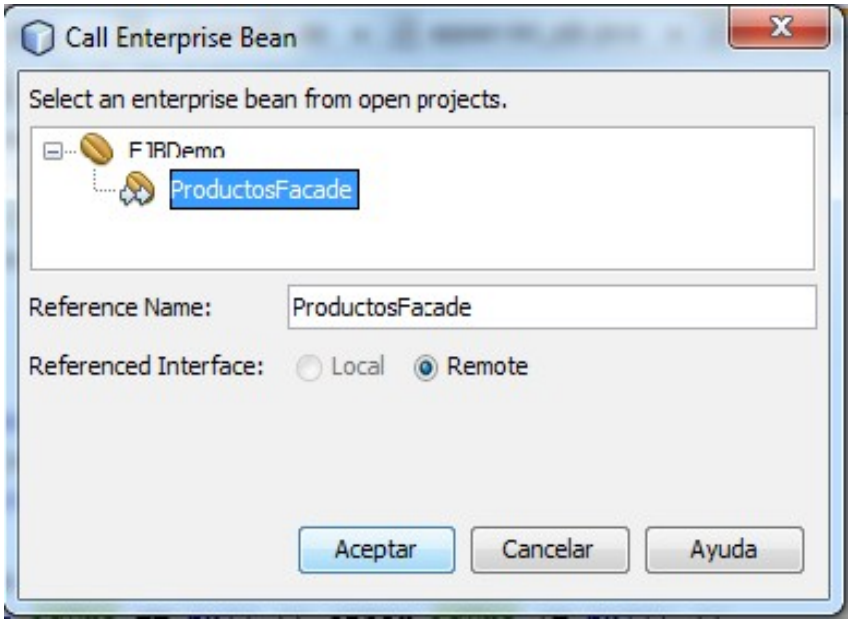



***Figura 4.19*** *Selección de EJB*

Con esto último se nos ha copiado el código de llamada remoto a EJB. Solo nos queda utilizar los métodos proporcionados por este EJB. Para este caso utilizaremos el método *findAll* que se encarga de mostrar toda la lista de productos que hayan en la tabla de la base de datos seleccionada. El código que se encarga de hacer la llamada a *findAll* es el siguiente.

```
List productos=productosFacade.findAll();
for(int i=0;i<productos.size();i++){
    Productos it = (Productos) productos.get(i);
    out.println("<h1>" + it.getIdPro()+"-->"+it.getDescPro()+"</h1>");

}
```

***Figura 4.20*** *Código de findAll*

Y finalmente solo nos queda hacer referencia al servlet que acabamos de crear desde la página de index.jsp. Cuando ejecutamos la aplicación se nos muestra lo siguiente:


***Figura 4.21*** *Ejemplo de findAll*

### *4.10.1 Análisis de resultados*

Para hacer una comparación entre los distintos tiempos vamos a utilizar la API JAMon Java Application Monitor, que permite ver los tiempos entre capas, cuanto tiempo tarde en ejecutarse un determinado EJB, JSP o servlet. Cuanto tarda la Base de datos en ejecutar cierta consulta y pasarla a la clase que la ha pedido, etc.

No sólo podemos sacar estadísticas de tiempo en la ejecución, sino que también podemos ver cuantos usuarios simultáneos hay en la apliación y detectar errores de programación.

Para la ejecución de la API deberemos escribir el siguiente código. Dónde start es el momento desde donde empezamos a mediar y stop cuando paramos.

```
Monitor mimonitor = MonitorFactory.start("MiMonitor");
mimonitor.stop();
System.out.println(mimonitor);
```

***Figura 4.22*** *Código de medición de tiempo*

### *4.10.2 Tiempo de ejecución de un EJB*

Hemos analizado la diferencia de tiempo que hay entre la ejecución de un EJB con estado a diferencia de un EJB sin estado. El contenido de los dos EJBs es el mismo y solo varia el tipo.

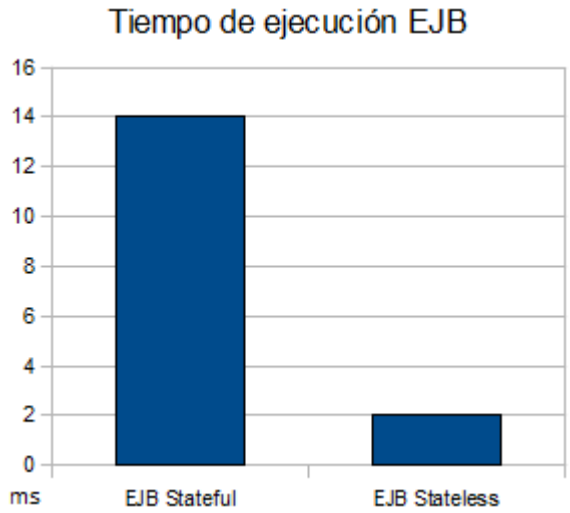



***Figura 4.23*** *Tiempo de ejecución EJB*

Como podemos observar en la gráfica la diferencia de tiempo de ejecución entre un EJB Stateful y un Stateless es muy superior, aún siendo bajo (14 ms), ya que éste se tiene que encargar de guardar el estado para una posible siguiente llamada al EJB.

### *4.10.3 Tiempo de ejecución de un Servlet*

Esta vez analizaremos el tiempo de ejecución de un Servlet concreto, que es el que hemos utilizado en el ejemplo anterior. Haremos la comparación utilizando EJBs con estado y sin estado.

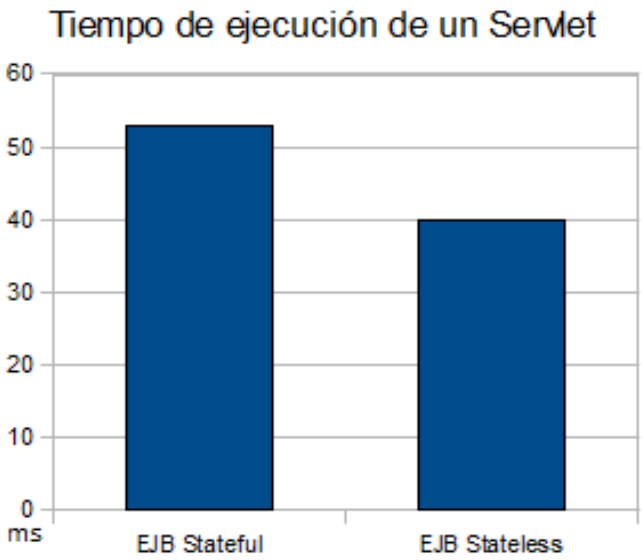



***Figura 4.24*** *Tiempo de ejecución de un Servlet*

### *4.10.4 Tiempo de acceso a la Base de Datos*

Haciendo el análisis del tiempo que se tarda en acceder a la base de datos podemos observar que no influye el tipo de EJB de sesión ya que el tiempo es el mismo.

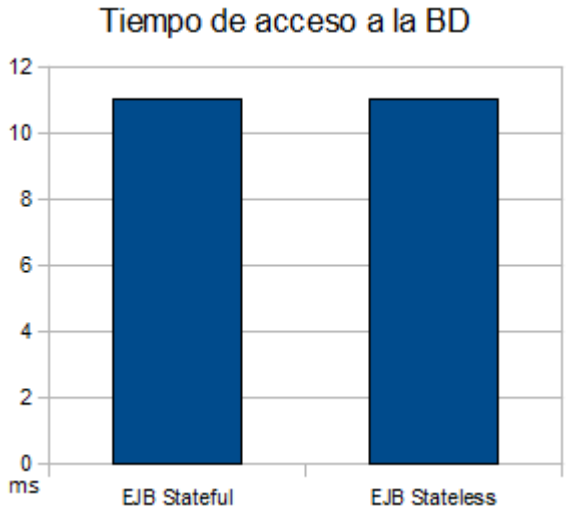



***Figura 4.25*** *Tiempo de acceso a la BD*

# CAPÍTULO 5

## Desarrollo de la capa de presentación

## *5.1 Alternativas para la capa de presentación*

Cuando nos planteamos el desarrollo de una aplicación, una de las cosas más importantes es identificar el tipo de aplicación que vamos a desarrollar. El tipo de aplicación que construiremos dependerá de las restricciones de despliegue que tengamos y del servicio que se quiere ofrecer. Por ejemplo, se puede tener la restricción de no requerir ningún tipo de instalación en los clientes, en cuyo caso tendríamos una aplicación web. Tenemos los siguientes tipos de aplicaciones.

### *5.1.1 Aplicaciones basadas en web*

Estas aplicaciones se pueden ejecutar en cualquier navegador sin la necesidad de realizar ninguna instalación en el cliente.

