# ESTUDIO Y DESCRIPCIÓN DE LA ARQUITECTURA JAVA EE Y DE LAS PRINCIPALES TECNOLOGÍAS SUBYACENTES

# ESTUDIO Y DESCRIPCIÓN DE LA ARQUITECTURA JAVA EE Y DE LAS PRINCIPALES TECNOLOGÍAS SUBYACENTES

**MAURICIO ARCHBOLD BARBOZA**
**MILTON JAVIER OCHOA LARIOS**

**UNIVESIDAD TECNOLÓGICA DE BOLÍVAR**
**PROGRAMA DE INGENIERÍA DE SISTEMAS**
**CARTAGENA DE INDIAS D. T. y C.**
**2007**

# ESTUDIO Y DESCRIPCIÓN DE LA ARQUITECTURA JAVA EE Y DE LAS PRINCIPALES TECNOLOGÍAS SUBYACENTES

**MAURICIO ARCHBOLD BARBOZA**
**MILTON JAVIER OCHOA LARIOS**

**Monografía para obtener el título de**
**Ingeniero de Sistemas**

**ASESOR:**
**GIOVANNY RAFAEL VASQUEZ MENDOZA**
**Ingeniero de Sistemas**

**UNIVESIDAD TECNOLÓGICA DE BOLÍVAR**
**PROGRAMA DE INGENIERÍA DE SISTEMAS**
**CARTAGENA DE INDIAS D. T. y C.**
**2007**

Cartagena, febrero 5 del 2007

Señores:

**UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR**
Comité de evaluación de proyectos
La Ciudad

**Respetados señores:**

De la manera más cordial, nos permitimos presentar a ustedes la monografía titulada **"ESTUDIO Y DESCRIPCIÓN DE LA ARQUITECTURA JAVA EE Y DE LAS PRINCIPALES TECNOLOGÍAS SUBYACENTES**", para su estudio, consideración y aprobación, como requisito fundamental para obtener el título de Ingeniero de Sistemas, además para aprobar el Minor de Ingeniería de Software.

En espera que éste cumpla con las normas pertinentes establecidas por la institución.

Sinceramente,

| | |
|---|---|
| Mauricio Archbold Barboza | Milton Javier Ochoa Larios |
| Código: 0205020 | Código: 0105018 |

Cartagena, febrero 5 del 2007

Señores:

**UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR**
Comité de evaluación de proyectos
La Ciudad

**Respetados señores:**

Tengo el agrado de presentar a su consideración, estudio y aprobación, la monografía titulada "**ESTUDIO Y DESCRIPCIÓN DE LA ARQUITECTURA JAVA EE Y DE LAS PRINCIPALES TECNOLOGÍAS SUBYACENTES**", desarrollada por los estudiantes Mauricio Archbold Barboza y Milton Javier Ochoa Larios.

Al respecto me permito comunicar que he dirigido el citado trabajo, el cual considero de gran importancia y utilidad.

Atentamente,

_____

Giovanny R. Vásquez Mendoza
Ingeniero de Sistemas

Nota de aceptación

_____
_____
_____
_____

_____

Firma del presidente del jurado

_____

Firma del jurado

_____

Firma del jurado

Cartagena Febrero de 2007

**AUTORIZACIÓN**

Cartagena de Indias, Febrero de 2007

Nosotros **Mauricio Archbold Barboza**, identificado con número de cédula 73.208.215 de la ciudad de Cartagena y **Milton Javier Ochoa Larios,** identificado con número de cédula 73.197.994 de la ciudad de Cartagena, autorizamos a la Universidad Tecnológica de Bolívar para hacer uso de nuestro trabajo de grado y publicarlo en el catálogo online de la Biblioteca.

Atentamente,

| | |
|---|---|
| Mauricio Archbold Barboza | Milton Javier Ochoa Larios |

## DEDICATORIA

Quiero dedicar, este trabajo a Dios, por ser el que nos acompaña día a día.

A mis padres, por acompañarme y apoyarme en todos los momentos de mi vida.

A mi hermana, por su apoyo y tolerancia, a mi hermano que esta en el cielo que se que desde allá me esta ayudando a ser una mejor persona.

A mi novia, por estar con migo en los buenos y malos momentos.

Por última, a todas las personas que en el transcurso de mi carrera, han estado allí, cuando los he necesitado.

**MAURICIO ARCHBOLD BARBOZA**

## DEDICATORIA

Este trabajo se lo dedico a quien para mi son las mejores personas del mundo, mis padres, Arnaldo Ochoa Díaz y Yamile Larios Posada.

A mis hermanos (Igor y Rosiris) y mi sobrino Sebastián con quienes día a día comparto los buenos y malos ratos que nos da la vida.

A Dios por darme la vida, por acompañarme cada día y por haberme dejado alcanzar una más de mis metas.

**MILTON JAVIER OCHOA LARIOS**

## AGRADECIMIENTOS

Los autores de este trabajo de grado, desean brindarle al Ingeniero Giovanny R. Vásquez Mendoza, los más sinceros agradecimientos, por ayudarnos, a través de su colaboración, paciencia, y orientación, a la culminación exitosa de nuestra monografía.

También a todas aquellas personas, que de una u otra forma, hicieron posible la realización del Minor Ingeniería de Software, al cual le damos finalización, con la presentación del siguiente trabajo investigativo.

Queremos agradecerle, a Dios, por habernos llenado de fortaleza, paciencia y esmero, para no desfallecer en nuestra labor estudiantil, y ponerle amor a cada una de las actividades que realizamos durante el transcurso de ésta.

Por último y no menos preciado le damos un especial agradecimiento al señor Moisés Quintana, por habernos inundado de conocimiento en lo largo de toda nuestra carrera.

**TABLA DE CONTENIDO**

## LISTADO DE TABLAS Y FIGURAS

# INTRODUCCIÓN

Los desarrolladores de hoy ***cada vez más*** reconocen la necesidad de aplicaciones distribuidas, transaccionales y portables, que se apoyen en las tecnologías de un servidor como es la velocidad, seguridad y confiabilidad. En el mundo de la tecnología de la información, las aplicaciones empresariales deben ser diseñadas, construidas y producidas por menos dinero, mayor velocidad y con pocos recursos.

Con la **Edición Empresarial de Java**, se puede desarrollar una aplicación empresarial de una manera muy sencilla y rápida. El objetivo de la plataforma Java EE es ofrecer a los desarrolladores un poderoso conjunto de APIs que reducen drásticamente el tiempo de desarrollo, reducen la complejidad de la aplicación y mejoran el rendimiento en la aplicación.

La arquitectura Java EE 5 introduce un modelo de programación simplificado. Con esta tecnología, los descriptores de despliegue XML ahora son opcionales. En lugar de eso, un desarrollador puede ingresar toda esa descripción en una *anotación* directamente en el archivo fuente Java, y el servidor Java EE configurara el componente para su despliegue y su ejecución.

Java EE esta diseñado para soportar aplicaciones que implementen servicios empresariales para clientes, empleados, socios y otros quienes lo demanden o contribuyan a la empresa. Dichas aplicaciones son inherentemente complejas, potencialmente acceden a datos desde una gran variedad de fuentes, y son aplicaciones distribuidas con una gran variedad de clientes.

Para tener un mejor control y manejo de esas aplicaciones, las funciones de negocio que soportan a esos diferentes usuarios, son manejadas en el *middle tier*. El *middle tier* representa un ambiente que es controlado por alguna tecnología de algún departamento.

El middle tier generalmente se esta ejecutando sobre un servidor dedicado y el tiene acceso a todos los servicios de la empresa.

El modelo de una aplicación Java EE define una arquitectura que puede implementar servicios de aplicaciones multi-tier que entregan escalabilidad, accesibilidad, y manejabilidad, estas características son necesarias en todas las aplicaciones de nivel empresarial. Este modelo de partición, separa el trabajo necesario para implementar un servicio multi-tier en dos partes: lógica de negocio y lógica de presentación que serán implementadas por el desarrollador, y los servicios de sistema estándar ofrecidos por la plataforma Java EE. El desarrollador puede confiar en la plataforma para que esta ofrezca una solución a los problemas difíciles de los servicios multi-tier.

A continuación, se tratada una tecnología en particular de la arquitectura Java EE por capitulo, en el capitulo 1 trataremos de introducir una verdadera definición de lo que es Java EE, además de conceptos esenciales que ésta maneja; en los restantes capítulos solo dedicamos a: JSP, JSF, Servlets, EJB y Servicios Web respectivamente. Ilustrando al final de cada capitulo un breve ejemplo que explique esa tecnología.

# 1. FUNDAMENTOS DE JAVA EE

Las aplicaciones empresariales solucionan problemas de negocio. Esto usualmente involucra almacenamiento de información en forma segura, recuperación y manipulación de los datos del negocio, que pueden ser: facturas de clientes, aplicaciones de crédito hipotecario, reservación de vuelo, entre muchos otros. Tal vez estas aplicaciones tengan múltiples interfaces de usuario: una interfaz Web para los clientes y una aplicación con una interface grafica de usuario (GUI) corriendo en las oficinas, por decir algún ejemplo.

Las aplicaciones empresariales deben establecer una comunicación entre sistemas remotos, coordinar datos en múltiples almacenamientos, y asegurarse que los sistemas cumplen las reglas impuestas por el negocio.

Los desarrolladores empresariales construyen sus aplicaciones sobre sistemas llamados servidores de aplicación. Así como los GUI *toolkits* suministran servicios para las aplicaciones con interfaz grafica, los servidores de aplicaciones suministran servicios para desarrollar aplicaciones empresariales, estas facilitan mucho las cosas, como lo es: las comunicaciones que se realizan entre computadoras, manejan conexiones a base de datos, posibilidad de servir páginas Web, y posibilidad al manejo de transacciones.

Por último, cabe mencionar que Java EE ofrece una manera uniforme para desarrollar aplicaciones empresariales sobre cualquier servidor de aplicación. El conjunto de librerías desarrolladas por ***Sun Microsystems*** y la Comunidad Java, es lo que representan esta uniformidad, a eso anterior es lo que nosotros llamamos la Arquitectura Java, que es el tema de esta monografía.

## 1.1 ¿QUÉ ES JAVA EE?[1]

**Java Platform, Enterprise Edition** o **Java EE**, es una plataforma de programación para 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.

Java EE incluye varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc., y define como coordinarlos. Java EE también establece algunas especificaciones únicas para Java EE de algunos componentes. Estas incluyen Enterprise JavaBeans, Servlets, JavaServer Pages y varias tecnologías de servicios web. Esto permite al desarrollador crear una aplicación de Empresa que sea portable entre plataformas y escalable. Otros beneficios añadidos son, por ejemplo, que el servidor de aplicaciones puede manejar las transacciones, seguridad, escalabilidad, concurrencia y gestión de los componentes que son desplegados, significando que los desarrolladores pueden concentrarse más en la lógica de negocio de los componentes en lugar de las tareas de mantenimiento de bajo nivel.

## 1.2. ARQUITECTURA MULTI–TIER

Uno de los temas más recurrentes que se ven en Java EE es la noción de soportar aplicaciones que son divididas en varios niveles, o *tiers*, que es la arquitectura fundamental de Java EE. Si pensáramos en la composición de una aplicación multiniveles, nos daríamos cuenta que podemos romperla en tres secciones principales, o capas lógicas:

- La primera área concierne a la exhibición de las cosas al usuario y recolección de datos del usuario: La capa de interfaz de usuario, a menudo es llamada la ***capa de presentación***, su compromiso es mostrarle objetos, datos, cosas al usuario y suministrar una forma en la cual el usuario pueda interactuar con el sistema. La

[1] http://es.wikipedia.org/wiki/Java_EE

capa de presentación incluye la parte del software que crea y controla la interfaz de usuario y valida las acciones de usuario.

- Debajo de la capa de presentación está la parte lógica, quien hace que la aplicación funcione y maneje el procesamiento de datos importantes de la misma. Esta capa lógica, es llamada la ***capa de negocio***, o más informalmente *the middle tier*.

- Todas las aplicaciones empresariales necesitan leer y escribir datos; y la parte del software que es responsable para leer y escribir esos datos (cualquiera que sea la fuente) es la ***capa de acceso de datos***.

### 1.2.1. Arquitectura Single–Tier.

Las aplicaciones sencillas están diseñadas para que corran en una sola computadora. Todos los servicios suministrados por la aplicación (la interfaz de usuario, acceso a datos persistentes y la lógica que procesa los datos de entrada por el usuario y lectura desde algún almacenamiento) existen en la misma computadora, y a menudo todo eso es agrupado en la misma aplicación.

Los sistemas de este tipo, son relativamente fáciles de manejar, y la consistencia de los datos es simple. Sin embargo, esta arquitectura tiene algunas desventajas, los sistemas ***Single–Tiers*** no son escalables, no pueden ser manipulados por muchos usuarios, y no proveen fácil manejo para compartir los datos en una empresa.

### 1.2.2. Arquitectura Cliente–Servidor (Two–Tier).

Muchas aplicaciones toman ventaja de un servidor de base de datos y acceso persistente a datos, utilizando y enviando sentencias SQL al servidor de base de datos para salvar y recuperar datos. En este caso, la base de datos corre como un proceso diferente de la aplicación, incluso corre en otra computadora.

Los componentes para el acceso de datos están segregados del resto de la lógica de la aplicación. El fundamento de esta estrategia, esta en la centralización de datos, donde múltiples usuarios pueden acceder simultáneamente y trabajar con una base de datos común, y de esta manera proporcionar la capacidad de tener la base de datos en un servidor que pueda compartir una comunicación con la aplicación.

Usualmente se refiere a la arquitectura Cliente–Servidor a cualquier arquitectura donde un cliente se comunica con un servidor, cualquiera sea el tipo de datos o servicio que el servidor suministre. Es mucho más conveniente y significativo conceptualizar la división de responsabilidades en *tiers*.

Una de las desventajas de la arquitectura cliente–servidor es que la lógica que manipula los datos y las reglas de especificación de la aplicación, están agrupadas en la misma aplicación. Esto se convierte en un problema cuando múltiples aplicaciones usan una base datos compartida.

Para evitar todo este desperdicio, lo más lógico que se tiene que hacer es separar físicamente las reglas de negocio, colocando estas reglas de negocio en un servidor por separado, así para cuando se necesite actualizar las reglas del negocio, solo se necesite actualizar una sola vez y no por cada aplicación que corre en las distintas computadoras.

### 1.2.3. Arquitectura N-Tier.

La arquitectura N-Tier, como su nombre lo implica, es aquella arquitectura donde la aplicación es dividas en N capas, donde N es un numero elegido por los analistas de aplicación, ya que cualquier aplicación puede ser sub divida en cualquier cantidad de capas.

Una arquitectura común en sistemas empresariales es aquella donde participan 3 tiers. En este modelo, toda la lógica de negocio es extraída fuera de la aplicación que corre en la computadora de escritorio. La aplicación de escritorio es responsable solo de mostrar la interfaz de usuario y establecer la comunicación entre la capa lógica de negocio. Esta

capa ya no es responsable de la lógica del negocio ni del acceso a los datos, su trabajo es solo la capa de presentación.

Normalmente, en una aplicación distribuida, la capa de negocio se ejecuta en un servidor aparte de la estación de trabajo (aunque esto no es absolutamente necesario). La capa de negocios realiza una conexión lógica entre la capa de presentación y la base de datos. Como ésta se encuentra en un servidor, ésta puede ser accedida por cualquier cantidad de usuarios en la red. A medida que la demanda de usuarios crece, los servicios también crecen, y eso produce que la lógica de negocio incremente su complejidad e intensidad de procesamiento, esto no presenta ninguna dificultad en términos de hardware, ya que el servidor puede ser actualizado o se pueden ir agregando más servidores.

En este modelo también es posible seguir dividiendo la aplicación en pequeñas capas según su funcionalidad. La arquitectura de una aplicación está en la libertad de dividir en cualquier cantidad de capas, siempre y cuando esta sea la más apropiada, y teniendo en cuenta el desempeño de computo y de acceso a redes, en el que el sistema se despliega. Sin embargo, se debe tener mucho cuidado a la hora de buscar el punto de división apropiado, ya que el rendimiento de la aplicación se puede ver afectado por la comunicación que se tiene que realizar cada vez que se necesita accesar a cada una de las capas.

### 1.3. INDEPENDENCIA DE PROVEEDORES

***Sun Microsystems*** ha promovido la plataforma Java como una estrategia sólida para la construcción de aplicaciones que no están atadas a una sola plataforma. De la misma manera, la arquitectura de Java EE tiene creada una especificación abierta que puede ser implementada por cualquiera. En el momento existen muchos servidores de aplicación basados en las especificaciones de Java EE que ofrecen una plataforma para construir y desplegar aplicaciones escalables *N–tiers*.

Cualquier servidor de aplicación de Java EE que se venda, debe proveer la misma suite de servicios, usando las interfaces y las especificaciones que ***Sun*** hizo como parte integral de Java EE.

Un servidor de aplicación Java EE, ofrece al desarrollador varias opciones a la hora de implantar un proyecto, y opciones similares a medida que se le van añadiendo aplicaciones a la organización. Al construir una aplicación sobre la arquitectura Java EE, se ofrece un sustancial desacople entre la lógica de la aplicación que se escribe y el resto de cosas (seguridad, base de datos, acceso, soporte a transacciones, etc.) que son suministradas por el servidor de aplicación Java EE.

Recuerde que todos los servidores Java EE deberían soportar las mismas interfaces definidas en la especificación de Java EE. Esto quiere decir, que se puede diseñar una aplicación sobre un servidor y desplegar en otro. También es posible decidir más adelante si cambiar de servidor de aplicación Java EE para cambiar el ambiente de producción. Mover la aplicación hacia un nuevo ambiente de producción debería ser algo muy trivial.

Un punto a tener en cuenta es que cada proveedor de Java EE le agrega diferentes valores para su particular implementación de Java EE. La especificación que cubre Java EE es grande, pero también existen ciertas cosas que no están en la especificación de Java EE. El rendimiento, la fiabilidad y la estabilidad son solo unas cuantas áreas que no hacen parte de la especificación de Java EE, pero son áreas donde los proveedores ponen su mayor atención. Agregan valor a la facilidad en el uso de sus herramientas de despliegue, alta optimización en el rendimiento, soporte para servidores *cluster* (permitiendo a un grupo de servidores atender a todas las aplicaciones de clientes, como si fuese un *single super-fast, super-big Server*), además de otras cosas.

Los puntos claves a tener en cuenta aquí son:

- La producción en las aplicaciones puede ser potencialmente beneficiada por las capacidades no soportadas por la implementación de Java EE. El que una aplicación corra lenta en una computadora portátil no quiere decir que Java EE es inherentemente lento.

- Todos los proveedor especifican sus capacidades; esto, para que se tenga en cuenta como estas capacidades pueden impactar a tu aplicación.

## 1.4. ESCALABILIDAD

La definición del rendimiento y las exigencias en una aplicación, es un paso vital en la definición de requerimientos. La arquitectura de Java EE ofrece bastante flexibilidad a la hora de acomodar cambios en el rendimiento, desempeño y capacidad de carga. Aplicaciones con arquitectura *N-Tier* le permitente a los desarrolladores aplicar adicional poder computacional donde sea necesario. Dividiendo la aplicación en tiers, es posible hacer *refactoring*[2] a puntos delicados de la aplicación, evitando un gran impacto a los demás componentes en la aplicación.

*Cluestering*, *connection pooling* y *failover* se convierten en términos familiares en el momento en que se empieza a trabajar en una aplicación hecha bajo Java EE. Muchos de los proveedores de servidor de aplicaciones Java EE han trabajado minuciosamente para mejorar las formas en como se manejan el rendimiento, desempeño y disponibilidad (cada uno con alguna técnica en particular usando el *framework* de Java EE).

## 1.5. CONCEPTOS EN JAVA EE

### 1.5.1. Clientes y Servidores en Java EE.

Un cliente de Java EE puede ser una aplicación de consola escrita en Java, o una aplicación con interfaz grafica (GUI) usando JFC (*Java Foundation Classes*) y *Swing* o AWT. Estos tipos de clientes a veces también son llamados clientes gordos (*fat clients*) ya que estos tienden a tener bastante código solo para la interfaz de usuario.

También existen otros clientes en Java EE, los clientes basados en páginas Web; estos son clientes que viven dentro de un navegador. Como la mayor parte procesamiento es realizada en lado del servidor Web, estos clientes tienen muy poco código. Este tipo de clientes es también a veces llamado cliente liviano (*thin client*). Un cliente liviano puede ser una página Web usando solo código HTML, puede ser una página enriquecida con

[2] Refactoring: es el proceso de reescribir material (código) ya escrito para mejorar su funcionalidad.

*JavaScript*[3], o una página que contiene un *applet* muy sencillo donde escasamente se muestra una interface.

Seria muy vago decir que la lógica de la aplicación utilizada por un cliente es el "servidor", aunque esto es cierto, desde la perspectiva del desarrollador cuando esta escribiendo el código de la aplicación cliente, esta ilusión no es de ninguna manera la magia que la plataforma Java EE ofrece. Es un hecho, que el servidor de la aplicación Java EE es quien conecta el cliente con la lógica de negocio.

Los componentes creados del lado del servidor por el desarrollador pueden ser componentes de Web y componentes de negocio. Los componentes Web vienen de la forma JSPs o Servlets. Los componentes de negocio, en el mundo de Java EE son EJBs.

Estos componentes se basan en el *framework* Java EE. Java EE proporciona soporte a los componentes del lado del servidor en forma de contenedores (*containers*).

### 1.5.2. Contenedores.

Los contenedores es una pieza central en la arquitectura Java EE. En un servidor de aplicación, los componentes de web y componentes de negocio existen dentro de contenedores e interactúa con la infraestructura Java EE por medio de interfaces bien definidas.

De la misma manera en que los desarrolladores de aplicación pueden dividir la lógica de una aplicación en *tiers,* para darle a cada *tier* una funcionalidad especifica, los diseñadores de Java EE han dividido la infraestructura lógica en capas lógicas. Ellos se han tomado el trabajo de escribir todo el soporte de la infraestructura, esto incluye, seguridad, acceso a datos, manejo de transacciones, *binding*, localización de recursos, y las distintas formas de comunicación para conectar un cliente con el servidor. Java EE ofrece un conjunto de interfaces que permite conectar la lógica de la aplicación a esta infraestructura y acceder a esos servicios.

[3] JavaScript: Es una lenguaje de programación interpretado y diseñado para complementar las capacidades del HTML.

Java EE ofrece contenedores del lado de servidor por una razón: Ofrecer una interface bien definida, junto con unos servicios que permiten a los desarrolladores de aplicación enfocarse solo en los problemas de negocio que se intentan resolver, sin tener que preocuparse por la tubería y el cableado que hay detrás de eso para funcionar. Los contenedores manejan todos los mínimos detalles como es el inicio de servicios del lado del servidor, activación de la lógica en la aplicación y limpieza de componentes.

Java EE y la plataforma Java ofrecen contenedores para componentes Web y componentes de negocio. Estos ofrecen un entorno e interfaces para los componentes que están alojados en el contendor. Los contenedores definidos en Java EE incluyen contenedores para Servlets, JSPs y EJBs.

### 1.5.3. Java Servlets

Indudablemente ya se estará familiarizado con el acceso simple a páginas estáticas HTML, usando un navegador que envía una petición a un servidor Web, el cual, este regresa una página Web que está almacenada en el servidor. Aquí, el servidor Web está usándose como una librería virtual que devuelve un documento que le fue pedido.

Este modelo, de servir páginas Web estáticas no esta provisto para el contenido generado dinámicamente.

Los *Servlets* son una de las tecnologías desarrolladas para enriquecer un servidor. Un *Servlet* es un componente de Java que es invocado como resultado de una petición hecha por un cliente a un *Servlet* en particular .Un servidor Web recibe una petición para entregar un *Servlet* de la forma HTML. El servidor Web a su vez, invoca el *Servlet* y lo regresa como resultado a la petición del cliente. El *Servlet* es libre de realizar cualquier cálculo si lo necesita, y retornar al cliente en forma de HTML.

Los *Servlets* son manejados e invocado por el contendor *Servlet* de Java EE. Cuando el servidor Web recibe una solicitud de algún *Servlet*, este notifica al contenedor *Servlet*, el cual cargara los *Servlets* necesarios, e invoca la interface apropiada con sus métodos para satisfacer la petición.

### 1.5.4. JavaServer Pages (JSPs)

Los JSP, así como los Servlets, están afectados por el contenido generado dinámicamente. Estos dos componentes Web (Servlets y JSPs) abarcan un gran porcentaje del contenido en aplicaciones Java EE en el mundo real.

Construir Servlets implica construir componentes Java que emite un HTML en la mayoría de los casos. Sin embargo, este enfoque no es muy factible para las personas que gastan su tiempo en lo concerniente a la parte visual de aplicaciones Web y no necesariamente les interesa saber mucho sobre el desarrollo de software.

Pasando a JSP, las páginas JSP son HTML basado en documentos planos que contienen un trozo de código Java llamados *scriptlets*, estos se encuentran embebidos en el documento HTML.

Tal vez tengas cierta experiencia con los *JavaScript*, que es muy parecido al lenguaje Java. Este puede ser incluido en las páginas Web, para que se ejecute en el navegador web cuando es completamente descargado. Los JSP tienen un cierto parecido a los *JavaScripts*, pero en cambio de ser ejecutados en el navegador Web, el código es compilado y ejecutado en el servidor, luego, el resultado HTML es retornado a la petición del cliente. Las páginas JSP son muy livianas y rápidas (después de la primera compilación del Servlet) y proporcionan una gran escalabilidad en las aplicaciones basadas en Web.

### 1.5.5. JavaServer Faces (JSF)

JSF es una tecnología relativamente nueva, que intenta ser lo mas robusta posible, una enriquecida interface de usuario para aplicaciones Web. JSF es usada junto con Servlets y JSPs.

Cuando solo usamos JSPs o Servlets para generar la presentación, la interfaz de usuario esta limitada a solo lo que puede ser implementado en HTML. El HTML tiene un buen

conjunto de componentes para la interfaz de usuario, *check-boxex, radio-buttons, fields, labels* y *buttons*.

JSF ofrece un componente (API) para construir interfaces de usuario. Los componentes en JSF son componentes de interfaz de usuario que pueden ser fácilmente agrupados para crear la interfaz de usuario del lado del servidor. La tecnología JSF también hace sencilla la conexión entre componentes de interfaz de usuario con los *data source* de la aplicación, y además muy sencilla la conexión de los eventos generados por los clientes con el manejador de eventos en el servidor.

Los componentes JSF manejan toda la complejidad de la interfaz de usuario, dejando al desarrollador concentrarse solo en la lógica de negocio. La flexibilidad viene del hecho que los componentes de interfaz de usuario no generan directamente un código de presentación. La creación del código de presentación del cliente es un trabajo de los diferentes tipos de trazado (*custom renderers*). Con el adecuado trazado, los componentes de la interfaz de usuario podrían ser usados para generar código de presentación para cualquier dispositivo. Incluso, si el dispositivo del cliente cambiase, simplemente tendrías que configurar el sistema para que usara otro trazado para el nuevo cliente, sin necesidad de cambiar ningún código JSF. Por el momento, el formato de presentación más común es el HTML, y los JSF vienen con distintos trazados para crear interfaces de usuario HTML.

### 1.5.6. EJBs

Cuando se menciona Java EE, lo primero que llega a la mente es EJBs. Al principio mencionamos que Java EE no solo son EJBs, con esto no queremos decir que los EJBs es algo sin importancia; la atención que ésta tecnología tiene es verdaderamente merecida.

Para entender mejor que son y que hacen los EJBs, seria bueno empezar con los métodos de invocación remota de Java (RMI – *Remote Method invocation*). Si no se está familiarizado con RMI, seria bueno visitar la página *http://java.sun.com/rmi*.

RMI permite que un objeto Java corra en una computadora y sus métodos sean llamados por otro objeto que esta corriendo en otra computadora. Para crear objetos con RMI, se diseña una interface, en esa interface se definen los métodos que tenga el objeto remoto. Luego el objeto remoto implementa la interface que se definió. Esta clase extiende de la clase *java.rmi.server.UnicastRemoteObject*, la cual proporciona todo lo necesario para que exista una comunicación entre este objeto y el objeto quien lo llama. Finalmente, se escribe una aplicación que cree una instancia de esta clase y registre esa instancia en el RMI *registry*.

El RMI *registry* es un servicio sencillo de *lookup*, que asocia un nombre con un objeto, Este mismo servicio es usado por la aplicación cliente, la cual solicita un objeto del registro por medio de un nombre. Una vez este recibe la referencia local del objeto remoto, este puede usar los métodos del objeto; sin embargo, al ejecutar el método en la computadora cliente, esta pasa por la red la llamada del método y es ejecutada en la computadora donde el objeto remoto reside.

Lo que en realidad RMI ofrece es la implementación esencial de una arquitectura cliente-servidor: un registro para *lookup*, establece la comunicación para invocar operaciones y pasar parámetros desde y hacia los objetos remotos, y un mecanismo básico para el control de acceso a los recursos del sistema como medida de protección.

