# *2*

# *Java y el mundo Web*

## 2.1 El protocolo HTTP

En la actualidad, difícilmente se imagina la vida sin el uso de internet: Comprar un boleto para un concierto, enviar correo electrónico, dejar un mensaje en el blog que seguimos, revisar el estado de cuenta bancario, son actividades cotidianas y transparentes para la persona promedio del siglo XXI. Sin embargo, detrás de estas existe un mundo virtual llamado *World Wide Web*, que está formado por una inmensa cantidad de equipos llamados servidores, que proveen recursos específicos a otros tantos, llamados clientes.

Para que la comunicación entre cliente y servidor se lleve a cabo, se requiere de un lenguaje que ambos puedan interpretar. Este lenguaje en común, HTTP (*HyperText Transfer Protocol*), fue definido por el W3C en 1990 y se trata de un protocolo de petición/respuesta entre cliente y servidor por medio de alguno de sus métodos.

La comunicación comienza cuando el cliente lanza una petición que será atendida por el servidor. El cliente recibe como respuesta los datos solicitados, así como un código que indica el procesamiento de dicha petición.

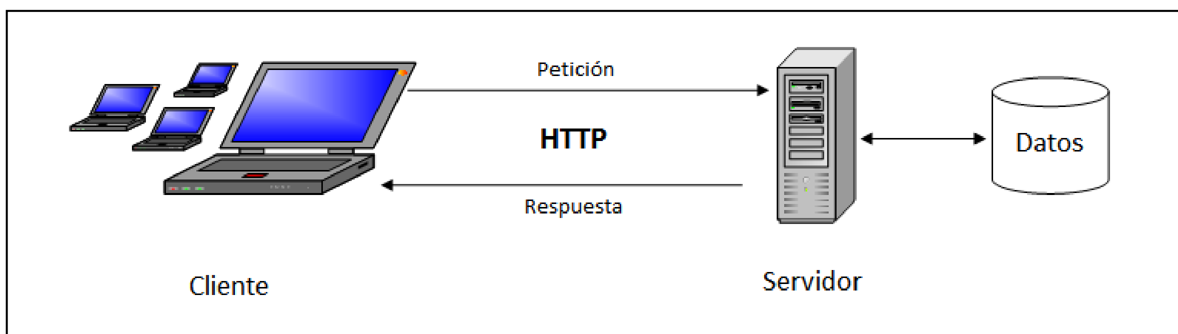


Figura 2.1: Funcionamiento básico del protocolo HTTP

### 2.1.1 Solicitud de la petición

Cada petición hecha por el cliente debe contener la información necesaria para poder ser tratada. Esto es, a qué servidor va dirigida y qué recurso es el que la atenderá. Dicha información se encuentra en la dirección URL:

[protocolo]://[servidor]:[puerto]/[recurso]

Ejemplos:

http://localhost:8084/Uniformes/paginasProduccion/Inicio.jsp

http://www.oracle.com/index.html

https[1]://boveda.banamex.com.mx/serban/index.htm

El servidor puede ser expresado por su nombre canónico, su dirección IP o el nombre del equipo en red. En caso de que se omita el puerto, se conecta al asignado por defecto según el protocolo utilizado. Puerto 80 para http y 443 para https.

### 2.1.2 Métodos de petición

Cada petición enviada al servidor debe ser atendida por alguno de los métodos explicados a continuación:

1. GET: Es el método que el navegador elige por default cuando se hace clic en algún enlace o se escribe directamente la URL. Generalmente es utilizado para *obtener* información de sólo lectura del servidor. Para encaminar la petición, la URL debe contener los parámetros que se desean enviar. Por ejemplo */recurso.jsp?param1=1¶m2=2*
2. POST: Empleado cuando se requiere enviar información, comúnmente proveniente de un formulario para actualizar los datos del servidor (en una base de datos). La información del cliente es enviada por medio del cuerpo de la petición y no por la URL como es el caso del método GET.
3. PUT: Se utiliza normalmente para almacenar un archivo en el servidor.
4. DELETE: Es la contraparte de PUT, y es utilizado para eliminar un recurso. En algunos servidores, tanto PUT como DELETE, no se encuentran disponibles debido a cuestiones de seguridad.
5. HEAD: Es muy similar al método GET, sin embargo, este sólo regresa las cabeceras de la respuesta y no su cuerpo.



1. https (*http secure*): Es una combinación del protocolo http con protocolos criptográficos para obtener peticiones más seguras.

6. OPTIONS: Devuelve al cliente los diferentes métodos HTTP que pueden ser ejecutados por el servidor. Por lo general: GET, POST y HEAD.
7. TRACE: Este método se utiliza para saber si existe el receptor del mensaje y usar dicha información para tareas de diagnóstico y depuración.

Por lo general, GET y POST son los métodos más empleados por el servidor y la elección de estos debe depender del tipo de petición que se haga, teniendo siempre en cuenta que con el uso de GET, los datos enviados al servidor serán visibles por medio de la URL, mientras que POST los esconde en el cuerpo de la petición haciendo que viajen de manera más segura.

## 2.2 La evolución de Java en el mundo web

La primera aportación que hizo Java al mundo web, en el año 1995, fue el uso de los applets, programas hechos bajo la tecnología Java, que pueden ser embebidos en una página HTML y ejecutados por la Máquina Virtual de Java (JVM por sus siglas en inglés) del navegador. Pese a tratarse de un recurso muy útil y novedoso, no estaba especificado propiamente para el desarrollo de aplicaciones web y se encontraba condicionado al soporte que cada navegador diera a esta tecnología.

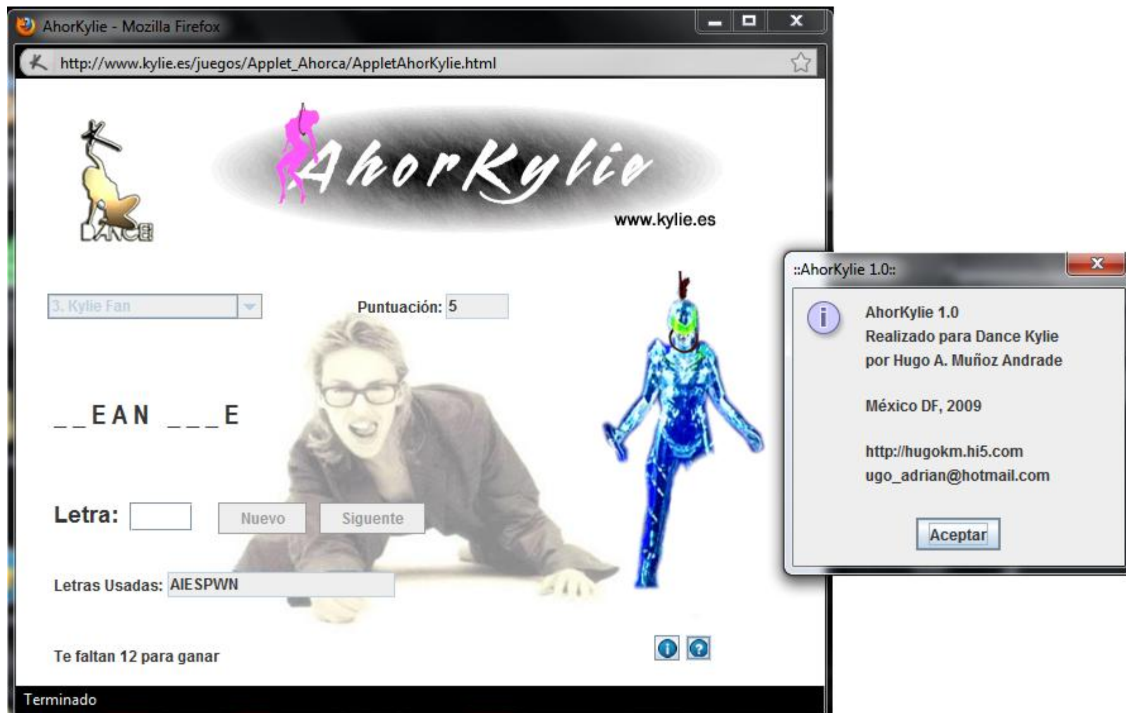


Figura 2.2: Juego hecho con la tecnología applet de Java y embebido en una página web

Es así como en 1997, Sun Microsystems, libera la primer especificación de los servlets como una alternativa de desarrollo web, cuyo mercado era mayormente dominado por lenguajes como PHP, ASP y Perl.

### 2.2.1 Servlets

Un servlet es una clase que tiene como función principal atender peticiones hechas por el navegador, procesarlas del lado del servidor y regresar una respuesta al cliente, la cuál puede ser un documento HTML, XML u otro formato como imágenes o datos binarios. Para que un servlet pueda funcionar, debe poder entender las peticiones, por lo que debe usar el protocolo HTTP.

#### 2.2.1.1 Funcionamiento de los Servlets

Un servlet está compuesto principalmente de 3 métodos: `init()`, `service()` y `destroy()`. Su ciclo de vida comienza cuando es instanciado y cargado en memoria por el contendedor[2] haciendo una llamada a su método principal `init()` tras la primer petición recibida para este. El método `init()` será ejecutado sólo una vez durante el ciclo de vida, por lo que existirá únicamente una copia en memoria de cada servlet. Por cada petición interceptada, el contenedor creará un hilo de ejecución, transparente para el programador, llamando al método `service()`, que se encargará de procesarla y enviar su respuesta al cliente. Finalmente, cuando el servlet termina, el contenedor ejecuta el método `destroy()` para limpiar la memoria.

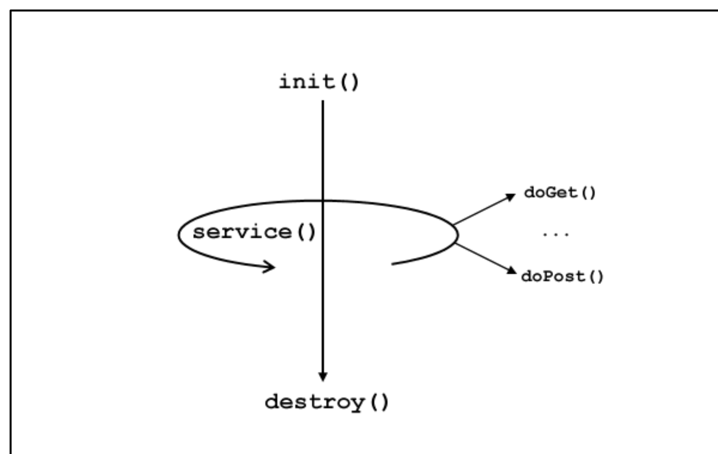


Figura 2.3: Funcionamiento del servlet.



2. El contenedor de servlets es el servidor encargado de ejecutar los servlets.

### 2.2.1.2 El API (*Application Programming Interface*) de los Servlets

Java tiene una extensa propuesta de clases e interfaces que forman el API completa de los servlets. Quizá la más importante sea la interface `Servlet`, que define los métodos que deben ser implementados por todos los servlets.
La implementación de aquellos cuyo protocolo sea independiente, se logra heredando a la clase `GenericServlet` por lo que el método `service()` debe ser sobreescrito para manejar apropiadamente la petición.

Por otro lado, los servlets que usen el protocolo HTTP deben heredar su funcionalidad a la clase `HttpServlet`, subclase de `GenericServlet`, pero con funciones específicas del protocolo, en cuyos métodos se encuentran definidos los 7 tipos de petición HTTP: `doGet()`, `doPost()`, `doDelete()`, `doHead()`, `doOptions()`, `doPut()` y `doTrace()`. En este caso, `service()` se encarga de llamar al método `doXxx()` apropiado, el cual debe ser sobreescrito para el manejo de la petición. Estos métodos, reciben objetos `HttpServletRequest` y `HttpServletResponse`, que contienen la petición del cliente y la respuesta del servidor, respectivamente.