#### *5.1.1.1 Aplicaciones Servlet/JSP clásicas*

JSP permite agregar muchas funcionalidades a una página HTML dinámicamente. Las aplicaciones Servlet/JSP permiten ejecutar EJB y presentar los resultados en formato html mediante el comando println. Cuando la página JSP se despliega, el contenido se ejecuta de adentro hacia fuera, un servlet se crea basándose en las etiquetas scriplets incrustados en el código Java. Todo esto sucede de manera transparente para el usuario. Una de las ventajas de JSP es puede ser desarrollado por un programador web sin necesidad de tener conocimientos de un desarrollador.

En el ejemplo utilizado que hemos explicado en el capítulo anterior de EJB, utilizamos una página JSP simple como página de inicio que mediante una llamada a un Servlet, éste se encarga de llamar a los EJBs necesarios para la ejecución de la aplicación y nos devuelve como resultado en formato html los datos que hemos pedido.

En la siguiente figura se explica el proceso de JSP.

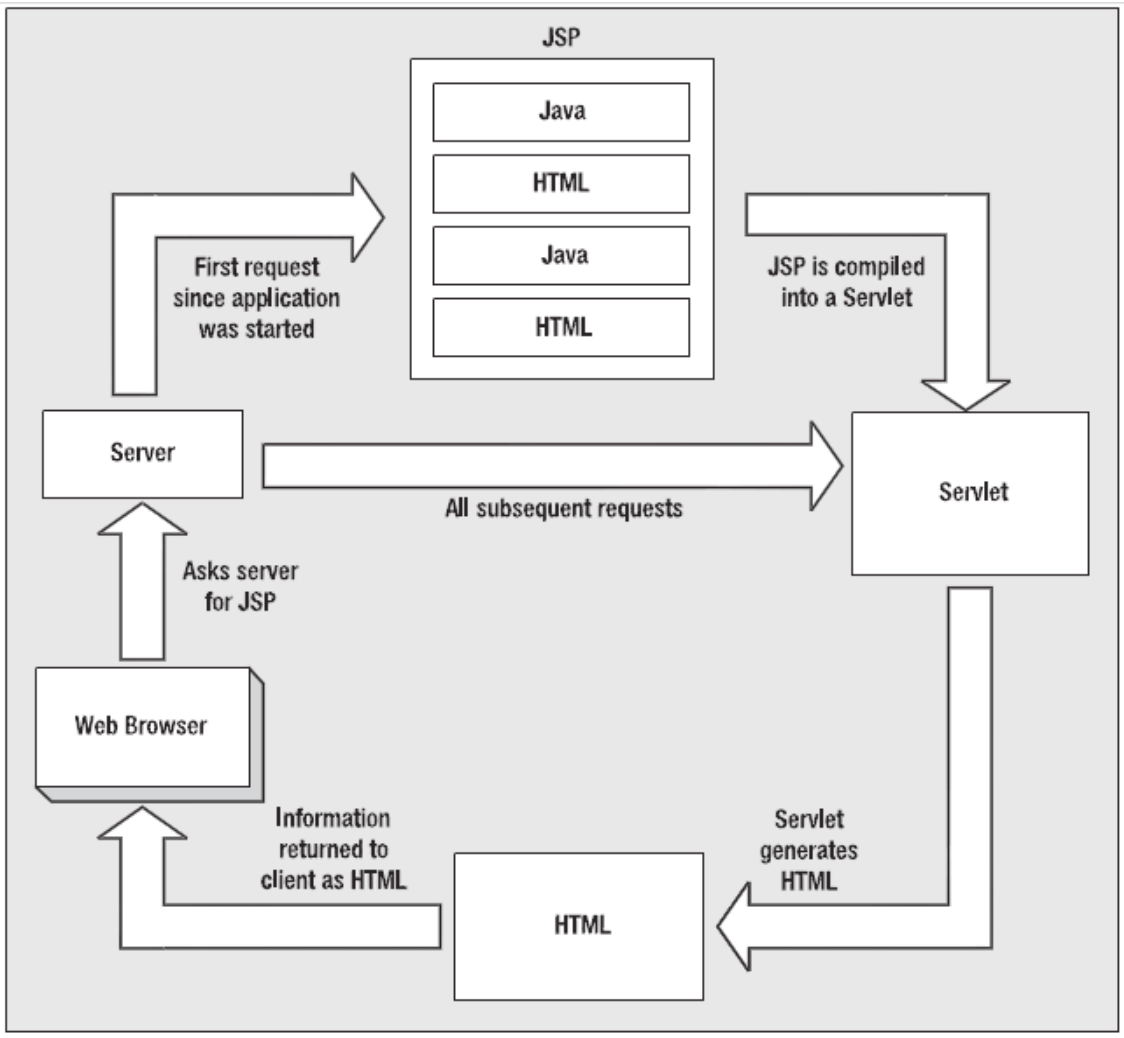



***Figura 5.1*** *Esquema JSP/Servlet*

A continuación exponemos la resolución final del programa realizado, que se adjuntará junto con la memoria del proyecto.


***Figura 5.2** Servlet/JSP*

### *5.1.1.2 Aplicaciones RIA*

Las aplicaciones RIA (Rich Internet Applications), son la elección más adecuada cuando queremos dar una versión más visual y con mejor respuesta a través de la red. Estas aplicaciones ofrecen la calidad gráfica de una aplicación de escritorio y las ventajas de despliegue y mantenimiento de una página web.

Las aplicaciones RIA son un nuevo tipo de aplicaciones con más ventajas que las tradicionales aplicaciones web. Surgen como una combinación entre aplicaciones web y aplicaciones tradicionales.

Para desarrollar una aplicación RIA hemos utilizado ICEfaces, que es una framework Ajax habilitado con J2EE y que utiliza lenguaje Java.

Esta aplicación que hemos desarrollado consiste en una página de Login que mediante EJBs se comunica con la base de datos y comprueba si el usuario y la contraseña son correctos. El programa se adjunta junto con la memoria y el resultado es el siguiente.


***Figura 5.3*** *Página de login con la RIA ICEfaces*

Si el login es correcto nos devuelve la palabra “logueado” y sino es correcto o ha habido algun error de conexión nos devuelve un mensaje de error.

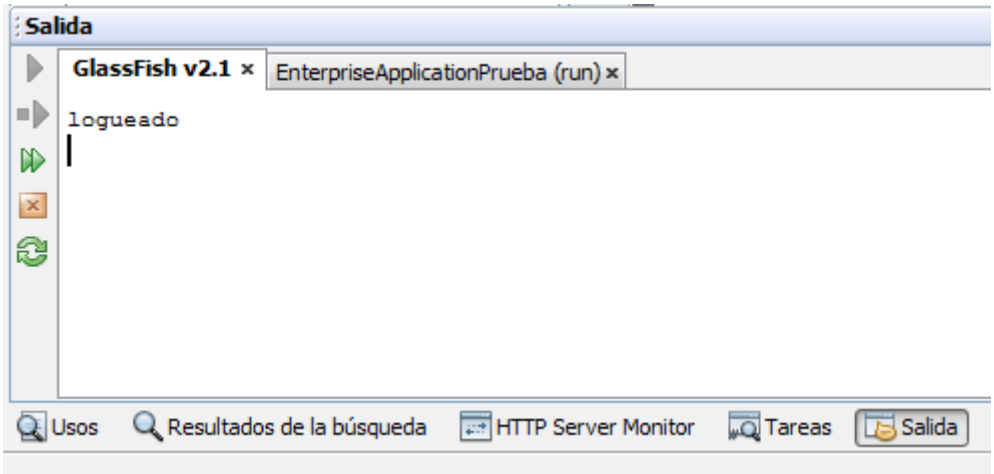



***Figura 5.4*** *ICEfaces login correcto*

### *5.1.1.3 Desarrollo mediando struts*

Los struts son un framework de la capa de presentación que implementa el patrón MVC, modelo-vista-controlador, en Java. Y, como todo framework, intenta simplificar la implementación de la arquitectura.

MVC es un patrón de diseño aportado originariamente por el lenguaje SmallTalk a la Ingeniería del Software. Consiste principalmente en dividir las aplicaciones en tres partes: Controlador, modelo y vistas. El controlador es el encargado de redirigir o asignar una aplicación a cada petición. El modelo se corresponde con la lógica de negocio y una vez realizadas las operaciones necesarias el flujo vuelve al controlador y la vista es la gestión de la interfaz de los datos a los usuarios.