Aunque RMI es un componente *lightweight*, no esta diseñado para satisfacer los requerimientos de una aplicación distribuida. Esta carece de la infraestructura esencial en las que aplicaciones empresariales confían, como lo es la seguridad, acceso a datos, manejo de transacciones y escalabilidad.

A pesar de que este suple necesidades como lo es el *networking*, esta no ofrece un framework para un servidor de aplicación, que le permita alojar estos componentes, y tampoco es posible el escalamiento junto con la aplicación. Se debe escribir ambas aplicaciones cliente y servidor. Para resolver estos inconvenientes se diseñaron los EJBs.

Los EJBs son componentes Java que implementan lógica de negocio. Estos permiten que la lógica de negocio de una aplicación (o un conjunto de aplicaciones) sea

compartimentada en EJBs, aislando la aplicación *front–end* de la lógica de negocio. La arquitectura Java EE incluye un servidor que es un contenedor para los EJBs, este carga los *beans* cuando se les necesite, invoca operaciones expuestas, aplica reglas de seguridad, y ofrece soporte de transacción para los *beans*.

Al desarrollar EJBs, se siguen los mismos pasos que se usan para crear objetos con RMI. Se crea una interface que ofrece las operaciones o servicios provistos por los EJBs. Luego se crea una clase que implemente esta interface. Cuando se despliega un EJB en un servidor de aplicación, el EJB es asociado con un nombre en un registro. Los clientes pueden buscar el EJB en el registro, y llamar a sus métodos remotos.

Los EJBs son de tres tipos: *sesion beans*, *entity beans* y *driven-message beans*. Los *beans* de sesión, como su nombre lo indica, vive mientras exista una conversación o sesión, entre la aplicación cliente y los *beans*. Dependiendo del diseño, un *bean* de sesión, puede mantener el estado durante la sesión o pueden ser sin estado.

Los *beans* de entidad representan objetos de negocio (como clientes, facturas y productos) en el domino de la aplicación. Estos objetos de negocio son persistentes, por eso ellos pueden ser guardados y recuperados cuando se requieran. La arquitectura Java EE ofrece bastante flexibilidad para el modelo persistente. El trabajo de almacenamiento y recuperación de la información del estado del *bean* se le deja al contenedor.

El tercer tipo de EJB, son los *beans* dirigidos por mensajes, que ofrecen un modelo por componentes de servicios que están a la escucha de mensajes, esto se ilustra en la figura 1. La plataforma Java EE incluye un servicio de cola de mensajes (Message Queue) que permite a las aplicaciones postear mensajes en una cola. La ventaja de comunicarse así cae en que los emisores y receptores de mensajes no necesitan saber nada sobre el otro. Solo necesitan saber sobre el servicio de cola de mensajes.

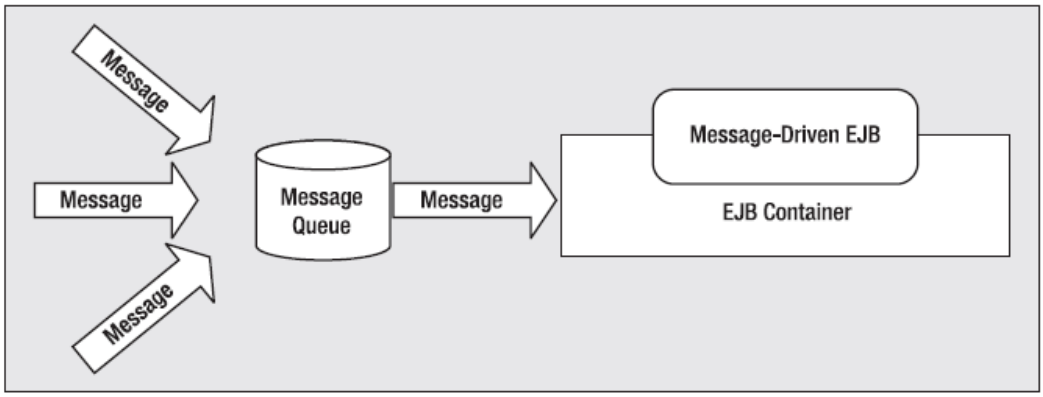


**Figura 1**. Diagrama de un sistema de mensajería asíncrona típico.

Los emisores de mensajes pueden enviar un mensaje a la cola, sabiendo que algún servicio obtendrá el mensaje, pero no sabiendo exactamente quien recibe el mensaje o incluso si será recibido. Los receptores pueden suscribirse a la cola y obtener los mensajes que le son interesados, sin saber quien envió. Esta manera de comunicación es la que se le denomina asíncrona.

### 1.5.7. Servicios Web

La Internet se esta convirtiendo cada vez más en el *backbone* de las aplicaciones de negocio. Sistemas externos que suplen necesidades de las aplicaciones webs que manejan reglas de negocio son considerados servicios web. El World Wide Web Consortium (W3C), en un esfuerzo de unificar, como los servicios de Web deberían ser publicados, descubiertos y accedidos, ha buscado proveer una definición más concreta para los servicios web. Aquí esta una definición de la arquitectura de los servicios Web (www.w3.org/TR/ws-arch):

*Un servicio Web es un sistema de software diseñado para soportar la interoperabilidad maquina-a-maquina sobre una red. Teniendo ésta una interfase descrita en un formato procesable (específicamente WSDL). Los otros sistemas interactuaran con el servicio web de la manera descrita por su descripción usando mensajes SOAP, usualmente convenido utilizar HTTP con una serialización XML en conjunción con otros estándares Web relacionados.*

Esta definición contiene algunos requerimientos específicos:

- Un servicio web permite a una computadora solicitar algún servicio de otra computadora.
- La descripción del servicio es procesable por una maquina.
- Los sistemas acceden a un servicio usando mensajes XML enviados sobre HTTP.

El W3C ha establecido el *Web Service Description Languaje* (WSDL) como el formato XML que es usado por los servicios web para describir sus servicios y además como los clientes pueden acceder a estos servicios. Para llamar estos servicios, los clientes deben estar en capacidad de poder obtener estas definiciones. El registro XML ofrece la posibilidad de publicar la descripción del servicio, búsqueda de servicios, y obtención de información WSDL que describe las especificaciones de dicho servicio.
No solo existe XML para la especificación de servicios, también están: ebXML y Universal Description, Discovery e Integration (UDDI). La API JAXR ofrece una implementación independiente para acceder a esos registros XML.

El SOAP (Simple Object Access Protocol) es el *lingua franca* usado por servicios web y usado por clientes para la invocación de estos servicios, paso de parámetros y obtención de resultados. SOAP define los estándares de un mensaje XML y mapeo de datos requerido para que la aplicación de un cliente llame un servicio web y pase parámetros. La API JAX-RPC ofrece una fácil manera para el desarrollo de interfaces que enmascara la complejidad de la tubería.

No sorpresivamente, la arquitectura Java EE ofrece un contenedor que aloja los servicios web y un componente para el fácil despliegue de los servicios Web.

# 2. TECNOLOGÍA JAVASERVER PAGES (JSP)

Los JSP son componentes en una aplicación Java EE que consiste de código HTML y Java. Esto no se debe confundir.

La tecnología JavaServer Pages (JSP) nos permite poner segmentos de código *Servlet* directamente dentro de una página HTML estática. Cuando el navegador carga una página JSP, se ejecuta el código del *Servlet* y el servidor de aplicaciones crea, compila, carga y ejecuta un Servlet en segundo plano para ejecutar los segmentos de código Servlet y devolver una página HTML o imprimir un informe XML.

## 2.1. DESARROLLO DE JSP

El proceso de desarrollar una página JSP que sea capaz de responder a las peticiones de los clientes implica tres pasos principales:

- **Creación:** el desarrollador crea el archivo fuente JSP que contiene HTML y código Java embebido.
- **Despliegue:** el JSP es instalado en el servidor. Este puede ser un servidor Java EE completo o un servidor JSP independiente.
- **Traducción y compilación:** el contenedor JSP traduce el código HTML y Java en un archivo de código fuente Java. Este archivo es compilado en una clase Java, que luego es ejecutada por el servidor. El archivo clase creado del JSP es conocido como *the JSP page implementation class*.

## 2.2 CICLO DE VIDA DE UN JSP

Una vez que la compilación es realizada, el ciclo de vida de un JSP realiza las siguientes fases:

- **Carga e instanciación:** el servidor encuentra o crea *the JSP page implementation class* para la pagina JSP y lo carga en la JVM. Después de que la clase es cargada, la JVM crea una instancia de la clase. Este puede ocurrir inmediatamente después de la carga, o a la primera petición que es solicitada.
- **Inicialización**: el objeto página del JSP es inicializado. Si se necesita ejecutar código durante la inicialización, se puede añadir un método a la página para que sea llamado durante la inicialización.
- **Procesamiento de petición:** el objeto página responde a peticiones. Note que una simple instancia de un objeto tratará todas las peticiones. Después de realizar su procesamiento, una respuesta es retornada al cliente. La respuesta consiste únicamente en etiquetas de HTML u otros datos; ninguna de los códigos fuente de Java es enviado al cliente.
- **Fin de vida**: el servidor deja de enviar peticiones al JSP. Después de que todas las peticiones terminaron su procesamiento, cualquier instancia de la clase es liberada. Esto por lo general ocurre cuando el servidor se cierra, pero también puede ocurrir en otros momentos, cuando el servidor necesita conservar recursos, cuando es detectada una actualización del archivo fuente JSP, o cuando necesita terminar la instancia por otros motivos.

La figura 2 muestra el ciclo de vida de una petición hecha por un cliente.

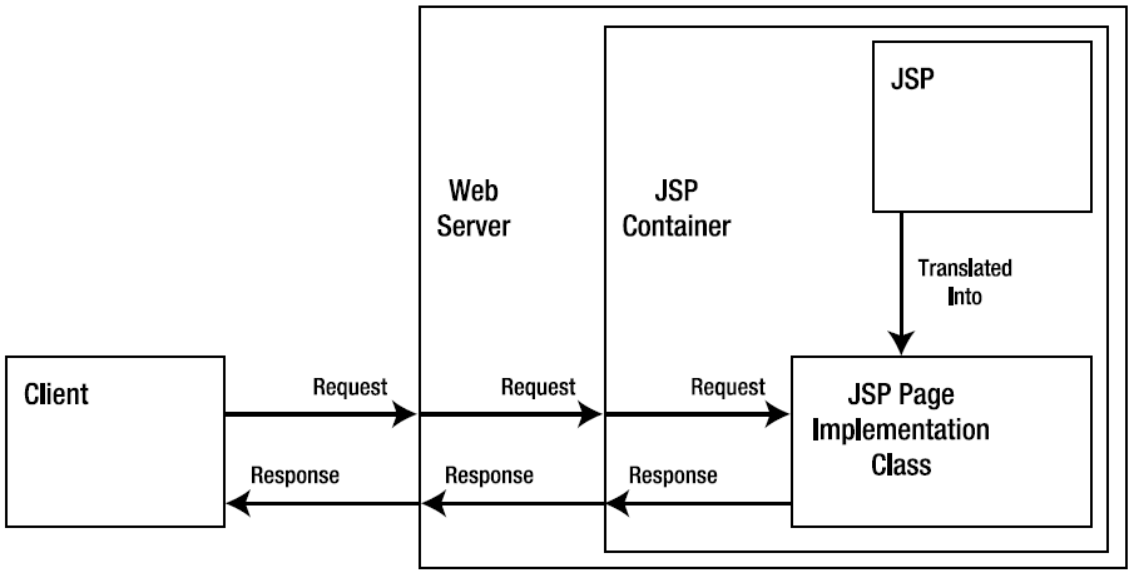


**Figura 2.** Un archivo fuente JSP es compilado en una *JSP page implementation class.* Cuando el servidor recibe una petición del JSP, la petición es enviada al contenedor, la cual pasa la petición al JSP correcto, luego la respuesta sigue el camino inverso.

Cuando un cliente envía una petición de un JSP, el servidor Web pasa esa petición al contenedor JSP, y el contenedor JSP determina cual *JSP page implementation class* debería manejar la petición. El contenedor JSP entonces llama a un método de la *JSP page implementation class,* el cual procesa la petición y devuelve una respuesta a través del contenedor y el servidor Web al cliente. En general, a este proceso se le es referido como “una petición es enviada a un JSP”.

### 2.3. COMPONENTES DE LOS JSP

Ahora que ya se sabe como es funcionamiento de un JSP, se mostrara como están compuestos. Miremos la siguiente línea de código JSP:

```
<html><body><p>Hello, World!</p></body></html>
```

Evidentemente, esto no es un ejemplo de un JSP. Sin embargo las etiquetas HTML están de la forman correcta y son validas para un archivo JSP. Se podría guardar este código en un archivo llamado *HelloWorld.jsp* e instalarla en una aplicación web, y el servidor

podría acceder a este como una fuente JSP. El punto es que las paginas JSP lucen como paginas HTML.

El anterior no es un buen ejemplo, ya que este no es dinámico de ninguna forma. Si las paginas JSP no contienen código Java, entonces se debería hacerla simplemente como paginas HTML estáticas. Las paginas JSP tienen la intensión de tener un comportamiento dinámico; ellas cambian dependiendo de las solitudes de los clientes. Este comportamiento dinámico se da por el código Java embebido que estas tienen. Se puede pensar en las paginas JSP como paginas webs con *bits* Java embebido.

Sin embargo, no solo hay que escribir código Java en la página, se necesita una manera de decirle al traductor JSP que *bits* son código Java y que *bits* son HTML normal. Para hacer eso, la especificación JSP define etiquetas parecidas a las HTML o XML, que encierran el código Java. Estas etiquetas se dan en tres categorías:

- Elementos de Directivas
- Elementos de Script
- Elementos de Acción

### 2.3.1. Elementos de directivas

Los elementos de directivas proporcionan información de las páginas al contenedor JSP. Tres directivas están disponibles: **page**, **include**, y **taglib**. Una página JSP sencilla puede tener múltiples instancias de las directivas **page** e **include**.

#### 2.3.1.1 Directiva Page

La directiva **page** es usada para especificar los atributos de página. La forma de la directiva **page** en estilo JSP es:

```
<%@ page attributes %>
```

Así como todos los atributos HTML, los atributos deben ser pares de nombre/valor, con un signo igual (=) separando el nombre del valor, y el valor entre comillas. Se puede encontrar la lista completa de atributos y sus manejos en la especificación JSP, que se puede descargar de http://java.sun.com/products/jsp

La tabla1 muestra los atributos más usados cuando se empiezan a desarrollar paginas JSPs:

**Tabla1.** Tipos de atributos de la directiva *page* con sus descripciones

| **Atributo** | **Descripción** |
|---|---|
| import | Permite importar clases y paquetes Java en la página JSP, al transformar la página JSP a un *Servlet* estas directivas se sustituirán por instrucciones import en el *Servlet* resultante. Es la única directiva que se puede utilizar más de una vez dentro de la misma página. |
| Session | Indica si la página participa en una sesión. Los valores válidos son verdaderos o falsos. El valor por defecto es verdadero. Si es verdadero, la página participa en una sesión; si es falso, entonces no lo hace, y no puede acceder a ninguna información de sesión. |
| Info | Un String arbitrario. Este puede tener cualquier valor. Es proporcionado de modo que el JSP puede proveer un instrumento de dirección de la información sobre sus contenido, objetivo, nombre, etcétera. |
| isErrorPage | Esto es si la página es una página de error. Por defecto esto es falsa. |
| ContentType | Permite definir el tipo MIME de la respuesta |

| | |
|---|---|
| | que se enviara al usuario. |
| IsThreadSafe | Admite valores true o false, si se indica true el *Servlet* resultado implementara la interfaz SingleThreadModel. |
| pageEncoding | Tipo de codificación de la página. Por defecto ISO-8859-1 (escritura latina) para JSP-style y UTF-8 (codificación 8-bits en Unicode) para etiquetas de XML-style. |
| errorPage | Es el URL de la pagina que se debería enviar al cliente, si ocurriera un error en la pagina. Si no se especifica ningún URL, el contenedor usa página por defecto. |

#### 2.3.1.2 Directiva Include.

La directiva *include* es usada para incluir otra página dentro de la actual página. La directiva *include* de la forma JSP es:

```
<%@ include attributes %>
```

Generalmente esta se incluye en la cabecera o en el pie de pagina, pero en realidad no se esta limitado a eso, esta puede estar en cualquier parte del contenido. Esta se usa cuando se tiene datos estándares que se desean incluir en múltiples paginas JSP. El archivo que contiene los datos estándares es incluido cuando la página es traducida en su código Java.

Esta directiva tiene un simple atributo llamado *file.* El atributo *file* específica el nombre del archivo que va a ser incluido en la posición actual del archivo. El archivo incluido puede ser cualquier pagina HTML o JSP, o fragmento de una página. El archivo es especificado usando un URL a un archivo dentro de la aplicación Web; la ruta es relativa al archivo JSP.

#### 2.3.1.3 Directiva *taglib*.

Esta directiva permite importar dentro de la página JSP una biblioteca de *tags* personalizados. Con esto se pretende conseguir una mayor separación entre la lógica de negocio y la presentación de modo que en los *tags* personalizados se puede incluir lógica de negocio que debe implementar un programador (por ejemplo acceso a bases de datos), y el diseñador sólo tiene que hacer usos de esos *tags* personalizados dentro de su JSP.

La sintaxis para incluir una *taglib* es la siguiente:

```
<%@taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

Una vez incluida la biblioteca de *tags* dentro del JSP se pueden usar los *tags* definidos dentro de esta biblioteca con la siguiente sintaxis:

```
<prefix:tagName attributeName="attributeNameValue" >body</prefix:tagName>
```

### 2.3.2. **Elementos de scripting**

Los elementos scripting son los elementos en la página que incluyen el código Java. Hay tres subformas de este elemento: declaraciones, scriptlets, y expresiones.

#### 2.3.2.1 Declaraciones.

Una declaración es usada para declarar, y opcionalmente definir, una variable Java o un método. Esto trabaja así como cualquier declaración dentro de un archivo de código fuente Java. La forma JSP del elemento de declaración es:

```
<%! declaración %>
```

La declaración aparece sólo dentro de la página JSP traducida, pero no en la salida al cliente. Por ejemplo, para declarar un vector en una pagina JSP, se usaría de la siguiente forma:

```
<%! Vector v = new Vector(); %>
```

Las declaraciones JSP nos permiten configurar variables para su uso posterior en expresiones o *scriptlets*. También podemos declarar variables dentro de expresiones o scriptlets en el momento de usarlas.

Las declaraciones van encerradas entre etiquetas de declaración <%! y %>. Podemos tener varias declaraciones. Por ejemplo:

```
 <%! double bonus; String text; %>
<%! String strMult, socsec; %>
<%! Integer integerMult; %>
<%! int multiplier; %>
<%! double bonus; %>
```

### 2.3.2.2. Scriptlets

Los *Scriptlets* contienen sentencias de código Java. El código en el *scriptlet* aparece en el JSP traducido, pero no en la salida al cliente. El elemento *scriptlet* de la forma JSP es:

```
<% scriptlet code %>
```

Cualquier declaración de código de Java legal puede aparecer dentro de un scriptlet Por ejemplo, para repetir el ¡frase “Hola, Mundo!” diez veces en la página de salida, se podría usar el siguiente scriptlet:

```
<%for (int i = 0; i < 10; i++) {%>
        Hello, World!
<%}%>
```

Los scriptlets JSP nos permiten embeber segmentos de código java dentro de una página JSP. El código embebido se inserta directamente en el *Servlet* generado que se ejecuta cuando se pide la página. Este *scriptlet* usa las variables declaradas en las directivas descritas arriba. Los *Scriptlets* van encerradas entre etiquetas <% y %>.

```
<%
        strMult = request.getParameter("MULTIPLIER");
        socsec = request.getParameter("SOCSEC");
        integerMult = new Integer(strMult);
        multiplier = integerMult.intValue();
        bonus = 100.00;
%>
```

#### 2.3.2.3. Expresiones

Permiten indicar una expresión que se evaluará y cuyo resultado se transformará en un *String* y se enviará como parte del HTML que se devuelva como resultado de la petición. En el siguiente ejemplo se muestras la hora a la que se realizó la petición de la página:

```
Hora Actual: <%= new java.util.Date() %>
```

Las expresiones son usadas para sacar el valor de una expresión al cliente. La forma JSP del elemento de expresión es:

```
<%= expression %>
```

Las expresiones JSP nos permiten recuperar dinámicamente o calcular valores a insertar directamente en la página JSP. En este ejemplo, una expresión recupera el número del seguro social desde el bean de entidad *Bonus* y lo pone en la página JSP.

```
<H1>Bonus Calculation</H1>
Social security number retrieved:
<%= record.getSocSec() %>
<P>
Bonus Amount retrieved: <%= record.getBonus() %>
<P>
```

### 2.3.3. Elementos de Acción

Las acciones estándares son definidas por la especificación JSP. Son parecidas a las etiquetas HTML, pero estas hacen que la página realice una acción, de ahí el nombre. También es posible crear acciones personalizadas, que son conocidas como *custom actions*, estas nos permiten extender la implementación JSP con nuevas características y

ocultar mucha complejidad a los diseñadores visuales que necesitan buscar la página JSP y modificarla.

Las etiquetas específicas JSP son las siguientes:

<jsp:useBean>
<jsp:setProperty>
<jsp:getProperty>
<jsp:param>
<jsp:include>
<jsp:foward>
<jsp:plugin>
<jsp:params>
<jsp:fallback>
<jsp:attribute>
<jsp:body>
<jsp:invoke>
<jsp:doBody>

jsp:forward y jsp:include indican al motor JSP que pase de la página actual a otra página JSP.

jsp:useBean, jsp:setProperty, jsp:getProperty nos permiten embeber y utilizar tecnología JavaBeans en páginas JSP.

jsp:plugin descarga automáticamente el Plug-In Java al cliente para ejecutar applet en la plataforma Java adecuada.

Un JavaBean es simplemente una clase Java que sigue ciertos requerimientos.

## 2.4. UTILIZACIÓN DE OBJETOS IMPLÍCITOS.

Las páginas JSPs también pueden tener acceso a la petición del cliente directamente. Las peticiones echas por los clientes pasan por un objeto llamado *request*. Estos objetos están implícitos porque un JSP accede a ellos y pude usarlos sin necesidad de declarar ni inicializar los objetos.

- request
- response
- out
- session
- config
- exception
- application

### 2.4.1 El Objeto request

Las paginas JSP son componentes web que responden a un proceso de petición HTTP. El objeto implícito *request* representa esta petición HTTP. A través del objeto *request*, se puede tener acceso a los encabezado HTTP, los parámetros de petición, y otras información acerca de las peticiones. Con frecuencia se usara este objeto para leer parámetros de petición.

Cuando un navegador suministra una petición al servidor, este puede enviar información junto con la petición en la forma de parámetros de petición, esto se puede hacer de dos formas:

- Parámetros URL-Codificados: Estos parámetros son adjuntados a la petición URL como un String. El parámetro comienza con un símbolo de interrogación (?), seguido por las parejas nombre/valor de todos los parámetros, con cada nombre y valor lo delimitamos por un signo igual (=), y cada pareja delimitada por un ampersand (&):

```
http://www.myserver.com/path/to/resource?name1=value1&name2=value2
```

- Parámetros codificados: estos parámetros son suministrados como un resultado de una sumisión de forma. Ellos tienen el mismo formato que los URL-decodificados pero estos van incluidos con el cuerpo de petición y no anexados al URL solicitado.

Estos parámetros de petición pueden ser leídos a través de varios métodos del objeto *request*:

```
String request.getParameter(String name);
String[] request.getParameterValues(String name);
Enumeration request.getParameterNames();
Map getParameterMap();
```

El método *getParameter(String)* devuelve el valor del parámetro con el nombre dado. Si el nombre del parámetro tiene varios valores (por ejemplo, cuando en una forma suministra el valor de un checkboxes), este método devuelve el primer valor. Para parámetros con varios valores, *getParameterValues(String)* devuelve todos los valores para determinado nombre. El método *getParameterMap()* devuelve todos los parámetros como parejas de nombres/valores.

### 2.4.2. EL Objeto response

Este encapsula la respuesta a la aplicación cliente. Algunas de las tareas que se puede hacer usando el objeto respuesta es poner cabeceras, poner cookies para los clientes, y redirigir la respuesta al cliente. Se puede ejecutar estas funciones con los siguientes métodos:

```
public void addHeader(String name, String value)
public void addCookie(Cookie cookie)
public void sendRedirect(String location)
```

El objeto *response* es una instancia de java.servlet.HttpServletResponse, y su alcance es en toda la pagina.

### 2.4.3. El objeto out

El objeto implícito *out* es una referencia a un flujo de salida que se puede usar con *scriptles.* Usando el objeto *out*, el scriptlet puede escribir datos a las respuestas que son enviadas al cliente. Por ejemplo:

```
<%Iterator categories = faqs.getAllCategories();
  while (categories.hasNext()) {
        String category = (String)categories.next();
        out.println("<p><a href=\"" + replaceUnderscore(category) + "\">" +
        category + "</a></p>");
}
%>
```

### 2.4.4. El Objeto session

HTTP es un protocolo *stateless*[4]. Así como un servidor Web es concebido, cada petición de un cliente es una nueva petición, con nada para unirlo a peticiones anteriores. Sin embargo, en una aplicación Web, la interacción de un cliente con la aplicación a menudo atravesará muchas peticiones y respuestas.

Para unir todas estas interacciones separadas en una conversación coherente entre el cliente y la aplicación, la aplicación Web usa el concepto de *Sesión.* Una sesión se refiere a la conversación entre un cliente y un servidor.

Los componentes JSP en una aplicación Web automáticamente participan en la sesión de un cliente dado, sin necesidad de hacer algo especial. Cualquier pagina JSP que use la directiva *page* para poner el atributo de sesión en falso, no le es posible acceder al objeto *session*, y así no puede participar en la sesión.

Usando el objeto *session*, la página puede guardar información acerca del cliente o de la interacción con el cliente. La información es guardada en el objeto *session,* así como cuando se guarda en un *Hashtable* o en un *HashMap*. Esto significa que una pagina JSP

[4] Stateless: Literalmente significa sin estado, lo que quiere decir que no guarda ninguna información trascedente.

solo puede guardar objetos, y no primitivas Java durante la sesión. Para almacenar primitivas Java, se necesita usar una de las clases *wrapper* como *Integer* o *Boolean*. Los métodos para almacenar y recuperar datos de sesión se muestran a continuación:

```
Object setAttribute(String name, Object value);
Object getAttribute(String name);
Enumeration getAttributeNames();
void removeAttribute(String name);
```

Cuando otros componentes en la aplicación Web reciben una petición, ellos pueden acceder a los datos de sesión que fueron guardados por otros componentes. Ellos pueden cambiar la información en la sesión o añadirle nueva información.

Normalmente, el desarrollador no tiene que escribir el código en su página para manejar la sesión. El servidor crea el objeto de sesión y asocia peticiones de cliente con una sesión particular. Sin embargo, esta asociación normalmente sucede por medio del uso de las *cookies*, que son enviadas al cliente. La *cookie* mantiene la ID de la sesión; cuando el navegador devuelve la *cookie* al servidor, el servidor usa la sesión ID para asociar la petición a una sesión.

Cuando el navegador no acepta *cookies*, el servidor retrocede a un esquema llamado *URL rewriting* para mantener la sesión. Si el servidor tiene la posibilidad de usar URL *rewriting*, la página tiene que volver a reescribir cualquier URL embebido. Esto se hace por medio un método del objeto *response*:

```
response.encodeURL(String);
response.encodeRedirectURL(String);
```

El segundo método es usado cuando el URL será enviado como una redirección al navegador, usando el método *response.sendRedirect().* El primer método es usado para todo los otros URLs.

El objeto de *session* tiene el alcance de sesión, y todos los objetos almacenados en el objeto de sesión también tienen el alcance de sesión. El objeto de sesión es una instancia de *javax.servlet.http.HttpSession*.

### 2.4.5. El Objeto config.

El objeto de *config* es usado para obtener parámetros de inicialización JSP específicos. Estos parámetros de inicialización son puestos en el descriptor de despliegue, pero son específicos a una sola página. Los parámetros de inicialización de JSP son puestos en el elemento <servlet> del descriptor de despliegue. Este es el porque la implementación del JSP (la clase Java que es compilada de la página del JSP) es una clase *Servlet*. El elemento <servlet> con el elemento <init-param> tiene la siguiente forma:

```
<servlet>
        <servlet-name>StockList</servlet-name>
        <servlet-class>web.StockListServlet</servlet-class>
        <init-param>
                <param-name>name</param-name>
                <param-value>value</param-value>
        </init-param>
</servlet>
```

Si los parámetros de inicialización JSP son definidos en el descriptor de despliegue, se puede tener acceso a ellos usando este método:

```
config.getInitParameter(String name);
```

### 2.4.6. El Objeto exception

El objeto *exception* está disponible sólo dentro de páginas de error. Esto es una referencia al objeto *java.lang.Throwable* que hizo que el servidor llamara la página de error. Se usaría como cualquier otro objeto de *Error* o *Exception* en un bloque *try-catch* de un segmento de código. El objeto *exception* tiene el alcance de página.

### 2.4.7. El Objeto application

El objeto *application* representa el ambiente de la aplicación web. Se usará este objeto para obtener parámetros de configuración a nivel de aplicación. Dentro del descriptor de despliegue, se puede poner parámetros de aplicación usando este elemento:

```
<webapp>
        <context-param>
                <param-name>name</param-name>
                <param-value>value</param-value>
        </context-param>
</webapp>
```

El valor del parámetro puede ser accedido usando el siguiente método:

```
application.getInitParameter(String name);
```

# 3. LA TECNOLOGÍA JAVASERVER FACES

Los JSF están diseñados para simplificar el desarrollo de aplicaciones web, estos hacen fácil la construcción de componentes de interfaz y paginas de usuario, además facilitan la conexión de esos componentes con los objetos de negocio.

Los principales componentes de la tecnología JavaServer Faces son:

- Un API y una implementación de referencia para: representar componentes UI y manejar su estado; manejo de eventos, validación del lado del servidor y conversión de datos; definir la navegación entre páginas; soportar internacionalización y accesibilidad; y proporcionar extensibilidad para todas estas características.
- Una librería de etiquetas JavaServer Pages (JSP) personalizadas para dibujar componentes UI dentro de una página JSP.

Este modelo de programación bien definido y la librería de etiquetas para componentes UI facilitan de forma significativa la tarea de la construcción y mantenimiento de aplicaciones Web con UIs del lado del servidor.

Con un mínimo esfuerzo, podemos:

- Conectar eventos generados en el cliente al código de la aplicación en el lado del servidor.
- Mapear componentes UI a una página de datos del lado del servidor.
- Construir un UI con componentes reutilizables y extensibles.
- Grabar y restaurar el estado del UI más allá de la vida de las peticiones de servidor.

Como se puede apreciar en la figura 3, la interfase de usuario que creamos con la tecnología JavaServer Faces (representado por *myUI* en el gráfico) se ejecuta en el servidor y se renderiza en el cliente.

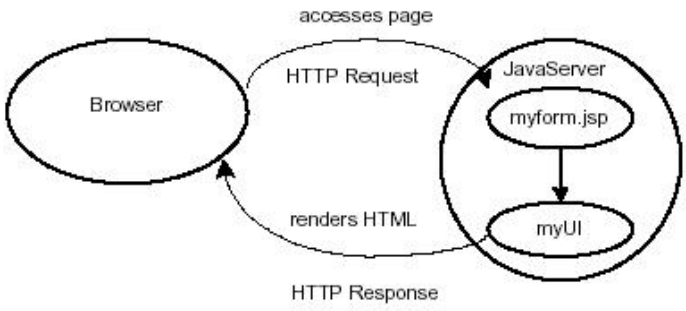


**Figura 3.** Procesamiento de una petición hecha por un cliente, usándose JavaServer Faces

La página JSP, *myform.jsp*, dibuja los componentes de la interface de usuario con etiquetas personalizadas definidas por la tecnología JavaServer Faces. El UI de la aplicación Web maneja los objetos referenciados por la página JSP:

- Los objetos componentes que mapean las etiquetas sobre la página JSP.
- Los manejadores de eventos, validadores, y los conversores que está registrados en los componentes.
- Los objetos del modelo que encapsulan los datos y las funcionalidades de los componentes específicos de la aplicación.

### 3.1. ¿POR QUE USAR JSF?

Una de las grandes ventajas de la tecnología JavaServer Faces es que ofrece una clara separación entre el comportamiento y la presentación. Las aplicaciones Web construidas con tecnología JSP conseguían parcialmente esta separación. Sin embargo, una aplicación JSP no puede mapear peticiones HTTP al manejo de eventos específicos de los componentes o manejar elementos UI como objetos con estado en el servidor. La tecnología JavaServer Faces nos permite construir aplicaciones Web que implementan una separación entre el comportamiento y la presentación tradicionalmente ofrecida por arquitectura UI del lado del cliente.

La separación de la lógica de la presentación también le permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas. Por ejemplo, los Autores de páginas sin experiencia en programación pueden usar

las etiquetas de componentes UI de la tecnología JavaServer Faces para enlazar código de la aplicación desde dentro de la página Web sin escribir ningún script.

Otro objetivo importante de la tecnología JavaServer Faces es mejorar los conceptos familiares de componente-UI y capa-Web sin limitarnos a una tecnología de script particular o un lenguaje de marcas. Aunque la tecnología JavaServer Faces incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, los APIs de la tecnología JavaServer Faces se han creado directamente sobre el API JavaServlet. Esto nos permite hacer algunas cosas: usar otra tecnología de presentación junto a JSP, crear nuestros propios componentes personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente.

Pero lo más importante, la tecnología JavaServer Faces proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos.

### 3.2. ¿QUÉ ES UNA APLICACIÓN JAVASERVER FACES?

En su mayoría, las aplicaciones JavaServer Faces son como cualquier otra aplicación Web Java. Se ejecutan en un contenedor Servlet Java, y típicamente contienen:

- Componentes JavaBeans (llamados objetos del modelo en tecnología JavaServer Faces) conteniendo datos y funcionalidades específicas de la aplicación.
- Manejadores de eventos.
- Páginas, cómo páginas JSP.
- Clases de utilidad del lado del servidor, como beans para acceder a las bases de datos.

Además de estos ítems, una aplicación JavaServer Faces también tiene:

- Una librería de etiquetas personalizadas para dibujar componentes UI en una página.

- Una librería de etiquetas personalizadas para representar manejadores de eventos, validadores, y otras acciones.
- Componentes UI representados como objetos con estado en el servidor.
- Validadores, manejadores de eventos y manejadores de navegación.

Toda aplicación JavaServer Faces debe incluir una librería de etiquetas personalizadas que define las etiquetas que representan componentes UI y una librería de etiquetas para representar otras acciones importantes, como validadores y manejadores de eventos. La implementación de JavaServer Faces proporciona estas dos librerías.

La librería de etiquetas de componentes elimina la necesidad de codificar componentes UI en HTML u otro lenguaje de marcas, resultando en componentes completamente reutilizables. Y, la librería "core" hace fácil registrar eventos, validadores y otras acciones de los componentes.

La librería de etiquetas de componentes puede se la librería **html_basic** incluida con la implementación de referencia de la tecnología JavaServer Faces, o podemos definir nuestra propia librería de etiquetas que dibuje componentes personalizados o que dibuje una salida distinta a HTML.

Otra ventaja importante de las aplicaciones JavaServer Faces es que los componentes UI de la página están representados en el servidor como objetos con estado. Esto permite a la aplicación manipular el estado del componente y conectar los eventos generados por el cliente a código en el lado del servidor.

Finalmente, la tecnología JavaServer Faces nos permite convertir y validar datos sobre componentes individuales y reportar cualquier error antes de que se actualicen los datos en el lado del servidor.

## 3.3. EL CICLO DE VIDA DE UNA PÁGINA JAVASERVER FACES

El ciclo de vida de una página JavaServer Faces page es similar al de una página JSP: El cliente hace una petición HTTP de la página y el servidor responde con la página traducida a HTML. Sin embargo, debido a las características extras que ofrece la tecnología JavaServer Faces, el ciclo de vida proporciona algunos servicios adicionales mediante la ejecución de algunos pasos extras.

Los pasos del ciclo de vida se ejecutan dependen de si la petición se originó o no desde una aplicación JavaServer Faces y si la respuesta es o no generada con la fase de renderizado del ciclo de vida de JavaServer Faces.

### 3.3.1 Escenarios de Procesamiento del Ciclo de Vida de una Petición

Una aplicación JavaServer Faces soporta dos tipos de diferentes respuestas y dos tipos de diferentes peticiones:

- Respuesta Faces: Una respuesta servlet que se generó mediante la ejecución de la fase Renderizar[5] la Respuesta del ciclo de vida de procesamiento de la respuesta.

- Respuesta No-Faces: Una respuesta servlet que no se generó mediante la ejecución de la fase Renderizar la Respuesta. Un ejemplo es una página JSP que no incorpora componentes JavaServer Faces.

- Petición Faces: Una petición servlet que fue enviada desde una Respuesta Faces previamente generada. Un ejemplo es un formulario enviado desde un componente de interfase de usuario JavaServer Faces, donde la URI de la petición identifica el árbol de componentes JavaServer Faces para usar el procesamiento de petición.

[5] Renderizar: Es la acción de asignar y calcular todas las propiedades de un objeto antes de mostrarlo en pantalla.

- Petición No-Faces: Una petición Servlet que fue enviada a un componente de aplicación como un servlet o una página JSP, en vez de directamente a un componente JavaServer Faces.

La combinación de estas peticiones y respuestas resulta en tres posibles escenarios del ciclo de vida que pueden existir en una aplicación JavaServer Faces:

- Escenario 1: Una Petición No-Faces genera una Respuesta Faces: Un ejemplo de este escenario es cuando se pulsa un enlace de una página HTML que abre una página que contiene componentes JavaServer Faces. Para dibujar una Respuesta Faces desde una petición No-Faces, una aplicación debe proporcionar un mapeo FacesServlet en la URL de la página que contiene componentes JavaServer Faces. FacesServlet acepta peticiones entrantes y pasa a la implementación del ciclo de vida para su procesamiento.

- Escenario 2: Una Petición Faces genera una Respuesta No-Faces: Algunas veces una aplicación JavaServer Faces podría necesitar redirigir la salida a un recurso diferente de la aplicación Web diferente o generar una respuesta que no contiene componentes JavaServer Faces. En estas situaciones, el desarrollador debe saltarse la fase de renderizado (Renderizar la Respuesta) llamando a FacesContext.responseComplete. FacesContext Contiene toda la información asociada con una Petición Faces particular. Este método se puede invocar durante las fases Aplicar los Valores de Respuesta, Procesar Validaciones o Actualizar los Valores del Modelo.

- Escenario 3: Una Petición Faces genera una Respuesta Faces: Es el escenario más común en el ciclo de vida de una aplicación JavaServer Faces. Este escenario implica componentes JavaServer Faces enviando una petición a una aplicación JavaServer Faces utilizando el FacesServlet. Como la petición ha sido manejada por la implementación JavaServer Faces, la aplicación no necesita pasos adicionales para generar la respuesta. Todos los oyentes, validadores y

conversores serán invocados automáticamente durante la fase apropiada del ciclo de vida estándar, como se describe en la siguiente sección.

### 3.3.2 Ciclo de Vida Estándar de Procesamiento de Peticiones

La mayoría de los usuarios de la tecnología JavaServer Faces no necesitarán conocer a fondo el ciclo de vida de procesamiento de una petición. Sin embargo, conociendo lo que la tecnología JavaServer Faces realiza para procesar una página, un desarrollador de aplicaciones JavaServer Faces no necesitará preocuparse de los problemas de renderizado asociados con otras tecnologías UI. Un ejemplo sería el cambio de estado de los componentes individuales. Si la selección de un componente como un checkbox afecta a la apariencia de otro componente de la página, la tecnología JavaServer Faces manejará este evento de la forma apropiada y no permitirá que se dibuje la página sin reflejar este cambio.

La figura 4 ilustra los pasos del ciclo de vida petición-respuesta JavaServer Faces

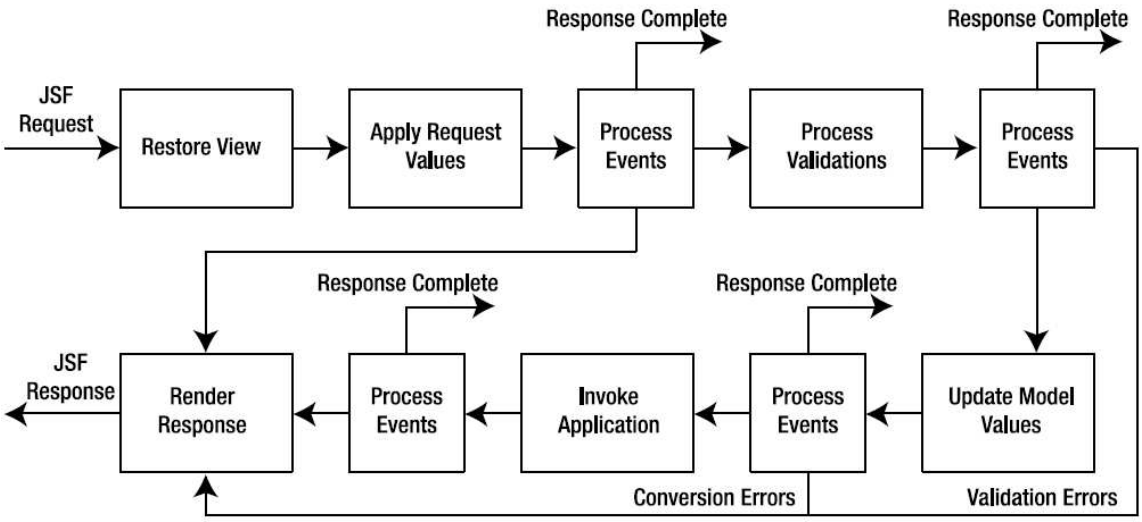


**Figura 4.** Tratado de una petición enviada a un JSF, pasando por las seis faces.

#### 3.3.2.1. Reconstituir el Árbol de Componentes

Cuando se hace una petición para una página JavaServer Faces, como cuando se pulsa sobre un enlace o un botón, la implementación JavaServer Faces comienza el estado Reconstituir el Árbol de Componentes.

Durante esta fase, la implementación JavaServer Faces construye el árbol de componentes de la página JavaServer Faces, conecta los manejadores de eventos y los validadores y graba el estado en el *FacesContext*.

#### 3.3.2.2. Aplicar Valores de la Petición

Una vez construido el árbol de componentes, cada componente del árbol extrae su nuevo valor desde los parámetros de la petición con su método decode. Entonces el valor es almacenado localmente en el componente. Si falla la conversión del valor, se genera un mensaje de error asociado con el componente y se pone en la cola de FacesContext. Este mensaje se mostrará durante la fase **Renderizar la Respuesta**, junto con cualquier error de validación resultante de la fase **Procesar Validaciones**.

Si durante esta fase se produce algún evento, la implementación JavaServer Faces emite los eventos a los oyentes interesados.

En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

En el caso del componente userNumber de la página greeting.jsp, el valor es cualquier cosa que el usuario introduzca en el campo. Como la propiedad del objeto del model unida al componente tiene un tipo Integer, la implementación JavaServer Faces convierte el valor de un String a un Integer.

En este momento, se han puesto los nuevos valores en los componentes y los mensajes y eventos se han puesto en sus colas.

#### 3.3.2.3. Procesar Validaciones

Durante esta fase, la implementación JavaServer Faces procesa todas las validaciones registradas con los componentes del árbol. Examina los atributos del componente que especifican las reglas de validación y compara esas reglas con el valor local almacenado en el componente. Si el valor local no es válido, la implementación JavaServer Faces añade un mensaje de error al FacesContext y el ciclo de vida avanza directamente hasta la fase Renderizar las Respuesta para que la página sea dibujada de nuevo incluyendo los mensajes de error. Si había errores de conversión de la fase Aplicar los Valores a la Petición, también se mostrarán.

En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se han disparado eventos durante esta fase, la implementación JavaServer Faces los emite a los oyentes interesados.

#### 3.3.2.4. Actualizar los Valores del Modelo

Una vez que la implementación JavaServer Faces determina que el dato es válido, puede pasar por el árbol de componentes y configurar los valores del objeto de modelo correspondiente con los valores locales de los componentes. Sólo se actualizarán los componentes que tenga expresiones valueRef. Si el dato local no se puede convertir a los tipos especificados por las propiedades del objeto del modelo, el ciclo de vida avanza directamente a la fase Renderizar las Respuesta, durante la que se dibujará de nuevo la página mostrando los errores, similar a lo que sucede con los errores de validación.
En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se han disparado eventos durante esta fase, la implementación JavaServer Faces los emite a los oyentes interesados.

#### 3.3.2.5. Invocar Aplicación

Durante esta fase, la implementación JavaServer Faces maneja cualquier evento a nivel de aplicación, como enviar un formulario o enlazar a otra página. En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

El oyente pasa la salida al NavigationHandler por defecto. Y éste contrasta la salida con las reglas de navegación definidas en el fichero de configuración de la aplicación para determinar qué página se debe mostrar luego.

Luego la implementación JavaServer Faces configura el árbol de componentes de la respuesta a esa nueva página. Finalmente, la implementación JavaServer Faces transfiere el control a la fase Renderizar la Respuesta.

#### 3.3.2.6. Renderizar la Respuesta

Durante esta fase, la implementación JavaServer Faces invoca las propiedades de codificación de los componentes y dibuja los componentes del árbol de componentes grabado en el FacesContext.

Si se encontraron errores durante las fases Aplicar los Valores a la Petición, Procesar Validaciones o Actualizar los Valores del Modelo, se dibujará la página original. Si las páginas contienen etiquetas output_errors, cualquier mensaje de error que haya en la cola se mostrará en la página.

Se pueden añadir nuevos componentes en el árbol si la aplicación incluye renderizadores personalizados, que definen cómo renderizar un componente. Después de que se haya renderizado el contenido del árbol, éste se graba para que las siguientes peticiones puedan acceder a él y esté disponible para la fase Reconstituir el Árbol de Componentes de las siguientes llamadas.

# 4. SERVLET

Se podría definir un Servlet como un programa escrito en Java que se ejecuta en el marco de un servicio de red, (un servidor HTTP, por ejemplo), y que recibe y responde a las peticiones de uno o más clientes.

Los Servlets son parecidos a los applets, pero en ves de corren en lado del cliente, este se ejecuta en el servidor de aplicación. Así como las paginas JSP, los Servlets son clases Java que son cargadas y ejecutadas en por un contenedor Servlet que puede corren por si solo como un componente, o pertenecer a un servidor Java EE.

Los *Servlets* estas diseñados para aceptar una petición de un cliente (usualmente, un navegador web), procesar esa petición y retornar una respuesta al cliente. Aunque todo ese proceso puede ocurrir en un Servlet, usualmente clases helper u otros componentes web como los Enterprise JavaBean realizaran el proceso de la lógica de negocio, dejando al Servlet libre para realizar solo el proceso de peticiones y respuestas.

## CARACTERÍSTICAS DE LOS *SERVLETS*

- Son independientes del servidor utilizado y de su sistema operativo, lo que quiere decir que a pesar de estar escritos en *Java*, el servidor puede estar escrito en cualquier lenguaje de programación, obteniéndose exactamente el mismo resultado que si lo estuviera en *Java*.

- Los *servlets* pueden llamar a otros *servlets*, e incluso a métodos concretos de otros *servlets*. De esta forma se puede distribuir de forma más eficiente el trabajo a realizar. Por ejemplo, se podría tener un *servlet* encargado de la interacción con los clientes y que llamara a otro *servlet* para que a su vez se encargara de la comunicación con una base de datos.

De igual forma, los *servlets* permiten redireccionar peticiones de servicios a otros *servlets* (en la misma máquina o en una máquina remota).

- Los *servlets* pueden obtener fácilmente información acerca del *cliente* (la permitida por el protocolo *HTTP*), tal como su dirección *IP*, el *puerto* que se utiliza en la llamada, el método utilizado (***GET***, ***POST***), etc.

- Permiten además la utilización de *cookies* y *sesiones*, de forma que se puede guardar información específica acerca de un usuario determinado, personalizando de esta forma la interacción cliente-servidor.

- Los *servlets* pueden actuar como enlace entre el cliente y una o varias *bases de datos* en arquitecturas *cliente-servidor de 3 capas* (si la base de datos está en un servidor distinto).

- Asimismo, pueden realizar tareas de *proxy* para un *applet*. Debido a las restricciones de seguridad, un *applet* no puede acceder directamente por ejemplo a un servidor de datos localizado en cualquier máquina remota, pero el *servlet* sí puede hacerlo de su parte.

- Al igual que los *programas CGI*, los *servlets* permiten la generación dinámica de código *HTML* dentro de una propia página *HTML*. Así, pueden emplearse *servlets* para la creación de contadores, banners, etc.

### 4.1. HTTP Y PROGRAMAS SERVIDOR

Aunque los Servlets fueran al principio queridos para trabajar con cualquier servidor, en la práctica los Servlets son usados sólo con servidores de web. De manera que en las aplicaciones Java EE, se desarrollarán Servlets que responden a peticiones de HTTP. El Servlet API provee una clase llamada HttpServlet exclusivamente para tratar con estas peticiones. La clase HttpServlet es diseñada para trabajar estrechamente con el protocolo HTTP.

El protocolo HTTP define la estructura de las peticiones que un cliente envía a un servidor de web, el formato el cual el cliente pueda presentar parámetros de petición, y el modo que el servidor responde. Los *HttpServlets* usan el mismo protocolo para manejar el servicio de peticiones que ellos reciben y devuelven respuestas a clientes.

### 4.1.1. Métodos de petición.

La especificación HTTP define varias peticiones que un cliente web (típicamente un navegador) puede hacer en un servidor web. Estos tipos de petición son llamados métodos. La tabla 2 muestra los siete métodos que son definidos por la especificación HTTP:

**Tabla 2.** Métodos de la especificación HTTP y sus descripciones

| **Método** | **Descripción** |
|---|---|
| GET | Recupera la información identificada por una petición de Identificador de Recurso Uniforme (URI). |
| POST | Se requiere que el servidor pase el cuerpo de la petición a la fuente identificada como *URI* por la petición, para procesar. |
| HEAD | Devuelve solo la cabecera de las respuestas que podrían ser devueltas por una petición *GET*. |
| PUT | Carga datos al servidor para ser almacenado en la petición dada *URI*. La diferencia principal entre este y *POST* es que el servidor no debería procesar más requerimientos de el put, pero simplemente almacenarlo en la petición URI. |
| DELETE | Borra los recursos identificados por la petición URI. |
| TRACE | Hace que el servidor devuelva la petición de mensaje. |
| OPTIONS | Pide al servidor la información sobre un recurso específico o sobre las capacidades del servidor en general. |

### 4.1.2. Como un Servidor Responde a Peticiones

Ya se sabe como un servidor web responde cuando se le hace la petición *GET* de una página Web HTML estática. Cuando se digita una dirección o se hace un *click* a un *link*, el servidor localiza el recurso identificado por el URI y devuelve ese recurso como parte de un mensaje HTTP al navegador web. En caso de una página Web, el navegador muestra la página Web.

Sin embargo, ¿Qué sucede, cuándo el recurso es un programa de lado de servidor? En este caso, el servidor tiene que interpretar el URI como una petición de un programa de lado de servidor, formatear los parámetros de petición en una forma que el programa los reconozca, y pase la petición a aquel programa. En los primeros días de la Web, un formato estandarizado llamado la Interfaz de Entrada Común (CGI) fue desarrollado para este fin.

Siempre que usted vea un URL que tiene /cgi/ dentro de la dirección, usted estará creando una petición a un programa de lado de servidor de ese tipo. El programa debe interpretar los parámetros de petición, ejecutar el procesamiento apropiado, y devolver una respuesta al cliente. En el pasado, el programa de servidor era por lo general escrito en lenguaje C o Perl, los cuales eran ejecutados en un proceso separado del servidor. Cada petición hace que un nuevo proceso fuera engendrado, y cuando el programa se completaba el procesamiento de la petición era terminado. Este era por lo general un recurso intensivo.

Java Servlets, y específicamente *HttpServlets*, proporciona algunas ventajas sobre programas CGI para aplicaciones de lado de servidor. Ellos pueden correr en el mismo proceso que el servidor, entonces los nuevos procesos no tienen que ser engendrados para cada petición. También, ellos son portables entre servidores (mientras ellos no usan ningún código específico de plataforma). Los programas de CGI escritos y compilados en C, por ejemplo, tienen que ser compilados de nuevo para un sistema operativo diferente.

## 4.2. EL MODELO SERVLET Y HTTPSERVLETS

La figura 5 muestra una vista simplificada de lo que pasa cuando un cliente hace una petición que es tratada por un Servlet.

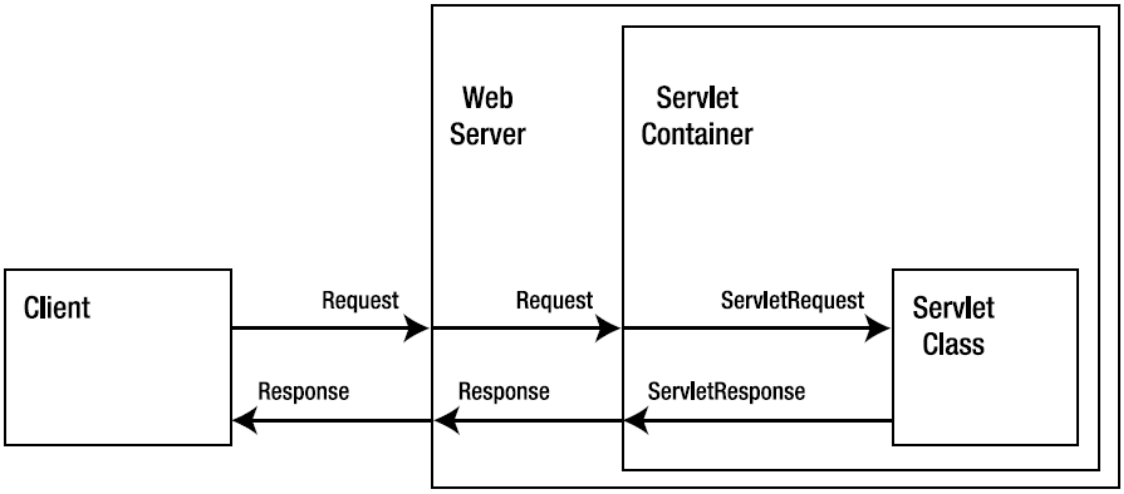


**Figura 5.** Una petición de un Servlet es pasada por el servidor al contenedor Servlet, que lo pasa a la clase Servlet.

Cuando un cliente (por lo general, pero no necesariamente, un navegador web) hace una petición al servidor, y el servidor determina que la petición es para un recurso Servlet, este pasa la petición al contenedor Servlet. El contenedor es el programa responsable de cargarlo, iniciación, llamado, y liberación de instancias de Servlet.

El contenedor Servlet toma la petición de HTTP; analiza su petición URI, las cabeceras, y el cuerpo; y almacenan todos aquellos datos dentro de un objeto que implementa la interfaz *javax.servlet.ServletRequest*.

Esto también crea una instancia de un objeto que implementa *javax.servlet.ServletResponse*. El objeto *response* encapsula la respuesta al cliente. El contenedor entonces llama un método de la clase Servlet, pasando los objetos de respuesta y petición. El Servlet trata la petición y devuelve una respuesta al cliente.

El Servlets puede enviar datos de salida directamente por la respuesta, como se mostro en la Figura 5. Sin embargo, en muchas aplicaciones web, el Servlet aceptará y tratará la petición, usando algún otro componente para generar la respuesta al cliente.

### 4.2.1. Diseño básico de un servlet

Así como los programas CGI, los Servlets están diseñados para responder a peticiones GET y POST, junto con todas las otras peticiones definidas para HTTP. Sin embargo, probablemente nunca tendrá que responder a algo además de GET o POST. La figura 6 muestra las clases que se usaran escribiendo Servlets.

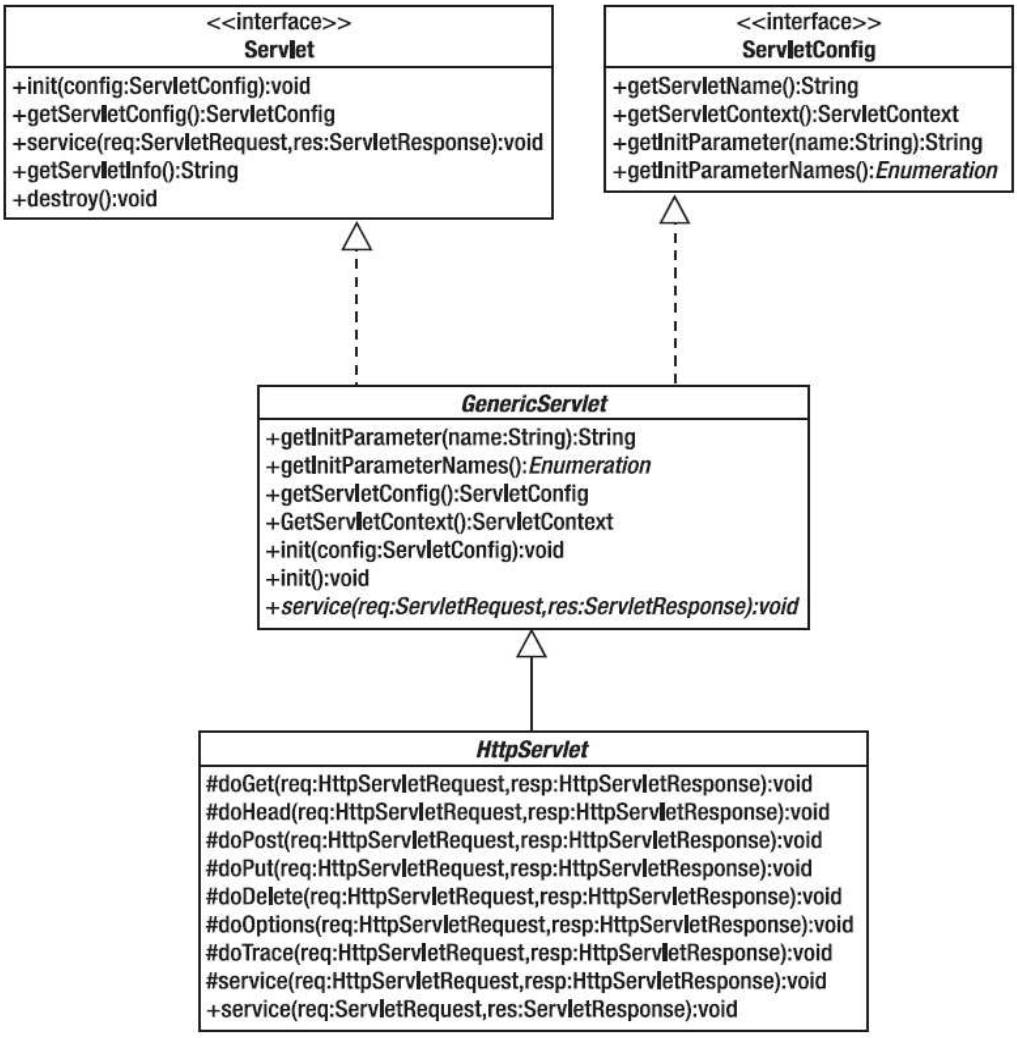


**Figura 6.** Las clases padres que proporcionan la funcionalidad básica de un Servlet

Cuando se escriben Servlets, usualmente extiendes de una clase llamada javax.servlet.http.HttpServlet. Este es una clase base que proporciona el apoyo a peticiones HTTP. La clase HttpServlet, por su parte, extiende javax.servlet.GenericServlet, que proporciona la funcionalidad básica de un Servlet. Finalmente, GenericServlet pone en práctica la interfaz Servlet, javax.servlet.Servlet. Esta también implementa un interfaz llamada ServletConfig, que permite y ofrece el acceso fácil a la información de configuración del Servlet.

Note que Servlet define sólo un pequeño número de métodos. Se puede adivinar probablemente que *init*() y *destroy*() no manejan ninguna petición. Los métodos *getServletConfig*() y *getServletInfo*() tampoco manejan peticiones. Esto deja sólo el *service*() para manejar peticiones.

#### 4.2.1.1 El método service().

Cuando un contenedor Servlet recibe una petición de un Servlet, el contenedor llama el método *service()* de los Servlets. Asi, un Servlet que implemente la interfaz *Servlet* puede implementar el método *service()* para manejar las peticiones. Sin embargo, los Servlet no necesitan implementar *service().* Las API Servlet proveen una clase llamada *HttpServlet,* la cual es una subclase de *GenericServlet*, así mismo una subclase de Servlet. Como se muestra en la Figura 6**.** La clase *HttpServlet* implementa dos versiones sobrecargadas de *service()*. Cuando el contenedor Servlet recive una petición, este llama *service(ServletRequest, ServletResponse)*, el cual llama *service(HttpServletRequest, HttpServletResponse).* Según el tipo de respuesta, *HttpServlet*, luego llama su Servlet para manejar un tipo de petición específico: GET, POST, o uno de los otros métodos HTTP.

#### 4.2.1.2. El método doPost() y doGet()

El HttpServlet es querido para responder a peticiones de HTTP, y debe manejar peticiones GET, POST, HEAD, ETC…. Por consiguiente, El HttpServlet define métodos adicionales. Esto define un *doGet()* para manejar peticiones *GET*, *doPost()* para manejar peticiones POST, etcétera; hay un *doXXX()* el método para cada método HTTP. A

excepción de *doTrace()* y *doOptions()*, estos métodos simplemente devuelven un mensaje de error al cliente que dice que el método no es soportado. Se espera que, los desarrolladores, escriban el Servlet para ampliar HttpServlet y sobrescriban los métodos que deseen soportar.

El HttpServlet ya pone en práctica un método *service()*, y esto determina el método correcto de *doXXX() de* pedir la petición HTTP. En una aplicación del mundo real, se debería determinar los métodos HTTP para ser soportados por su Servlet y sobrescribir la correspondencia del método *doXXX()*. Este casi siempre será *doPost()* o *doGet()*. Los métodos *doTrace()* y *doOptions()* son totalmente implementados por HttpServlet, entonces no se necesita sobrescribir en su Servlet.

Cuando el contenedor Servlet recibe la petición de HTTP, se traza un mapa del URI a un Servlet. Entonces llama el método *service*() del Servlet. Aumiendo que el Servlet extiende HttpServlet, y sobrescribe solo a *doPost()* o *doGet()*, la llamada al *service()* irá a la clase padre HttpServlet. El método *service()* determina qué método HTTP uso la petición y llama al método correcto *doXXX()*, como se muestra en la Figura 6.

Si su Servlet tiene aquel método, será llamado porque esto sobrescribe el mismo método en *HttpServlet*. El método *doXXX()* procesa la petición, genera una respuesta HTTP, y lo devuelve al cliente.

En la Figura 7, note que aunque HttpServlet y MyServlet sean mostrados en cajas separadas, ambos constituyen un solo objeto en el sistema: una instancia de MyServlet.

La firma actual de todos los métodos *doXXX()* es como se muestra a continuación:
*public void doXXX(HttpServletRequest request, HttpServletResponse response).* Cada método -*doPost()*, *doGet()*, etc. aceptan dos parámetros. El objeto de HttpServletRequest encapsula la petición al servidor. Esto contiene los datos para la petición, así como un poco de información de cabecera acerca de la petición.

Usando métodos definidos por el objeto *request*, el Servlet puede tener acceso a los datos presentados como parte de la petición. El objeto HttpServletResponse encapsula la respuesta al cliente. Usando el objeto *response* y sus métodos, se puede devolver una respuesta al cliente.

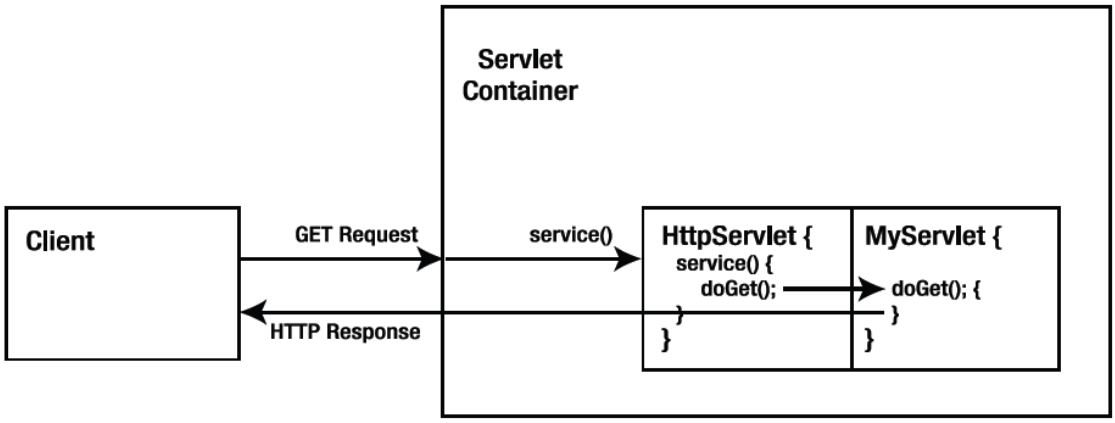


**Figura 7**. La petición es pasada al método *service()* de HttpServlet, que llama el método correcto en la subclase Servlet.

### 4.3. CICLO DE VIDA DE UN SERVLET

La especificación de Servlet define las cuatro etapas siguientes de lifecycle de Servlet:

1. Loading and instantiation
2. Initialization
3. Request handling
4. End of life

Estas cuatro etapas son ilustradas en la Figura 8:

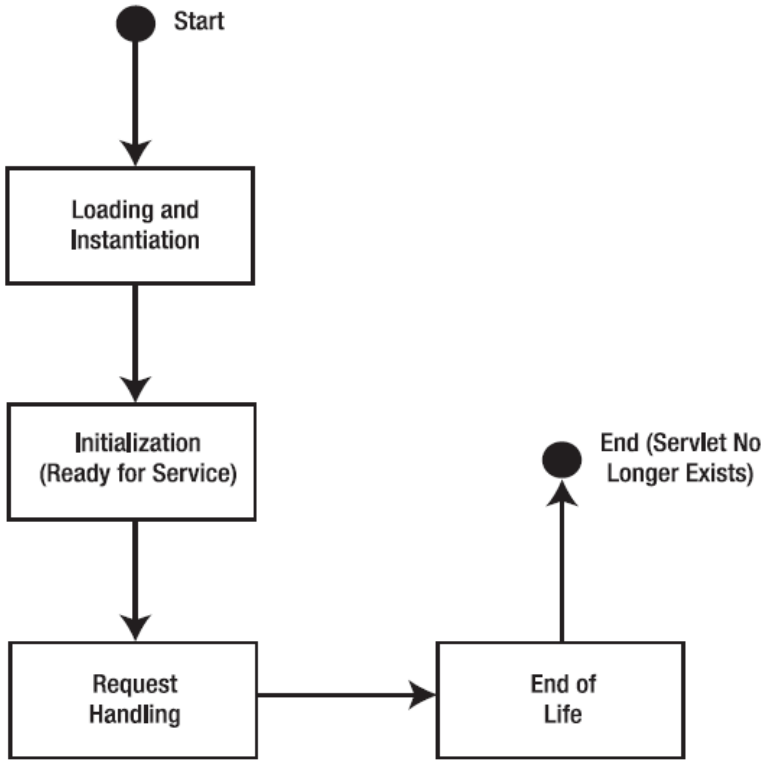


**Figura 8.** Desde la carga hasta el final de la vida, un Servlet pasa por varias fases distintas durante su *lifecycle*.

### 4.3.1. Carga e instanciación

En el primer estado del lifecycle, las clases servlet son cargadas desde los classpath y inicializadas por los contenedores Servlet. Los métodos que realizan estas páginas

El método que realiza esta etapa es el constructor Servlet. Sin embargo, a diferencia de las otras etapas, no tiene que proporcionar explícitamente el método para esta etapa.
¿Cómo sabe el contenedor Servlet cual Servlets cargar? Esto sabe leyendo el descriptor de despliegue. El contenedor Servlet lee cada archivo web.xml, y carga las clases Servlet identificadas en el descriptor de despliegue. Entonces el contenedor instancia cada Servlet llamando a su constructor sin argumento.

Ya que el contenedor Servlet dinámicamente carga e instancia Servlets, el no sabe sobre ningún constructor que usted crea lo que podría tomar parámetros. Así, puede llamar sólo

el constructor sin argumentos, y es inútil para usted para especificar a cualquier constructor además de uno que no toma ningunos argumentos. Ya que el compilador de Java proporciona a este constructor automáticamente cuando usted no suministra a un constructor, no hay ninguna necesidad de escribir un constructor en todos los Servlet. Este es por qué la clase Servlet no es necesario definir un constructor explícito.
¿Si usted no proporciona un constructor, cómo se inicializa el Servlet? Este es manejado en la siguiente fase del lifecycle, inicialización de Servlet.

### 4.3.2. Inicialización

Después de que el Servlet es cargado e instanciado, el Servlet debe ser inicializado. Este ocurre cuando el contenedor llama el método init(ServletConfig). Si su Servlet no tiene que realizar alguna inicialización, el Servlet no tiene que poner en práctica este método. El método es proporcionado por la clase padre GenericServlet.
Por eso el clase Login del Servlet no tenía un método init(). El init() método permite que el Servlet lea parámetros de inicialización o datos de configuración, inicialice recursos externos como uniones de base de datos, y realice otras actividades antiguas. El GenericServlet proporciona dos formas sobrecargadas del método:

public void init() throws ServletException
public void init(ServletConfig) throws ServletException

El descriptor de despliegue puede definir parámetros que se aplican al Servlet por el elemento <init-param>. El contenedor Servlet lee estos parámetros del archivo web.xml y los almacena como pares de nombre/valor en un objeto de ServletConfig. Como la interfaz Servlet define sólo init(ServletConfig), este es el método que el contenedor debe llamar. El GenericServlet pone en práctica este método de almacenar la referencia de ServletConfig, y luego llamar el método parameterless init() que esta definido.

Por lo tanto, para realizar inicialización, el Servlet tiene que implementar el método parameterless init(). Si se implementa init(), el init() será llamado por GenericServlet; y porque la referencia de ServletConfig ya esta almacenada, el método init() tendrá el acceso a todos los parámetros de inicialización almacenados en el.

Si se decia realmente poner en práctica init(ServletConfig) en el Servlet, el método en el Servlet debe llamar el método superclase init(ServletConfig):