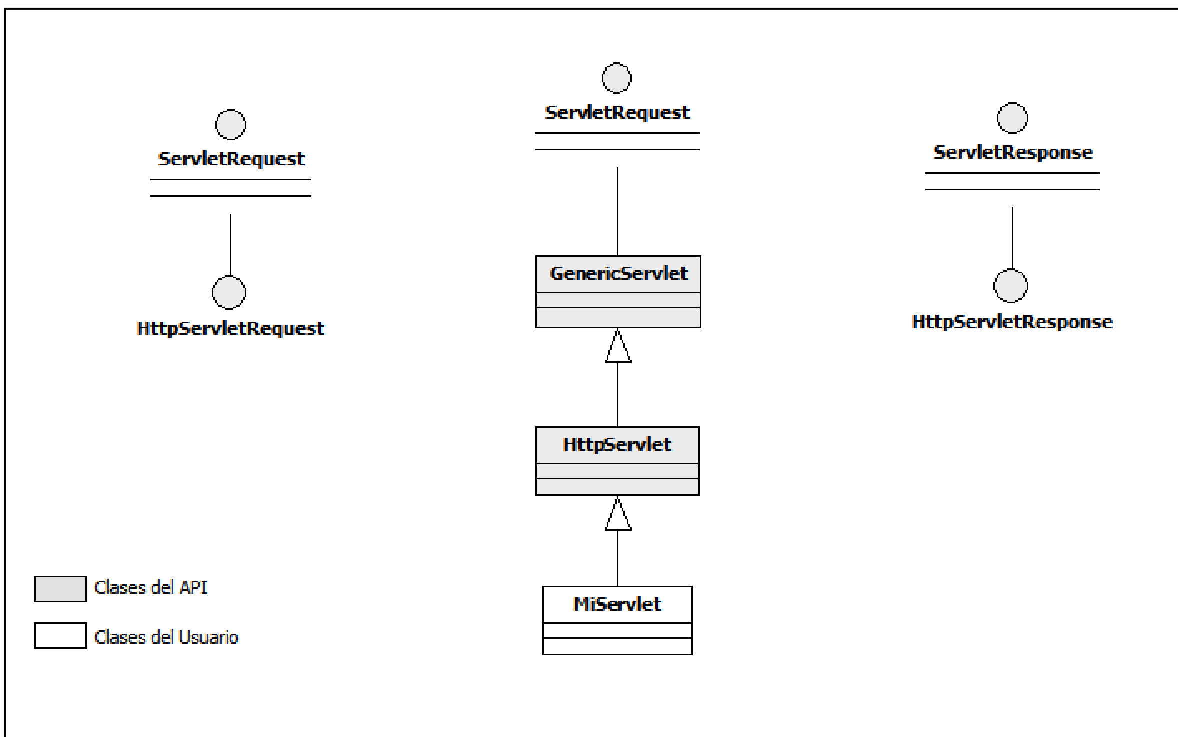


Figura 2.4: API general de los Servlets

En las siguientes tablas se muestra la descripción de los métodos principales de las clases e interfaces del API.

| Método | Retorno | Descripción |
|---|---|---|
| destroy() | void | Llamado por el contenedor para indicar que la ejecución del servlet ha terminado. |
| getServletConfig() | ServletConfig | Proporciona el acceso a la información de configuración del servlet. |
| getServletInfo() | String | Devuelve información del servlet, como el autor y la versión. |
| init(ServletConfig config) | void | Llamado por el contenedor para crear la instancia del servlet. |
| service(ServletRequest peticion, ServletResponse respuesta) | | Llamado por el contenedor para responder a una petición del cliente. |

Tabla 2.1: Métodos de la interface `Servlet`

| Método | Retorno | Descripción |
|---|---|---|
| getInitParameter(String nombre) | String | Regresa el nombre del ServletConfig de inicialización. |
| getInitParameterNames() | Enumeration | Regresa los nombres de los parámetros de inicialización. |
| getServletContext() | ServletContext | Regresa la referencia del ServletContext sobre el que el servlet está corriendo. |
| getServletName() | String | Regresa el nombre de la instancia del servlet. |
| log(String mensaje) | void | Escribe en una bitácora del servlet el mensaje indicado. |
| log(String mensaje, Throwable t) | | Escribe en una bitácora del servlet el mensaje de error en caso de excepción Throwable. |

Tabla 2.2: Métodos de la clase abstracta `GenericServlet` (implementa interface `Servlet`)

| Método | Retorno | Descripción |
|---|---|---|
| doXxx(HttpServletRequest pet, HttpServletResponse resp ) | void | Se llama en respuesta a una de los 7 tipos de petición de HTTP (siendo GET y POST los más comunes). |
| getLastModified(HttpServletRequest pet) | long | Regresa en milisegundos el tiempo en el que el objeto pet fue modificado por última vez. |

Tabla 2.3: Métodos de la clase abstracta `HttpServlet` (hereda a la clase abstracta `GenericServlet`)

| Método | Retorno | Descripción |
|---|---|---|
| getParameter(String nombre) | String | Obtiene el valor de un parámetro, con el nombre indicado, al servlet. |
| getParameterNames() | Enumeration | Devuelve el nombre de todos los parámetros asociados a la petición. |
| getParameterValues(String nombre) | String[] | Para un parámetro con múltiples valores, devuelve cada uno de estos. |
| getCookies() | Cookie[] | Devuelve un arreglo de objetos Cookie almacenados en el cliente por el servidor. |

Tabla 2.4: Algunos métodos de la interface `HttpServletRequest`

| Método | Retorno | Descripción |
|---|---|---|
| addCookie | void | Agrega un objeto Cookie al encabezado de la respuesta. |
| getOutputStream() | ServletOutputStream | Obtiene un flujo de salida en bytes para enviar datos binarios al cliente. |

Tabla 2.5: Algunos métodos de la interface `HttpServletResponse`

| Método | Retorno | Descripción |
|---|---|---|
| getWriter() | PrintWriter | Obtiene un flujo de salida en carácteres para enviar datos de texto al cliente. |
| setContentType(String tipo) | void | Especifica el tipo MIME[3] de la respuesta para mostrarlo adecuadamente en el navegador. |

Tabla 2.5 (*continuación*)

El servlet que se presenta a continuación recoge los valores de *nombre* y *edad* del cliente, el servidor lo procesa y envía como respuesta una página en formato HTML con los datos recogidos.

```
1   public class ServletNombreEdad extends HttpServlet {
2       public void doPost(HttpServletRequest request,
3               HttpServletResponse response)
4               throws ServletException, IOException {
5
6           PrintWriter out = response.getWriter();
7           response.setContentType("text/html");
8           String nombre = request.getParameter("nombre");
9           String edad = request.getParameter("edad");
10          out.println("<html>");
11          out.println("<head>");
12          out.println("<title>Servlet de Prueba</title>");
13          out.println("</head>");
14          out.println("<body>");
15          out.println("<br />");
16          out.println("<h1>"+nombre);
17          out.println("<br />usted tiene "+edad+" años </h1>");
18          out.println("</body>");
19          out.println("</html>");
20      }
21  }
```

Código 2.1: Servlet que procesa una petición del cliente



3. MIME (Multipurpose Internet Mail Extensions): Es un estándar que ayuda al navegador a comprender y manejar contenidos, por ejemplo: image/jpeg, text/html, video/mpeg.

En la línea 2 y 3 se indica que se trabajará con una petición de tipo POST y se lleva a cabo la sobreescritura del método `doPost()` recibiendo los objetos `request` y `response`. Se crea un objeto `PrintWriter` en la línea 6 para poder mandar caracteres al cliente y en la línea 7 se define el tipo MIME de la respuesta, en este caso un documento de texto HTML. El método `getParameter()` es el encargado de recoger los datos de la petición y en las líneas 8 y 9 se almacenan en su respectiva variable. Finalmente, de las líneas 10 a 19, se construye la respuesta que será enviada al cliente en forma de HTML.

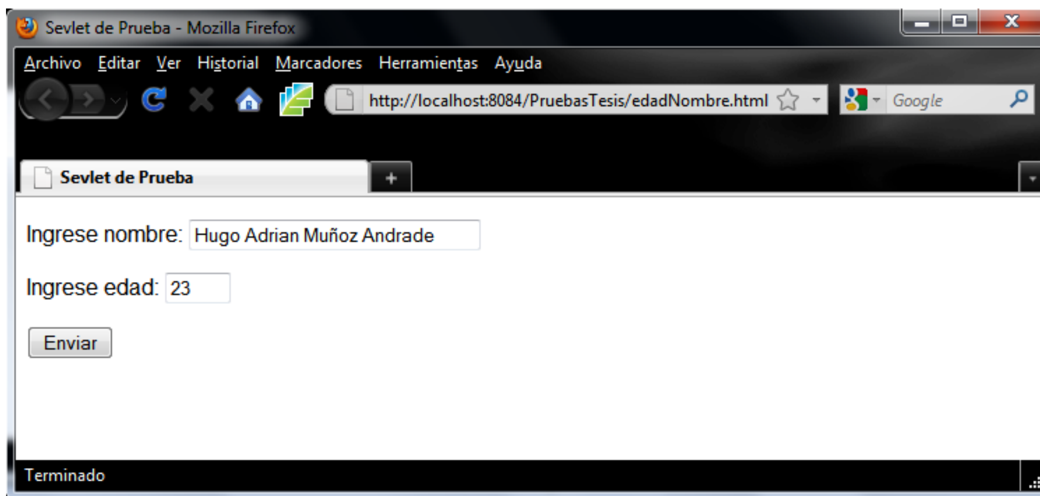


Figura 2.5: Petición de cliente

Figura 2.6: Respuesta del servidor

Esta primer solución de *Sun* para desarrollar aplicaciones web en Java no fue aceptada con facilidad debido al desconocimiento de la programación orientada a objetos y a la complicada manera de construir páginas sencillas por la inclusión de etiquetas HTML en el código del servlet. Las demás tecnologías disponibles, mencionadas con anterioridad, se encontraban basadas en soluciones tipo script y ofrecían un desarrollo más rápido aunque no necesariamente eficiente. Dada esta situación, se introduce JSP (*Java Server Pages*) que facilitaría la creación de contenido dinámico del lado del servidor.

## 2.2.2 Java Server Pages (JSP)

En sus inicios fue fuertemente criticado por los programadores debido a las similitudes que presentaba con otros lenguajes que hacían uso de scripts, y aunque a primera vista así parecía, JSP era más que eso, se trataba de una extensión de la tecnología servlet. El contenedor transforma cada JSP en un servlet que compila y procesa cada petición recibida.

### 2.2.2.1 El uso de scriptlets

Aunque el trabajo se reducía considerablemente, la solución que JSP daba a los problemas tenía ciertas debilidades, como piezas de código Java embebido en forma de *scriptlets*, que provocaba desorden, complejidad en la JSP y no se promovía su reutilización. A continuación se muestra una JSP que resuelve el problema previamente hecho en el servlet:

```
1   <html>
2       <head>
3           <title>JSP Prueba con Sriptlets</title>
4       </head>
5       <body>
6           <%
7               String nombre = request.getParameter("nombre");
8               String edad = request.getParameter("edad");
9           %>
10           <h1>Hola <%=nombre%><br />
11               usted tiene <%=edad%> años
12           </h1>
13       </body>
14   </html>
```

Código 2.2: JSP que procesa la petición, usando scriptlets. El elemento <% %> permite la inclusión de código Java. Mientras que <%= %> inserta el valor de una variable.

En 1999 aparecen las etiquetas personalizadas que ofrecen un código más limpio, comprensible y reutilizable siguiendo la sintaxis de HTML, pero no es sino hasta el año 2002, en el que son introducidas como parte de la especificación de Sun para JSP, cuando comienza el auge de estas etiquetas.

#### 2.2.2.2 JSTL (*JSP Standard Tag Library*) y el lenguaje EL (*Expression Language*)

La librería JSTL, creada por la *fundación Apache* y *Sun,* introduce el uso de etiquetas personalizadas que permiten el control de flujo de datos, evitando con esto, la inclusión de código Java en la JSP. De esta manera, se favorecía el aprendizaje de dicha tecnología, pues aunque no se tuviera el conocimiento de la programación orientada a objetos o del propio lenguaje, usando las etiquetas personalizadas de JSTL se aprovechaban las ventajas que Java ofrecía.