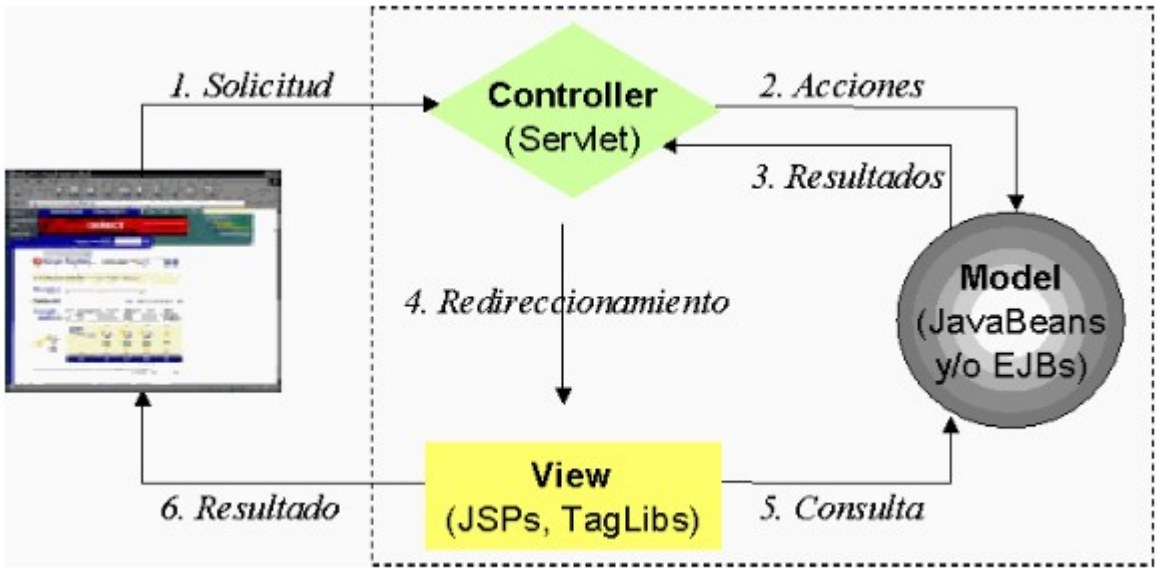



***Figura 5.5*** *Esquema de MVC*

Para este caso hemos desarrollado un ejemplo que consiste en la validación de un login. El navegador genera una solicitud que es atendida por el controlador. El mismo se encarga de analizar la solicitud, seguir con la configuración que se le ha programado en el xml y llamar al Action pasándole los parámetros enviados. El Actionform se encarga de comprobar si los datos introducidos son correctos.

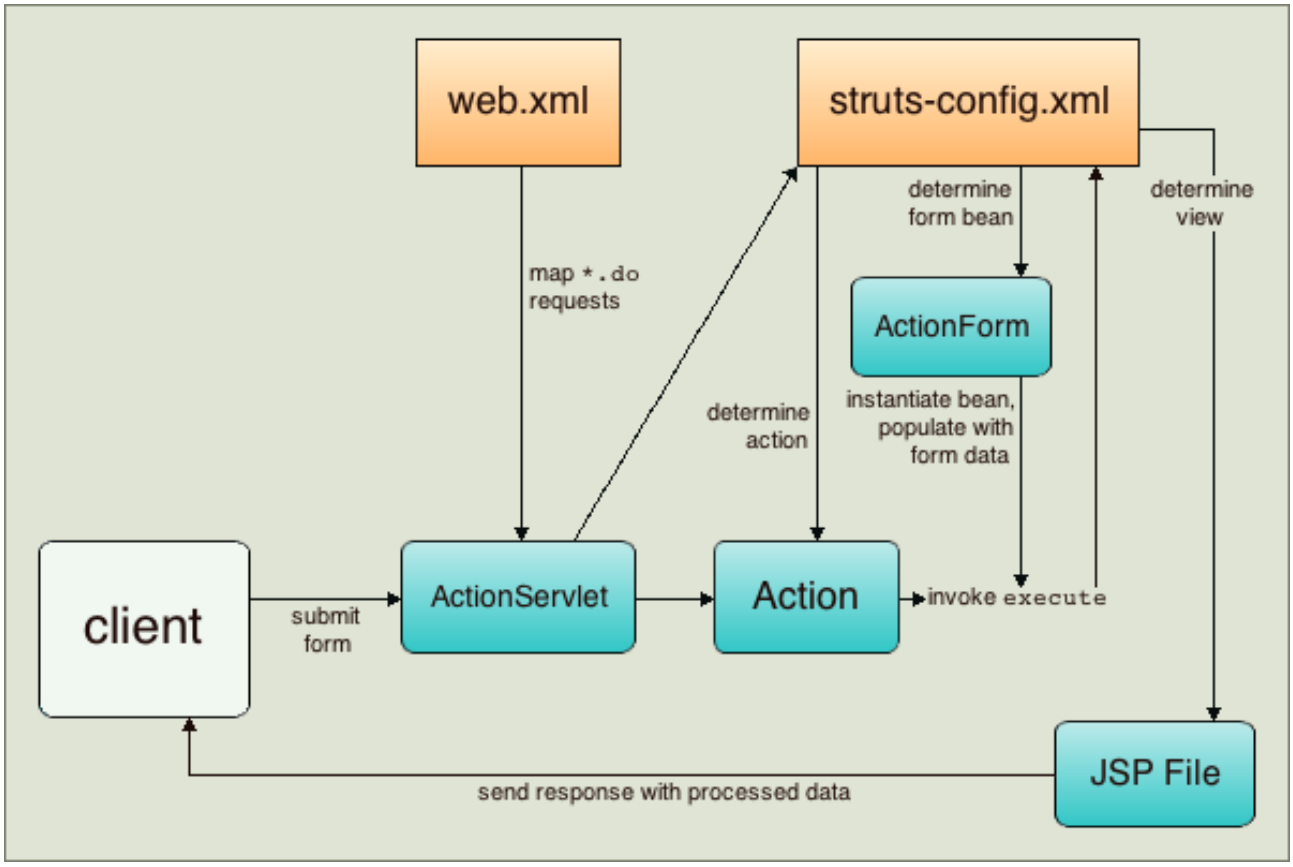



***Figura 5.6*** *Esquema Struts de ejemplo*

El ejemplo está adjunto junto con la memoria y el resultado visual es el siguiente:

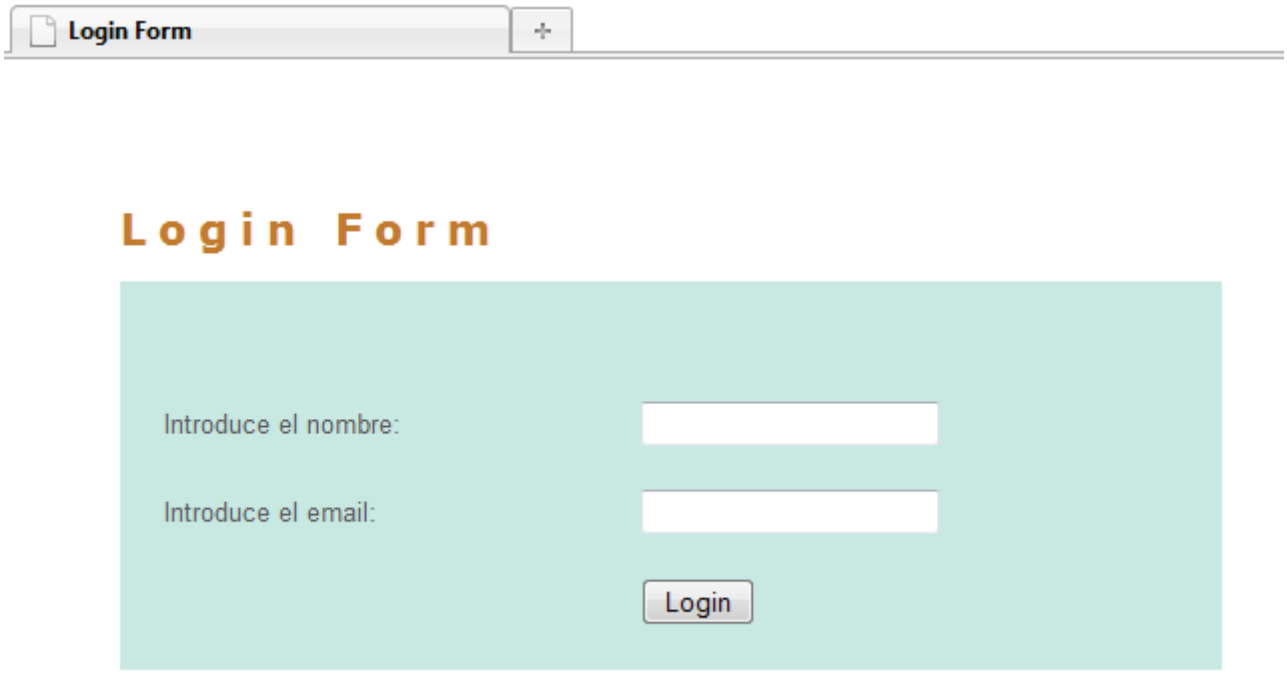



***Figura 5.7** Formulario*

Si los valores son incorrectos muestra un mensaje de error y si es correcto te muestra la siguiente página jsp.