```
public class Login extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
      //...Remainder of init() method
    }
        //...Rest of Servlet
}
```

La especificación Servlet requiere que init(ServletConfig) se completen con éxito antes de que cualquier petición puede ser atendida por el Servlet. Si el código encuentra un problema durante init(), se debería lanzar un ServletException o una subclase UnavailableException. Este dice al contenedor que había un problema con la inicialización y que esto no debería usar el Servlet para ninguna petición. UnavailableException de Utilización permite que se especifique una cantidad de tiempo que el Servlet no esta disponible. Después de este tiempo, el contenedor podría procesar de nuevo la llamada a init(). Usted puede especificar el tiempo no disponible para el UnavailableException que usa a este constructor:

```
public UnavailableException(String msg, int seconds)
```

El parámetro int puede ser cualquier número entero: negativo, cero, o positivo. Un valor negativo o cero indica que el Servlet no puede determinar cuando estará disponible otra vez. Por ejemplo, este podría ocurrir si el Servlet determina que un recurso exterior no está disponible; obviamente, el Servlet no puede estimar cuando el recurso exterior estará disponible.

Un valor positivo indica que el servidor debería tratar de inicializar el Servlet otra vez después de aquel número de segundos. Si usted quiere señalar que el Servlet es permanentemente no disponible, use a este constructor:

```
public UnavailableException(String msg)
```

Después de que el Servlet inicializa con éxito, se permite que el contenedor use el Servlet para manejar peticiones.

### 4.3.3. Fin de vida

Cuando el contenedor Servlet tiene que descargar el Servlet, porque está siendo cerrado o por alguna otra razón como un ServletException, el contenedor Servlet llamará al método *destroy()*. Sin embargo, previo a la destrucción por el método *destroy()*, el contenedor debe permitir el tiempo para cualquier hilo de petición que todavía intenta completar su procesamiento. Después de que estos hilos son terminados de procesar, o después de un período de tiempo de espera definido por servidor, se permite que el contenedor llame *destroy()*. Note que *destroy()* realmente no destruye el Servlet o hace que sea un garbage-collector (colector de basuras). Esto simplemente proporciona una oportunidad del Servlet para limpiar cualquier recurso que se uso o fue abierto. Obviamente, después de que este método es llamado, el contenedor no enviará más peticiones al Servlet. La firma de método destroy() es como se muestra a continuación:

```
public void destroy()
```

El método *destroy()* permite que el Servlet libere o limpie cualquier recurso que esto usa. Por ejemplo, esto puede cerrar uniones de base de datos o archivos, limpiar con agua cualquier corriente, o cerrar cualquier sockets.

Este método es implementado por la clase padre GenericServlet, si el Servlet no tiene que realizar alguna limpieza, el Servlet no tiene que poner en práctica este método. Después de que el método completa *destroy()*, el contenedor liberará sus referencias a la instancia Servlet, y la instancia Servlet será elegible para el colector de basura.

Aunque este método sea público, se supone para ser llamado sólo por el contenedor Servlet. Nunca se debería llamar el método *destroy()* dentro su Servlet, y no debería permitir que otro código llame este método.

# 5. ENTERPRISE JAVA BEANS

Hasta el momento hemos venido discutiendo temas concernientes al desarrollo de aplicaciones Java EE como: la interfaz de usuario y lógica de negocio. El principal mecanismo discutido hasta este momento para expresar la lógica de negocio ha sido los *JavaBean*s accedidos desde un código JSP o *Servlet*. Java EE tiene una poderosa facilidad dedicada a expresar la lógica de negocio en una aplicación y el acceso a una base de datos usando conceptos de JavaBeans. Esta facilidad son los *Enterprise JavaBeans*, conocidos como EJBs.

A continuación se explicaran los EJBs, los cuales son una muy importante capacidad que tiene la plataforma Java EE. Los EJBs ofrecen una infraestructura para el desarrollo y despliegue en las aplicaciones empresariales.

## 5.1 ENTENDIENDO LOS EJBs

Con frecuencia, la arquitectura de una aplicación esta formada de varias *tiers*, que cada una tiene su propia responsabilidad. Una arquitectura que consiste de tres *tiers* se muestra en la figura 9; se usó el lenguaje de modelado unificado (UML).

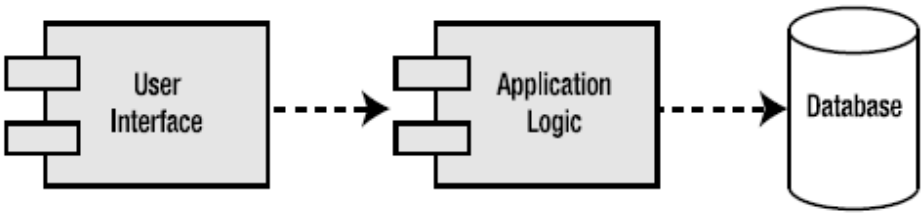


**Figura 9.** Un modelo clásico de una arquitectura *multi-tier*.

Los dos elementos de la izquierda en la figura 9 son llamados componentes en la notación UML. Estos componentes representan módulos de software. El diagrama describe que esta arquitectura es *multi-tier*. Las arquitecturas *multi-tier* tienen mucha

ventajas, como lo es la posibilidad de modificar una de las capas sin afectar ninguna de las otras.

Esta arquitectura (*multi-tier*) ofrece un excelente modelo para ver cómo los EJBs encajan en el diseño de un programa. Los EJBs ofrecen una capa lógica de aplicación y una abstracción en la capa de base datos. La capa lógica en la aplicación también es conocida como *middle tier*.

Los *JavaBeans* y Los *Enterprise JavaBeans* son dos cosas diferentes, pero por sus similitudes (y razones de *marketing*), comparten el nombre en común. Los JavaBeans son componentes construidos en Java que pueden ser usados en cualquier capa de una aplicación. Ellos están pensados para estar relacionados con los *Servlets*, y algunos componentes GUI. Los *Enterprise JavaBeans* son elementos más especializados, son componentes basados para funcionar en servidores, usados para construir la lógica de negocio y funcionalidad al acceso de datos en una aplicación.

### 5.1.1 ¿Por que Utilizar los EJBs?

Hoy en día, existen muchos servidores de aplicación, los cuales tienes una basta lista de características mostrando todo lo que pueden ofrecer. Ellos soportan la funcionalidad de la capa lógica. Algunas de las características de un servidor de aplicación incluyen lo siguiente:

- **Comunicación con el cliente:** El cliente, que a menudo es una interfaz de usuario, debe ser capas de llamar los métodos de los objetos que están el servidor de aplicación por medio de protocolos aceptados.

- **Administración de los estados de sesión:** Esto es, el monitoreo que se le puede aplicar a una sesión hecha por un cliente. Relacionémoslo con la sesión que se realizan en JSP y Servlets,

- **Administración de transacciones:** Algunas operaciones, como por ejemplo cuando se están actualizan los datos, deben ocurrir como una unidad de trabajo. Si una sola actualización falla, todas las demás deberían fallar.

- **Administración de conexiones a la base de datos:** Un servidor de aplicación debe conectarse a una base de datos, usando una piscina de conexiones para optimizar la disponibilidad.

- **Autenticación y autorización de usuarios:** Los usuarios de una aplicación deben logearse para propósitos de seguridad. La funcionalidad de una aplicación en la que un usuario se le permite el acceso esta basada en la asociación de un usuario con un ID.

- **Mensajería asíncrona:** A menudo, las aplicaciones necesitan comunicarse con otros sistemas de una manera asíncrona; esto es, sin esperar respuesta inmediata (o de recibido) del otro sistema. Esto requiere de un sistema de mensajería que este corriendo por debajo de la aplicación, adema que ofrezca garantía de entrega de esos mensajes asíncronos.

- **Administración del Servidor de Aplicación:** Los servidores de aplicación deben ser administración. Por ejemplo, necesitan ser monitoreados y reajustados.

Todo eso es posible realizarlo mediante el uso de EJBs, los cuales son muy flexibles a la hora de hacerles cualquier modificación.

### 5.1.2 Especificación de los EJBs

La especificación de los EJBs define una arquitectura común, que es impuesta a muchos proveedores para que desarrollen sus servidores de aplicación cumpliendo con esa especificación. En estos días los desarrolladores pueden escoger entre distintos servidores de aplicación, que cumplen con un estándar común.

Algunos de los servidores de aplicación EJB comerciales son: WebLogic (BEA), Sun ONE (Sun), Contenedores OC4J para base de datos Oracle 10g y WebSphere (IBM). También existen unos muy buenos servidores *open-source* como JBoss and JOnAS. Sun ofrece una implementación referencial *open-source* de las especificaciones de Java EE5 y EJB 3.0 que los desarrolladores puede utilizar para construir y probar sus aplicaciones para conformidad con esas especificaciones.

Actualmente esta bajo desarrollo, la referencia de implementación que tiene el codename "*Glassfish*" y esta disponible en *http://glassfish.dev.java.net/*. Estos servidores de aplicación, en conjunción con las capacidades definidas en la especificación EJB, soportan todas las características aquí expuestas y muchas otras más.

La especificación EJB fue creada por miembros experimentados de la comunidad de desarrolladores. Los miembros de este prestigiado grupo son de distintas organizaciones como: JBoss, Oracle y Google. Gracias a ellos, tenemos un estándar, nos basamos en una especificación para desarrollar y desplegar aplicaciones empresariales.

### 5.1.3 LOS TRES TIPOS DE EJBS

Actualmente existe tres tipos de EJBs: beans de sesión, beans de entidad y beans dirigidos por mensajes. A continuación se describe cada uno de estos beans.

#### 5.1.3.1 Beans de Sesión

Una forma de pensar en la capa lógica (*middle tier*) de la arquitectura mostrada en la figura 10, es en un conjunto de objetos que, juntos implementan la lógica de negocio de la aplicación. Los beans de sesión están construidos bajo el diseño de EJB para ese propósito, implementación de la capa lógica. Como se muestra en la figura 10, puede haber muchos beans de sesión en una aplicación. Cada uno manejando un segmento de la aplicación en la lógica de negocio.

Un bean de sesión tiende a ser responsable de un grupo de funcionalidades relacionadas.

Existen dos tipos de bean de sesión, los cuales son definidos por su uso en una interacción con un cliente:

- **Sin estado (*Stateless*)**: Estos beans no declaran ninguna variable de instancia y sus métodos solo actúan sobre parámetros locales. No hay manera de mantener un estado por medio de los métodos.

- **Con estado (*Stateful*):** Estos beans pueden mantener el estado del cliente a través de invocación de métodos. Esto es posible mediante el uso de variables de instancia, declaradas en la definición de la clase. El cliente pondrá valores a esas variables y usar esos valores en invocación de sus métodos.

El proceso de compartir un bean de sesión con estado es más complicado que un bean de sesión sin estado, esto refiriéndonos al trabajo que tiene que realizar el servidor de aplicación. Almacenar el estado de un EJB es proceso muy intensivo que consume mucho recurso, por eso una aplicación que use un beans de sesión con estado no será fácilmente escalable, esto en contrastes a los beans de sesión sin estado, que ofrecen una excelente escalabilidad, esto es debido a que los contenedores EJB no necesitan seguir su estado a través de las llamadas de métodos.

Todos los EJBs, incluyendo los beans de sesión, operan bajo el contexto de un servidor EJB, como se ilustra en la figura 10. Un servidor EJB contiene una "construcción" conocida como contenedores EJB, que son responsables de ofrecer un ambiente operativo para el manejo de los beans; además, ofrece servicios para los EJB que se encuentran activos.

Un típico escenario es: una interfaz de usuario (UI) que llama a los métodos de un bean de sesión para que le suministren cierta funcionalidad que ellos ofrecen. Un bean de sesión podría llamar a otros beans de sesión. La figura 10 ilustra una típica iteración entre la interfaz de usuario, beans de sesión, beans de entidad y una base de datos.

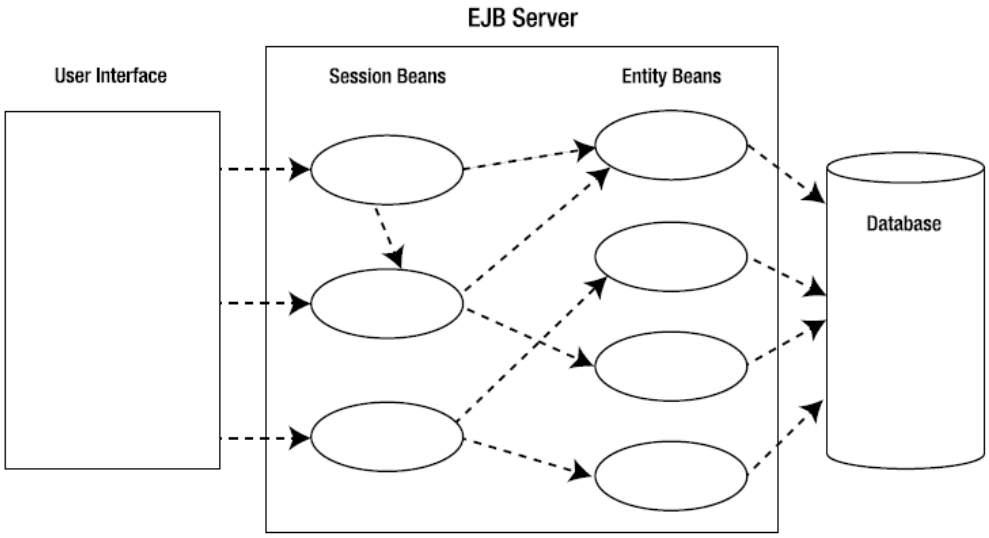


**Figura 10.** Beans de sesión y de entidad en una aplicación.

#### 5.1.3.1.1. Anatomía de un bean de sesión.

Cada bean de sesión requiere de la presencia de un bean interface y un bean clase. Un bean interface es el mecanismo por el cual el código del cliente interactuará con las entrañas del bean, siendo el bean de clase la implementación de estas entrañas, esto se ilustra en la figura 11.

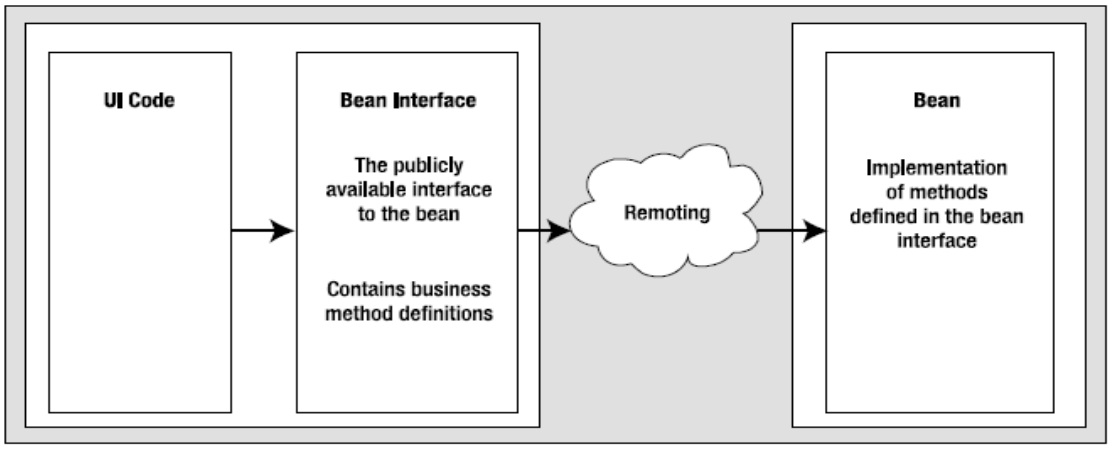


**Figura 11**. Elementos en un bean de sesión.

La implementación de la lógica de negocio en un bean de sesión esta localizada en el bean de clase. Este bean de clase debe implementar la interface *javax.ejb.SessionBean* o definir un prefijo a esta clase con el **descriptor metadata** @*Stateless.*

El *descriptor metadata* es una nueva adición al lenguaje Java con la introducción de J2SE 5.0. Esto permite a los desarrolladores elaborar definiciones en las clases, agregando descripción adicional e información de comportamiento a la generación del archivo clase. La especificación EJB 3.0 ha utilizado esta facilidad para describir la mayor parte de estos trabajos. Para saber más sobre esta nueva adición al lenguaje Java, puede leer la especificación oficial en http://jcp.org/en/jsr/detail?id=175.

El bean de interface (o interface de negocio, como algunos suelen llamar) expone los métodos por medio de los cuales se llama al bean de clase, puede ser implementada por el desarrollador, o esta puede ser generada automáticamente. El compilador generara un bean de interface, con todos los métodos públicos disponibles en el bean de clase.

#### 5.1.3.1.2. ¿Como escoger entre un bean de sesión con estado y sin estado?

Como se menciono anteriormente los beans de sesión son una buena opción para la implementación de la lógica de negocio, procesos y flujos de trabajo. Cuando decides utilizar un bean de sesión para implementar esa lógica, necesitas además hacer otra selección, y es si el bean de sesión es con estado o sin estado.

Consideremos una aplicación ficticia de una tienda, donde el cliente usa los métodos *buy()* y *getTotalPrice()* de un bean de sesión llamado *StockTrader*. Si un usuario tuviese distintas cotizaciones de compra y quisiera ver el precio subtotal de cada una de ellas, entonces ese “estado” necesitaría ser almacenado en alguna parte. Un lugar para almacenar este tipo de información trascendente se encuentra en las variables de instancia de los beans de sesión. Esto requiere que el bean de sesión sea definido como *stateful* (con estado), esto se hace utilizando el descriptor metadata @*Stateful*.

Existen ciertas ventajas y desventajas en los bean de sesión con estado. Las siguientes son algunas ventajas:

- Información trascendente, así como se describió en el escenario de la tienda, esta puede ser almacenada fácilmente en las variables de instancia de los beans de sesión, esto es opuesto al hacer uso de bean de entidad (o JDBC) para almacenarla en una base de datos.

- Ya que la información trascendental es almacenada en un bean de sesión, el cliente no necesita guardarla, así evitando el desperdicio de ancho de banda al enviar el bean de sesión la misma información por cada llamada a un método del bean de sesión.

La principal desventaja de los beans de sesión con estado es que no son escalables en un servidor EJB, esto eso, porque ellos requieren más recursos del sistema que los bean de sesión sin estado. El bean de sesión con estado no solo requiere más memoria para ser almacenados, sino que además se les tiene que manejar de una forma robusta, ellos son intercambiados (swapped) a y desde la memoria (activados y desactivados) en un contenedor EJB demandando así, bastantes recursos para su manejo del servidor. El estado queda almacenado de una forma aun más permanente cuando un bean de sesión es desactivado, y este estado se vuelve a cargar cuando el bean es activado.

**Nota.** La interface genérica de un bean de los cuales todos los EJBs se derivan, definen muchos métodos en el ciclo de vida de una sesión, incluyendo *ejbActivate()* y *ejbRemove()*. La clase de un bean de sesión con estado puede implementar estos métodos para causar un procesamiento especial para cuando estos son activados o removidos.

#### 5.1.3.2 Beans de entidad

Antes de que el modelo de programación orientado a objetos se volviera popular, habitualmente los programas eran escritos en lenguajes procedimentales y a menudo se empleaban bases de datos para guardar datos .A causa del poder y madures de la tecnología de base de datos relacionales, ahora se toma provecho para desarrollar aplicaciones orientadas a objetos que usen bases de datos relacionales. El problema con este acercamiento es que existe una diferencia inherente entre la orientación a objetos y la tecnología de base de datos relacionales, produciendo esto cierta incompatibilidad a coexistir en una aplicación. El uso de beans de entidad es una manera de tener lo mejor de ambos mundos, por las siguientes razones:

- Los bean de entidad son objetos, y ellos pueden ser diseñados usando principios de orientación a objetos y usados en las aplicaciones como objetos.

- Los datos en esos objetos de entidad son persistentes en algunos almacenes de datos, usualmente en base de datos relacionales. Todos los beneficios de la tecnología de base de datos relacionales, incluyendo la madures de productos, velocidad, fiabilidad, capacidad de recuperación y borrado de consultas, pueden ser aprovechados.

Un típico escenario EJB es, cuando un bean de sesión necesita acceder a los datos de la aplicación, este llama a los métodos de un bean de entidad. Los beans de entidad representan los datos persistentes en una aplicación EJB. Por ejemplo, una aplicación de una institución educativa puede tener un bean de entidad llamado *Estudiante* que tiene una instancia por cada estudiante que es matriculado en la institución. Los beans de entidad, con frecuencia son respaldados por una base de datos relacional, leen y escriben en las tablas de una base datos. Debido a esto, ellos ofrecen una abstracción orientada a objetos para algunos almacenes de información.

Como se muestra en la figura 11, es bueno en la práctica llamar solo a los bean de sesión por el cliente, y dejar a los beans de sesión sean los que llamen a los bean de entidad.

Aquí se señalan algunas razones del porque se debe hacer eso:

- Llamar directamente los métodos del bean de entidad evaden la lógica de negocio contenida en los beans de sesión y esto tiende a empujar la lógica de negocio en el código de la interfaz grafica.

- Los bean de sesión pueden proteger la interfaz grafica de los cambios que se hagan en un bean de entidad.

- Restringiendo el acceso del cliente a un bean de sesión, hace más seguro al servidor y recursos de red.

#### 5.1.3.2.1. ¿Como los beans de entidad se relacionan junto con los bean de sesión?

Los beans de entidad pueden facilitar una abstracción orientada a objetos de una base de datos relacional. Ellos son envueltos por un almacén de datos persistentes.

Los beans de entidad trabajan muy bien con los beans de sesión suministrando funcionalidad de la aplicación del lado del servidor. La figura 12 bosqueja como los bean de sesión y entidad trabajan juntos.

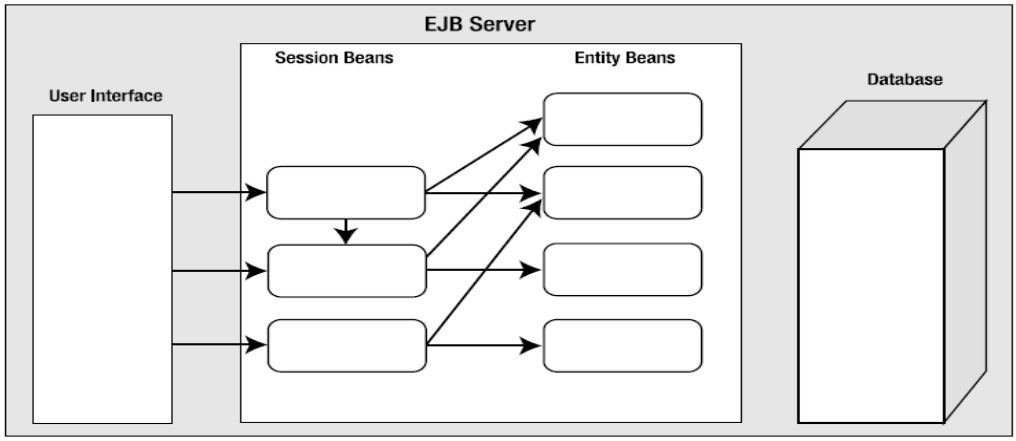


**Figura 12.** Iteración entre el cliente, lógica de beans y la base de datos.

Los beans de sesión generalmente implementan la lógica de negocio, procesos y flujo de trabajo de una aplicación, y los beans de entidad son objetos de datos persistentes. La diferencia entre los dos cae en la implementación. Los beans de sesión que presentan cierta funcionalidad, están compuesto de una vieja interfaz Java plana y de un objeto. Los beans de entidad, actúan solo como objetos de datos, están constituido meramente de un objeto plano Java. Estos dos tipos de beans son inmediatamente complementarios.

#### 5.1.3.2.2. Anatomía de un bean de entidad

Los beans de entidad son estructuralmente diferentes a los bean de sesión, ya que ellos no son construidos con una interface bean. La construcción de un bean de entidad al más básico nivel es simplemente la creación de un objeto Java plano *serializable* que es anotado por el descriptor *@Entity* (*javax.persistence.Entity*). Todos los campos de instancia descritos en la anotación de la clase son persistentes en el almacén de entidad, excepto los campos que son anotados con el descriptor *@Transient* (*javax.persistence.Transient*).

Los beans de sesión pueden crear y remover beans de entidad del almacén usando los métodos apropiados del *EntityManager*. Los bean de entidad son creados en el almacén por medio del método *persist()*, y son removidos con el método *remove()*. Estrictamente hablando, otros clientes como lo son las interfaces de usuario y sistemas externos pueden acceder a los beans de entidad directamente, pero como se dijo anteriormente, esto no es lo mejor que se debe hacer. La interface *javax.persistence.EntityManager* facilita la manipulación del almacén de datos, pudiendo realizar tareas de búsqueda, eliminación y uso de los beans de entidad.

#### 5.1.3.2.3. La clase del bean de entidad

La clase del bean de entidad contiene varios tipos de métodos:

**Métodos de obtención y colocación de datos persistentes:** Por ejemplo, una clase *Stock* puede tener un campo llamado *tickerSymbol*, con un método de obtención llamado *getTickerSymbol()* y un método de colocación llamado *setTickerSymbol()*. Los nombres de los métodos de obtención y colocación solo son una denotación del nombre del campo, similarmente a las propiedades de los JavaBeans.

**Métodos que contienen lógica de negocio:** Esos métodos típicamente acceden y manipulan los campos de un bean de entidad. Por ejemplo, si se tiene un bean de entidad llamado *StockTransactionBean* con dos campos llamados *price* y *quantity,* un metodo llamado *getTransactionAmount()* podría ser la multiplicación de los dos campos y regresar el monto de la transacción.

**Métodos del ciclo de vida que son llamados por el contenedor EJB:** Por ejemplo, en un bean de sesión, un método anotado por el descriptor *@PostConstruct* es llamado después de que un bean de entidad halla terminado su inicialización, pero antes de que cualquier método de negocio sea llamado. Estos métodos pueden ser controlados por el paso de valores de inicialización.

Los métodos en el ciclo de vida son llamados por cada evento significativo de un bean que uno se pueda imaginar. Los bean de entidad en particular reciben un trato especial en esos métodos que son ejecutados cuando una iteración ocurre con el *EntityManager*. Estos métodos son invocados antes, antes después, y después de acciones de persistencia, actualización, eliminación y cargados; La anotación de esos métodos son: *@PrePersist @PostPersist), @PreRemove, @PostRemove, @PreUpdate, @PostUpdate y @PostLoad. Se* puedes utilizar estos descriptores para reaccionar de una forma más perspicaz a las interacciones que sucedan en el almacén persistente.

La clase del bean de entidad es anotada con el descriptor metadata *@Entity* y debe implementar la interfase *java.io.Serializable* si se desea usar con interfaces remotas.

#### 5.1.3.2.4. Persistencia Controlada por el contenedor y el EntityManager

Los beans de entidad son envueltos por algún tipo de almacén de datos persistente, muchas veces es una base de datos relacional. Esta persistencia puede ser manejada automáticamente por el contenedor EJB, el cual es llamado *container-managed persistence* (CMP). Recordemos que un contenedor EJB es una pieza de un servidor EJB que maneja y ofrece servicios para los EJBs. Con CMP, un bean de entidad es mapeado a una tabla de una base de datos que esta dedicada a ese propósito, el de almacenar instancias de un bean de entidad. Por ejemplo, un bean de entidad llamado *Stock* puede tener una tabla llamada *stock*, *stock_table* o quizás *StockBeanTable* dedicada a eso. Cada registro en la tabla podría representar una instancia del bean de entidad.

La interface *EntityManger* (*javax.persistence.EntityManager*) permite crear, quitar y encontrar beans de entidad en el almacén de datos. Esta interface representa una abstracción del almacén de datos persistentes que facilita la existencia de un bean de entidad. La interface *EntityManager* contiene los siguientes métodos para búsqueda y actualización del almacén de datos:

- Al llamar al método *persist()*, se crea una nueva instancia de un bean de entidad en el almacén de datos. Es importante notar que el método *persist()* no crea una nueva instancia de un bean de entidad en realidad; este crea una entrada asociada con un bean de entidad al almacén, basado en la instancia del bean de entidad que le fue pasado.
- Al llamar el método *remove()*, se borra el bean de entidad especificado del almacén de datos.
- Al llamar al método *find()*, se busca la instancia del bean de entidad apropiada mediante su llave primaria. Las llaves primarias se discuten más adelante.
- Al llamar al método *createQuery()*, permite acceder al almacén de una manera mas generalizada, similar al uso del lenguaje SQL.

El lenguaje de consulta EJB (EJB QL) permite ejecutar consultas en los bean de entidad. El EJB QL, es, lo que su nombre sugiere, es un lenguaje de consulta muy similar al SQL.

Así como el SQL, EJB QL tiene avanzadas características para realizar consultas a una base datos, como funciones de suma y sub consultas. Estas consultas son encapsuladas en los métodos del bean de entidad, permitiendo a los desarrolladores utilizar esos métodos de consultas SQL.

- Al llamar a los métodos de obtención y recuperación, correspondientes a los campos del bean de entidad que se relacionan a los campos de la tabla en una base de datos para poder leer y escribir en ella, respectivamente. Esos campos a menudo son llamados *CMP Fields*, ya que ellos son manejados por el contenedor.

Cuando se usa CMP, el esquema de la base de datos refleja directamente el diseño del bean de entidad. La figura 13 muestra esta relación.

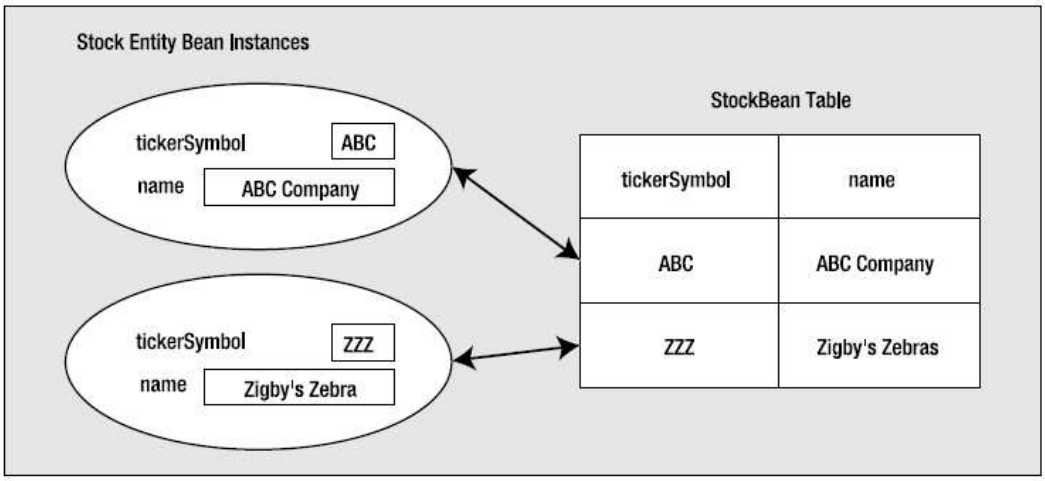


**Figura 13:** instancias del bean de entidad *Stock* con respecto a su tabla

Como en una base de datos relacional, los bean de entidad tienen un esquema abstracto que define los campos CMP de cada bean de entidad, y todas las relaciones que son manejadas en el contenedor entre todos los beans de entidad. Por cada relación, existe un método correspondiente en la relación del bean de entidad que se refiere al otro.

Los campos CMP son formalmente definidos como campos privados en el bean de entidad. El desarrollador se encuentra restringido a usar tipos primitivos de Java; la clase Java *String*, tipos Java *serializable* y otros bean de entidad para miembros de clase de

valores persistentes. Multivalores persistentes (colecciones de valores, como lo es un arreglo) están restringidos a esas clases que derivan de *java.util.Collection* o *java.util.Set*.

#### 5.1.3.2.4.1. Llaves Primarias

Un requisito de un bean de entidad es que uno o más de sus campos CMP debe ser único. Este campo (o combinación de campos) es conocido como llave primaria, que es muy útil a la hora de buscar un bean de entidad específico. Una llave primaria puede ser cualquier objeto *serializable*, como lo es un *String* o un *Integer*. Las llaves primarias también tienen otra buena característica en los bean de entidad y es la habilidad que tiene el contenedor EJB, en manejar las relaciones entre los bean de entidad.

Se puede dejar que el contenedor haga el trabajo de mantener la llave primaria usando el descriptor metadata *@Id* (*javax.persisntece.Id*) para indicarle como es la unicidad de la llave:

```
@Id(generate=javax.persistence.GeneratorType.AUTO)
private int id;
```

Este código alerta al contenedor que el campo en cuestión requiere de algún valor único, este es generado de forma automática cuando una instancia de la entidad es creada en el almacén de datos.

#### 5.1.3.2.5. Bean-Managed Persistente

La alternativa al CMP es *Bean-Managed Persistente* (BMP), en la que se escribe toda la lógica de persistencia. Esta lógica de persistencia va dentro del bean de entidad (toda la conexión con la base de datos, consultas, y mecanismos de actualización serán contenidos con el bean de entidad).

Los beans BMP no han sido condicionados para la especificación EJB 3.0. Sin embargo, debes considerar usar BMP en las siguientes situaciones:

- Una base de datos ya existente, heredado de un sistema, y se esta construyendo una aplicación EJB basada en esa base de datos. Si el diseño de los bean de entidad no encajan con el esquema existente de base de datos, BMP será requerido.
- El servidor EJB no soporta CMP fusionada con la base de datos que se esta usando. Esto puede suceder en el caso donde la base de datos de cuestión carece de soporte JDBC, o la manera de interactuar con la base de datos es demasiado complejo para garantizar una solución.

#### 5.1.3.2.6. El Lenguaje de Consulta EJB

Los *beans* de entidad ofrecen una abstracción orientada a objetos de una base de datos, completada con la posibilidad de crear métodos de negocio que operan sobre los datos contenidos en un *bean* de entidad. EJB QL permite embeber consultas parecidas a la sintaxis SQL dentro de los bean de entidad, que pueden ser accedidos por medio de sus métodos. El resultado de las consultas del EJB QL a menudo son referencias a bean que pueden ser operados directamente, así se alcanzan las ventajas de la combinación del modelo de orientado a objetos y SQL.

Un bean de entidad es una representación de datos almacenados en una tabla de base de datos. Los datos en un sistema relacional de base de datos, son almacenados en tablas, y los datos de esas tablas pueden tener relaciones con otras tablas haciendo uso de llaves primarias y llaves foráneas. Ahora, como un bean de entidad representa datos almacenados en tablas de base datos, este puede tener una cierta similitud en las relaciones que hay entre ellos.

Las relaciones entre los EJBs de entidad son expresadas con anotaciones metadata en el archivo fuente. Generalmente, el contenedor EJB usara esa información de los bean de entidad para obtener todas las relaciones que existen entre ellos, eso con el fin de crear

consultas en un lenguaje parecido a SQL. Las relaciones son inferidas mediante el uso alguno de los descriptores de relación (como *@OneToMany*).

Una consulta EJB QL referencia un bean de entidad por el nombre del esquema abstracto. El mapeo del nombre del esquema abstracto de un bean de entidad es tomado del nombre adquirido en el uso del descriptor *@Entity*. Este nombre puede ser generado automáticamente (el cual es el nombre local de la clase del bean de entidad), o este puede ser asignado manualmente en el descriptor:

```
@Entity(name="ElPresidente")
public class President {
       ...
}
```

Este segmento de código asegura que el nombre del esquema abstracto para el bean de entidad *President*, es asignado *ElPresidente*. Las consultas que se realicen al almacén de entidades entonces será:

```
SELECT variable FROM ElPresidente variable [query errata]
```

#### 5.1.3.2.6.1. El objeto *javax.ejb.Query*

Las consultas EJB QL son realizadas en la capa de datos persistentes, atreves del uso de la interface *EntityManager*, la cual encapsula los objetos *Query*. Los objetos *Query* representan una vista de alto nivel de la ejecución de una consulta y pueden ser usados para afinar la interacción de una aplicación con los beans de entidad almacenados.

Asumiendo que se adquirió una referencia del *EntityManager* en alguno de los bean de sesión, un ejemplo de una interacción con un bean almacenado podría lucir así:

```
public List getAllPresidents(){
       Query q=manager.createQuery("SELECT p FROM President p");
       return q.getResultList();
}
```

Este ejemplo compila una consulta EJB QL en un objeto Query, y al ejecutarlo retorna una *List* de objetos *President* como resultado.

La interface *Query* no solo define la ejecución de una consulta, también puede incluir métodos que coloquen un numero máximo de resultados retornados, indicaciones que especifican los parámetros operacionales en el almacén de entidades.

#### 5.1.3.2.6.2. Construyendo consultas EJB

Una consulta EJB QL consiste en lo siguiente:

- Una clausula SELECT, que especifica el tipo de valores a ser seleccionados.
- Una clausula FROM, que especifica el dominio (o tabla) en la cual los objetos serán seleccionados.
- Una clausula opcional WHERE, la cual es usada para filtrar los resultados devueltos por la consulta.
- Una clausula opcional GROUP BY, la cual es usada para agrupar los resultados devueltos.
- Una clausula opcional ORDER BY, la cual es usada para ordenar los datos devueltos en la consulta.

La sintaxis de EJB QL es similar a la sintaxis SQL. Una forma común de una consulta EJB QL es:

```
SELECT variable
FROM abstractSchemaName [AS] variable
[WHERE value comparison value]
[GROUP BY ...]
[HAVING ...]
[ORDER BY ...]
```

**La clausula SELECT**

La clausula SELECT puede contener una variable de identificación, que puede ser usada para indicar el tipo de entidad solicitada. La clausula SELECT puede ser filtrada usando la clausula WHERE. Por ejemplo, para pedir los datos solo de los presidentes con apellido Bush, es usada la siguiente consulta EJB QL:

```
SELECT p
FROM MyPresidentsSchema p
WHERE p.last_name = 'Bush'
```

Note que la clausula FROM asocia la variable identificadora p con el nombre de esquema abstracto *MyPresidentsScheme*. El nombre del esquema es especificado en el descriptor de despliegue. La clausula WHERE filtra el resultado de la consulta mostrando solo los resultados que sean igual al parámetro suministrado por el cliente.

**Parámetros de Entrada**

También se puede usar parámetros de entrada en una consulta EJB QL. Cada parámetro de entrada es indicado por un literal precedido :, seguido por una cadena que es el nombre del parámetro. En otras palabras, el parámetro “lastname” es expresado como **:lastname**.

El anterior ejemplo si usáramos este parámetro, se podría tener más flexibilidad en la consulta:

```
SELECT p
FROM MyPresidentsSchema p
WHERE p.last_name = :lastname
```

Cuando un parámetro se es suministrado, se puede obtener los datos de los presidentes con cualquier apellido especificado, esto se hace usando un método como el siguiente:

```
public List findPresidentBySurname(string surname){
       Query q=manager.createQuery("SELECT p FROM~CCC
       MyPresidentsSchema p WHERE p.last_name=:surname");
       q.setParameter("surname",surname);
       return q.getResultList();
}
```

Cuando existen datos duplicados, se pueden usar múltiples parámetros para evitar eso. Por ejemplo, para obtener los datos de George Bush, se podría usar la siguiente consulta, asumiendo que se ha declarado el segundo parámetro en el descriptor de despliegue:

```
SELECT p
FROM MyPresidentsSchema p
WHERE p.last_name = :surname AND p.election_year=:electyear
```

**Wildcards**

También es posible usar la palabra clave LIKE junto con %caracteres, así como se usan en SQL.

**Funciones**

EJB QL tiene funciones numéricas y de cadena. La tabla 3 siguiente muestra esas funciones con su respectiva descripción:

**Tabla 3.** Funciones de EJB QL y sus descripciones.

| **Función** | **Descripción** |
|---|---|
| CONCAT (String, String) | Concadena dos cadenas y devuelve un string |
| SUBSTRING (String, start, length) | Retorna una cadena |
| LOCATE (String, String[, start] ) | Retorna un entero |
| TRIM (String) | Retorna una cadena libre de espacios en blanco |
| UPPER (String) | Retorna una cadena con todos los caracteres en mayúscula |
| LOWER (String) | Retorna una cadena con todos los caracteres en minúscula |
| LENGTH (String) | Retorna un entero |
| ABS (number) | Retorna un int, float o doble, del mismo tipo de argumento |
| SQRT (double) | Retorna un doble |
| MOD (int, int) | Retorna un int |

**Subconsultas**

EJB QL incluye la posibilidad de realizar una subconsulta en una consulta SELECT. La consulta puede tener solo una clausula WHERE o HAVING, y debe ser una consulta simple. Por ejemplo, se podría usar una subconsulta para obtener una lista de los presidentes cuyo total número de votos excedió el promedio de todos los votos:

```
SELECT p
FROM MyPresidentsSchema p
WHERE p.totalVotes > ( SELECT avg(po.totalVotes) FROM MyPresidentsSchema po )
```

**Funciones de Agregación**

EJB QL tiene las siguientes funciones de agregación: AVG(), SUM(), COUNT(), MAX() y MIN(). Las funciones AVG() y SUM() deben tener un argumento numérico. Las otras funciones tienen un argumente correspondiente al tipo de datos que corresponda el campo EJB. Estas funciones puede ser usadas solo cuando se agrupan el valor devuelvo mediante la clausula GROUP BY.

Los valores que contienen NULL son eliminados antes de que las funciones sean ejecutadas. La palabra clave DISTINCT puede ser usada para eliminar valores duplicados en conjunción con las funciones EJB QL.

Existen mas detalles acerca del EJB QL, los cuales pueden ser encontrados en la especificación EJB 3.0, esta puede ser descarda desde http://java.sun.com/products/ejb/docs.html.

#### 5.1.3.3. Beans dirigidos por mensajes (MDBs)

Cuando una aplicación, basada en EJBs necesita recibir mensajes asíncronos desde otros sistemas, estos pueden ser manejados por el poder y comodidad de los beans dirigidos por mensajes. Los mensajes asíncronos entre sistemas pueden ser comparados a los eventos que son lanzados desde los componentes UI a un manejador de eventos en la misma JVM.

Considerando que muchas de las grandes aplicaciones escalables deben poseer la habilidad de responder a algunos eventos externos, uniendo eso con los requerimientos que el mecanismo para activar dichas respuestas son ambas extensibles y eficientes, y encontraras que no existe ninguna necesidad de una herramienta de alguna aplicación empresarial que sea capaz de manejar esta tarea. Los bean dirigidos por mensajes es la solución EJB 3.0 para dichos escenarios en una aplicación.

##### 5.1.3.3.1. Vistazo general de los beans dirigido por mensajes

Los beans dirigidos por mensajes (a menudo abreviados MDBs) es la forma por la cual las aplicaciones EJB son capaces de responder a algunos eventos externos (más exactamente, un mensaje JMS[6]) sin necesidad de alguna aplicación externa. Estos mensajes son enviados atraves de un productor de mensajes JMS; sin embargo, también se podría usar los MDBs con otros frameworks de mensajería (o incluso se podría desarrollar alguno).

El propósito de los MDBs es el de recibir y procesar mensajes asíncronos. Esos mensajes podrían originarse de sistemas externos o desde componentes de la misma aplicación. Estos mensajes son llamados asíncronos ya que ellos pueden llegar en cualquier momento, lo que es opuesto al resultado directo que origina la invocación de un método remoto.

[6] JMS: Java Messaging Service

Similarmente a la forma de cómo una interfaz grafica maneja los eventos que son lanzados, los bean dirigidos por mensajes están a la “escucha” de mensajes asíncronos que han sido enviados a ellos, teniendo en cuenta que el emisor de los mensajes asíncronos no esta bloqueado por la espera de una respuesta.

Descrito generalmente, los MDBs ofrecen un procedimiento simplificado de responder a eventos asíncronos. Debido a la naturaleza del contenedor EJB que dirige esos mensajes al MDB, el desarrollador se libra de construir una solución garrafal que sea capas de manejar de forma segura todos los hilos o del mantenimiento de todos los objetos que están a la escucha. Cada MDB que se desarrolla solo necesita de un método *onMessage()*, que maneja los mensajes de un tipo especifico que esta esperando; la tarea de comunicar ese mensaje a través del framework es manejada por el desarrollador.

Los MDBs no solo son activados por el contenedor EJB, también pueden ser activados por una facilidad en Java EE conocida como la API de Servicio de mensajería de Java (JMS) y los sistemas externos que están disponibles a interactuar con un sistema JMS.

La figura 14 muestra el contexto en el cual los MDBs operarán, y servirá como base para la explicación de la API JMS.

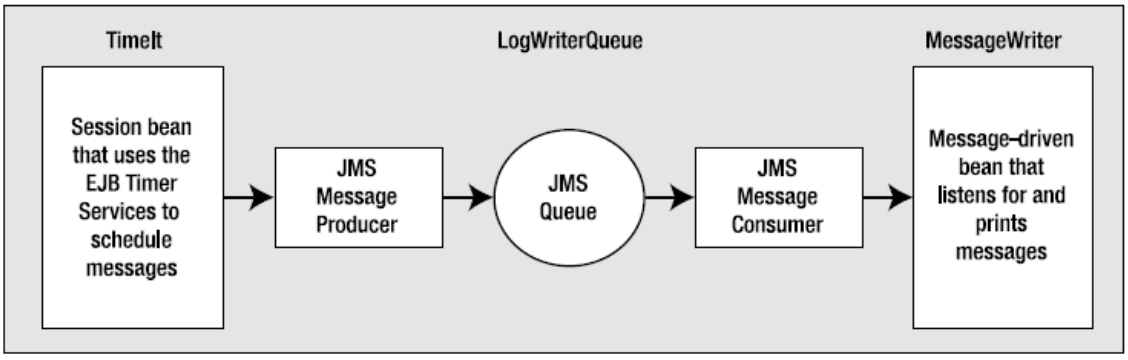


**Figura 14.** Diseño básico de la aplicación ejemplo

Así como los bean de sesión sin estado, los MDB son sin estado, objetos manejados por el contenedor que responde a solicitudes, mientras un MDBs responde a *messages (javax.jms.Message)* JMS recibidos desde el contenedor, un bean de sesión sin estado

responde a solicitudes de clientes hechas a través de una interface apropiada. La diferencia entre ellos ésta en que un MDB no implementa una interface local o remota, pero en cambio implementa una interface apropiada de mensajes JMS para describir su comportamiento. Para nuestro propósito, implementaremos la interface *MessageListener* (*javax.jms.MessageListener*), la cual define un comportamiento muy genérico y utilizable. Otros paquetes pueden describir comportamientos más especializados, así como el recipiente de mensajes XML enviados por los clientes.

Los MDBs no son solo objetos que responden mensajes, los MDBs tiene un enriquecido modelo de programación que incluye soporte para conceptos de alto nivel como transacción de mensajes (permitiendo el procesamiento de mensajes para darle cierto nivel de atomicidad) y un método de intercepción.

Tomando ventaja de estas herramientas ofrecidas, una aplicación empresarial de alto nivel de funcionalidad seria fácilmente construida a bajo costo.

#### 5.1.3.3.2. Descripción de los MDBs

Así como los otros tipos de bean presentados en la especificación EJB 3.0, los MDBs son descriptos inicialmente por una anotación, para este caso *@MessageDriven* (*javax.ejb.MessageDriven*). Esta anotación es usada por el contenedor para determinar el comportamiento a un bean (así como los bean de entidad y de sesión), más notable la configuración de activación. La configuración de activación de un bean dirigido por mensaje puede instruir al contenedor para que tome el método apropiado para el mensaje para aplicar al bean o restringir los tipos de mensajes permitidos para ser procesador por el bean.

El siguiente segmento de código ilustra como un MDB esta definido:

```
@MessageDriven(activateConfig =
        {
                @ActivationConfigProperty(propertyName="destinationType",
                                propertyValue="javax.jms.Queue"),
                @ActivationConfigProperty(propertyName="destination",
                                propertyValue="queue/LogWriter")
        })

public class MessageWriter implements MessageListener {
```

Se puede ver aquí que el MDB tiene una configuración de activación especificada por un arreglo de anotaciones *@ActivationConfigProperty (javax.ejb.ActivationConfigProperty)*. Este segmento en particular alerta al contenedor EJB de los mensajes recibidos por este bean.

#### 5.1.3.3.3. El contexto de los MDBs

Los MDBs ofrecen algo más que un simple mecanismo de respuesta a mensajes. La especificación EJB fue hecha para que los MDBs puedan usar transacciones (ofrecida por el contenedor EJB o por el mismo bean), por medio de métodos especiales de intercepción que enriquecen la funcionalidad del proceso de mensaje, además de invocación programada a los beans. Esto último se hace posible mediante el uso del servicio *Timer*. El acceso a la funcionalidad de transacciones y el servicio de timer es ofrecida por la interfase de los MDBs *MessageDrivenContext* (*javax.ejb.MessageDrivenContext*). Esta interface puede ser adquirida a través del uso de "inyección".

Inyección es la forma por el cual el contenedor inserta automáticamente una referencia al objeto apropiado para un miembro **anotado**. Por ejemplo:

```
@PersistenceContext
private EntityManager _manager;
```

La anotación *@PersistenceContext* efectúa una inyección especial, esta puede ser comparada a la inyección que se le realiza a la interface *EntityManager*. El tipo de inyección discutida aquí es una forma inyección más genérica, llamada *resource injection*. Esta inyección permite al contenedor insertar el objeto apropiado basado en una solicitud.

Consideremos el siguiente segmento de código:

```
@Resource
private MessageDrivenContext _context;
```

El contenedor EJB pondrá el miembro *_context* a referenciar el valido *MessageDrivenContext* para el actual MDB. El miembro estará disponible para su uso automáticamente después que el contenedor haya inicializado el bean y antes de que cualquier mensaje sea procesado.

#### 5.1.3.3.4. Transacciones MDB

Como se menciono, los MDBs en EJB 3.0 están habilitados para que usen transacciones permitiéndoles un alto control sobre el proceso de mensajes. Por medio de las transacciones, se puede hacer un "*roll back*" al procesamiento de un mensaje en particular si en caso tal este falla. El rollback presenta cascada del método de negeocio que inicializacion requerida la transacción a los metodos es invocada que soporte procesamiento de transacciones. Esta funcionalidad es semánticamente idéntica a la idea de las transacciones en base de datos, donde un completo conjunto de operaciones de base de datos pueden ser rolled back o comprometido basado en el éxito de intrucciones individuales en el conjunto.

##### 5.1.3.3.4.1. Transacciones manejadas por el contenedor

La especificación EJB hizo provisiones en los contenedores para que ofrezcan su propia funcionalidad en las transacciones. Esta funcionalidad puede ser agregada directamente a un MDB mediante la anotación *@TransactionManagement* (*javax.ejb.TransactionManagement*). Una vez la clase sea anotada, los métodos de negocio de la clase deben ser anotados para especificarle al contenedor como interpretara la funcionalidad de las transacciones. Se puede especificar este comportamiento usando la anotación *@TransactionAttribute* (*javax.ejb.TransactionAttribute*) sobre los métodos de negocio junto con alguno de los dos siguientes atributos: *REQUIRED* o *NOT_SUPPORTED*.

Actualmente *@TransactionAttribute* puede mantener diferentes valores para conllevar el comportamiento de la transacción: *MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED y NEVER*. Sin embargo, los MDBs pueden tener métodos de negocios anotados con solo *REQUIRED* o *NOT_SUPPORTED*.

El tipo de transacción *NOT_SUPPORTED* alerta al contenedor que cualquier otra transacción existente deberá ser suspendida mientras el MDB ejecuta el método en cuestión. Una vez el método haya finalizado su ejecución, la transacciones suspendidas serán reanudadas y todo vuelve a continuar como se encontraba.

Las transacciones *REQUIRED* por otro lado, escalan la funcionalidad de las transacciones a los métodos invocados con la función original. Por consiguiente, si una sencilla instrucción SQL de INSERT falla en la aplicación, todos los metodos invocados en este momento que usen la transacción son alertado para que se devuelvan, tomando su estado anterior antes del error.

Miremos el siguiente segmento de código que indica lo anteriormente expuesto:

```
@TransactionManagement
@MessageDriven (...) // Simplified for this example
public class MyMDB {
        @TransactionAttribute(REQUIRED)
        public void onMessage(javax.jms.Message){
                ...
        }
}
```

#### 5.1.3.3.4.2. Transacciones definidas por usuarios

No es requerido que las transacciones sean controladas por el contenedor. Simplemente agregamos el parámetro BEAN a la anotación de la clase @*TransactionManagement* y el contenedor asumirá que el MDB esta manejando todos los detalles para el manejo de la transacción. Aunque la tarea de manejar sus propias transacciones no es difícil, esta requiere un cierto grado de vigilancia en términos de adherirse a los estándar de código.

El primer paso para manejar las transacciones manualmente de una particular clase es obtenerla via *resource injection*. La siguiente línea de código muestra como hacer esto:

```
@Resource javax.transaction.UserTransaction customTransaction;
```

Una vez se halla agregado esto a la definición de clase MDB, todo lo que se necesita hacer es agregar la llamada correcta al objeto *customTransaction* en la clase. El siguiente código presenta un rápido ejemplo:

```
@TransactionManagement(BEAN)
@MessageDriven (...) // Simplified for this example
public class MyMDB {
        @Resource javax.transaction.UserTransaction customTransaction;
        public void onMessage(javax.jms.Message){
                customTransaction.update(); // Start the transaction
                ...
                customTransaction.commit();
                // Assume transaction went well, commit changes
        }
}
```

#### 5.1.3.3.5. Invocación de un interceptor

El modelo de la programación orientada a aspectos (AOP) usada en EJB 3.0 es una forma muy simplificada, donde los métodos de negocio de un bean de sesión y un MDB pueden ser precedidos o seguidos automáticamente por algún "interceptor". Esos interceptores son llamados solo antes o después de la ejecución de un método de negocio de un bean.

Se le puede indicar a un método en particular que interceptara a otros métodos de negocio (como *onMessage(),* en el caso de los MDBs) en el propio bean mediante la anotación del descriptor *@AroundInvoke (javax.ejb.AroundInvoke)*, por ejemplo:

```
@AroundInvoke
public Object messageFoo(InvocationContext ctx) throws Exception{
        System.out.println("Method intercepted!");
        return ctx.proceed();
}
```

Este método en particular será invocado antes de la ejecución de un método de negocio. Antes de hacer la llamada a *ctx.procced()*, el contenedor chequeara por cualquieras métodos interceptores. Cuando se haya agotado la lista de métodos interceptores, se ejecutara método de negocio.

También se podría indicar un conjunto objetos externos que actuaran como interceptores a través del uso de la anotación *@Interceptors (javax.ejb.Interceptors):*

```
@MessageDriven
@Interceptors({"util.logger","util.profiler","util.decompressor"})
public class GzipNetMessageActor{
        // ...
}
```

El contenedor EJB envolverá a todas las llamadas de los métodos de negocio para la instancia de GzipNetMessageActor con invocaciones de los métodos anotados por *@AroundInvoke* en *util.logger*, *util.profiler*, y *util.decompressor* (en ese orden). Esta es una herramienta de manejo para agregar rápidos valores a tu aplicación sin modificar el código existente.

#### 5.1.3.3.6. La API Java Message Service

La API JMS se encuentra localizada en el paquete *javax.jms*, que ofrece una interface a las aplicaciones que requieran de los servicios de un sistema de mensajería. Un sistema de mensajería admite mensajes que contienen texto, objetos y otro tipo de mensajes para ser enviados y recibidos asíncronamente. Esto es en contrastes con el modelo RPC que hemos estado usando en los EJBs, donde las interacciones entre componentes ocurren síncronamente.

Una implementación de un sistema de mensajería que cumple con la API JMS es llamado *JMS provider*.

En la figura 14, el bean de sesión *TimeIt* de la izquierda envía un mensaje asíncrono al MDB *MessageWriter* que esta a la derecha. Los beans *TimeIt* y *MessageWriter* son conocidos como clientes JMS porque ellos son clientes del sistema del mensajería que esta por debajo.

Los sistemas de mensajerías activan la comunicación asíncrona usando un **destino** para el mensaje, que se mantiene hasta que ellos puedan ser entregados al recipiente. El circulo en el medio, *LogWriterQueue*, es el destino que mantiene los mensajes del bean de sesión *TimeIt* que van con destino para el MDB *MessageWriter*.

Existen dos tipos de destinos en JMS:

- Un *queue* es usado para mantener los mensajes en cola que son enviados desde un cliente JMS para ser enviados a otro. Este modelo de mensajería es conocido como punto a punto.
- Un *topic* es usado para mantener los mensajes que son enviados desde muchos clientes JMS para ser entregados a múltiples clientes JMS. Este modelo de mensajería es conocido como: publicar/suscribir.

En la figura 14 también existen unas cajas para un *productor de mensajes JMS* y un *consumidor de mensajes JMS*. Ellos representan clases en un JMS *provider* que trabaja en representación de los clientes JMS para enviar y recibir mensajes. No se tiene que crear un consumidor de mensajes JMS cuando se trabaja con MDBs, porque el contenedor EJB lo hace en representación de los MDBs.

#### 5.1.3.3.7. Servicio Timer EJB

La especificación EJB 2.1 introdujo un servicio llamado el EJB *Timer Service*. Su propósito es registrar a los Enterprise beans para que reciban *callbacks* después de un especifico lapso de tiempo, o en intervalos específicos. Esto es bastante útil, por ejemplo, si se quiere que un bean de sesión inicialice un proceso a las 2:00 am todo los días para obtener datos desde un sistema externo.

El EJB Timer Service no está intencionado a actuar como un horario de procesos en tiempo real. En otras palabras, no soporta precisión en nanosegundo ni alta confiabilidad; esto principalmente intenta es servir como un modelo para aplicaciones de tiempo.

Los eventos pueden ser disparados por:

- Un momento en particular. Quizás se pueda notificar a los EJB a la media noche el 28 de julio del 2061 para que realice algo especial.
- Después de un tiempo enlazado.
- Sobre un periodo, después de un intervalo de tiempo dado.

Estos servicios de tiempo son ofrecidos por el contenedor EJB y son posibles obtenerlos a través de la misma forma como los otros servicios a nivel de contenedor. Para obtener un objeto *TimerService*, se puede:

- Inyectar el *TimerService* dentro un miembro de clase usando resource injection.

```
@Resource javax.ejb.TimerService myTimer
```

- Obteniendo una instancia del objeto *TimerService* a través del método *getTimerService()* del objeto *MessageDrivenContext* o del objeto *SessionContext*.

```
@Resource javax.ejb.MessageDrivenContext initialContext;
...
public void onMessage(javax.jms.Message){
        javax.ejb.TimerService ts=initialContext.getTimerService();
}
```

- Adquirir una instancia del objeto *TimerService* a través de JNDI.

Como se debe esperar, TimerService ofrece a los EJB objetos de tiempo para que realicen alguna acción sobre un intervalo específico. Para especificar la acción a realizar, un EJB debe anotar uno de sus métodos con la anotación @*Timeout*.

## 6. SERVICIOS WEBs

Un servicio web consiste de funcionalidad disponible para aplicaciones usando protocolos asociados con la web. Ejemplos de protocolos normalmente asociados con la web son HTTP, XML y SOAP. Usando esos protocolos, una aplicación puede hacer uso de la funcionalidad ofrecida por un servicio web.

Por ejemplo, un vendedor de libros llamado Larios puede tener un servicio web corriendo sobre su servidor web que ofrece la posibilidad de hacer pedidos de libros. Podemos llamar a este servicio web LariosBookService. Una aplicación pdria usar este servicio cuando se necesite saber el precio de un libro o para pedir un libro. Este servicio web deberia tener operaciones, cada una de las cuales ejecutara alguna funcionalidad, por ejemplo:

- Una operación getPrice podria tomar el numero ISBN del libro como entrada, para devolver el precio del libro.
- Una operación orderBook podria tomar el numero ISBN y un numero de tarjeta de credito, para luego procesar la orden del libro.

De la misma manera como el servicio web LariosBookService de ofrece, las organizaciones ofrecen servicios a las aplicaciones usando protocolos estándares. La accesibilidad de los servicios web sobre estos protocolos hacen atractiva el desarrollo de aplicaciones distribuidas.

El hacer la lógica de negocio abierta y disponible para cualquier cliente, soportando las diferentes capas de servicios web, le proporciona a la aplicación mucho más valor y funcionalidad.

Ejemplos de servicios web los podemos encontrar en www.xmethods.com. Este sitio es un de los varios sitios que permite a los desarrolladores *postear* información sobre los servicios web que se tienen desarrollado, incluyendo una descripción del servicio e instrucciones de cómo usarlo. Unos de los ejemplos que podemos encontrar allí son:

**Tabla 4.** Servicios que se encuentran en internet

| Nombre del Servicio Web | Descripción |
|---|---|
| Google Search API | Ofrece el uso del servicio de búsqueda de Google para las aplicaciones |
| BibleWebservice | Retoma texto biblico |
| Currency Exchange Rate | Devuelve el intercambio entre dos monedas |
| Image Converter | Convierte un tipo de imagen a otra |
| Weather – Temperature | Rertorna la termperatura actual de una región de Estados unidos. |

### 6.1. ESTANDARES Y MODELOS

La capacidad que tiene una aplicación distribuida para comunicarse con cada uno de los componentes y el poder hacer llamado a los métodos de esos componentes, asi como lo hacen los servicios web, debería serle familiar ya que esos conceptos se mencionaron en los EJBs. Es un hecho que muchos estándares han evolucionado tanto que permiten a los clientes de una maquina invocar las operaciones o métodos de un servidor de otra maquina, por ejemplo:

- Llamada a procedimientos remotos (RPC): Existen dos tipos de RPC,
    - Ambientes de computo distribuidos (DCE) RPCs
    - Sun RPC
- Common Object Request Broker Architecture (CORBA): CORBA fue definido y está controlado por el Object Management Group (OMG) que define las APIs, el protocolo de comunicaciones y los mecanismos necesarios para permitir la interoperabilidad entre diferentes aplicaciones escritas en diferentes lenguajes y ejecutadas en diferentes plataformas, lo que es fundamental en computación distribuida.
- Distributed Component Object Model (DCOM): es una tecnología propietaria de Microsoft, utilizada para desarrollar componentes software distribuidos sobre varios ordenadores y que se comunican entre sí. Extiende del modelo COM de

Microsoft y proporciona el sustrato de comunicación entre la infraestructura del servidor de aplicaciones COM+ de Microsoft.

- Java Remote Method Invocation (Java RMI): Esta tecnología facilitan el uso de lo los EJBs.

Todos estos estándar son excelente, pero cada uno a algún grado, plataforma o dependiente del lenguaje de programación. Pero en un futuro, una aplicación necesitará invocar los métodos de cualquier otra aplicación, para realizar eso se requiere un estándar que:

- Se accesible con la mayoría de los lenguajes de programación más populares.
- Pueda ser usado sobre casi cualquier hardware /sistema operativo.
- Usen protocolos de comunicación ubicuos (presentes).
- Incitar la comunicación sobre puertos que probablemente no tengan firewall.

El modelo RPC es una forma de implementar servicios web. En este modelo, una aplicación de servicio web ofrece una interfase asequible a los clientes sobre la red; muy similar al modelo de los bean de sesión. Los programas clientes pueden encontrar e invocar los métodos de esta interface así como si estuviesen residiendo en la misma maquina. Los datos comunicados entre el cliente y el servicio web son expresados usando Simple Object Acess Protocol (SOAP) y XML.

Otra forma de implementar un servicio web es usando un modelo de mensajería. Donde cada aplicación puede enviar mensajes SOAP al otro, sin esperar un valor de respuesta. Las aplicaciones se comunicarían de forma asíncrona, así como los hacen los componentes MDB. El servicio web será iniciado por una llamada de petición de un cliente, pero el servidor necesitara iniciar su propio proceso de respuesta, esta puede ser enviando un señal al cliente por medio de un evento de la interfase.

## 6.2. ¿PORQUE UTILIZAR SERVICIOS WEB?

La naturaleza de plataforma cruzada de Java facilitan a las aplicaciones distribuidas correr sobre múltiples hardware y sistemas operativos. Si todos los componentes de una aplicación distribuida están escrita en Java, entonces se usara EJBs con Java RMI, esta seria una buena elección. Sin embargo, los servicios web son una gran elección para integrar aplicaciones que son escritas en varios lenguajes, porque la mayoría de plataformas tiene soporte para SOAP, el cual es un protocolo usado en servicios web para el paso de objetos de una aplicación a otra. Esto permite que una aplicación desarrollada en Java pueda usar las operaciones de servicio web de una aplicación desarrollada en

Una ventaja sustanciosa de los servicios web es que ellos normalmente usan HTTP como protocolo de comunicación. Esto es porque así, ellos pueden fácilmente ser implementado sobre el puerto TCP/IP, que normalmente tienen abierto los firewalls: 80, 8080 y 443.

## 6.3. PROTOCOLO STACK

En la tabla 5 se muestran los protocolos usados en los servicios web, desde el mas arriba hasta el de mas bajo nivel.

**Tabla 5.** Protocolos de servicios webs y sus capas asociadas

| Capa | Tecnologia |
|---|---|
| Service Discovery | UDDI |
| Service Description | WSDL |
| Messaging | SOAP |
| Encoding | XML |
| Transport | HTTP |

### 6.3.1. Capa de transporte

La capa de transporte es Hypertext Transfer Protocol (HTTP), protocolo en el cual la mayoría del tráfico web viaja. Los servicios web también pueden ser transportados sobre mensajes de e-mail, usando Simple Mail Transport Protocol (SMTP).

### 6.3.2. Capa de codificación

Todo el tráfico de los servicios web estan expresando en Extensible Markup Language (XML). Para mas información sobre XML visitar http://java.sun.com/xml/docs.html, www.xml.org y www.xml.com.

### 6.3.3. Capa de Mensajes

Todos los datos de la aplicación enviados por los servicios web están encerrados en mensajes SOAP. El protocolo SOAP esta basado completamente sobre XML y contiene estructuras de cómo un SOAP es envuelto: el encabezado SOAP y el cuerpo SOAP. En el cuerpo, se encuentra todos los datos de instancia de un objeto que están siendo transportados.

Una aplicación Java puede utilizar la API definida SOAP con anexos de Java (SAAJ) para crear, enviar y recibir mensajes SOAP. Esta API esta contenida en el paquete javax.xml.soap, el cual viene incluido en Java EE 5 SDK. Para saber sobre este protocolo visitar la página http://java.sun.com/xml/downloads/saaj.html

En Java existe una facilidad conocida como JAX-WS utilizada para desarrollar un servicio web a un alto nivel de abstracción, JAX-WS usa la API SAAJ para producir y consumir mensajes SOAP.

### 6.3.4. Capa de descripción

La funcionalidad ofrecida por un servicio web esta descrito por “El lenguaje de descripción de servicios web” (Web Service Description Language WSDL). Cuando se usa JAX-WS los servicios web son implementados en Java, con WSDL usando XML para describir las interfaces, métodos, argumentos, valores de retorno, y el URL del servicio web.

Algunos términos relacionados con el WSDL son:

- Puerto WSDL es un elemento que se puede hacer una analogía con una interface en Java.
- Operación WSDL es un elemento que se puede hacer una analogía con un método en Java.
- Partes WSDL (contenido dentro de los mensajes) son análogos a los argumentos y valores de retorno en Java.

### 6.3.5. Capa de descubrimiento

Los servicios web pueden ser registrados para que sean usados, y luego ser descubiertos. Por ejemplo, una aplicación cliente en busca de un servicio web que ofrezca la funcionalidad de un diccionario de sinónimos podría chequear en los servicios webs que están registrados. Esto es conocido como Universal Description, Discovery, and Integration (UDDI). El UDDI fue originalmente una colaboración entre Microsoft, IBM y Ariba; el consorcio UDDI ahora incluye cientos de miembros.

Los registros UDDI mantienen información sobre negocios y servicios web que ofrecen. Esos registros UDDI son hospedados por varios miembros UDDI y están disponibles en internet, pero ellos también podrían estar localizando en una intranet corporativa. Se puede encontrar una lista de registros UDDI disponibles en internet, además de más información sobre UDDI en www.uddi.org.

Existe una API conocida como Java API for XML Registries (JAXR) que ofrece una interface uniforme para registros, asi como UDDI. Mas información visitar http://java.sun.com/xml/jaxr.

### 6.3.6. Capas emergentes

Algunas capas emergentes de la pila de servicios web tratan temas sobre seguridad, identidad del cliente, coordinación de transacciones, interfaces de usuario de servicios web, y flujo de un procesos. Se puede chequear algunas de estas tecnologías emergentes con el Java Web Services Pack (WSDP), disponible en http://java.sun.com/webservices/index.jsp.

## 6.4. ¿CÓMO FUNCIONAN LOS SERVICIOS WEB?

En un escenario típico de Web Services, una aplicación de negocio envía una petición vía HTTP a un servicio situado en una URL. El servicio recibe la petición, la procesa y devuelve una respuesta también sobre HTTP.

La idea es sencilla pero requiere:

- Un protocolo de intercambio de mensajes petición/respuesta sobre HTTP.
- Una forma de que clientes y proveedores puedan interactuar a través de los mensajes, es decir, un *lenguaje de especificación de interfaces*.

Se ha optado por utilizar **SOAP** (Simple Objet Access Protocol) como protocolo de intercambio de mensajes. Es un protocolo sencillo basado en XML y estandarizado por el W3C.

El *lenguaje de especificación de interfaces* utilizado en servicios web es **WSDL** (Web Services Description Language). WSDL permite especificar en XML las operaciones y tipos de datos de un servicio web. Así, aunque el cliente y el servidor estén escritos en lenguajes distintos (por tanto, con sintaxis y tipos de datos diferentes) pueden interactuar al utilizar un lenguaje neutral para comunicarse.

Una petición de un servicio web constaría de los siguientes pasos:

1. En el cliente se elabora una petición de una operación con unos parámetros
2. La petición se transforma a formato XML utilizando WSDL
3. La petición transformada se envía vía HTTP utilizando SOAP
4. El servidor de servicios web recibe la petición
5. El servidor determina que operación debe realizarse y transforma los parámetros de formato XML a su representación correspondiente en el lenguaje utilizado para implementar el servidor
6. El servidor invoca la operación con los parámetros enviados, elabora una respuesta y se la envía al cliente de la misma forma que se ha explicado

## CONCLUSIONES

Al culminar este documento, se podrá observar que la arquitectura Java EE es una completa y basta plataforma que puede resolver cualquier problema de aplicación empresarial, dándole al desarrollador una gran gama de herramientas que le permiten construir de manera sencilla diversos proyectos a nivel empresarial. Basados en las explicaciones que se muestran en esta monografía nos damos cuenta que existe una gran interoperabilidad entre los distintos tecnologías (Web services, Enterprice Java Beans, Java Server Pages, etc.).

JSP es una tecnología con una característica muy importante que es la de combinar o incrustar código java junto con código HTML. Esta combinación es posible realizarla de dos formas como lo son el insertar código java a través de Scripts, y la de realizar un llamado de un archivo contenedor de código java. El trabajar con JSP le aporta grandes beneficios al gremio de programadores Web, debido a la facilidad de proteger el código, logrando así un alto grado de confidencialidad de su forma de programar. En pocas palabras JSP hace muy práctico separar la lógica del negocio de la representación de los datos.

La tecnología JSF ofrece una clara separación entre el comportamiento y la presentación. Esto se da gracias a sus librerías de etiquetas Core y HTML; la primera consta de un conjunto de etiquetas personalizadas para representar manejadores de eventos, validadores, y otras acciones; la segunda que corresponde al conjunto de etiquetas personalizadas para dibujar componentes UI en una página, todo esto se podría resumir en que los JSF son la parte dinámica de una pagina HTTL. Además la tecnología JSF nos permite convertir y validar datos sobre componentes individuales y reportar cualquier error antes de que se actualicen los datos en el lado del servidor.

Un servlet es un objeto que se ejecuta en un servidor o contenedor JEE, que fue especialmente diseñado para ofrecer contenido dinámico desde un servidor Web, generalmente es HTML. Esta tecnología vino en respuesta a la programación CGI.

Los EJBs proporcionan un modelo de componentes distribuido estándar para el lado del servidor. El objetivo de los Enterprise beans es dotar al desarrollador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes nos permite que éstos sean flexibles y sobre todo reutilizables.

Por otro lado, se concluye que los Servicios Web Aportan interoperabilidad entre aplicaciones de software independientemente de sus propiedades o de las plataformas sobre las que se instalen. Estos fomentan los estándares y protocolos basados en texto, que hacen más fácil acceder a su contenido y entender su funcionamiento. Al apoyarse en HTTP, los servicios Web pueden aprovecharse de los sistemas de seguridad *firewall* sin necesidad de cambiar las reglas de filtrado. Permiten que servicios y software de diferentes compañías ubicadas en diferentes lugares geográficos puedan ser combinados fácilmente para proveer servicios integrados.

Finalmente, podemos pensar que la meta de Java EE es definir un estándar que ayude a suplir los retos tecnológicos en esta nueva era. Definiendo estándares que son implementados por distintos proveedores y fabricantes, no forzando a emplear ningún producto específico, proveyendo soporte tanto para el lado del servidor como para el lado del cliente para aplicaciones corporativas multi-tier.

## RECOMENDACIONES

El presente documento solo es una muy breve descripción en lo que respecta sobre la arquitectura Java EE, y solo se describieron las tecnologías más esenciales de esta plataforma, esto con la intensión de motivar al lector a seguir el material bibliográfico, principalmente en los links expuestos, ya que estos se remiten a páginas "oficiales" de la tecnología Java.

Una recomendación muy importante para la utilización de este documento guía, es que el lector tenga un cierto grado de conocimiento acerca del lenguaje programación Java, pues como se ha descrito en el documento, toda la arquitectura Java EE cae sobre ello. El lenguaje de programación Java es el núcleo de la arquitectura Java EE.

Por ultimo, seria excelente que la Universidad Tecnológica de Bolívar fomentara la investigación sobre este tema, particularizando cada una de estas tecnologías, ya que todo en lo que concierne a Java esta tomando un gran auge a nivel mundial, y seria bueno estar a la altura de todo este conocimiento. Y Además seria muy gratificante para nosotros los estudiantes que la biblioteca adquiriera el material bibliográfico utilizado para realizar este documento; estos libros se pueden adquirir vía Internet en www.ChooseBooks.com.

## TÉRMINOS Y DEFINICIONES

**API**: Interfaz de programacion de aplicaciones (Applications Programming Interface): es un conjunto de fnciones que están disponible para realizar programa para un cierto entorno.

**Acoplamiento:** Capacidad que tiene un sistema, de comunicarse internamente.

**APPLET**: Pequeñas aplicaciones escritas en Java que se incluye en una página Web y que se puede ejecutar en cualquier navegador que disponga de un interprete Java, sin que para su uso necesite intercambiar informacion con el servidor ya que siempre se ejecuta en el cliente.

**BEAN**: Es un componente de software quetiene la particularidad de ser reutilizable y que evita la tediosa tarea de programarlos uno a uno. Se puede decir que existen con la finalidad de ahorrarnos tiempo al programar. Es el caso de la mayoria de componentes que manejan los editores visuales más comunes.

**CGI**: Software que facilita la comunicacion entre un servidor Web y los programas que funcionan fuera del servidor.

**Componente:** Los componentes de software son unidades ejecutables de producción independiente, adquisición e implementación que pueden formar parte de un sistema funcional.

**Cokkie:** Pequeño trozo de datos que entrega el programa servidor de HTTP al navegador WWW para que este lo guarde.

**Deployment:** Herramienta de despliegue, o máquina virtual de J2EE.

**Encapsular:** Proteger, los datos de tal forma que no se pueda acceder fácilmente, de forma externa.

**Escalabilidad:** Cualidad que tiene un sistema, para aumentar o mejorar su capacidad de procesamiento**.**

**Framework:** Marco de trabajo de una aplicación.

**Interfaz:** Pieza de software que permite comunicar un componente con otro o con su entorno.

**JNDI:** es una Interfaz de Programación de Aplicaciones para servicios de directorio. Esto permite a los clientes descubrir y buscar objetos y nombres a través de un nombre.

**Modulo:** Parte fundamental de un sistema.

**Plataforma:** Tecnología, que brinda soporte para el funcionamiento de un sistema.

**SCRIPT**: En la programacion de computadoras es un programa o una secuencia de instrucciones que e interprertado y llevado a cabo por otro programa en lugar de ser procesado por el procesador de la computadora.

**SCRIPTLETS**: es una pieza de codigo Java incrustado en codigo HTML.
**Sistema:** Es una colección de componentes, interrelacionados, que trabajan de manera conjunta, para alcanzar un objetivo común.

**Software:** Es el nombre, que recibe el conjunto que forman, un programa, su documentación asociada y la configuración de datos, que son necesarias para su buen funcionamiento.

**Renderizar**: Es la accion de asignar y calcular todas las propiedades de un objeto antes de mostrarlo en patlla. Proceso mediante el cual la computadora crea una imagen partiendo de la descripcion de las caracteristicas de los objetos que contiene.

**RMI:** Invocación de métodos remotos (Remote Method Invocation), consiste en que un objeto acceda a un método (una de las funcionalidades) de otro objeto remoto (que esté situado en otro punto de una red).

**URL:** (Uniform Resource Locator) Refiere la situacion de un documento en Internet. Direccion global de documentos y otras fuentes en la World Wide Web.

## BIBLIOGRAFIA


- **Kevin Mukhar and Chris Zelenak**, Beginning Java EE 5 : Apress 2005
- **Rod Johnson**, J2EE Development without EJB: Wrox 2004
- **Mike Keith and Merrick Schincariol**, Pro EJB 3: Apress 2006
- **http://java.sun.com/products/jsp/** Conceptos de Java Server Pages
- **http://java.sun.com/javaee/javaserverfaces/** Introducción a Java Server Faces
- **http://java.sun.com/products/servlet/** Conceptos de Servlets
- **http://java.sun.com/products/ejb/** Documentación Acerca de los Enterprise Java Beans
- **http://java.sun.com/javaee/** Principal Descripción de Java EE
- **http://java.sun.com/webservices/** Explicación de los Servicios Web
- **http://java.sun.com/javase/technologies/database/index.jsp** Funcionamientos de las bases de datos junto con Java
- **http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5/** Breve Introducion a Java EE 5

**ANEXO A. Ejemplo De Una Aplicación Jsp**

Miremos un ejemplo donde se muestra un pagina JSP con un comportamiento dinámico: una pagina de bienvenida que se dirija a una aplicación que maneje las preguntas más frecuentes (FAQ).

Empecemos por crear el archivo welcome.jsp. Esta es la primera página a la cual será accedida por un usuario en la aplicación web.