El lenguaje EL surgió como parte de la especificación JSTL para acceder de manera directa a los datos de petición y a las propiedades de los objetos. Su sintaxis básica es: ${objeto.propiedad}.

Las principales librerías que ofrece JSTL se muestran en la siguiente tabla:

| Función JSTL | Prefijo | Descripción |
|---|---|---|
| Core | c | Se encarga de las condiciones, bucles, asignación de variables y redirecciones. Algunas etiquetas son: <c:choose>, <c:if>, <c:redirect>, <c:set>, <c:forEach>. |
| Format | fmt | Útil para dar formato a la información usando los patrones dados. Algunas etiquetas son: <fmt:formatDate> y <fmt:formatNumber> |
| SQL | sql | Maneja la conectividad con la base de datos. Algunas etiquetas son: <sql:query>, <sql:transaction>, <sql:update>. |
| Functions | fn | Se trata de un conjunto de funciones útiles para el acceso de datos con EL. Algunas de estas son: fn:containts(cadena, subc), fn:endsWith(cadena,sufijo), fn:strartsWith(cadena,prefijo), fn:indexOf(cadena, sub), fn:length(cadena), fm:trim(cadena) |

Tabla 2.6: Funciones de JTSL. Ver apéndice A para mayor información.

El siguiente código muestra una JSP que resuelve el problema de los ejemplos anteriores usando la librería JSTL y el lenguaje EL:

```
1   <html>
2       <head>
3           <title>JSP Prueba con JSTL y EL</title>
4       </head>
5       <body>
6           <h1>Hola ${param.nombre}<br />
7               usted tiene ${param.edad} años
8           </h1>
9       </body>
10  </html>
```

Código 2.3: JSP que procesa la petición, usando la librería JSTL y lenguaje EL. El objeto param es una colección de los parámetros de la petición.

Tras esta evolución, es como los desarrolladores web comenzaron a considerar JSP como una tecnología eficaz, ágil, de fácil aplicación y potente, pues seguía conservando su metodología orientada a objetos así como el uso de los servlets, aunque ya de manera transparente para el programador. No obstante seguía habiendo un problema: El código de la lógica de negocio y de acceso a datos estaba mezclado con el de presentación en la JSP, resultando en sistemas cuyo mantenimiento era difícil y no siempre resultaba apropiado. Ante esta complicación, se vio la necesidad de separar en capas la aplicación, usando un modelo que permitiera gestionar de manera independiente el acceso a datos, la lógica de negocio y la presentación de la información al cliente.

## 2.3 El patrón MVC (Model, View, Controller)

MVC es un patrón de diseño en el que se separan los componentes de una aplicación (generalmente web) de acuerdo a su funcionalidad:

- Modelo: Esta capa encapsula la lógica de negocio y el acceso a datos.
- Vista: Es la representación gráfica de los datos generados en el modelo para ser enviados al usuario.

- Controlador: Dirige las peticiones del cliente hacia el modelo para su procesamiento y espera su respuesta para ser enviada a la vista.

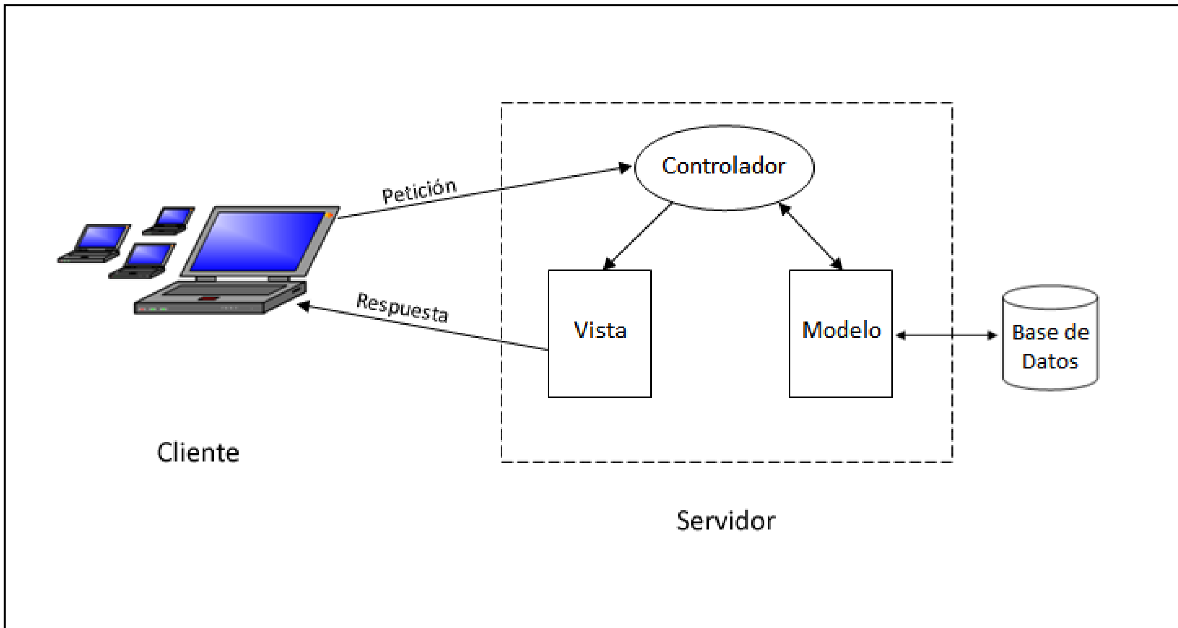


Figura 2.7: Funcionamiento básico del patrón MVC

El uso de este patrón facilita el mantenimiento, debido a que los componentes del sistema se encuentran muy bien ubicados, y por su desacoplamiento, la modificación de uno de estos no afecta al funcionamiento de los demás. A su vez, promueve la reutilización de código y aumenta la escalabilidad.

En este punto se ve con claridad cómo se pueden combinar la simplicidad que ofrece JSP para crear contenido HTML dinámico y la potencia que brinda la gestión de peticiones HTTP por parte de los servlets, para crear aplicaciones web rápidas, robustas y escalables.

A pesar de que las especificaciones de Java permiten un desarrollo ágil de sistemas web, para la construcción de esta aplicación serán utilizados tres frameworks, que en conjunto, simplifican la implementación del patrón MVC: Struts, Spring e Hibernate.

## 2.4 Struts

Struts es el pionero de los frameworks que implementan MVC. Su historia comienza en mayo del 2000 cuando Craig McClanahan lanza el proyecto de crear un estándar MVC basado en Java, en 2001, año en que se libera la versión 1.0, fue adoptado por buena parte de los desarrolladores a nivel mundial, siendo hasta ahora el líder de los frameworks MVC de Java. Sus beneficios directos se ven reflejados sólo en las capas de la *Vista* y el *Controlador*.

### 2.4.1 Componentes y funcionamiento de Struts

Struts requiere de un archivo de configuración llamado `struts-config.xml` en el que se registran los elementos del API contenidos en la aplicación.

#### 2.4.1.1 API de Struts

– `ActionServlet`: Este servlet actúa como el cerebro de la aplicación pues se encarga de interceptar todas las peticiones recibidas del cliente delegando su tratamiento a un objeto del tipo `RequestProcessor` responsable de encaminar la petición para su respetivo proceso.

– `Action`: Si `ActionServlet` es el cerebro, la clase `Action` puede ser considerada como el corazón, pues sirve como puente de enlace entre la petición del cliente y el Modelo. Es aquí donde se hacen las llamadas a la lógica de negocio y la transferencia de datos a la Vista (generalmente páginas JSP).

– `ActionForm`: Se trata de JavaBeans que contienen los datos de captura del cliente (usualmente procedentes de formularios) y únicamente sirve para el transporte de datos, no deben ser usados por la lógica u otras capas de la aplicación.

– `ActionMapping`: Asocia una petición con algún objeto `Action` que la procesará. Contiene la información de la URL que provoca la ejecución del `Action`, así como las posibles vistas a las que serán enviados los datos de la respuesta.

– ActionForward: Se trata de objeto que contiene la dirección lógica de algún recurso al que será direccionado el usuario una vez que el Action haya terminado de ejecutarse. Esto trae una ventaja de mantenimiento, pues cuando una URL cambie por motivos de actualización, no se tendrá que modificar cada referencia con su ubicación física desde código, sino simplemente modificar en el archivo struts-config.xml dicha referencia.

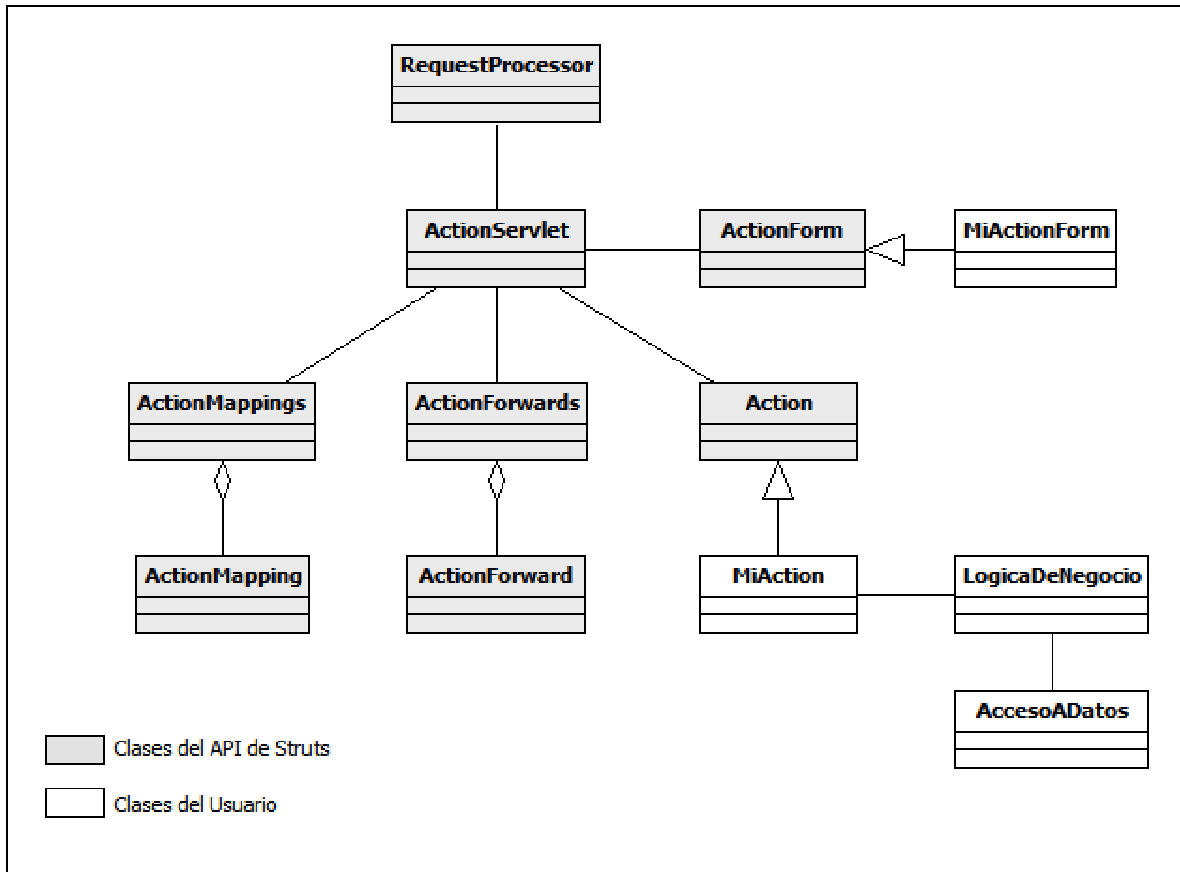


Figura 2.8: API general de Struts

### 2.4.1.2 Ciclo de vida de una petición en Struts

Como convención, para que una petición sea interceptada por ActionServlet, el recurso de la URL debe contar con la terminación .do (por ejemplo http://localhost:8084/Uniformes/verEmpleados.do), cuando esto sucede por primera vez, el contenedor instancia un servlet del tipo ActionServlet y la petición es enviada a un objeto RequestProcessor, el cual compara la terminación de la URL (ignorando el .do) con la información de los ActionMapping configurados en struts-config.xml para determinar

la subclase `Action` que debe encargarse de la petición e instanciar un objeto `ActionForm` rellenándolo con los datos del cliente. Una vez que se encuentra y ejecuta el objeto `Action`, se llama a su método principal, `execute()`, en el que se encuentran las llamadas a las diferentes clases del Modelo que se encargarán de manejar dicha petición. Cuando se obtiene la respuesta dada, `Action` devuelve un objeto `ActionForward` al `ActionServlet` para buscar la dirección lógica del recurso en el archivo de configuración y finalmente ser direccionado.