***Figura 5.8*** *Login correcto*

### *5.1.1.4 Desarrollo mediante JSF*

Al igual que Struts, JSF pretende normalizar y estandarizar el desarrollo de aplicaciones web. JSF es posterior a Struts y por lo tanto se ha basado en ella para mejorar las deficiencias. Además comparten el mismo creador.

JSF trata el interfaz de forma visual, similar a Swing o Visual Basic. De forma que la programación del interfaz se haga a través de componentes mediante clicks.

La gran virtud de JSF es su sencillez, que además nos permite crear nuestros propios componentes.

Esta vez hemos utilizado Visual Web JSF que es una herramienta mediante la cual podemos arrastrar objetos de manera visual y el propio programa te genera código sin necesidad de introducirlo manualmente. De esta manera obtenemos una interfaz muy atractiva para el usuario sin necesidad de conocimientos complejos de jsp. Visual Web JSF utiliza la librería Woodstock creada por Sun.

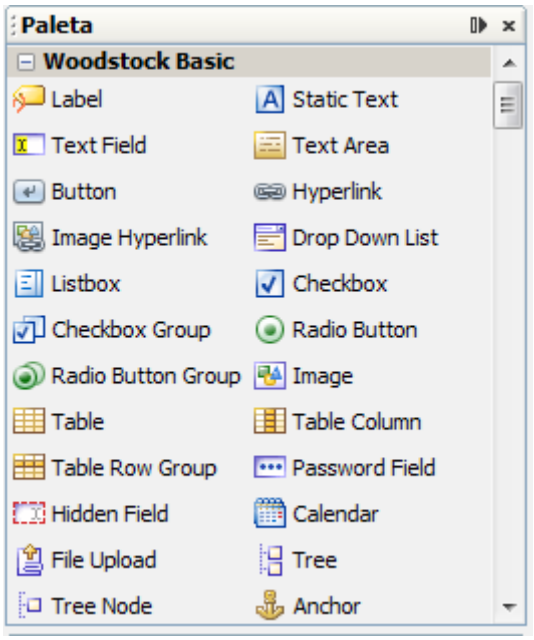



***Figura 5.9*** *Elementos de Woodstock*

El siguiente ejemplo es una simulación de como podría ser una aplicación empresarial para recursos humanos, donde tenemos una base de datos con personas relacionadas con la empresa y los datos de los empleados, junto con los estudios de cada uno.

A continuación veremos como sería cuando se ejecuta la página que muestra todos los empleados que hay en la empresa.


***Figura 5.10*** *Lista de empleados*

Otro ejemplo serían formularios implementados para introducir datos en la base de datos.

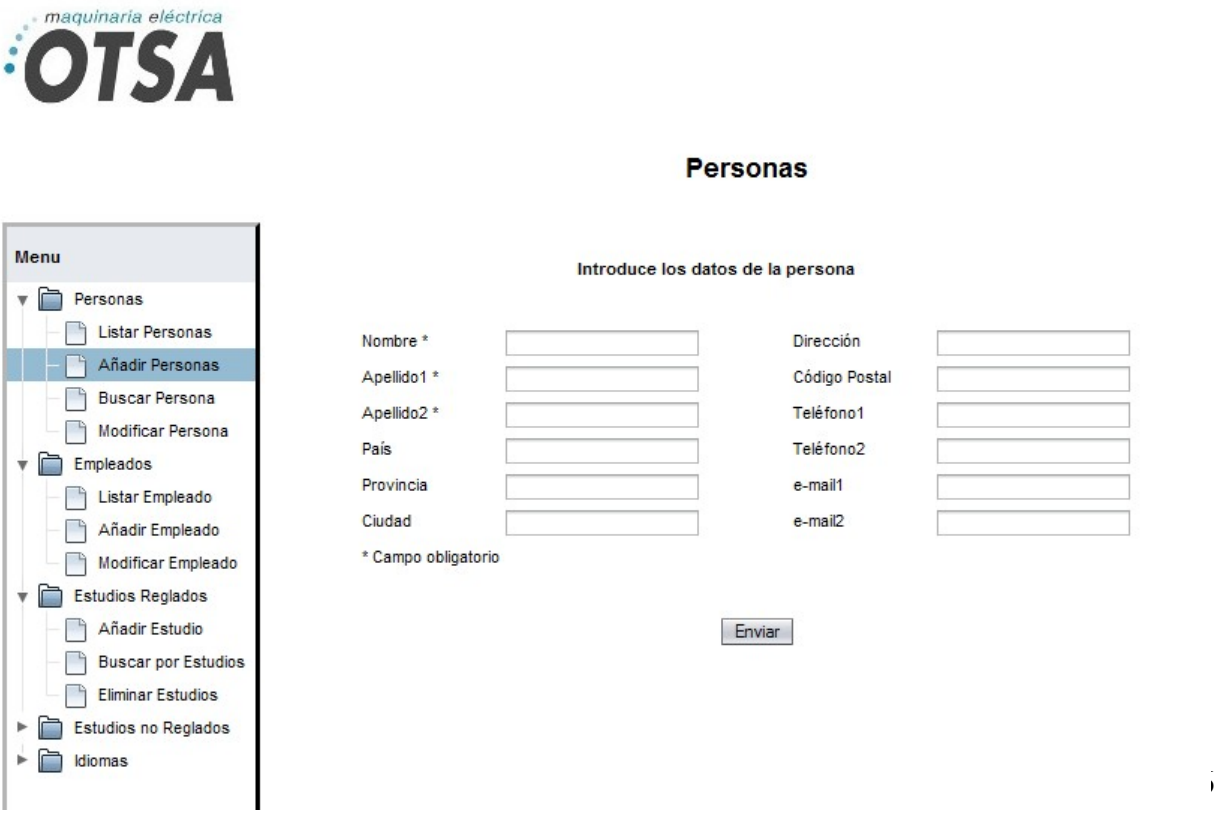



***Figura 5.11*** *Formulario*

### *5.1.2 Aplicaciones de escritorio*

Las aplicaciones de escritorio, o Rich Client, son de tipo aplicación tradicional. Ofrecen potentes interfaces gráficas y alto rendimiento. Pueden funcionar en todo tipo de entornos, con conexión o sin conexión.

Para la realización del siguiente ejemplo hemos utilizado Swing, que es una biblioteca gráfica para Java que incluye widgets como cajas de texto, botones, tablas, etc.

Las ventajas que ofrece Swing son varias, por ejemplo el diseño en Java puro provee menos limitaciones a la plataforma y el desarrollo de componentes de Swing es más activo. Por contraposición, al ser una plataforma de escritorio necesita tener instalados los plug-in de Java en los ordenadores que quieran ejecutarlo y la necesidad de tener que instalar actualizaciones en cada ordenador.

El ejemplo consiste en una aplicación que interaccione con la base de datos para que se puedan añadir, modificar, actualizar o eliminar la misma.