```
<%@ page errorPage="/WEB-INF/errorPage.jsp"
        import="java.util.Iterator,com.apress.faq.FaqCategories" %>

<html>
        <head>
                <title>Java FAQ Welcome Page</title>
        </head>

        <body>
                <h1>Java FAQ Welcome Page</h1>
                Welcome to the Java FAQ

                <%! FaqCategories faqs = new FaqCategories(); %>
                Click a link below for answers to the given topic.

                <%
                Iterator categories = faqs.getAllCategories();
                while (categories.hasNext()) {
                        String category = (String) categories.next();
                %>

                <p>
                <a href="<%= replaceUnderscore(category) %>.jsp"><%= category %>
                </a></p>

                <%}%>

                <%@ include file="/WEB-INF/footer.jspf" %>
        </body>
</html>

<%!
public String replaceUnderscore(String s) {
return s.replace(' ','_');
}
%>
```

La pagina *welcome.jsp* tiene una directiva *include* que agrega un estándar *footer* (pie de pagina). Ya que el archivo que será incluido solo es un fragmento y no un completo archivo JSP, se usa la convención de nombre de archivos, se le coloca una extensión .jspf, esto es recomendado por la especificación JSP.

Mostremos ahora que contiene el *archivo footer.jspf*.

```
<hr>
Page generated on <%= (new java.util.Date()).toString() %>
```

Con este ejemplo no esperamos que ocurra ningún error, ya que la página es muy simple. Pero de igual manera, si un error ocurriera se mostraría la siguiente pagina: *errorPage.jsp*.

```
<%@ page isErrorPage="true" import="java.io.PrintWriter" %>

<html>
	<head>
		<title>Error</title>
	</head>

	<body>
		<h1>Error</h1>
		There was an error somewhere.
		<%@ include file="/WEB-INF/footer.jspf" %>
	</body>
</html>
```

Y finalmente, el siguiente código muestra el archivo de ayuda que será usado por *welcome.jsp*, *FaqCategories.java*.

```
package com.apress.faq;
import java.util.Iterator;
import java.util.Vector;

public class FaqCategories {
	private Vector categories = new Vector();

	public FaqCategories() {
		categories.add("Dates and Times");
		categories.add("Strings and StringBuffers");
		categories.add("Threading");
	}

	public Iterator getAllCategories() {
		return categories.iterator();
	}
}
```

En el archivo *welcome.jsp* podemos observar muchas de las nuevas características que han sido mencionadas en la sección de JSP. Empecemos con la directiva page:

```
<%@ page errorPage="/WEB-INF/errorPage.jsp"
import="java.util.Iterator, com.apress.faq.FaqCategories" %>
```

Esta directiva tiene dos atributos. El primero, un *errorPage* que está definido, al cual el explorador re direccionara si un error ocurriese en la pagina *welcome.jsp*. El otro atributo usado con la directiva page es el *import*. La página importa dos clases Java: la clase *Iterator* de la API Java y la clase *FaqCategories* que es parte de esta aplicación.

Note que la página también puede usar esta sintaxis para el import:

```
<%@ page errorPage="/WEB-INF/errorPage.jsp"
import="java.util.*, com.apress.faq.*" %>
```

Esta es seguida por un código HTML plano. Más abajo en la página esta una declaración de elementos *scripting*:

```
<%! FaqCategories faqs = new FaqCategories(); %>
```

Este elemento declara una variable llamada *faqs* e se inicializa por la llamada del constructor de la clase *helper FaqCategories*. Se puede observar que la declaración de esos elementos siguen las reglas de codificación Java, incluyendo el uso del punto y coma para terminar una sentencia.

El siguiente elemento JSP en la página es un scriptlet:

```
<%
Iterator categories = faqs.getAllCategories();
while (categories.hasNext()) {
       String category = (String)categories.next();
       %>
       <p><a href="/<%= replaceUnderscore(category) %>"><%= category %></a></p>
<%
}
%>
```

Este *scriplet* obtiene un *Iterator* de la instancia de *FaqCategories*. Este *Iterator* es usado para moverse atreves de cada una de las categorías definidas en la clase *FaqCategories*. Caca categoría es cargada en una variable String llamada *category*, y esta es usada para crear un link HTML. Cada categoría es imprimida dos veces usando elementos de expresión: primero con el atributo href del tag <a> para poner la pagina a la cual el link se refiere, y luego con el cuerpo del link. El primer elemento de expresión llama al método *replaceUnderscore()* e imprime el resultado; el otro elemento expresión simplemente imprime el valor de *category*.

Note que con el *scriplet*, se debe usar la sintaxis Java. Sin embargo, con un elemento de expresión, solo se usa la expresión, sin punto y coma para finalizar la sentencia. Al final de la página, una directiva *include* incluye un pie de página estándar.

```
<%@ include file="/WEB-INF/footer.jspf" %>
```

Este ultimo elemento declara el método *replaceUnderscore(),* el cual remplaza los espacios en una cadena con un guion de piso. Este es llamado por el *scriptlet*.

El siguiente archivo es *footer.jspf*, Como se puede ver este no es un archivo completo JSP. Este archivo usa un elemento de expresión para imprimir la hora y fecha actual del servidor, cuando la pagina es cargada al usuario.

El archivo *errorPage.jsp* es usado para cuando una excepción sea atrapada en la pagina *welcome.jsp*. Esta incluye un pie de página estándar. Sin embargo, asumimos que todo en la pagina *welcome.jsp* esta de forma correcta, y por ende este pagina de error no será cargada. Por otro lado, como se supone que es el archivo de error, este contiene el atributo *isErrorPage*, y es definido como true.

El ultimo archivo fuente *FaqCategories.java*. Este es un *helper class* que suministra tres categorías a la pagina *welcome.jsp*. En una aplicación del mundo real, las categorías podrían venir de algún almacén persistente como una base de datos o un directorio. Para este ejemplo, la *helper class* maneja el código de las categorías para *welcome.jsp*. Las categorías son almacenadas en un objeto *Vector*, el cual es una miembro de la instancia de la clase. En el constructor de la clase, las categorías son agregadas a un vector. Finalmente, la clase define un método *getAllCategories(),* el cual simplemente devuelve el *Iterator* para el *Vector*. La pagina JSP usa este *Iterator* para moverse atreves de cada categoría.

**ANEXO B. Ejemplo De Un Javaserver Faces**

En este ejemplo, simularemos un sistema de reservación de vuelos.

Iniciaremos por la implementación de una clase JavaBean, que representa la capa de negocio de la aplicación web. Este bean será conectado a la capa de presentación por el sistema JSP. Este representa la información necesaria para realizar una búsqueda de vuelo en el sistema. El código de la clase *FlightSearch* se muestra a continuación, y es utilizado para almacenar los parámetros de la búsqueda que fueron ingresados por el usuario. Los parámetros de que se utilizan aquí son: aeropuerto de origen, aeropuerto destino, fecha y hora de partida, y hora y fecha de llegada.

```
package com.apress.jsf;

public class FlightSearch {
        String origination;
        String destination;
        String departDate;
        String departTime;
        String returnDate;
        String returnTime;

        public String getDepartDate() { return departDate;}

        public void setDepartDate(String departDate) {
                this.departDate = departDate;
        }

        public String getDepartTime() { return departTime;}

        public void setDepartTime(String departTime) {
                this.departTime = departTime;
        }

        public String getDestination() {return destination;}

        public void setDestination(String destination) {
                this.destination = destination;
        }

        public String getOrigination() {return origination;}

        public void setOrigination(String origination) {
                this.origination = origination;
        }

        public String getReturnDate() {
                return returnDate;
        }

        public void setReturnDate(String returnDate) {
                this.returnDate = returnDate;
        }

        public String getReturnTime() {return returnTime;}

        public void setReturnTime(String returnTime) {
                this.returnTime = returnTime;
        }
}
```

Mirando la el código, se puede obsercar que es un estándar JavaBean. No existe un constructor explicito, dejando al compilar colocar el constructor sin argumentos por defecto. Existen un capos para los parámetros que queremos almacenar, y unos métodos para obtener y poner esos parámetros. Esto significa que todas las propiedades de la clase están expuestas de lectura y escritura para la aplicación web. Esto permitirá a una parte de la aplicación colocar las propiedades y en otra diferente parte leer las propiedades.

La siguiente parte de nuestro ejemplo es una pagina web que acepte entradas de usuarios para realizar la búsqueda de un vuelo. Eta será una pagina JSP con campos de entrada para el origen, destino, fecha y hora de partida, y fecha y hora de llegada. El código que se muestra a continuación es del archivo *searchForm.jsp*.

```
<html>
	<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
	<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
	<f:view>
	<head>
		<title>Freedom Airlines Online Flight Reservation System</title>
	</head>
	<body>
	<h:form>
		<h2>Search Flights</h2>
		<table>
		<tr><td colspan="4">Where and when do you want to travel?</td></tr>
		<tr>	<td colspan="2">Leaving from:</td>
			<td colspan="2">Going to:</td>
		</tr>
		<tr>	<td colspan="2">
			   <h:inputText value="#{flight.origination}" size="35"/>
			</td>
			<td colspan="2">
			   <h:inputText value="#{flight.destination}" size="35"/>
			</td>
		</tr>
		<tr>	<td colspan="2">Departing:</td>
			<td colspan="2">Returning:</td>
		</tr>
		<tr>	<td>	<h:inputText value="#{flight.departDate}"/></td>
			<td>	<h:inputText value="#{flight.departTime}"/></td>
			<td>	<h:inputText value="#{flight.returnDate}"/></td>
			<td>	<h:inputText value="#{flight.returnTime}"/></td>
		</tr>
		</table>

		<p>
			<h:commandButton value="Search" action="submit"/>
		</p>
	</h:form>
	</body>
	</f:view>
</html>
```

Al ver el anterior código, se puede ver que no existe una sola declaración o scriptlet en la página. Solo existen dos directivas *taglib*, algunas etiquetas HTML, y algunas etiquetas que lucen como *tags custom actions*. El tag que inicia con f: o h: viene de la librería de tags definidas en las directivas taglib.

Los tags que usan el prefijo f: ofrecen el núcleo de funcionalidad JSP para la página, y los tags que usan el prefijo h: ofrecen elementos HTML para la pagina. En la página existe solo un tag JSP, el tag *view.* Cualquier pagina que incluya elementos JSP debe tener el tag view como el tag JSP *outermost*. El resto de los tags en la pagina crean elementos HTML en la pagina. El tag *form* crea un formulario HTML. El tag *input* crea una entrada para un campo de texto en el formulario. El tag *commandButton* crear un botón en el formulario.

Si esta familiarizado con formularios HTML, se debe saber que cada formulario HTML requiere un atributo de acción, que además puede incluir un método opcional de atributo. El atributo de acción, le dice al navegador donde enviar los datos del formulario. El método del atributo le dice al navegador si se trata de una petición GET o una petición POST. Los tags del JSP no usan ninguno de estos atributos. La especificación JSP requiere que todos los formularios se envíen al mismo URL desde el cual fueron pedidos. La especificación también requiere que todos los formularios JSP que usen el método POST para enviar datos del formulario a la aplicación web. Se puede notar que los tags de input tienen una sintaxis diferente a la que se usa en HTML. Por otro lado, vemos la expresión *#{flight.origination}* esta es usada cuando la página JSP quiere acceder a una propiedad de un objeto en la página. El nombre a la izquierda del punto es el nombre del objeto accesible por la página; el nombre a la derecha del punto es una propiedad del objeto al cual se desea acceder. Como este JavaBean esta hecho para que sea disponible en la pagina JSP, esta pagina puede leer o escribir en la propiedad.

La pagina *searchForm.jsp* usa esta expresión con campos de entrada. Cuando se envía esta pagina a la aplicación, los valores ingresados en los campos serán usados para ponerlos en el JavaBean. En una aplicación web real que se ofrezca este servicio, el de reservación de vuelos, este buscara en la base de datos, y se le mostrara al usuario. Sin

embargo, para este ejemplo solo haremos un eco con los parámetros de búsqueda. Esto es concluido con la página *searchResult.jsp*, a continuación se muestra su código:

```
<html>
	<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
	<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
	<f:view>
	<head>
		<title>Freedom Airlines Online Flight Reservation System</title>
	</head>

	<body>
		<h3>You entered these search parameters</h3>
		<p>Origination: <h:outputText value="#{flight.origination}"/>
		<p>Depart date: <h:outputText value="#{flight.departDate}"/>
		<p>Depart time: <h:outputText value="#{flight.departTime}"/>
		<p>Destination: <h:outputText value="#{flight.destination}"/>
		<p>Return date: <h:outputText value="#{flight.returnDate}"/>
		<p>Return time: <h:outputText value="#{flight.returnTime}"/>
		<p>Trip type : <h:outputText value="#{flight.tripType}"/>
	</body>
	</f:view>
</html>
```

Así como en la pagina *searchForm.jsp*, el tag JSP *outermost* es el tag f:view. Con view, la pagina usa los tags h:outputText. El tag *outputText* es usado para mostrar el texto en la página. Así como el tag *inputText*, estos usan la sintaxis #{object.property} para acceder a una propiedad de un objeto en la pagina. Para este caso, el objeto es un JavaBean identificado por el nombre *flight*. El tag *outputText* lee la propiedad del objeto y la muestra en la página web genera por este JSP.

Ahora ya se ha visto las tres partes principales de una aplicación web: una pagina de entrada, una pagina de salida y un JavaBean para guardar los datos de negocio. En términos del patrón MVC, *FlightSearch* es el modelo, y las paginas *searchForm* y searchResult.jsp son la vista. Lo que no hemos mostrado aun es el controlador. Tampoco se ha explicado como el controlador sabe donde encontrar el modelo o la vsta, y además como el controlador sabe el flujo lógico atreves de la aplicación web. En los códigos expuestos en este ejemplo, se puede observar que no se tiene ninguna información que indique como controlar la transferencia de pagina a pagina. Veamos como es manejado este control.

La información sobre los componentes de vista en la aplicación web, y la información sobre como controlar el flujo a través de la aplicación es contenida en un archivo de configuración especial llamado *faces-config.xml*, que se muestra a continuación:

```
<?xml version="1.0"?>

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
       version="1.2">

       <navigation-rule>
       <from-view-id>/searchForm.jsp</from-view-id>
       <navigation-case>
              <from-outcome>submit</from-outcome>
              <to-view-id>/searchResults.jsp</to-view-id>
              <redirect/>
       </navigation-case>
       </navigation-rule>

<managed-bean>
       <managed-bean-name>flight</managed-bean-name>
       <managed-bean-class>com.apress.jsf.FlightSearch</managed-bean-class>
       <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>
```

Aunque este archivo puede contener distinta información sobre una aplicación web, para este ejemplo, solo se necesita que haga dos cosas: identificar el flujo de control desde *searchForm.jsp* a *searchResult.jsp*, e identificar el JavaBean usado por la aplicación.

Este archivo identifica el JavaBean usado por la aplicación web en el elemento managed-bean. Se tendrá un elemento managed-bean por cada JavaBean usado en la aplicación web. El elemento managed-bean contiene tres sub elementos:

- El primer sub elemento es el nombre usado para identificar el bean en una pagina JSP. En el segmento de código el nombre utilizado es *flight*; esto es porque ambas paginas searchForm.jsp y searchResult.jsp pueden acceder a una instancia del bean usando la expresión #{flight…}
- El segundo elemento es el nombre de la clase calificadora del JavaBean. Este nombre le dice al contenedor JSP que clase cargar e instanciar para crear una instancia del JavaBean.
- El tercer elemento identifica el alcance del objeto. Alcance de sesión significa que el objeto existe para la interacción entre el usuario y la aplicación. El contenedor debe mantener el objeto atraves de multiples ciclos de peticiones/respuestas, hasta que la sesión con el usuario halla terminado.

El archivo faces-config.xml tambien es usado para decirle al controlador como navegar atravez de la aplicación. El flujo de navegación es especificado en el elemento navigation-rule. Para nuestro ejemplo solo necesita un elemento. En general, un elemento de navigation-rule identifica la pagina de inicio, una condición, y una pagina a la cual navegara cuando la condición ocurra.

En nuestro ejemplo, la página de inicio es *searchForm.jsp*. Si la página solicitada es procesada con un resultado, el control es transferido a *searchResult.jsp*. Mirando el código del archivo *searchForm.jsp*, se puede observar que el elemento *commandButton* tiene una acción de *submit*; cuando el botón es presionado y el formulario es enviado, esta acción es igual al *from-outcome* del *navigation-rule*.

El elemento *navigation-rule* también incluye un elemento *redirect*. Con este elemento, la respuesta es creada, causando que el navegador se dirija a la pagina searchResult.jsp, la cual también actualiza la barra de dirección en el navegador. Sin este elemento, la respuesta también es creada correctamente y se enviara al navegador, pero la barra de direcciones no actualizara, quedándose con el nombre de la anterior página.

Finalmente, solo necesitamos una pieza final para nuestra aplicación web. Crear una pagina por defecto que estará en código HTML, esta redireccionara a la pagina correcta de una aplicación JSP. Un archivo index.html:

```
<html>
	<head>
		<meta http-equiv="Refresh" content= "0; URL=searchForm.faces"/>
	</head>
</html>
```

Se puede observar que esta pagina redirecciona hacia searchForm.faces. Sin embargo no existe ningún componente en nuestra aplicación llamado así. Entonces, ¿Cómo es que la aplicación web sabe que pagina servir? Todas las peticiones que son peticiones JSP son dirigidas hacia el controlador de la aplicación, el cual es un Servlet suplido como parte de la implementación de referencia JSP. Este Servlet convierte la petición *searchForm.faces* a *searchForm.jsp,* se procesa la pagina JSP y se envía la respuesta al navegador.

**ANEXO C. Ejemplo De Un Servlet**

Ahora que se sabe lo básico de los objetos Servlets, miremos un sencillo ejemplo donde usaremos los métodos de *HttpServletRequest* y *HttpServletResponse*. Crearemos un ejemplo que responda a peticiones HTTP POST. Iniciemos por crear un sencillo Servlet. A continuación se muestra el código del archivo *Login.java*:

```
package com.apress.servlet;

import javax.servlet.http.*;
import java.io.*;

public class Login extends HttpServlet {
	public void doPost(HttpServletRequest request,
		HttpServletResponse response){

		String username = request.getParameter("username");
		try {
			response.setContentType("text/html");
			PrintWriter writer = response.getWriter();
			writer.println("<html><body>");
			writer.println("Thank you, " + username +
				". You are now logged into the system.");
			writer.println("</body></html>");
			writer.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Este Servlet implementa el método doPost(). Así, para llamar al Servlet, se necesita un cliente que puede enviar peticiones POST. A continuación se muestra el código de una pagina HTML, login.html que tiene un formulario que envía al Servlet:

```
<html>
	<head>
		<title>Login</title>
	</head>

	<body>
		<h1>Login</h1>
		Please enter your username and password
		<form action="Login" method="POST">
		<p>Username: <input type="text" name="username" length="40">
		<p>Password: <input type="password" name="password" length="40">
		<p><input type="submit" value="Submit">
		</form>
	</body>
</html>
```

El Servlet Login, ilustra algunos de los principales conceptos vistos. La clase por si misma es solo como cualquier otra clase. En este caso, esta es subclase de HttpServlet. Como una subclase de HttpServlet, la clase Login solo necesita sobre escribir los métodos de HttpServlet para implementar su comportamiento, o alternativamente, agregar nuevos métodos para un nuevo comportamiento. En este ejemplo, se necesita solo sobrescribir el método *doPost()* de HttpServlet.

Cuando se hace clic en el botón Submit de la pagina estática login.html, el navegador web envía una petición de POST al servidor. Los formularios web pueden ser usados para enviar cualquiera de las dos peticiones GET o POST. El tag <form> en la pagina web tiene un método de atributo que tiene el valor POST. Esto le dice al navegador que envie una petición POST a la fuente indicada por el atributo *action* del tag <form>. Si no se usase un método de atributo, el formulario por defecto usaría un método GET.

Cuando el servidor recibe una petición POST, este analiza el URL para determinar que fuente enviar a la petición. Existen dos tipos de rutas URLs:

**Rutas relativas:** Estas rutas no tienen el carácter slash (/). El HTML en código usa una ruta relativa para identificar la fuente. El atributo de *action* del tag <form> es simplemente la dirección relativa de *Login*, y el navegador convierte esta a la apropiada URL. Cuando el navegador convierte el *Login* a un URL, esta se anexa en el contexto de la aplicación actual ya que este es una ruta relativa. Este mapea Login a un URL relativo a la actual pagina. Por ejemplo, *login,html* fue servida de el URL http://localhost:8080/Servlet_Ex01/login.html. El navegador anexa Login a la misma ruta de la pagina login.html para forma el URL http://localhost:8080/Servlet_Ex01/Login.

**Rutas absolutas:** Esta ruta inicia con un slash (/). Las rutas absolutas están formadas relativamente al servidor. Si el formulario en login.html contiene la ruta absoluta /Login, el navegador debería enviar la petición a http://localhost:8080/Login. Este no identifica una fuente valida en el servidor, y debería resultar un error 404 Not Found.

Se pueden utilizar cualquiera de estas dos rutas en una pagina web. Si se usa rutas absoluta, se necesitara incluir el contexto del componente web como parte de la ruta. Por

ejemplo, la ruta absoluta mostra en el siguiente tag <form> debería causar que el navegador envie el formulario a la correcta fuente:

```
<form action=”/Servlet_Ext01/Login” method=POST”>
```

Cuando se despliega este Servlet, para del descriptor de despliegue le dice al servidor y al contenedor Servlet como mapear la petición de la ruta a la clase Servlet.

En este ejemplo, el elemento de ruta /Servlet_Ex01 del URL le dice al servidor pasar la petición al contenedor Servlet. La porción /Login de la ruta mapea el Servlet Login, el cual es implementado por la clase *com.apress.servlet.Login*. El contenedor Servlet construye instancias de HttpServletRequest y HttpServletResponce, y llama al método *service()* de *Login*. Como *Login* no tiene implementado *service()*, el método de la clase padre es llamada. El método *service()* de HttpServlet determina si es una petición POST y llama al método *doPost().* Como *Login* define *doPost*(), este método es usado para procesar la petición.

Junto con el método *doPost(),* el Servlet *Login* lee el parámetro de petición del objeto *HttpServletRequest*. El método que es usado para hacer esto es *getParameter(String),* el cual devuelve un String que tiene el valor del parámetro de petición con el nombre dado. Si el parámetro no existe, entonces se retornara un valor nulo. El parámetro usado por el Servlet es “*username*”:

```
String username = request.getParameter("username");
```

Este es el mismo como el nombre del atributo usado en el formulario web para la entrada del campo *username*:

```
<p>Username: <input type="text" name="username" length="40">
```

El Servlet Login usa el objeto response para devolver una respuesta al cliente. Esta inicia por configurar el tipo de contenido de respuesta a “text/html”:

```
response.setContentType("text/html");
```

Después de configurar el tipo de contenido, el Servlet obtiene un objeto Writer del objeto response. Este Writer es usado para enviar las cadenas que constituyen la respuesta al cliente:

```
try {
       response.setContentType("text/html");
       PrintWriter writer = response.getWriter();
       writer.println("<html><body>");
       writer.println("Thank you, " + username +
       ". You are now logged into the system");
       writer.println("</body></html>");
writer.close();
} catch (Exception e) {
       e.printStackTrace();
}
```

Como escribir por un flujo puede lanzar una IOException, todo el bloque esta encapsulado en un bloque try-catch.

**ANEXO D. Ejemplo de un bean de session**

Es momento de pasar de toda esta teoría a la práctica. A continuación desarrollaremos un bean de sesión basándonos en un ejemplo tradicional, el programa “Hello World!”.

Ya que este es el primer ejemplo de los EJBs, detallaremos bástate el código. Para este ejemplo existen tres archivos fuentes: SimpleSession.java, SimpleSessionBean.java y SimpleSessionClient.java.

A continuación se muestra el código para *la interface de negocio remota*, SimpleSession.java.

```
package beans;

import javax.ejb.Remote;

@Remote
public interface SimpleSession{
        public String getEchoString (String clientString);
}
```

El siguiente código muestra la implementación del bean de sesión, SimpleSessionBean.java.

```
package beans;

import javax.ejb.Stateless;

@Stateless
public class SimpleSessionBean implements SimpleSession {
        public String getEchoString(String clientString) {
                return clientString + " - from session bean";
        }
}
```

Finalmente el siguiente muestra el código del cliente que sirve para probar el bean de sesión.

```
package client;

import beans.SimpleSession;
import javax.naming.InitialContext;

public class SimpleSessionClient {
        public static void main(String[] args) throws Exception{
                InitialContext ctx = new InitialContext();
                SimpleSession simpleSession
                        = (SimpleSession) ctx.lookup(SimpleSession.class.getName());
                for (int i = 0; i < args.length; i++) {
                        String returnedString = simpleSession.getEchoString(args[i]);
                        System.out.println("sent string: " + args[i] +
                                ", received string: " + returnedString);
                }
        }
}
```

Tenemos tres archivos fuentes que iremos detallando. Iniciaremos con el código del cliente y luego miraremos la interface del bean de sesión y la clase.