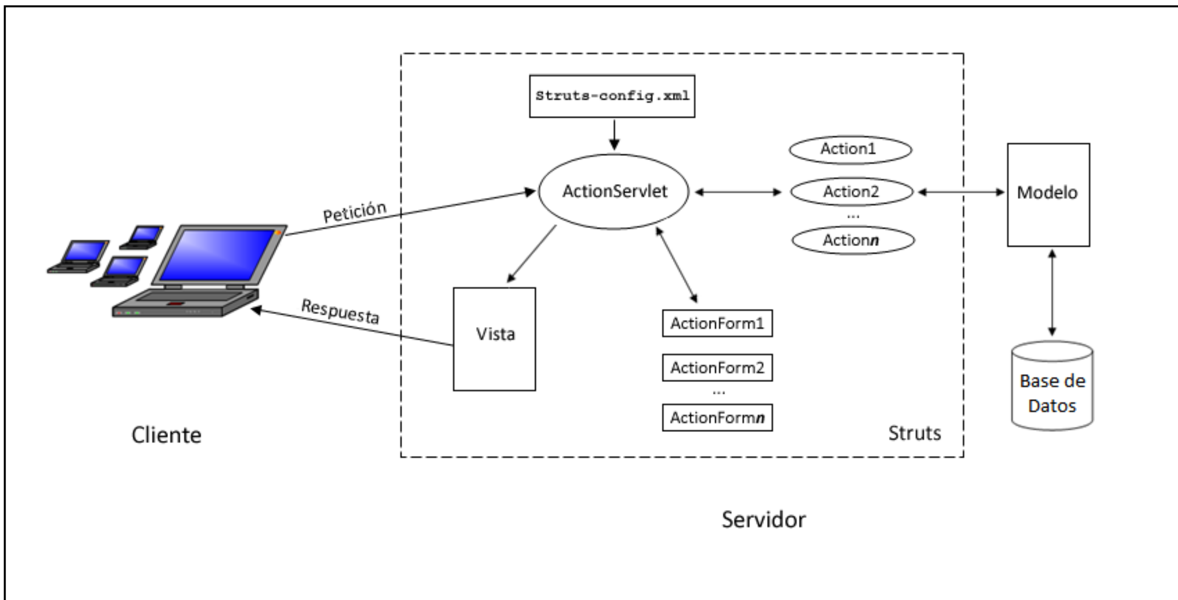


Figura 2.9: Funcionamiento de Struts

En las siguientes tablas se especifican los métodos principales de las clases fundamentales de Struts:

| **Método** | **Retorno** | **Descripción** |
|---|---|---|
| execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response) | ActionForward | Procesa la petición HTTP y envía la respuesta a un recurso lógico. |

Tabla 2.7: Método principal de la clase Action

| Método | Retorno | Descripción |
|---|---|---|
| findForward(String nombre) | ActionForward | Devuelve el objeto ActionForward cuya dirección lógica se específica en el nombre que recibe. |
| getInputForward() | | Devuelve un ActionForward que corresponde al input (definido en struts-config.xml) de la Action. |

Tabla 2.8: Métodos de la clase `ActionMapping`

### 2.4.1 Configuración de Struts

Como ya se mencionó, para que los elementos que conforman Struts puedan ser utilizados, deben estar debidamente registrados y configurados en `struts-config.xml`, cuya estructura básica se muestra a continuación:

```
1   <?xml version="1.0" encoding="UTF-8" ?>
2   <!DOCTYPE struts-config PUBLIC
3   "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
4   "http://jakarta.apache.org/struts/dtds/struts-config_1_3.dtd">
5   <struts-config>
6        <form-beans>
7           <!– Declaración de ActionForm -->
8        </form-beans>
9
10       <global-exceptions>
11          <!– Declaración de excepciones (de ser requeridas)  -->
12       </global-exceptions>
13
14       <global-forwards>
15          <!– Declaración de forward globales (de ser requeridos) -->
16       </global-forwards>
17
18       <action-mappings>
19           <!– Declaración de ActionMapping -->
20       </action-mappings>
21   </struts-config>
```

Código 2.4: Estructura básica del archivo de configuración de Struts

#### 2.4.1.1 Configuración de **ActionForm**

```
1    <form-beans>
2        <form-bean name="ProduccionForm" type="ActionForm.ProduccionForm"/>
3        <form-bean name="UniformeForm" type="ActionForm.UniformeForm"/>
4        <form-bean name="PedidoForm" type="ActionForm.PedidoForm"/>
5                                      …
6    </form-beans>
```

Código 2.5: Registro de objetos ActionForm en `struts-config.xml`

En la etiqueta `<form-beans>` de la línea 1 se declara la lista de objetos `ActionForm` que serán utilizados a lo largo de la aplicación. Su declaración individual se lleva a cabo con la etiqueta `<form-bean>` que tiene como atributos principales:

- `name`: Es un identificador único del bean que será usado como referencia por Struts.
- `type`: Se trata del nombre calificado de la clase.

#### 2.4.1.2 Configuración de **ActionMapping – Action – ActionForward**

```
1    <action-mappings>
2        <action name="UniformeForm" path="/UniformeAction" type="Action.UniformeAction"
3         scope="request">
4            <forward name="verUniformes" path="/verUniformes2.jsp"/>
5        </action>
6        <action name="PedidoForm" path="/PedidoAction" type="Action.PedidoAction"
7         scope="request">
8           <forward name="cambiosP" path="/verPedido.jsp"/>
9           <forward name="cambioPedido" path="/PedidoAction.do?accion=nuevoPedido"/>
10           <forward name="nuevoPedido" path="/verPedido2.jsp"/>
11       </action>
12    </action-mappings>
```

Código 2.6: Registro de objetos `Action`, `ActionMapping` y `ActionForward` en `struts-config.xml`

En la línea 1, con la etiqueta `<action-mappings>` se declaran todos los `ActionMapping` que serán utilizados. Dentro de este, se deben definir los objetos `Action` implementados con la etiqueta `<action>` cuyos atributos principales son:

- `name`: El nombre del objeto `ActionForm` que estará asociado con este `Action`, debe coincidir con el identificador `name` de la etiqueta `<form-bean>`
- `path`: Se trata en realidad del nombre del `Action` que provoca la captura de la petición por el `ActionServlet` y está ubicado entre el último carácter **/** y antes de la extensión `.do` de la URL (/PedidoAction.do)
- `type`: Es el nombre calificado de la clase.
- `scope`: Se trata del alcance que tendrán las variables del `ActionForm` en la apliación, sus posibles valores son: `session` (ámbito de sesión), `request` (ámbito de petición) y `application` (ámbito de aplicación). De no ser especificado, `session` es el valor por defecto.

Para cada objeto `Action`, se deben especificar los posibles `Forward`, a los que será direccionado después de su ejecución, con la etiqueta `<forward>` que tiene como elementos principales:

- `name`: Es el nombre lógico del recurso al que es enviado el usuario.
- `path`: Contiene la URL relativa de dicho recurso (puede ser un documento JSP, HTML o bien otro objeto `Action`) .

### 2.4.1.3 Configuración de `ActionServlet` en `web.xml`

Este archivo contiene las definiciones necesarias para la aplicación, servlets, algunos mapeos y en ocasiones parámetros de conexión a bases de datos.

```
1      <servlet>
2           <servlet-name>action</servlet-name>
3           <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
4              <init-param>
5                    <param-name>config</param-name>
6                    <param-value>/WEB-INF/struts-config.xml</param-value>
7             </init-param>
8     </servlet>
```

Código 2.7: Registro de `struts-config.xml` en el descriptor de despliegue `web.xml`

```
9     <servlet-mapping>
10              <servlet-name>action</servlet-name>
11              <url-pattern>*.do</url-pattern>
12     </servlet-mapping>
```

Código 2.7 (*continuación*)

De la línea 1 a 8 se lleva a cabo la definición del `ActionServlet` con la etiqueta `<servlet>`, en el que se especifica el nombre y clase del Servlet (líneas 2 y 3), así como la dirección relativa del archivo de configuración `struts-config.xml` en el parámetro de inicialización `config` (líneas 4 a 7).

También es importante definir el tipo de extensión que debe tener toda URL para ser atendida por Struts. De las líneas 9 a 12, se indica mediante el mapeo, que toda URL con terminación `.do` (por convención) sea atendida por el `ServletAction`, previamente configurado.

### 2.4.2 Librerías de Acciones JSP

Hasta ahora, sólo se ha hablado de lo que Struts hace con el *Controlador*, sin embargo, como se mencionó antes, Struts trabaja también para la *Vista*, y es aquí donde ofrece una serie de librerías de acciones para ser utilizadas en JSP y facilitar la manipulación de los datos obtenidos desde el servidor.

#### 2.4.2.1 Librería Bean

Esta librería es útil para acceder a las propiedades de los JavaBeans usados en la aplicación, así como la creación de nuevos a partir de datos disponibles en los diferentes ámbitos.

Para ser utilizada en una página JSP, debe ser incluida mediante la directiva

```
<%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
```

Sus acciones principales son las siguientes:

**`<bean:cookie>`**

Almacena el valor de una cookie en una variable JSP. Sus atributos principales son:

- `id`: Especifica el nombre de la variable JSP que se creará.
- `name`: Es el nombre de la cookie cuyo valor será almacenado en la variable JSP.

– `value`: En caso de que la cookie no sea encontrada, se crea una con el valor indicado en este atributo.

**<bean:define>**

Almacena el valor de un objeto existente en la aplicación en una variable JSP. Sus principales atributos son:

– `id`: Especifica el nombre de la variable JSP que se creará.
– `name`: Es el nombre del objeto cuyo contenido será almacenado en la variable JSP.
– `property`: Indica el atributo del objeto especificado en `name`, para que sea este el que se almacene en la variable JSP.
– `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.
– `toScope`: Define el ámbito que tendrá la nueva variable.
– `type`: Indica el tipo de dato de la variable creada (ejemplo java.lang.Integer) el valor por defecto es String.

**<bean:message>**

Muestra un mensaje almacenado en el archivo `ApplicationResource.properties`. Este es un archivo de texto plano y es usado por Struts para guardar parejas del tipo *clave = valor* y argumentos opcionales (*{0}…{4}*), útil para mostrar mensajes y no modificarlos desde código, también ayuda a la internacionalización de la aplicación mostrando dichos mensajes en diferentes idiomas. Los atributos principales de esta etiqueta son:

- `key`: Clave asociada en el archivo de propiedades al mensaje de texto.
- `arg0`, `arg1`, `arg2`, `arg3`, `arg4`: Indica los argumentos que deben ser mostrado en el mensaje, en sustitución de {0},{1},{2},{3} y {4}.

**<bean:write>**

Obtiene el valor de un objeto y lo muestra en la página. Sus principales atributos son:

– `name`: Es el nombre del objeto cuyo contenido será mostrado.
– `property`: Indica el atributo del objeto especificado en `name`, para que este se muestre en la página.

- `format`: Especifica el formato en que será mostrado el objeto.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.

#### 2.4.2.2 Librería Logic

Es en esta librería donde se definen las diferentes maneras de manejar el control de flujo de datos recibidos por el servidor.

Para ser utilizada en una página JSP, debe ser incluida mediante la directiva

```
<%@taglib uri="http://struts.apache.org/tags-logic" prefix="logic"%>
```

Sus acciones principales se muestras a continuación:

**<logic:empty> y <logic:notEmpty>**

La primera evalúa si un objeto se encuentra vacío, su contraparte es la etiqueta `notEmpty`, que buscará que un objeto contenga elementos. Sus atributos principales son:

- `name`: Es el nombre del objeto que será evaluado.
- `property`: Indica el atributo del objeto especificado en `name`, para su evaluación.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.