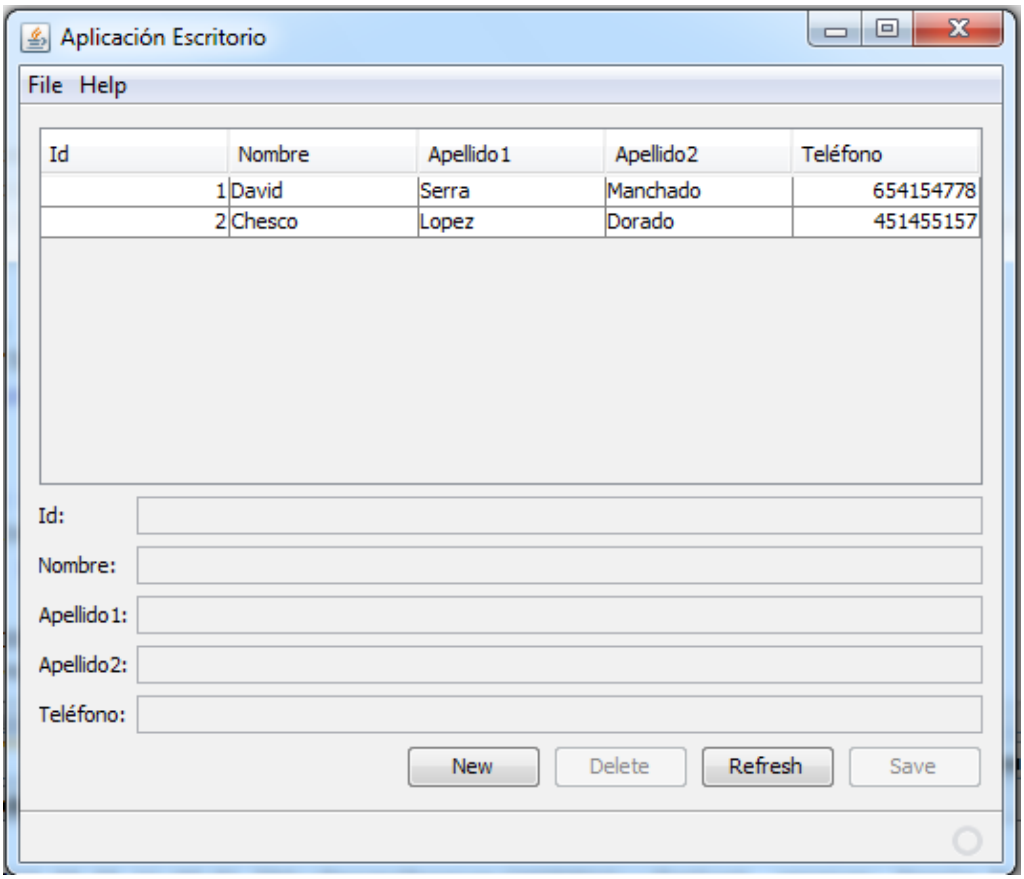



***Figura 5.12*** *Aplicación de escritorio*

## *5.2 Comparativa de los diferentes tipos de capas de presentación*

Hemos realizado una comparativa de los tiempos de carga de los diferentes tipos de capas de presentación y los resultados son los siguientes.

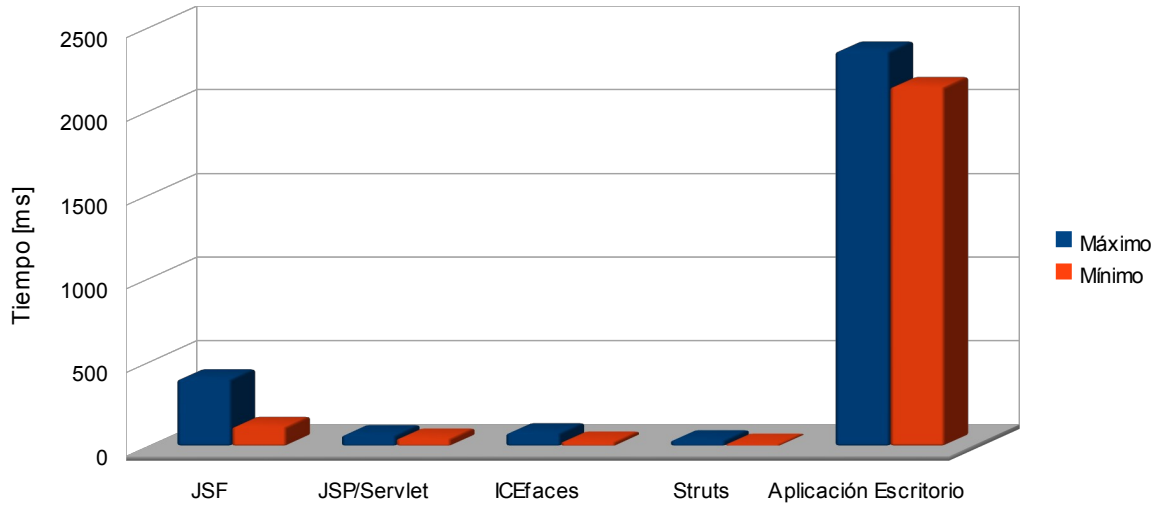



***Figura 5.13*** *Resultados de tiempos*

Como podemos observar en la gráfica hay una diferencia muy grande entre la aplicación de escritorio y las demás aplicaciones. Esto es debido a que debe cargarse toda la aplicación con todos los componentes desde el principio, a diferencia de las aplicaciones web que éstas se cargan progresivamente a medida que se van ejecutando las distintas páginas.

Cabe destacar también que cuantos más elementos dispongamos en la aplicación, más tardará en desplegarse. JSF al disponer de elementos visuales más elaborados es lógico que el tiempo sea mayor. Aun así, consideramos que el tiempo máximo de la aplicación mas lenta es de 2342ms, con lo que al ser un tiempo no muy elevado no supone un problema para su utilización.

# CAPÍTULO 6

## Conclusiones

## *6.1 Conclusiones*

Para este proyecto, hemos realizado un estudio sobre las tecnologías que debemos utilizar para realizar una aplicación empresarial, por lo tanto a continuación expondremos qué hemos elegido finalmente y porque.

Antes de nada, cabe remarcar que las conclusiones de este estudio están condicionadas por el estado actual del mercado. El mundo informático se encuentra en constante evolución sin detenerse. Es por ello que lo que hoy puede ser la solución más adecuada, en unos años pueda convertirse en algo obsoleto que requiera una actualización. De todos modos, y teniendo en cuenta este universo cambiante, el estudio de este proyecto se ha realizado para tener un período de validez de entre 3 y 5 años.

La primera decisión que hay que tomar es elegir la plataforma de desarrollo. En este caso claramente distinguimos dos plataformas que pueden competir en cuanto a recursos, filosofía y prestaciones, que son .NET y J2EE. Se analizó .NET Framework 2.0 ya que es la especificación compatible con Mono, que es la implementación open source. Actualmente está disponible la versión 4.0, pero solo es compatible con sistemas operativos Windows.

La elección se ha decantado por J2EE. Principalmente hay que comentar que las dos plataformas están destinadas a un mercado similar, pero con un despliegue diferente. Mientras con .NET te ves restringido a utilizar toda la tecnología de Microsoft, lo que comporta una gran inversión, la gran ventaja de J2EE es que es una especificación abierta que puede adaptada por multitud de aplicaciones. Si trabajas con .NET estás ligado principalmente a tecnologías de Microsoft, como por ejemplo BizTalk o Sharepoint. Es por esta razón que para minimizar riesgos se ha decidido trabajar con J2EE que se basa en procesos de estandarización de la Java Community Process en las tecnologías que implementa. Además J2EE también permite la integración de otras plataformas, como ya hemos comentado en su correspondiente capítulo, mediante RMI o Web Services.

Una vez elegida la plataforma de desarrollo se procede a elegir el servidor de las aplicaciones. De todos los servidores que hemos estudiado en los capítulos anteriores del proyecto los que destacan por sus características y por ser open source (recordamos que en principio la empresa no está dispuesta a realizar ningún gasto adicional) son Jboss y Glassfish. Los dos servidores de aplicaciones encajan perfectamente con la plataforma J2EE y ofrecen servicios para EJB, Tomcat, etc.

Glassfish pertenece a Sun Microsystems y Jboss fue comprada por RedHat. Hay que remarcar que los dos son productos fiables y con comunidades de desarrollo grandes. Para la elección de la plataforma se ha valorado principalmente el nivel de adopción de estándares y las tecnologías opcionales que éstos pueden implementar. Se escoge Glassfish por ser la especificación de referencia de J2EE por un lado, y por tener soporte de Sun en su desarrollo. Además, si a la empresa en algún momento le interesa realizar una importante ampliación, existe la versión comercial Sun Glassfish Enterprise Server, que ofrece servicios extra y un soporte 24 horas. Otro punto a favor de Glassfish es la integración con todos los productos de Sun, como por ejemplo con Netbeans. El propio programa en su IDE dispone de un entorno para ejecutar las aplicaciones en el servidor, de manera que se vuelve muy simple y eficaz para el desarrollador. Por otro lado, hay que añadir que durante el transcurso de este proyecto Sun microsystems fue comprado por Oracle, que también poseía una plataforma J2EE, por lo que las implicaciones de futuro de esta compra son desconocidas.

Por último, hemos de decidir que tipo de tecnologías para la interfaz de usuario tenemos que elegir para el desarrollo de la UI. Esta última depende tanto de las necesidades como de criterios de usabilidad. Deberán valorarse las características requeridas en cada aplicación en particular. Si por ejemplo se necesita una aplicación con gran carga de interacción de UI, como aplicativos CAD o de gráficos, la mejor opción sería una aplicación de escritorio, mientras que si prima la movilidad la opción sería aplicaciones web.