El método *main()* de la clase SimpleSessionClient facilita las cosas haciendo uso de JNDI, esto nos ayuda a obtener una referencia a la interface de negocio del bean de sesión. JNDI ofrece una interface común para los servicios de directorio (esto incluye LDAP, proveedores CORBA, y para nuestro caso, archivos locales del sistema). El directorio que nosotros estamos tratando aquí es interno al servidor EJB y mantiene una referencia a la interface de negocio de nuestro bean de sesión. A esta referencia se accede haciendo uso de JNDI *lookup*, al cual se le especifica un nombre calificador de la interface, para nuestro caso el nombre de la interface del bean. También podríamos especificar la localización del bean mediante una ruta absoluta JNDI, pasándole a la función *lookup()* la cadena literal *ejb/beans.SimpleSession*.

Una vez que la función *lookup()* devuelva el objeto, inmediatamente le hacemos un *casting* con la interface *SimpleSession*. Mucha atención en la decisión de adquirir una referencia JDNI usando el nombre de la clase *SimpleSession*. Al solicitar una referencia JNDI de una interface con el descriptor metadata *@Remote*, estamos instanciando implícitamente al método que queremos usar para comunicarnos con el bean.

```
InitialContext ctx = new InitialContext();
SimpleSession simpleSession
        = (SimpleSession) ctx.lookup(SimpleSession.class.getName());
```

En el código cliente para este ejemplo, se demuestra que podemos pasar un argumento a un método de un bean de sesión que exista en el servidor, el servidor realizar las operaciones necesarias a ese argumento y devuelve un diferente valor al cliente. Mas exactamente, el código “mira” los argumentos que le pasamos por el método *main()* del cliente vía línea de comando, direccionado uno a uno al método *getEchoString()* del bean de sesión. Esto se logra haciendo llamado del método *getEchoString()* de la interface del bean que existe en el cliente.

```
for (int i = 0; i < args.length; i++) {
       String returnedString = simpleSession.getEchoString(args[i]);
       System.out.println("sent string: " + args[i] +
              ", received string: " + returnedString);
}
```

Note que al invocar el método *getEchoString()* de la interface que se encuentra en el cliente, este invoca el método *getEchoString()* del bean de sesión que esta en el servidor. Esto se hace posible gracias al uso de *remoting*, el mecanismo por el cual la llamada a métodos que se encuentre en el servidor sean invocados remotamente.

El método de negocio de nuestro bean de sesión que la clase cliente llama, toma una cadena como argumento que es provista por el cliente, y regresa la misma cadena con un mensaje de estado anexado.

```
public String getEchoString(String clientString) {
       return clientString + " - from session bean";
}
```

Este método también es definido en la interface del bean, especificada en el código de SimpleSession:

```
public String getEchoString(String clientString);
```

La interface del bean merece una segunda ojeada por una importante razón: el descriptor metadata.

```
import javax.ejb.Remote;
@Remote
public interface SimpleSession
```

De pronto parecerá que @*Remote* no signifique mucho, pero su presencia es extremadamente importante para el desarrollo de un bean de sesión. Esta anotación le dice al compilador y al contenedor EJB que la interface en cuestión esta realizada para que sea utilizada por clientes remotos. El compilador generara la interface con toda la funcionalidad necesaria para permitir que clientes remotos invoquen métodos desde un contenedor EJB directamente, y solo lo que se necesito fue agregar un argumento extra a la declaración de la interface. Si fuera más sencillo, el código se escribiera por si mismo.

Hablando de sencillez, démosle otro vistazo al archivo SimpleSessionBean.java, mira la notación particular @*Stateless*.

```
package beans;

import javax.ejb.Stateless;

@Stateless
public class SimpleSessionBean implements SimpleSession
```

El descriptor @*Stateless* es una instrucción para el compilador y el contenedor EJB, indicando que la clase en cuestión debería ser desplegada y manipulada como un bean de sesión sin estado. Esta anotación es el procedimiento principal que se usara para desarrollar los EJBs.

**ANEXO E. Ejemplo de un JavaBean de entidad**

Los bean de entidad CMP tienen sus datos persistentes manejados por el contenedor EJB atreves del uso de una base de datos. Consecuentemente, el diseño de un bean de entidad en una aplicación puede ser muy parecido al diseño de tablas en una base de datos relacional, ten en cuenta que los bean de entidad son objetos y por ello pueden tener métodos de negocio así como datos.

Durante el análisis, las entidades que son sustantivos, son consideradas candidatas para ser representadas como un bean de entidad. Por ejemplo, una aplicación que ayuda al manejo de los estudiantes debe tener los siguientes beans de entidad: *Student*, *Institution*, *Counselor*, *Course* y *Program*. Otro paso lógico es descubrir los campos persistentes para cada bean de entidad. Por ejemplo, el bean de entidad *Course* debería tener campos CMP para el nombre del curso y descripción.

**Nota.** Mucho cuidado, el tamaño de la información que decides que sea persistente calculara que cantidad de espacio es ocupada en el almacén de datos y cuanto tiempo le toma al contenedor en llenar la entidad con estos datos persistentes.
El ejemplo que trabajaremos a trabes de esta sección, es una simple aplicación en la cual un usuario puede crear, buscar, actualizar y borrar las existencias de artículos en una tienda. Esta aplicación usa dos EJBs:

- Un bean de entidad llamado *Stock* que mantendrá información sobre las existencias. Existe una instancia de este bean de entidad por cada artículo.
- Un bean de sesión llamado *StockList* que usa el bean *Stock* y ofrecerá métodos de negocio a la interfaz grafica que le permiten controlar el bean *Stock*.

El siguiente segmento muestra el código para el bean de entidad, Stock.java.

```
package beans;

import javax.persistence.Entity;
import java.io.Serializable;
import javax.persistence.Id;

@Entity
public class Stock implements Serializable {
	// The persistent fields
	private String tickerSymbol;
	private String name;

	// Constructors
	public Stock() { }
	public Stock(String tickerSymbol, String name) {
		this.tickerSymbol = tickerSymbol;
		this.name = name;
	}

	// The access methods for persistent fields
	// tickerSymbol is the id
	@Id
	public String getTickerSymbol() {
		return tickerSymbol;
	}

	public void setTickerSymbol(String tickerSymbol) {
		this.tickerSymbol = tickerSymbol;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

Los siguientes dos segmentos muestra el código fuente para el bean de sesión, StockList.java y StockListBean.java

```
package beans;

import javax.ejb.Remote;

@Remote
public interface StockList {
	// The public business methods on the StockList bean
	public String getStock(String ticker);
	public void addStock(String ticker, String name);
	public void updateStock(String ticker, String name);
	public void deleteStock(String ticker);
}
```

```
package beans;

import beans.Stock;
import javax.persistence.PersistenceContext;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;

@Stateless
public class StockListBean implements StockList {

	// The reference to the entity manager
	@PersistenceContext
	private EntityManager _manager;

	// The public business methods. these must be coded in the
	// interface also.
	public String getStock(String ticker) {
		Stock stock = _manager.find(Stock.class, ticker);
		return stock.getName();
	}

	public void addStock(String ticker, String name) {
		_manager.persist(new Stock(ticker, name));
	}

	public void updateStock(String ticker, String name) {
		Stock stock = _manager.find(Stock.class, ticker);
		stock.setName(name);
	}

	public void deleteStock(String ticker) {
		Stock stock = _manager.find(Stock.class, ticker);
		_manager.remove(stock);
	}
}
```

Finalmente, el siguiente segmento muestra el código que define la interfaz de usuario, *StockClient.java*.

```
package client;

import beans.StockList;
import javax.naming.InitialContext;

// General imports
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class StockClient extends JFrame implements ActionListener {
        private StockList _stockList;
        private JTextField _ticker = new JTextField();
        private JTextField _name = new JTextField();
        private JButton _get = new JButton("Get");
        private JButton _add = new JButton("Add");
        private JButton _update = new JButton("Update");
        private JButton _delete = new JButton("Delete");

        public StockClient() {

                // Get the stock lister
                _stockList = getStockList();

                // Add the title
                JLabel title = new JLabel("Stock List");
                title.setHorizontalAlignment(JLabel.CENTER);
                getContentPane().add(title, BorderLayout.NORTH);

                // Add the stock label panel
                JPanel stockLabelPanel = new JPanel(new GridLayout(2, 1));
                stockLabelPanel.add(new JLabel("Symbol"));
                stockLabelPanel.add(new JLabel("Name"));
                getContentPane().add(stockLabelPanel, BorderLayout.WEST);

                // Add the stock field panel
                JPanel stockFieldPanel = new JPanel(new GridLayout(2, 1));
                stockFieldPanel.add(_ticker);
                stockFieldPanel.add(_name);
                getContentPane().add(stockFieldPanel, BorderLayout.CENTER);

                // Add the buttons
                JPanel buttonPanel = new JPanel(new GridLayout(1, 4));
                _get.addActionListener(this);
                buttonPanel.add(_get);
                _add.addActionListener(this);
                buttonPanel.add(_add);
                _update.addActionListener(this);
                buttonPanel.add(_update);
                _delete.addActionListener(this);
                buttonPanel.add(_delete);
                getContentPane().add(buttonPanel, BorderLayout.SOUTH);

                addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                setSize(330, 130);
                setVisible(true);
        }
```

```
	private StockList getStockList() {
		StockList stockList = null;
		try {
			// Get a naming context
			InitialContext ctx = new InitialContext();
			// Get a StockList object
			stockList
				= (StockList) ctx.lookup(StockList.class.getName());
		} catch(Exception e) {
			e.printStackTrace();
		}
		return stockList;
	}

	public void actionPerformed(ActionEvent ae) {
		// If get was clicked, get the stock
		if (ae.getSource() == _get) {
			getStock();
		}
		// If add was clicked, add the stock
		if (ae.getSource() == _add) {
			addStock();
		}
		// If update was clicked, update the stock
		if (ae.getSource() == _update) {
			updateStock();
		}
		// If delete was clicked, delete the stock
		if (ae.getSource() == _delete) {
			deleteStock();
		}
	}

	private void getStock() {
		// Get the ticker
		String ticker = _ticker.getText();
		if (ticker == null || ticker.trim().length() == 0) {
			JOptionPane.showMessageDialog(this, "Ticker is required");
			return;
		}

		// Get the stock
		try {
			String name = _stockList.getStock(ticker.trim());
			_name.setText(name);
		}catch (Exception e) {
			e.printStackTrace();
		}
	}
	private void addStock() {
		// Get the ticker
		String ticker = _ticker.getText();
		if (ticker == null || ticker.trim().length() == 0) {
			JOptionPane.showMessageDialog(this, "Ticker is required");
			return;}
		// Get the name
		String name = _name.getText();
		if (name == null || name.trim().length() == 0) {
			JOptionPane.showMessageDialog(this, "Name is required");
			return;
		}
		// Add the stock
		try {  _stockList.addStock(ticker.trim(), name.trim());
			JOptionPane.showMessageDialog(this, "Stock added!");
		}catch (Exception e) {
			e.printStackTrace();
		}
	}
```

```
	private void updateStock() {
		// Get the ticker
		String ticker = _ticker.getText();
		if (ticker == null || ticker.trim().length() == 0) {
			JOptionPane.showMessageDialog(this, "Ticker is required");
			return;
		}

		// Get the name
		String name = _name.getText();
		if (name == null || name.trim().length() == 0) {
			JOptionPane.showMessageDialog(this, "Name is required");
			return;
		}

		// Update the stock
		try {
			_stockList.updateStock(ticker.trim(), name.trim());
			JOptionPane.showMessageDialog(this, "Stock updated!");
		}catch (Exception e) {
			e.printStackTrace();
		}
	}

	private void deleteStock() {
		// Get the ticker
		String ticker = _ticker.getText();
		if (ticker == null || ticker.trim().length() == 0) {
			JOptionPane.showMessageDialog(this, "Ticker is required");
			return;
		}

		// Delete the stock
		try {
			_stockList.deleteStock(ticker.trim());
			JOptionPane.showMessageDialog(this, "Stock deleted!");
		}catch (Exception e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {
		StockClient stockClient = new StockClient();
	}
}
```

Para la explicación primero daremos un vistazo a la clase de bean entidad, ¿porque gastamos esfuerzo en tener dos constructores? La especificación EJB requiere que todos los bean de entidad ofrezcan al menos un constructor sin argumentos para el contenedor EJB. Nosotros ofrecemos el segundo constructor solamente para propósitos funcionales en la parte de llamadas de clientes.

A continuación mencionamos algunos datos resaltables sobre este bean de entidad:

- Implementación de la interface java.io.Serializable a nuestro bean de entidad. Hacemos esto para asegurarnos que este disponible para la interfaces remotas (más específicamente, nuestra interface *StockList*).
- No ofrecemos un tipo de generador con la anotación @id. Esto es porque confiamos en la llave suministrada por el cliente para identificar los objetos *Stock* en el almacén, y no es sencillo automatizar esta llave única basándonos en entradas pasadas al almacén.
- No inicializamos ninguna variable en nuestro constructor sin argumentos. Se hace esto para evitar que las variables persistentes tomen valores no nulos que erróneamente sean almacenadas en el almacén.

Dirijamos nuestra atención ahora al código del bean de sesión. El bean de sesión *StockList* usa nuestro bean de entidad. Los cuatro métodos en esa interface serán usados por el cliente para llevar las operaciones de obtención, agregar, actualización y de borrado mostrados en GUI del cliente. La clase del bean de sesión, *StockListBean.java* respaldan estas funcionalidades con los métodos *getStock*(), *addStock*(), *updateStock*() y *deleteStock*() respectivamente.

Cada una de esas llamadas a esos métodos le pertenecen a la interfase *EntityManager*, pero ¿como se instancia la clase implementa *EntityManager*? El lector seguramente se habrá dado cuenta de la anotación *@PersistenceContext* (*javax.persistence.PersistenceContext*) que precede nuestra declaración del miembro *EntityManager*. Pero ¿Qué es *@PersistenceContext*?

La anotación @PersistenceContext notifica al compilador que la variable o propiedad anotada representa un valor que será asignado a el por el contenedor de aplicación (en este caso, nuestro contenedor EJB). Cuando nuestro bean de sesión es instanciado, la variable *manager* automáticamente será colocada a la interface *EntityManager* del contenedor, ahorrándonos el tedioso trabajo de buscar y adquirir la instancia del contenedor *EntityManger*.

```
    public String getStock(String ticker) {
           Stock stock = _manager.find(Stock.class, ticker);
           return stock.getName();
    }
```

El método *getStock()* llama al método *find()* de la interface *EntityManager*, pasándole la clase del bean de entidad *Stock* y el símbolo del *ticker* deseado. ¿Por qué pasamos la clase al *EntityManager*? Recuerda que el bean de entidad esta asociado con una localización especifica en el almacén (en este caso, una tabla en la base de datos). Al pasar la clase al *EntityManager* es obvio dejarle saber en que tabla debería estar buscándola. Si este encuentra la tabla, este pregunta al bean de entidad Stock por el valor sujeto en el campo CMP *name*, y retorna el valor de la siguiente manera::

- El método *addStock()* llama al método *persist()* del *EntityManager*, pasándole a este una nueva instancia de la entidad *Stock*.
- El método *updateStock()* usa el método *find()* para obtener la referencia deseada del bean de entidad Stock, y este usa un método CMP para actualizar el campo *name*.
- El método *deleteStock()* usa el metodo *find()* para obtener la referencia deseada del bean de entidad *Stock*. Luego este llama el método *remove()* de la interface *EntityManeger*.

La clase *StockClient* obtiene una referencia a un bean de sesión:

```
private StockList getStockList() {
    StockList stockList = null;
    try {
           // Get a naming context
           InitialContext ctx = new InitialContext();
           stockList
                  = (StockList) ctx.lookup(StockList.class.getName());
    } catch(Exception e) {
           e.printStackTrace();
    }
    return stockList;
}
```

**ANEXO F. Ejemplo de MDBs y las API explicadas**

Ahora miraremos un ejemplo que demuestra las tecnologías mencionadas anteriormente en la sección: los MDBs, la API JMS, y el Servicio Timer EJB.

La figura 14 bosqueja el comportamiento de este ejemplo, que de describe a continuación:

- El bean de session *TimeIt* usa el servicio Timer EJB para ser notificado cada diez segundos.
- Cada vez que el bean *TimeIt* es notificado, este usa la API JSM para crear un mensaje productor JSM.
- El bean *TimeIt* usa el mensaje productor JSM para crear y enviar un mensaje al MDB *MessageWriter*. En nuestro ejemplo, este es un mensaje de texto que contiene la fecha y la hora en que el mensaje fue enviado.
- El JMS *message producer* envía el mensaje al *LogWriter*, el cual es un nombre arbitrario *JMS queue* creado para este ejemplo.
- Un JMS *message consumer*, el cual es creado y manejado por el contenedor EJB, recibe el mensaje.
- El contenedor EJB llama al método *onMessage()* del MDB *MesaggeWriter*, pasando el texto del mensaje en el método.
- El bean *MessageWriter* crea un *String*, que la concadena con el texto recibido, y lo envía al *System.out*. Note que el *System.out* es manejado por el servidor de aplicación, así iniciaremos una forma especial para ver la salida.

Para construir el ejemplo, crearemos los siguientes archivos .java:

- TimeIt.java en el paquete timer
- TimeItBean.java en el paquete timer
- TimerItTester.java en el paquete client
- MessageWriter.java en el paquete msg
- timer-message-service.xml en el directorio de despliegue JMS (c:\jboss\server\all\deploy\jms)

Implementaremos el código involucrado en el ejemplo, de acuerdo al flujo del mensaje desde donde es enviado. Primero, la interface remota del bean de sesión *TimeIt*:

```
package timer;

import javax.ejb.Remote;

@Remote
public interface TimeIt {
	// the public business method on the timer bean
	public void startTimer();
}
```

La interface remota solo tiene un método de negocio, *startTime()*. Este método será invocado por la aplicación cliente *TimeItTester*, que a continuación se muestra el código:

```
package client;

import timer.TimeIt;
import javax.naming.InitialContext;

public class TimeItTester {
	public static void main(String[] args) throws Exception {
		// Get a naming context
		InitialContext ctx = new InitialContext();

		// Create a TimeIt object
		TimeIt timeIt = (TimeIt) ctx.lookup(TimeIt.class.getName());
		timeIt.startTimer();
	}
}
```

Ahora tenemos el código corazón del ejemplo: Usando el servicio Timer EJB. Aquí tenemos la implementación del bean de sesión *TImeIt* contenido en el archivo *TimeItBean.java*:

```
package timer;

import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;
import javax.jms.Queue;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.Session;
import javax.jms.TextMessage;

// General imports
import java.text.*;
import java.util.*;

@Stateless
public class TimeItBean implements TimeIt {
	private @Resource SessionContext ctx;
	private @Resource TimerService timer;
	private @Resource(name="ConnectionFactory") ConnectionFactory factory;
	private @Resource(name="queue/LogWriter") Queue queue;
	private SimpleDateFormat sdf =
		new SimpleDateFormat("yyyy.MM.dd 'at' HH:mm:ss.SSS");

	// Public business method to start the timer
	public void startTimer(){
		// After initial five seconds, then every ten seconds
		timer.createTimer(5000, 10000, "timer");
	}

	// Timer method - timer expires - send message to queue
	@Timeout
	public void timeoutHandler(Timer timer) {
		Connection connection = null;
		try {
			// Get a connection from the factory
			connection = factory.createConnection();

			// Create a session
			Session session = connection.createSession(false,
				Session.AUTO_ACKNOWLEDGE);

			// Create a sender for the session to the queue
			MessageProducer sender = session.createProducer(queue);

			// Create a text message
			TextMessage message = session.createTextMessage();

			// Set the text of the message
			message.setText ("log entry, the time is: " + sdf.format(
				new Date()));

			// Send the message
			sender.send(message);
		}catch (Exception e) {
			System.out.println("Exception in message: " + e.toString());
			e.printStackTrace();
		}finally {
			if (connection != null) {
				try {
					connection.close();
				}catch (Exception e) {}
			}
		}
	}
}
```

Sigue el código de nuestro MDB, *MessageWriter.java*. No existe ninguna interface ya que los MDBs no las usan:

```
package msg;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(activateConfig ={
        @ActivationConfigProperty(propertyName="destinationType",
                propertyValue="javax.jms.Queue"),

        @ActivationConfigProperty(propertyName="destination",
                propertyValue="queue/LogWriter")
})

public class MessageWriter implements MessageListener {
        // Must implement this method for MessageDriven
        public void onMessage(Message message) {
                TextMessage msg = null;
                try {
                        if (message instanceof TextMessage) {
                                msg = (TextMessage) message;
                                System.out.println("Got message: " + msg.getText());
                        }else {
                                System.out.println("Got message of type: "
                                + message.getClass().getName() + " ==> ignored!");
                        }
                }catch (Throwable te) {
                        te.printStackTrace();
                }
        }
}
```

Soportando a los MDBs esta el archivo de despliegue XML (timer-message-service.xml), el cual le dice al subsistema JMS que cree un mensaje *queue* para ser usado por el MDB. Este archivo XML se coloca en el directorio de despliegue JMS del servidor de aplicación JBoss (c:\jboss\server\all\deploy\jms).

```
<?xml version="1.0" encoding="UTF-8" ?>
<server>
        <mbean code="org.jboss.mq.server.jmx.Queue"
                name="jboss.mq.destination:service=Queue,name=logQueue">
                <attribute name="JNDIName">queue/LogWriter</attribute>
                <depends optional-attribute-name="DestinationManager">
                        jboss.mq:service=DestinationManager
                </depends>
        </mbean>
</server>
```

Ahora que ya se conocen los archivos fuentes de la aplicación, miremos como trabajan juntas estas tecnologías (y porque son tan útiles). Primero examinemos como el Servicio Timer EJB esta implementado en la clase *TimeItBean*.

La directiva *@Resource SessionContext* en el bean de sesión es complementada por el contenedor EJB después que se ha creado el bean de sesión:

```
private @Resource SessionContext ctx;
```

El contenedor toma nota de la directiva *@Resource* (parecido a como hicimos en la anterior sección: Bean de entidad, cuando se usó *@PersistenceContext* para adquirir un *EntityManager* en el bean de sesión) y fijamos el valor del miembro privado igual al del objeto *SessionContext*, el cual representa el contexto del contenedor EJB en el que el bean de session esta corriendo. Como se menciono anteriormente, a este acto se le llama resource injection.

El resource injection también es usado para adquirir referencias al JMS *ConnectionFactory (javax.jms.ConnectionFactory),* mensajes *JMS Queue (javax.jms.Queue)* y objetos EJB *TimerService (javax.ejb.TimerService).*

Note que al usar la anotación *@Resource* para adquirir esos objetos, estas ahorrándote pasos extra que serian requeridos para usar los servicios de contenedor.

Una vez que una referencia del objeto TimerService para el contenedor halla sido obtenida, se usara en el método *startTimer()* para crear un objeto *timer* que se ejecuta sobre intervalos específicos.

Específicamente, el objeto Timer (javax.ejb.Timer) caducara, provocara, disparara, apagara – como sea como lo quieras decir – en cinco segundos, y después cada 10 segundos:

```
public void startTimer()
{
// After initial five seconds, then every ten seconds
timer.createTimer(5000, 10000, "timer");
}
```

Cuando el *timer* se dispara, el método del Enterprise bean anotado con el descriptor @TimeOut (javax.ejb.Timeout) es invocado. En nuestro ejemplo, este método (*public void timeoutHandler(Timer timer) )* usa la API JMS para enviar un mensaje asíncrono a un MDB, ahora cambiemos nuestra atención.

De acuerdo al diagrama y la descripción de comportamiento de este ejemplo, una cosa que el método anotado por @*Timeout* necesita hacer es crear un JMS *message producer* que pueda enviar mensajes al *LogWriter queue*. Para hacer eso, este usara el objeto *ConnectionFactory* adquirido por medio de la anotación @*Resource* para crear una Session (javax.jms.Session) que se comunica con *MessageWriter*. Note que el ConnectionFactory y el Queue han sido considerador ya han sido inicializados por el contendor, esto es necesario para que el bean Session los pueda conectar.

```
private @Resource(name="ConnectionFactory")
ConnectionFactory factory;
private @Resource(name="queue/LogWriter")
Queue queue;
```

El directiva *name* que se pasa a la anotación @*Resource* esta indicando al contenedor que este debería poblar la variable en cuestión (factory o queue) con una referencia a la instancia que el contenedor reconoce por ese nombre. ¿Recuerdas el archivo de despliegue XML timer-message-service.xml? Este archivo en particular instruye al subsistema JMS para crear un *queue* llamado *queue/LogWriter*. Aquí ese nombre JNDI es referenciado por la anotación @*Resource* para adquirirlo, donde este es subsecuentemente es usado por *timeoutHandler(Timer timer):*

```
Connection connection = null;
try {
       // Get a connection from the factory
       connection = factory.createConnection();
       // Create a session
       Session session =
       connection.createSession(false,
       Session.AUTO_ACKNOWLEDGE);
       // Create a sender for the session to the queue
       MessageProducer sender = session.createProducer(queue);
       // Create a text message
       TextMessage message = session.createTextMessage();
       // Set the text of the message
       message.setText
       ("log entry, the time is: " + sdf.format(new Date()));
       // Send the message
       sender.send(message);
}
```

El método *createProducer()* del objeto Session es pasado por referencia al *Queue*, y este crea un objeto *MessageProducer* (javax.jms.MessageProducer) que puede enviar mensajes a ese destino, así como se muestra en el siguiente código. El *MessageProducer* es representado por el *JMS message producer* en la figura 14.

```
// Create a sender for the session to the queue
MessageProducer sender = session.createProducer(queue);
```

Puesto que un *MessageProducer* es inservible sin un mensaje que producir, un objeto *TextMessage* es creado por la llamada al método *createTextMessage()* del objeto Session. Este particular tipo de mensajes es un *TextMessage*, pero puedes observar en javax.jms.Message (es una superintefase) de la API JMS, ver la documentación para tener una descripción de los otros cuatro tipos de mensajes disponibles en JMS. El String a ser enviado es construido, colocado en el objeto *TextMessage*, y enviado al destino (*Queue*):

```
// Create a text message
TextMessage message = session.createTextMessage();
// Set the text of the message
message.setText("log entry, the time is: " + sdf.format(new Date()));
// Send the message
sender.send(message);
```

No se va a enviar ningún otro mensaje que el método con la anotación @Timeout sea llamado, así entonces se cierra la Connection, que cierra la Session y el *MessageProducer* antes creados. Todo esto sucede en el código localizado en el bloque *finally* del ejemplo:

```
finally {
	if (connection != null) {
		try {
			connection.close();
		}catch (Exception e) {}
	}
}
```

Ahora es momento para el JMS *provider* para enviar el mensaje a el *LogWriter queue* y al bean *MessageWriter*, que es un MDB. No existe ninguna interfaz remota pues no van a ser utilizados fuera del servidor de aplicación.

Como se muestra en el código de la clase *MessageWriterBean*, los MDBs deben ser anotados con el descriptor *@MessageDriven (javax.ejb.MessageDriven)* y además deben implementar la interface *MessageListener*. Note cómo la declaración *@MessageDriven* contiene la configuración de información relevante respecto al *LogWriter queue:*

```
@MessageDriven(activateConfig =
{
	@ActivationConfigProperty(propertyName="destinationType",
	propertyValue="javax.jms.Queue"),
	@ActivationConfigProperty(propertyName="destination",
	propertyValue="queue/LogWriter")
})
```

Note como la anotación array es usada para llenar el parámetro *activateConfig* de la anotación @*MessageDriven*. Los elementos de este array son descriptores que informan al contenedor EJB como ciertos parámetros del bean *MessageDriven* son configurados cuando el bean es invocado. La anotación @*ActivationConfigProperty (javax.ejb.ActivationConfigProperty)* que configura la propiedad de destino igual a queue/LogWriter es de particular interés, es como el contenedor usara esto para conectar el LogWriter queue especificado en el JMS de despliegue XML. La propiedad *destinationType* es usada por el contenedor para informar que tipo de objeto será para el destino.

A su vez, la interface *MessageListener* concede posesión del método *onMessage()*. Este método, llamado cada vez que el MDB recibe un mensaje, es responsable para enviar un mensaje de informacion los otros componentes.

El método *onMesage()* de este ejemplo esta a la expectativa de un *TextMessage* y usara el método *getText()* para obtener el String que fue enviado. Luego, de haber recibido el mensaje este se imprime por medio del *System.out*.