**<logic:equal> y <logic:notEqual>**

La primer etiqueta evalúa que el contenido de un objeto sea igual al valor de especificado. Como contraparte está la etiqueta `<logic:notEqual>` encargada de evaluar que el contenido del objeto no sea igual al valor especificado. Sus atributos principales son:

- `cookie`: Nombre de la cookie cuyo contenido será evaluado.
- `name`: Es el nombre del objeto cuyo contenido será evaluado.
- `property`: Indica el atributo del objeto especificado en `name`, para que este sea evaluado.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.
- `value`: Especifica el valor a comparar con el objeto y cookie dados.

**<logic:greaterEqual> <logic:greaterThan> y**
**<logic:lessEqual> <logic:lessThan>**

El propósito de estas etiquetas, al igual que en las 2 anteriores, es evaluar el contenido de un objeto o una cookie, indicando si es *mayor o igual*, *mayor que*, *menor o igual* o *menor que* el valor especificado en `value`. Sus atributos principales son los mismos que en las etiquetas anteriores.

**<logic:iterate>**

Se encarga de recorrer una colección según las condiciones dadas. Sus principales atributos son:

- `name`: Es el nombre de la colección que será iterada.
- `property`: Indica el atributo del objeto especificado en `name`, que contiene la colección.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.
- `id`: Nombre de la variable JSP que almacenará la referencia del objeto actual en la iteración. Su ámbito estará limitado a la página.
- `indexId`: Nombre de la variable JSP que almacenará el número de iteración actual. Su ámbito estará limitado a la página.
- `length`: Limita el número de iteraciones en la colección.
- `offset`: Especifica el índice del cual debe comenzar la iteración.

Struts no ofrece un soporte directo para la capa del Modelo, el cual es conveniente que se divida en 2 subcapas: La lógica de negocio, que contendrá todas aquellas clases que encaminen a las peticiones de los objetos `Action` hacia la subcapa de acceso a datos responsable de obtener la información solicitada desde una fuente de datos.

## 2.5 JDBC (*Java DataBase Conectivity*)

JDBC es el API estándar proporcionado por Sun para las aplicaciones Java, que se encarga de la manipulación de datos por medio de un conjunto de clases e interfaces que permiten, una vez implementadas, llevar a cabo la comunicación entre una base de datos relacional y la aplicación.

La mayoría de los manejadores de bases de datos ofrecen su propia implementación de este API (llamados *drivers* o *controladores*), que por estar basado en interfaces, define los métodos, los tipos de objetos que recibe y los tipos de valores u objetos que debe regresar. Esto tiene como ventaja fundamental que la aplicación pueda ser portable en cuanto a manejador de base de datos se refiere. Así, si la aplicación lo llegara a requerir, los datos pueden ser migrados a un RDBMS (*Relational Database Magement System*) distinto sin que el código deba ser alterado, tan sólo bastaría con modificar la configuración y el driver del manejador específico.

### 2.5.1 API de JDBC

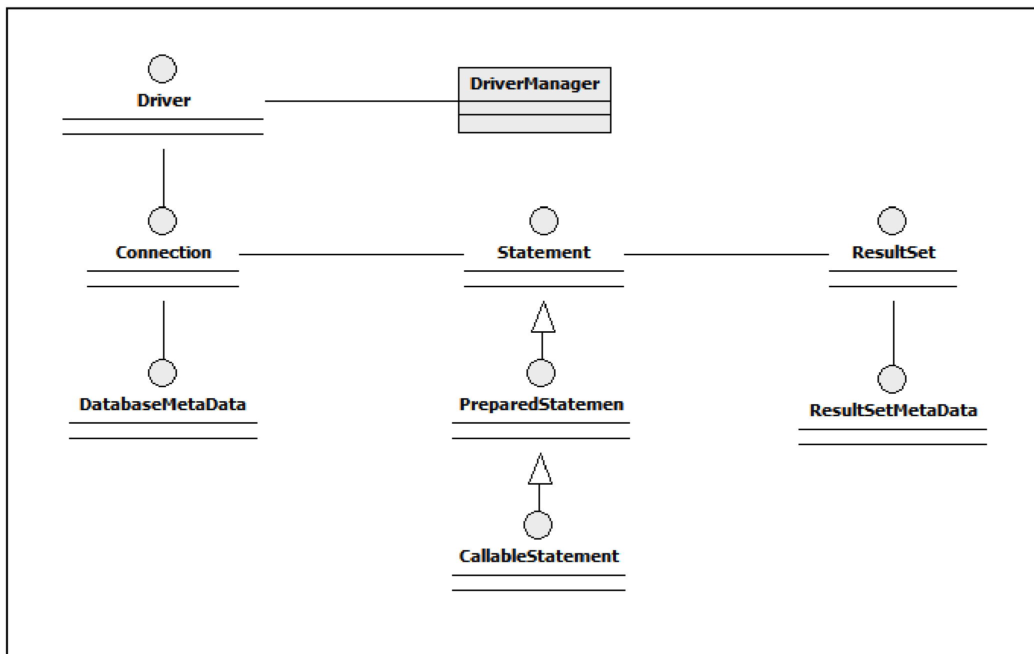


Figura 2.10: API general de JDBC

Las componentes principales del API se enlistan a continuación:

- `DriverManager`: Es la clase responsable de cargar el driver indicado y manejar las distintas conexiones con la base de datos.
- `Driver`: Es la interface que cada controlador debe implementar para ser llamada por el `DriverManager`.
- `Connection`: Se encarga de administrar la sesión entre la base de datos y la aplicación.
- `Statement`: Permite ejecutar sentencias SQL. Cuenta con 2 subclases: `PreparedStatement` y `CallableStatement`. La primera representa una sentencia SQL precompilada que puede ser ejecutada `n` veces con parámetros diferentes, útil para aumentar el rendimiento de sentencias `insert` múltiples. La segunda permite la manipulación de datos por medio de procedimientos almacenados.
- `DatabaseMetaData`: Encapsula la información relacionada con la base de datos, el driver y los metadatos.
- `Resultset`: Contiene las filas y columnas que regresa la base de datos tras la ejecución de la sentencia SQL.
- `ResultSetMetaData`: Contiene los metadatos del objeto `ResultSet`.

Las siguientes tablas se muestra el detalle de las clases e interfaces que componen el API de JDBC.

| **Método** | **Retorno** | **Descripción** |
|---|---|---|
| getConnection(String url) | Connection | Intenta establecer una conexión a la base de datos con la URL y parámetros dados. |
| getConnection(String url, String usuario, String password) | | |
| getConnection(String url, Properties info) | | |

Tabla 2.9: Métodos principales de la clase `DriverManager`

| Método | Retorno | Descripción |
|---|---|---|
| close() | void | Libera los recursos ocupados para la conexión. |
| commit() | | Guarda los cambios hechos, desde el último commit/rollback, de manera permanente en la base de datos. |
| createStatement() | Statement | Crea un objeto `Statement` para la ejecución de sentencias SQL |
| getMetaData() | DatabaseMetaData | Obtiene los metadatos de la conexión actual. |
| prepareStatement(String sql) | PrepareStatement | Crea un objeto `PrepareStatement` para la ejecución de sentencias SQL parametrizadas. |
| rollback() | void | Deshace todos los cambios hechos desde el último commit/rollback en la base de datos. |

Tabla 2.10: Métodos principales de la Interface `Connection`

| Método | Retorno | Descripción |
|---|---|---|
| close() | void | Libera los recursos ocupados por el objeto `Statement`. |
| execute(String sql) | boolean | Ejecuta una sentencia SQL que podría regresar múltiples valores. |
| executeQuery(String sql) | ResultSet | Ejecuta una sentencia SQL que regresa un objeto `ResultSet`. |
| executeUpdate(String sql) | int | Ejecuta una sentencia INSERT, DELETE o UPDATE. |
| setMaxRows(int max) | void | Fija el número máximo de filas que podrá tener el `ResultSet`. |

Tabla 2.11: Métodos principales de la Interface `Statement`

| Método | Retorno | Descripción |
|---|---|---|
| setTipo(int índice, Tipo valor)<br>*Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).* | Tipo | Determina el tipo de dato de la columna indicada por el índice. |

Tabla 2.12: Métodos principales de la Interface `PreparedStatement`

| Método | Retorno | Descripción |
|---|---|---|
| close() | void | Libera los recursos ocupados por el objeto `ResultSet`. |
| first() | boolean | Mueve el puntero a la primer fila del `ResultSet`. |
| getTipo(int índice)<br>*Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).* | Tipo | Regresa el contenido de la columna indicada por el índice, según sea el tipo de dato, en la fila actual del `ResultSet`. |
| last() | boolean | Mueve el puntero a la última fila del `ResultSet`. |
| next() | | Mueve el puntero a la siguiente fila de la actual en el `ResultSet`. |
| previous() | | Mueve el puntero a la fila anterior de la actual en el `ResultSet`. |
| relative(int n) | | Mueve el puntero `n` filas anteriores o posteriores a la actual el `ResultSet`. |

Tabla 2.13: Métodos principales de la Interface `ResultSet`

En el siguiente código se muestra cómo se realiza la conexión a una base de datos por medio de los componentes del API:

```
1   public class AccesoConJDBC {
2       Connection conexion = null;
3       Statement consulta = null;
4       PreparedStatement insert = null;
5       ResultSet resultados = null;
6       static final String Driver = "oracle.jdbc.OracleDriver";
7       static final String URL = "jdbc:oracle:thin:@127.0.0.1:1521:XE";
8       static final String usuario = "hugo";
9      static final String pass = "hugo";
10
11      public List<Proveedores> mostrarProveedores() {
12          List<Proveedores> lista = new ArrayList<Proveedores>();
13          try {
14              Class.forName(Driver);
15              conexion = DriverManager.getConnection(URL, usuario, pass);
16
17              consulta = conexion.createStatement();
18              resultados = consulta.executeQuery("SELECT idProveedor" +
19                      ",nombre,inicioactividad,finActividad," +
20                      "status FROM proveedores");
21              while (resultados.next()) {
22                  Proveedores proveedor = new Proveedores();
23                  proveedor.setIdproveedor(resultados.getInt(1));
24                  proveedor.setNombre(resultados.getString(2));
25                  proveedor.setInicioactividad(resultados.getDate(3));
26                  proveedor.setFinactividad(resultados.getDate(4));
27                  proveedor.setStatus(resultados.getString(5));
28                  lista.add(proveedor);
29              }
30          } catch (SQLException e) {
31              System.out.println("Se reportó una excepción: " + e);
32          } catch (ClassNotFoundException e) {
33              System.out.println("No se encontró el driver: " + e);
34          } finally {
35              try {
36                  consulta.close();
37                  resultados.close();
38                  conexion.close();
39              } catch (SQLException e) {
40                  System.out.println("Error al liberar recursos: " + e);
41              }
42          }
43          return lista;
44      }}
```

Código 2.8: Conexión y consulta a una base de datos por medio de JDBC

De las líneas 4 a 11 se declaran los distintos objetos utilizados para la manipulación de datos: `Connection`, `Statement`, `PreparedStatement` y `ResultSet`. Así como los parámetros requeridos por JDBC:

- La cadena que contiene la clase cualificada del Controlador (que implementa la interface Driver).
- La cadena que contiene la URL de la base de datos (siguiendo el formato [protocolo de comunicación]:[subprotocolo]:identificador propio de cada RDBMS).
- Nombre y contraseña del usuario con los permisos suficientes para acceder a los datos.

En la línea 14 se carga el driver y se inicia la conexión por medio del `DriverManager` con los parámetros previamente declarados. El objeto `Connection` hace referencia a un objeto `Statement` para poder ejecutar sentencias SQL en la línea 18 y se obtiene el objeto `ResultSet` tras la ejecución en la línea 18.

Para que los resultados puedan ser manejados se debe iterar sobre el `ResultSet`, y en este caso, de la línea 22 a la 27 los valores de cada fila se recuperan y almacenan (con los diferentes métodos que ofrece `ResultSet` para cada tipo de dato) en un objeto de tipo `Proveedores` que son agregados a una lista que contiene dichos objetos en la línea 28.