En caso que queramos utilizar una aplicación web, como hemos visto anteriormente disponemos de varias soluciones. Según la gráfica de comparaciones que se ha realizado, realmente no existe una diferencia significativa valorando el tiempo, por lo tanto descartamos este factor, sin embargo, en aplicaciones muy potentes, JSP solo envía http y se ejecuta en el servidor y JSF se ejecuta parte en el cliente, deberemos elegir qué nos interesa más. A partir de aquí, si queremos una aplicación más llamativa visualmente podemos optar por ICEFaces o JSF mediante Woodstock.

Para hacer el prototipo de la aplicación se decidió por Woodstock, básicamente por su fácil manejo de su edición y resultados eficaces. No obstante, en el futuro no se sabe que pasará con esta librería ya que, como he comentado anteriormente, la reciente compra de Oracle a Sun Microsystems, que ya disponía de su propia librería de JSF llamada Oracle ADF, hace que posiblemente en el futuro pueda haber una fusión de estas tecnologías o simplemente apoyar ésta última y dejar de respaldar a ICEFaces o Woodstock. Pero esto es algo que por el momento no podemos averiguar.

## *6.2 Conclusiones personales*

Como conclusiones personales, este proyecto me ha ayudado muchísimo a mejorar varios aspectos estudiados durante la carrera.

El conocimiento de Java que tenía antes de empezar este proyecto era muy básico y me ha ayudado a ampliar conocimientos sobre éste así como utilizar objetos simples con soltura.

Gracias a haber realizado varios ejemplos distintos de interfaces que en definitiva buscan lo mismo me ha ayudado a aprender un poco de cada tecnología, haciendo un incapié en JSF, que es donde he llegado a realizar una aplicación más elaborada.

Me ha ayudado a mejorar conocimiento de bases de datos ya que he tenido que realizar varias bases de datos por mí mismo partiendo desde cero. Así como utilizar tecnologías como JDBC para la interconexión de la base de datos con las aplicaciones.

También he aprendido a realizar un estudio amplio sobre un tema en concreto sabiendo utilizar la información útil y desechar lo que no interesa, teniendo en cuenta que en la mayoría de los casos la mayoría de la documentación está realizada por el propio fabricante y, lógicamente, solo se valoran los puntos a favor de su producto.

Como conclusión general podríamos decir que ha sido una ampliación y resumen de todo lo visto durante todos estos años de estudios.

## 6.3 Posibles ampliaciones

Las ampliaciones que se podrían hacer y no se han realizado son las siguientes:

Como medida para el futuro, se podría explotar el rendimiento de Glassfish en cluster, con varios servidores conectados entre sí y varios usuarios accediendo a la vez.

También se podría ampliar el retoque de las configuraciones de Glassfish en los siguientes térmitos:

- Augmentar el rendimiento y la estabilidad.
- Fortalecer la seguridad.
- Mejorar la administración.
- Creación de diferentes tipos de usuario.

Finalmente otra continuación hubiese sido haber implementado toda una aplicación empresarial que se dedicara a dirigir todos los recursos humanos de la empresa. Por ejemplo se podría añadir varios tipos de usuarios, como administradores, jefes y trabajadores, para que cada uno pudiese realizar sus operaciones pertinentes como revisar el currículum, el historial de bajas, etc.

Esta posible ampliación se podría haber realizado en varias versiones diferentes para ofrecer el producto a otras empresas por ejemplo como cliente de escritorio y como aplicación web.

# Bibliografía

## *Referencias bibliográficas*

[1] Kevin Mukhar and Chris Zelenak with James L. Weaver and Jim Crume, *Beginning Java EE 5 From Novice to Professional,* APRESS, 2006

[2] Justin Couch and Daniel H. Steinberg, *Java 2 Enterprise Edition Bible,* Hungry Minds, Inc., 2002.

## *Referencias Web*

[3] http://msdn.microsoft.com/es-es/architecture/default.aspx

[4] https://glassfish.dev.java.net/

[5] http://docs.sun.com/

[6] http://netbeans.org/kb/index.html

[7] http://www.adictosaltrabajo.com/

[8] http://www.programacion.com/

# Anexos

# *Anexo 1. Configuración de la plataforma de desarrollo*

Para este proyecto las plataformas que hemos elegido para su desarrollo serán PostgreSQL como base de datos, Glassfish como servidor de aplicaciones y Netbeans como IDE para el desarrollo de Java. A continuación describiremos los pasos de la instalación de cada uno de los elementos.

## *1.1 Instalación de Glassfish*

Para hacer pruebas y no ejecutar Glassfish directamente en el servidor hemos decidido instalar Máquina Virtual con el mismo Sistema Operativo que se utilizará para realizar una simulación.
Para estas pruebas se ha utilizado la máquina virtual Sun VirtualBox con la implementación Debian de GNU/Linux. Lógicamente el rendimiento será menor en una máquina virtual pero será útil para realizar pruebas antes de aplicarlas al servidor real.

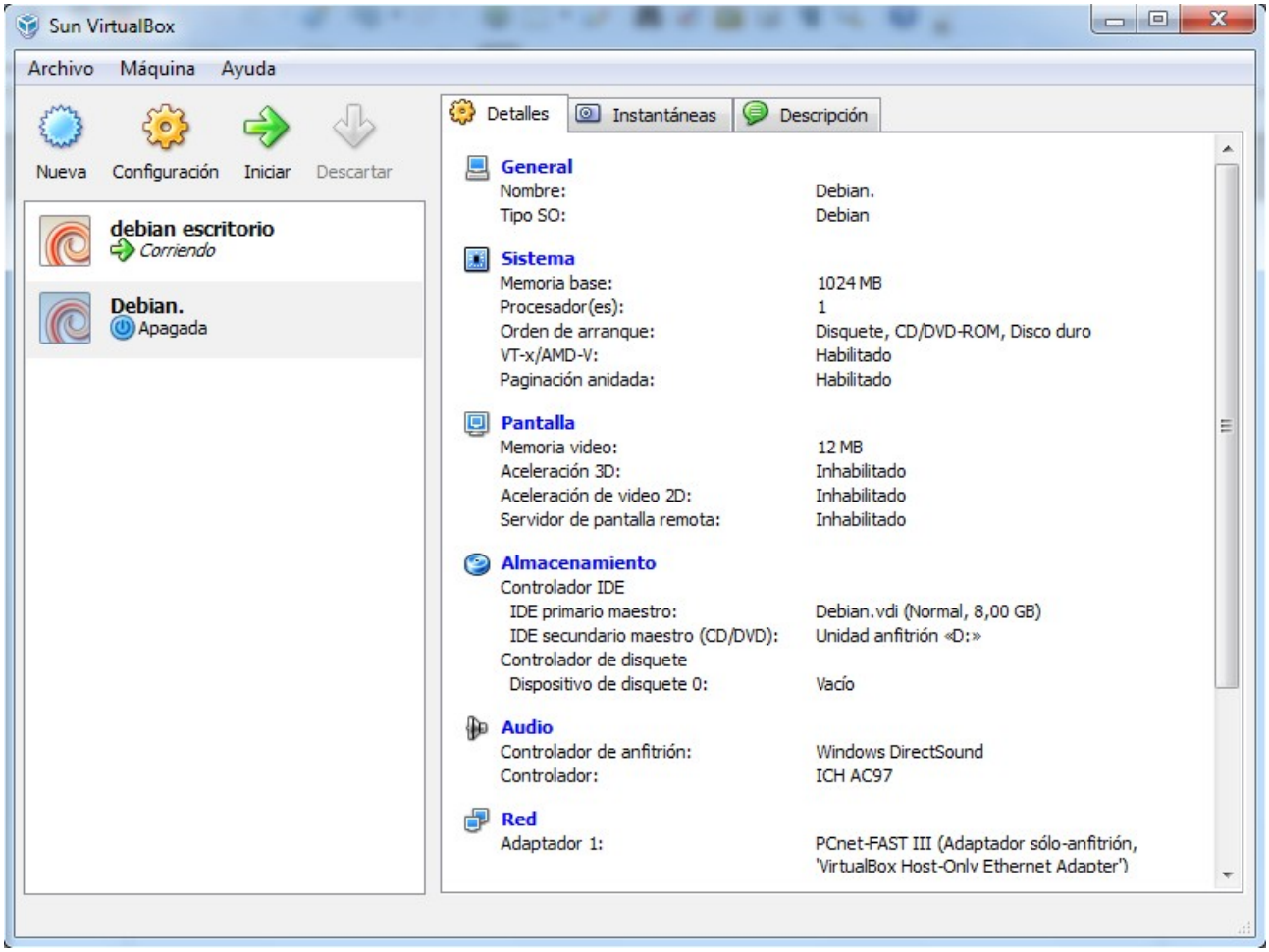
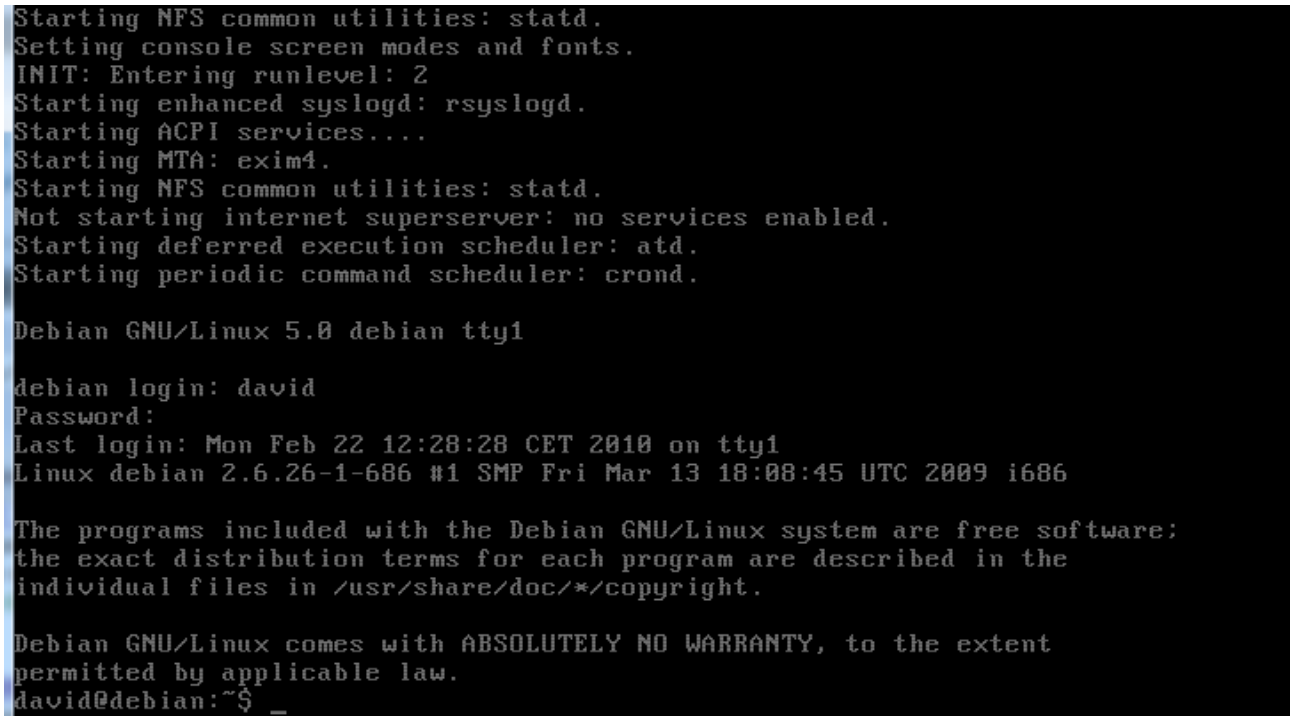