Finalmente de la línea 30 a la 33 se reportan las excepciones que puedan existir tanto por el Driver como por alguno de los elementos involucrados en la conexión y se liberan los recursos utilizados de la línea 36 a la 40.

Pese a las ventajas de portabilidad que ofrece JDBC, y tal como se observa en el ejemplo anterior, el código que se necesita escribir para la manipulación de datos es demasiado extenso, a pesar de que el problema queda resuelto en 3 líneas (18 a 20). Se estima que el 30% del código escrito en una aplicación se ocupa del problema de acceso a datos, desperdiciando con esto muchos de los recursos humanos que deberían ser invertidos en el problema principal: La lógica de negocio.

Otro problema de JDBC es que los objetos no pueden ser persistidos directamente de (o hacia) la base de datos sino que deben ser manejados atributo por atributo (líneas 23 a 27), lo cual se complica cuando la tabla contiene un número significativo de columnas.

## 2.6 Hibernate

Aquel tipo de problemas a los que se enfrentaban los programadores que usaban JDBC sirvieron como antecedente para el nacimiento de un nuevo paradigma. El paradigma de persistencia en objetos que permite almacenar el estado de un objeto de manera permanente para que sea recuperado cuando sea necesario. Esto trajo consigo una técnica que permite persistir los objetos en una base de datos relacional de manera eficaz: Mapeo Objeto-Relacional (ORM por sus siglas en inglés).

Hibernate se trata de una herramienta ORM que facilita las tareas de persistencia en la aplicación. Sus principales ventajas sobre el uso directo con JDBC son:

- Productividad: Debido a que elimina buena parte del código relacionado con el acceso y manejo de datos.
- Mantenimiento: Al contar con menos líneas de código, este se hace más comprensible y fácil de actualizar.
- Los cambios en el modelo de datos no afectan el funcionamiento programado.
- Simplicidad: Las transformaciones de los objetos para poder ser persistidos son innecesarias.
- Eficiencia: Realiza sus tareas con un mínimo de conexiones a la base de datos.
- Provee su propio lenguaje extendido de SQL: HQL (*Hibernate Query Language*) que de igual manera utiliza uniones (joins), funciones de agregación y funciones de agrupación, con la diferencia que HQL está totalmente orientado a objetos.
- Conserva la portabilidad que ofrece JDBC debido a que ocupa esta API por ser el estándar definido.

### 2.6.1 Componentes de Hibernate

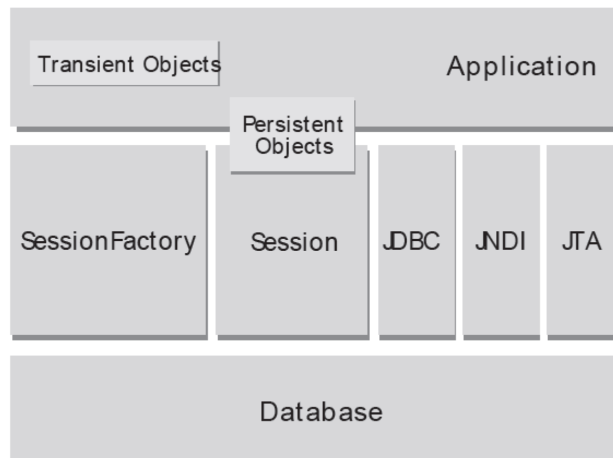


Figura 2.11: Componentes generales de Hibernate

La arquitectura básica de hibernate está formada por los siguientes elementos:

- `Session`: Es el objeto principal utilizado por Hibernate representando la conexión entre la base de datos y la aplicación, también llamada sesión. Se encarga de ejecutar las operaciones de almacenamiento y búsqueda de datos estando asociado a un sólo hilo de ejecución debido a que estos objetos son instanciados y destruidos por cada petición que Hibernate recibe de la aplicación.
- `SessionFactory`: Se encarga de crear los objetos `Session` cuando la aplicación los requiera. Existe sólo una instancia `SessionFactory` a lo largo de la aplicación (una por cada conexión a una base de datos diferente) y su estado es inmutable.
- `Configuration`: Es el responsable de la configuración de Hibernate de acuerdo a los archivos de mapeo y de crear el `SessionFactory` una vez que la configuración fue terminada. Es el primer objeto con el que se tiene contacto cuando se llama a Hibernate y es descartado una vez que se ha instanciado el `SessionFactory`.
- `Query`: Ejecuta una sentencia generalmente escrita en HQL permitiendo la parametrización de esta con resultados óptimos (análogamente con el objeto `PreparedStatement` de JDBC).

- Objetos persistentes: Se trata de la representación de las tablas de la base de datos en la aplicación por medio de JavaBeans, también llamados POJO (*Plain Old Text Object*). Se encuentran asociados a un objeto `Session` para ser persistidos.

A continuación se muestra el resumen de los métodos principales de clases e interfaces que conforman la arquitectura de Hibernate

| Método | Retorno | Descripción |
|---|---|---|
| buildSessionFactory() | SessionFactory | Crea un objeto `SessionFactory`. |
| configure() | Configuration | Busca las propiedades y mapeos del archivo de configuración para ser utilizados en la creación del `SessionFactory`.<br><br>Si no se especifica el archivo de configuración usa `hibernate.cfg.xml` por defecto. |
| configure(File configFile) | | |
| configure(String resource) | | |
| configure(URL url) | | |

Tabla 2.14: Métodos principales de Interface `Configuration`.

| Método | Retorno | Descripción |
|---|---|---|
| close() | void | Libera los recursos ocupados por el objeto `SessionFactory`. |
| getCurrentSession() | Session | Obtiene el objeto `Session` actual. |
| openSession() | | Instancia un nuevo `Session`. |

Tabla 2.15: Métodos principales de Interface `SessionFactory`.

| Método | Retorno | Descripción |
|---|---|---|
| beginTransaction() | Transaction | Encapsula una unidad atómica de trabajo (transacción). |
| cancelQuery() | void | Cancela la ejecución de la sentencia SQL. |
| close() | | Libera los recursos ocupados por el objeto `Session`. |
| createQuery(String hql) | Query | Crea una instancia de un objeto Query que estará asociado con el objeto `Session` y ejecutará una sentencia HQL. |
| delete() | void | Elimina un objeto persistente de la base de datos. |
| get(Class clase, Serializable id) | Object | Regresa el objeto persistente perteneciente a la clase e identificador dados, o nulo, de existir. |
| load(Class clase, Serializable id) | | Regresa el objeto persistente perteneciente a la clase e identificador dados, asumiendo que este existe. |
| save(Object objeto) | Serializable | Persiste en la base de datos el objeto dado. |
| update(Object objeto) | void | Actualiza el objeto indicado. |

Tabla 2.16: Métodos principales de Interface `Session`.

| Método | Retorno | Descripción |
| --- | --- | --- |
| iterate() | Iterator | Regresa el resultado de la sentencia en un objeto `Iterator`. |
| list() | List | Regresa el resultado de la sentencia en un objeto `List`. |
| scroll() | ScrollableResults | Regresa el resultado de la sentencia en un objeto `ScrollableResults`. |
| setTipo(int índice, Tipo valor)<br>*Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).* | Query | Establece el valor del parámetro indicado por el índice o el nombre para ser enviado en la consulta. |
| setTipo(String nombre, Tipo valor)<br>*Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).* | | |
| setFirstResult(int primerFila) | | Indica la primer fila que debe ser regresada. |
| setMaxResults(int maximoFilas) | | Fija el número máximo de filas que la sentencia debe regresar. |
| setParameter(int índice, Object valor) | | Establece el valor del parámetro indicado para una sentencia con nombre. |
| setParameter(String nombre, Object valor | | |
| uniqueResult() | Object | Regresa una instancia única del objeto resultado de la sentencia. |

Tabla 2.17: Métodos principales de Interface `Query`.

## 2.6.2 Configuración de Hibernate

### 2.6.2.1 Configuración de `Configuration`

Hibernate ofrece 2 alternativas de configuración: Un archivo de texto plano llamado `hibernate.properties` y un archivo XML llamado `hibernate.cfg.xml`, en ellos se especifican los parámetros necesarios para la conexión a la base de datos: URL, driver, dialecto, usuario, contraseña, etc. Así como el mapeo de las clases persistentes y algunos parámetros extra propios del framework. En caso de encontrarse los 2 archivos, `hibernate.cfg.xml` sobreescribe las propiedades configuradas en `hibernate.properties`. Para fines prácticos en la construcción de la aplicación, la configuración se llevará a cabo en el archivo `hibernate.cfg.xml`.

La estructura de este archivo se muestra a continuación:

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <!DOCTYPE hibernate-configuration PUBLIC
3    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5    <hibernate-configuration>
6      <session-factory>
7        <property name="hibernate.dialect">
8            org.hibernate.dialect.OracleDialect
9        </property>
10        <property name="hibernate.connection.driver_class">
11           oracle.jdbc.OracleDriver
12        </property>
13        <property name="hibernate.connection.url">
14            jdbc:oracle:thin:@127.0.0.1:1521:XE
15        </property>
16        <property name="hibernate.connection.username">usuario</property>
17        <property name="hibernate.connection.password">password</property>
18
19        <!— Mapeo de las clases persistentes -->
20
21      </session-factory>
22   </hibernate-configuration>
```

Código 2.9: Configuración de hibernate mediante el archivo `hibernate.cfg.xml`.

#### 2.6.2.2 Configuración de `SessionFactory`

Una vez que la configuración básica (la conexión a la base de datos, el usuario, contraseña, driver y las clases persistentes) se ha llevado a cabo, es conveniente tener por separado una clase de ayuda llamada `HibernateUtil`, que se encargue de instanciar el `SessionFactory` y optimice la creación de los objetos `Session`. Algunos ambientes de desarrollo como NetBeans, ya traen esta clase prediseñada, por lo que el programador no necesita crearla, sin embargo, es importante comprender su funcionamiento para que pueda ser usada correctamente.

```
1    public class HibernateUtil {
2        private static final SessionFactory sessionFactory;
3
4        static {
5            try {
6
7            sessionFactory = new Configuration().configure().buildSessionFactory();
9            } catch (Throwable e) {
11               System.out.println("Fallé al iniciar SessionFactory"+e);
12               throw new ExceptionInInitializerError(e);
13           }
14       }
15
16       public static SessionFactory getSessionFactory() {
17           return sessionFactory;
18       }
19   }
```

Código 2.10: Clase `HibernateUtil` encargada de crear el objeto `SessionFactory`

En la línea 2 se declara el objeto `SessionFactory` que se utilizará a lo largo de la aplicación y es inicializado en la línea 7 por medio de la creación de un objeto `Configuration`, el cual busca el archivo `hibernate.cfg.xml` para cargar las propiedades y mapeos con el método `configure()` y a partir de esto crear el objeto `SessionFactory` con el método `buildSessionFactory()`. Finalmente se regresa la referencia del `SessionFactory` en el método `getSessionFactory()` de la línea 16.

#### 2.6.2.3 Configuración de `Session`

En este punto, que ya se tiene el objeto `SessionFactory` listo para crear objetos `Session`, se debe contar con la clase que defina al objeto que será persistido en la base de datos, la cual debe estar registrada en `hibernate.cfg.xml`. Al conjunto de estas clases se le conoce como Modelo de Domino.

La configuración de las clases persistentes puede ser hecha de 2 maneras:

1. Por medio de un archivo XML en el que se definirán las propiedades de mapeo, tales como el nombre de la tabla equivalente a esa clase, el atributo id, las relaciones con otros objetos del modelo de dominio y el nombre de la columna correspondiente al atributo. Debe existir un archivo archivo `objeto.hbm.xml` por cada clase persistente.
2. A partir de la versión 5 de Java se introdujo el uso de anotaciones dentro del código para disminuir el tiempo de desarrollo. Hibernate ofrece soporte para esta funcionalidad y la incluye para definir las propiedades de mapeo directamente en los objetos POJO.

Antes de poder elegir entre estas dos opciones es conveniente analizar los siguientes puntos:

- El soporte para las anotaciones está dado a partir de Hibernate3 y si se está migrando una aplicación previa a esta versión, es conveniente conservar los archivos XML originales para no volver a configurar los mapeos y así evitar errores.
- Si no se cuenta con el código de las clases de dominio será imposible el uso de anotaciones.
- El uso de anotaciones hace que los mapeos se vuelvan más intuitivos y comprensibles.
- Para aplicaciones cuyo modelo de dominio es extenso, resulta complicada la gestión de los archivos XML por separado.

Retomando el ejemplo mostrado para JDBC (código 2.8) con la tabla proveedores se muestra a continuación los 2 tipos de configuración:

#### 2.6.2.4 Configuración del mapeo con `archivo.hbm.xml`

`Proveedores.class`

```
1   public class Proveedores {
2       private int idproveedor;
3       private Date inicioactividad;
4       private String nombre;
5       private char status;
6       private Date finactividad;
7
9    //métodos getter y setter
10  }
```

Código 2.11: Configuración del mapeo de la clase `Proveedores` en un archivo xml externo

`Proveedores.hbm.xml`

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <!DOCTYPE hibernate-mapping PUBLIC
3   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5   <hibernate-mapping>
6     <class name="Persistencia.Proveedores" table="proveedores">
7       <id name="idproveedor" column=" idproveedor"/>
8       <property name="nombre" type="string"/>
9       <property name="inicioactividad" type="java.util.Date"/>
10      <property name="status" type="string"/>
11      <property name="finactividad" type="java.util.Date"/>
12    </class>
14  </hibernate-mapping>
```

Código 2.11 (*continuación*)

La clase `Proveedores` únicamente cuenta con sus atributos y los métodos getter y setter para el transporte de los datos.

Por otro lado, en el archivo `Proveedores.hbm.xml` se lleva a cabo la configuración del mapeo a partir de la línea 5. En la etiqueta `class` de la línea 6 se declara el nombre calificado de la clase y la tabla a la que será mapeada. El atributo identificador se debe configurar dentro de la etiqueta `id` de la línea 7 y se especifican los nombres de este y de la columna, si ambos son iguales, este

último puede ser omitido. Para los atributos restantes se utiliza la etiqueta `property`, de las líneas 8 a 11, en la que se especifican el nombre y tipo de dato del atributo.

### 2.6.2.5 Configuración del mapeo con anotaciones

`Proveedores.class`

```
1   @Entity
2   @Table(name="proveedores")
3   public class Proveedores {
4
5       @Id
6       private int idproveedor;
7       @Temporal(TemporalType.DATE)
8       private Date inicioactividad;
9       private String nombre;
10      private char status;
11      @Temporal(TemporalType.DATE)
12      private Date finactividad;
13
14      //métodos getter y setter
15   }
```

Código 2.12: Configuración del mapeo de la clase `Proveedores` usando anotaciones.

La anotación `@Entity` indica que esa clase se trata de un objeto persistente hacia la tabla indicada con `@Table`. Cuando el nombre de la clase y la tabla es el mismo, esta última puede ser omitida. La anotación `@Id` de la línea 5 debe ser empleada en el atributo identificador. Cada una de las propiedades puede ser anotada con `@Column` indicando el nombre de su columna equivalente y puede ser excluida cuando el nombre de estos coinciden. Aquellos atributos que hagan referencia a columnas del tipo `DATE` o `TIMESTAMP` deben ser anotados con `@Temporal`.

Debido a las ventajas que presenta el mapeo de clases usando anotaciones, con respecto a los archivos XML, será esta la opción utilizada para la construcción de la aplicación.

Una vez que se ha configurado el mapeo, la clase debe ser registrada en hibernate.cfg.xml para que sea cargada cuando se crea la instancia de Configuration. Mediante el atributo class de la etiqueta mapping especificando el nombre calificado de la clase persistente (En caso de utilizar archivos XML, este es el que se registra).

```
19   <!– Mapeo de las clases persistentes -->
20   <mapping class="Persistencia.Proveedores"/>
```

Código 2.8 (*continuación*):

Ya en este punto, cuando la configuración previa se ha realizado, es el que pueden ser instanciados los objetos Session para comenzar con la persistencia de datos. El siguiente código muestra cómo recuperar de la base de datos una lista de objetos Proveedores.

```
1    public List<Proveedores> mostrarProveedores(){
2        Session = sesion;
3        List<Proveedores> proveedores;
4        sesion=HibernateUtil.getSessionFactory().openSession();
5        proveedor = sesion.createQuery("From Proveedores").list();
6        sesion.close();
7        return proveedor;
8     }
```

Código 2.13: Persistencia de los objetos Proveedores desde la base de datos con Hibernate.

En la línea 2 se declara un objeto Session inicializado por el método openSession() del SessionFactory que es referenciado a través de HibernateUtil en la línea 4. Una vez que se tiene abierta una nueva sesión, por medio del método createQuery() se ejecuta una sentencia HQL que será traducida al lenguaje nativo del RDBMS y se enviará a JDBC para su ejecución, que una vez realizada, regresa los datos en el formato indicado, en este caso, una lista de objetos Proveedores. Finalmente en la línea 6 se cierra la sesión para liberar recursos.

En el siguiente código se muestra la forma en que el estado del objeto es persistido a la base de datos:

```
1    public void guardarProveedores (Proveedores Pv){
2        Session sesion;
3        Transaction tran;
4        sesion=HibernateUtil.getSessionFactory().openSession();
5        tran=sesion.beginTransaction();
6        sesion.save(Pv);
7        tran.commit();
8        sesion.close();
9        tran.close();
10   }
```

Código 2.14: Persistencia de los objetos `Proveedores` hacia la base de datos con Hibernate.

Para modificar información en la base de datos (inserción, borrado o actualización), aparte del objeto `Session`, se necesita un objeto del tipo `Transaction` que maneje la transacción de la modificación. En la línea 3 es declarado e inicializado en la línea 5 con el método `beginTransaction()` del objeto `Session`. En la línea 6 se persiste el objeto recibido y para que su estado quede de manera permanente en la base de datos debe ser confirmado por medio del método `commit()`, de lo contrario no será almancenado. Finalmente en la líneas 7 y 8 se liberan los recursos de la sesión y la transacción.

Es importante tener en cuenta que Hibernate **no** sustituye a JDBC, pues la comunicación con la base de datos siempre se realiza por medio de este, el propósito de Hibernate es que el desarrollador sólo se enfoque en resolver qué datos necesita y no cómo acceder a estos. Para ello sólo se requiere configurar el mapeo de las entidades persistentes con las anotaciones precisas.

### 2.6.2.6 Anotaciones para el mapeo de atributos

- **`@Entity`**

  Sirve para indicarle a Hibernate las clases persistentes que deben ser mapeadas.

- **`@Id`**

  Debe ser empleada en el campo que funge como identificador o clave primaria de la clase.

- **`@GeneratedValue(strategy, generator)`**

  A pesar de que `@Id` es capaz por sí misma de elegir la mejor estrategia de generación de claves primarias, esta anotación sobreescribe dicha funcionalidad para ser asignada manualmente. Recibe 2 atributos:

  1. `strategy`: Define la estrategia de generación de claves, puede ser de 4 tipos:
     1.1. `GenerationType.AUTO`: Hibernate elige qué método usar de acuerdo al soporte dado por el RDBMS.
     1.2. `GenerationType.IDENTITY`: El RDBMS es el responsable de asignar este valor automáticamente conforme se van insertando lo datos.
     1.3. `GenerationType.SEQUENCE`: La asignación se lleva a cabo por medio de una secuencia creada en la base de datos.
     1.4. `GenerationType.TABLE`: Una tabla que contiene los valores primarios es la encargada de hacer la asignación.
  2. `generator`: Es el identificador de la generación de las claves.

- **`@SequenceGenerator(name,sequenceName,initialValue,allocationSize)`**

  Establece las propiedades de la generación de claves primarias a través de una secuencia. Sus atributos son:

  1. `name`: Es el nombre que identifica la secuencia y que deberá coincidir con el atributo `generator` de `@GeneratedValue`.

2. `sequenceName`: Es el nombre real de la secuencia en el RDBMS.
3. `initialValue`: Define el número con el que debe comenzar la secuencia
4. `allocationSize`: Indica el incremento que debe haber entre casa número creado.

- **`@EmbeddedId y @Embeddable`**

  Las claves primarias compuestas deben ser tratadas por JavaBeans aislados, en los que se declaren los atributos que forman la llave compuesta. Esta clase debe estar anotada con `@Embeddable` (en vez de `@Entity`) y el objeto persistente deberá indicar por medio de `@EmbeddedId` el objeto que define las clave compuesta (en sustitución de `@Id`).

- **`@Column(name, length, nullable, unique,…)`**

  Es una anotación opcional, útil cuando la columna y el atributo no tienen el mismo nombre o cuando se pretende crear las tablas a partir de los mapeos configurados. Sus atributos más importantes son:

  1. `name`: Indica el nombre de la columna.
  2. `length`: Especifica la longitud del atributo cuando es de tipo String. Si se trata de `double`, `float` o `int` se usan los atributos `scale` y `precision`.
  3. `nullable`: Indica si la columna debe ser NULL o NOT NULL.
  4. `unique`: Asigna la restricción de valor único a la columna.

- **`@Table(name, schema,…)`**

  Se trata de una anotación opcional que indica el nombre de la tabla a la que será mapeada la clase, útil cuando los nombres de estas no coinciden o cuando se intenta crear las tablas a partir de los mapeos configurados. Sus atributos más importantes son:

  1. `name`: Indica el nombre de la tabla.
  2. `schema`: Define el usuario de la base de datos sobre el cual se está trabajando.

- **`@Temporal(TemporalType)`**

  Los atributos que manejan valores temporales deben ser anotados con esta anotación para indicar cómo serán mapeados hacia la base de datos: `DATE`, `TIME` o `TIMESTAMP`.

#### 2.6.2.7 Anotaciones para el mapeo de asociaciones

Una asociación es la relación que guardan las clases persistentes entre sí: uno a uno, uno a muchos, muchos a uno y muchos a muchos. Existen 2 tipos de asociaciones:

- Unidireccionales: La navegabilidad entre las clases se lleva a cabo sólo en un sentido, y únicamente la clase en la que se define la asociación tiene conocimiento de la existencia de su clase asociada.
- Bidireccionales: La navegabilidad de las clases asociadas se lleva a cabo en los 2 sentidos. Cada clase es "consciente" de la existencia de la otra.

Es recomendable el uso de asociaciones bidireccionales con la finalidad de no limitar la navegabilidad en los datos, sin embargo, esto debe ser decidido por el desarrollador de acuerdo a las especificaciones. A continuación se muestran las anotaciones necesarias para mapear asociaciones:

- **`@OneToOne(cascade, fetch, mappedBy, optional,…)`**

  Esta anotación representa una asociación uno a uno entre 2 clases. Sus atributos principales son:

  1. `cascade`: Define las operaciones que podrán ser realizadas en cascada. Sus opciones son:

     1.1. `CascadeType.PERSIST`: Inserción de datos en cascada.

     1.2. `CascadeType.REFRESH`: Actualizaciones en cascada.

     1.3. `CascadeType.REMOVE`: Borrado de datos en cascada.

     1.4. `CascadeType.ALL`: Indica que todas las operaciones serán realizadas en cascada.

     En caso de no ser indicado este atributo, ninguna operación se realizará en cascada.

  2. `fetch`: Cuando el estado de un objeto es persistido desde la base de datos existen 2 maneras de manejar las relaciones con los diferentes objetos.

     2.1 `FetchType.EAGER`: Indica que el objeto persistido traerá consigo los datos de sus objetos asociados.

     2.2 `FetchType.LAZY`: Las asociaciones del objeto no serán persistidas cuando este lo sea.

Es recomendable usar siempre asociaciones con carga perezosa (LAZY) para mejorar el rendimiento de las consultas y sólo llamar a los objetos asociados cuando sea necesario (a través de los joins de HQL).

3. `mappedBy`: En las relaciones bidireccionales la clase padre debe indicar con este atributo los objetos que dependen de ella.
4. `optional`: Indica si el atributo mapeado puede ser nulo.