***Figura A.1*** *VirtualBox*

### *1.1.1 Instalación de Debian GNU/Linux 5.0*

Se ha instalado el paquete básico de la distribución de GNU/Linux Debian 5.0 sobre VirtualBox con una reserva de memoria de 1 Gb y una partición virtual de disco duro de 8 Gb.




### *1.1..2 Instalación de JDK*

Glassfish utiliza el Java Development Kit (JDK), que es un software que provee herramientas de desarrollo para la creación de programas en java. Glassfish utiliza la especificación J2EE y por lo tanto necesita la instalación de JDK.

Hay dos formas de instalar el paquete de java:

## 1. Instalación a través de la página web de java

Para este tipo de instalación realizamos los siguientes pasos:

a) Descargar el archivo de instalador binario **jdk_6u3_linux-i586.bin** desde la página web www.java.sun.com

b) En un terminal, nos vamos a la carpeta donde hemos descargado .bin:

**$ cd <carpeta>**

c) Le damos permisos de ejecución al archivo:

**$ chmod +x jkd-6ul-linux-i586.bin**

d) Instalamos:

**$ sudo ./jdk-6ul-linux-i586.bin**

e) Seguimos las instrucciones que aparecen en pantalla.

f) Movemos la carpeta creada después de la instalación a /usr/lib/jvm:

**$ sudo mv jdk1.6.0_01 /usr/lib/jvm**

g) Actualizamos el nuevo Java como una de las alternativas de Java:

**$ sudo update-alternatives --install "/usr/bin/java" "java" "/usr/lib/jvm/jdk1.6.0_01/bin/java" 1**

h) Actualizamos la nueva alternativa como la real de Java:

**$ sudo update-alternatives --set java /usr/lib/jvm/jdk1.6.0_01/bin/java**

i) Comprobamos si se ha instalado correctamente la versión 1.6.0:

**$ java -version**

## 2. Instalación a través de apt

APT (Advanced Packaging Tool) es un sistema de gestión de paquetes creado por el proyecto Debian. APT simplifica en gran medida la instalación y eliminación de programas en los sistemas GNU/Linux.

a) Actualización de la lista de sources. Editar el archivo /etc/apt/sources.list con la página web de descargas de Debian.

b) Actualizar lista

**$ apt-get update**

c) Procedemos a la instalacion de OpenJDK que es lo mismo pero la versión software libre.

d) Instalar OpenJDK

**$ install openjdk-6-jdk**

La instalación de OpenJDK se realizó con éxito y a continuación instalamos Glassfish 2.1

### *1.1.3 Instalación de Sun GlassFish Enterprise Server v2.1*

a) Descarga del GlassFish v2.1 utilizando wget guardándolo en la carpeta tmp.

**$ cd /tmp wget http://download.java.net/javaee5/v2.1_branch/promoted/Linux/glassfish-installer-v2.1-b60e-linux.jar**

b) Instalar Glassfish

**$ glassfish java -Xmx256M -jar glassfish-installer-v2.1-b60e-linux.jar**


c) Mover a la carpeta opt

d) Creamos el usuario “glassfish” que será el encargado de el servidor de aplicaciones.

**$ sudo adduser –home /opt/glassfish glassfish**

e) Les damos permisos de ejecución al usuario glassfish.

**$ sudo chown -R glassfish /opt/glassfish**
**$ sudo chgrp -R glassfish /opt/glassfish**

f) Volver ejecutables los scripts de Ant.

**$ chmod -R +x lib/ant/bin/**

g) Ejecutamos la configuracion.

**$ sudo -u glassfish lib/ant/bin/ant -f setup.xml**

h) Iniciar un Dominio

**$ sudo -u glassfish bin/asadmin start-domain domain1**

```
debian:/opt/glassfish/bin# ./asadmin start-domain domain1
Starting Domain domain1, please wait.
Default Log location is /opt/glassfish/domains/domain1/logs/server.log.
Redirecting output to /opt/glassfish/domains/domain1/logs/server.log
Domain domain1 is ready to receive client requests. Additional services are bein
g started in background.
Domain [domain1] is running [Sun GlassFish Enterprise Server v2.1 (9.1.1) (build
 b60e-fcs)] with its configuration and logs at: [/opt/glassfish/domains].
Admin Console is available at [http://localhost:4848].
Use the same port [4848] for "asadmin" commands.
User web applications are available at these URLs:
[http://localhost:8080 https://localhost:8181 ].
Following web-contexts are available:
[/web1  /__wstx-services ].
Standard JMX Clients (like JConsole) can connect to JMXServiceURL:
[service:jmx:rmi:///jndi/rmi://debian:8686/jmxrmi] for domain management purpose
s.
Domain listens on at least following ports for connections:
[8080 8181 4848 3700 3820 3920 8686 ].
Domain does not support application server clusters and other standalone instanc
es.
```

i) Configuramos el VirtualBox para que se conecte a través de la ip del host y escribimos en el navegador de Windows la dirección http://192.168.56.101:4848. Aquí tenemos la página principal de Glassfish, donde para acceder escribimos el usuario y contraseña por defecto.


***Figura A.2*** *Página principal de admin*

### *1.1.4 Primera prueba en Glassfish*

Para comprobar el correcto funcionamiento de Glassfish hemos descargado una prueba de la página oficial https://glassfish.dev.java.net/downloads/quickstart/hello.war

Para ejecutar la prueba tenemos que ubicar el archivo en el directorio /glassfish/domains/domain1/autodeploy/ y escribir en el navegador http://192.168.56.101:8080/hello y se nos muestra la siguiente pantalla.


***Figura A.3*** *Ejemplo Hello.war*

## 1.2 Instalación de PostgreSQL

Hacemos la instalación en Debian mediante el comando apt, se instala por defecto la versión 8.3

**$ sudo apt-get install postgresql**

Se crea automáticamente el usuario postgres así nos logueamos como ese usuario para crear una base de datos de prueba.

**$ su – postgres**

**$ createdb postgres**

Una vez creada la base de datos accedemos a la terminal de postgresql y definimos un usuario y su contraseña para la base de datos.

**$ psql postgres**

**$ alter user postgres with password 'XXXX';**

### *1.2.1 Configuración del servidor de Base de Datos*

Nuestro objetivo es manejar la base de datos instalada en Debian con la aplicación PgAdmin que permite el manejo de esta en Windows. Para ello tenemos que hacer una serie de modificaciones en los archivos de configuración de Postgresql.

Primero editamos el archivo postgresql.conf y modificamos las direcciones de escucha para podernos conectar a la base de datos desde el “exterior”.

**#listen_addressess = 'localhost'**
**#port=5432**

Pasará a estar así.

```
listen_addressess = '*'
port=5432
```

A continuación modificamos el archivo pg_hba.conf para definir la ip local desde la cual nos conectamos.

```
#Ipv4 local connections:
host   all      all      192.168.56.1md5
```

Ya tenemos configurado postgresql. A continuación configuramos el programa pgAdmin III desde Windows para que se conecte a la base de datos.

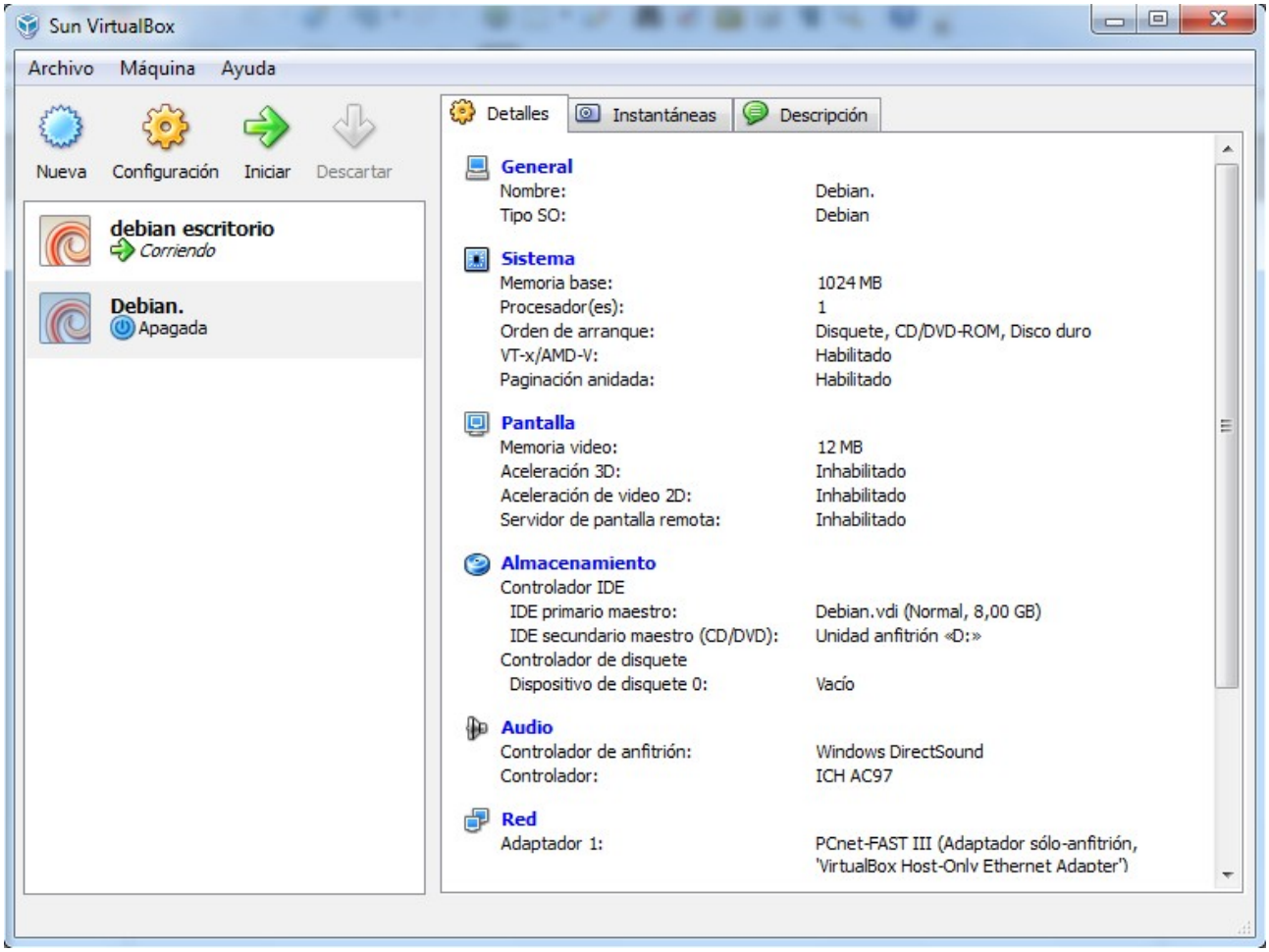



***Figura A.4*** *VirtualBox*

## *1.3 NetBeans*

La instalación de netbeans se ha realizado en windows, por lo que lo único que se ha tenido que hacer es descargarse el archivo ejecutable desde la página oficial http://netbeans.org/downloads/index.html.

Se ha elegido la versión 6.7.1 ya que es la última que cuenta con soporte a la aplicación Visual Web JSF y es compatible con la versión 2.1 de Glassfish.

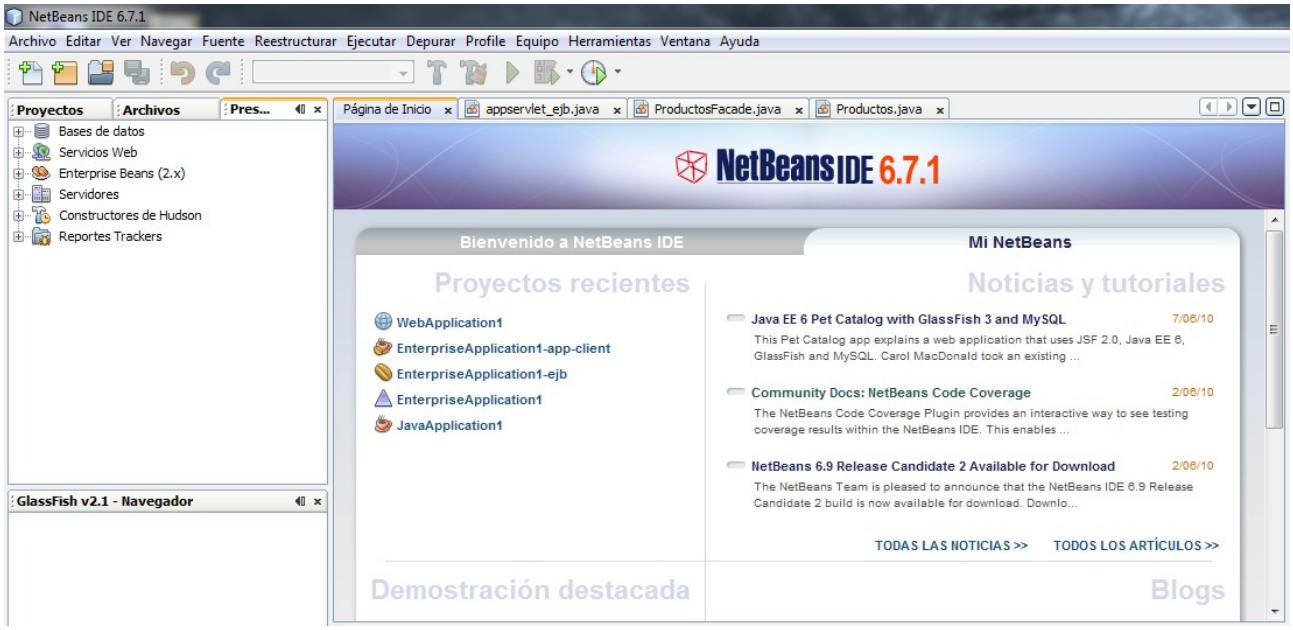



***Figura A.5*** *Entorno de desarrollo NetBeans 6.7.1*