- **`@OneToMany(cascade, fetch, mappedBy, optional,…)`**

  Representa la asociación uno a muchos entre 2 clases. El lado izquierdo al `To` indica la cardinalidad de la clase que lleva la anotación, el lado derecho es la cardinalidad de la clase asociada representada como una colección de objetos del tipo `Set`. Sus atributos principales son:

  1. `cascade`: Define las operaciones que podrán ser realizadas en cascada. En caso de no ser incluido ninguna operación se realizará en cascada.
  2. `fetch`: Especifica el tipo de carga que tendrán las asociaciones relacionadas con el objeto.
  3. `mappedBy`: En las relaciones bidireccionales la clase padre debe indicar con este atributo los objetos que dependen de ella.
  4. `optional`: Indica si el atributo mapeado puede ser nulo.

- **`@ManyToOne(cascade, fetch, optional,…)`**

  Representa la asociación muchos a uno entre 2 clases. Sus atributos principales son:

  1. `cascade`: Define las operaciones que podrán ser realizadas en cascada. En caso de no ser incluido ninguna operación se realizará en cascada.
  2. `fetch`: Especifica el tipo de carga que tendrán las asociaciones relacionadas con el objeto.
  3. `optional`: Indica si el atributo mapeado puede ser nulo.

- **`@JoinColumn(name,insertable,updatable)`**

  Indica la clave foránea en las relaciones hijas anotadas con `@ManyToOne`. Sus atributos principales son:

  1. `name`: El nombre del atributo que representa la clave foránea.
  2. `insertable`: Indica si el atributo debe ser insertado cuando sea mapeado.
  3. `updatable`: Indica si el atributo debe ser actualizado cuando sea mapeado.

Aquellas clases que tengan la clave foránea como parte de una clave primaria compuesta deben especificar los atributos `insertable` y `updatable` como `false` para evitar un doble mapeo de la clave foránea.

- **`@ManyToMany (cascade, fetch, mappedBy,…)`**

  Representa la asociación muchos a muchos entre 2 clases. Sus atributos principales son:

  1. `cascade`: Define las operaciones que podrán ser realizadas en cascada. En caso de no ser incluido ninguna operación se realizará en cascada.
  2. `fetch`: Especifica el tipo de carga que tendrán las asociaciones relacionadas con el objeto.
  3. `mappedBy`: En las relaciones bidireccionales la clase padre debe indicar con este atributo los objetos que dependen de ella.

Esta anotación sólo es útil cuando se trata de una relación muchos a muchos pura, es decir, que no tenga algún atributo extra a las claves foráneas que forman el identificador compuesto, de otra manera, la relación debe ser tratada por separado como 2 asociaciones uno a muchos.

Continuando con el ejemplo de los proveedores, a continuación se muestra cómo se realiza el mapeo de la asociación entre `Proveedores` y `Sucursales`.

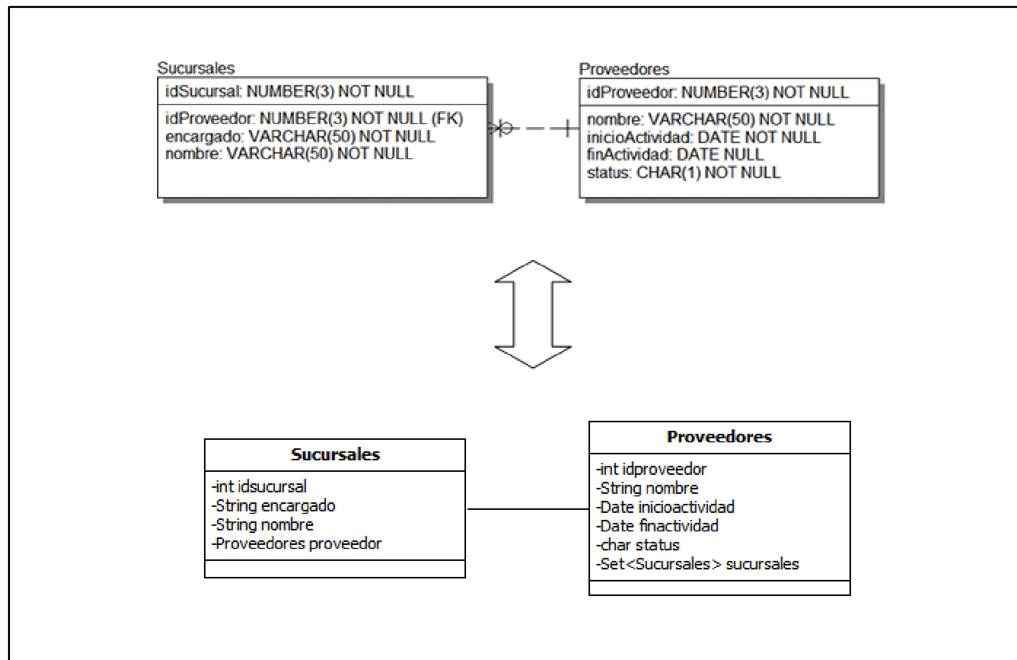


Figura 2.12: Mapeo de una relación `@OneToMany` bidireccional

Proveedores.class

```
1    @Entity
2    public class Proveedores {
3      @Id
4      @GeneratedValue(generator="secuencia2",strategy=GenerationType.SEQUENCE)
5      @SequenceGenerator(name="secuencia2",  sequenceName="secuencia_pv",
6      initialValue=1, allocationSize=1)
7      private int idproveedor;
8      @Temporal(TemporalType.DATE)
9      private Date inicioactividad;
10     private String nombre;
11     private char status;
12     @Temporal(TemporalType.DATE)
13     private Date finactividad;
14     @OneToMany(mappedBy="proveedor",fetch=FetchType.LAZY)
15     private Set<Sucursales> sucursales = new HashSet<Sucursales>();
16     //métodos getter y setter
17   }
```

Código 2.15: Mapeo entre `Proveedores` y `Sucursales`

Sucursales.class

```
1    @Entity
2    public class Sucursales{
3      @Id
4      @GeneratedValue(generator="secuencia3",strategy=GenerationType.SEQUENCE)
5      @SequenceGenerator(name = "secuencia3", sequenceName = "secuencia_suc",
6      initialValue = 1, allocationSize = 1)
7      private int idsucursal;
8      @ManyToOne(fetch = FetchType.LAZY)
9      @JoinColumn(name = "idproveedor")
10     private Proveedores proveedor;
11     private String encargado;
12     private String nombre;
13     //métodos getter y setter
14   }
```

Código 2.15 (*continuación*)

La asociación está definida como una relación bidireccional, pues en ambas clases se declara una referencia hacia la otra. De las líneas 4 a 6 en ambas clases, se define como estrategia de creación de llaves primarias una secuencia que comenzará en 1 e irá incrementándose a paso de 1.

La clase `Proveedores` define una colección `Set` de objetos `Sucursales` por medio de la asociación uno a muchos (un proveedor podrá tener muchas sucursales) en las líneas 14 y 15.

Por su parte, `Sucursales` define en la línea 8 la relación que guardará con `Proveedores` (una o más sucursales pueden pertenecer a un sólo proveedor) y establece la clave foránea en la línea 9.

## 2.7 Spring

Hasta el momento los 2 frameworks analizados cumplen con el cometido de simplificar el desarrollo de aplicaciones web con las ventajas de modularidad, fácil mantenimiento, ahorro de líneas de código y eficiencia en el tiempo de desarrollo. Sin embargo, las ventajas podrían acrecentarse aún más si se incluye un framework más a la lista: Spring.

El principal objetivo de Spring es facilitar el desarrollo de aplicaciones creadas en Java por medio de la unión de sus componentes (ya sea creados por el usuario, provenientes de otros frameworks o bien provenientes del API de Spring). Una de sus mayores virtudes es que se trata de un framework no invasivo, esto es, que en ninguna clase de la aplicación se programará código de Spring.

Spring está compuesto de 7 módulos que no necesariamente son útiles para todas las necesidades de negocio por lo que el programador tendrá la libertad de emplear aquellos que puedan tener un impacto tangible sobre su aplicación. Estos módulos son mostrados a continuación:

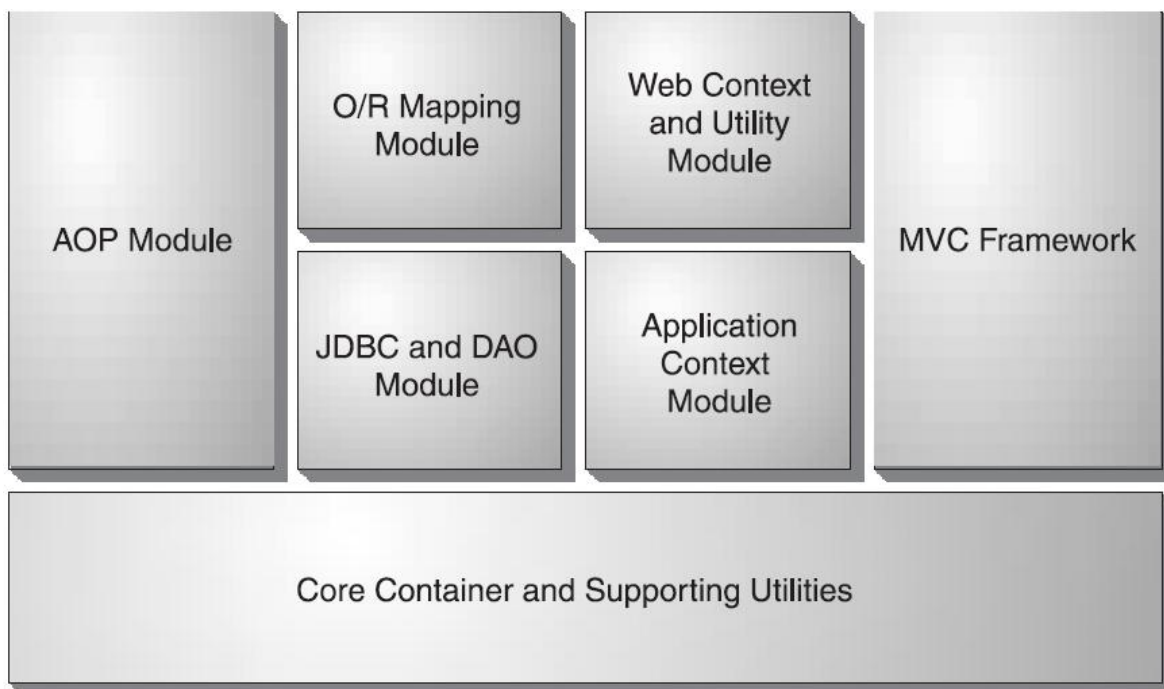


Figura 2.13: Arquitectura general de Spring

- **Core Container:** Spring también es conocido como un BeanFactory debido a que uno de sus roles fundamentales es la creación de objetos con las posibles dependencias que estos pudieran tener con otros. Este módulo es el encargado de proveer dicha funcionalidad a Spring siendo la base para cualquier aplicación que utilice el framework.

- **ApplicationContext:** Se trata de una subinterface de BeanFactory que se encarga de tener en memoria los beans declarados en el archivo de configuración `ApplicationContext.xml`.

- **AOP (Aspect-Oriented Programming):** Es una técnica que permite al programador separar la lógica de negocio con los aspectos técnicos de la aplicación, tales como logging, debugging y seguridad.

- **JDBC y DAO:** Spring ofrece un template o plantilla que contiene la funcionalidad del acceso a datos por medio de JDBC, que manejará la instanciación de los diferentes objetos derivados del API (`Connection`, `Statement`, `PreparedStatement` y `ResultSet`) así como el manejo de excepciones y liberación de recursos.

- **ORM:** Para aquellas aplicaciones que no hacen uso de JDBC para el manejo de datos sino que se basan en la implementación de un ORM que facilite esta tarea. Spring provee de un template o plantilla que soporta la interacción de sus objetos, liberación de recursos, manejo de excepciones y ejecución de transacciones, que unido, hace que el código responsable de la persistencia de datos se simplifique aún más.

- **Web:** Este módulo ofrece diversas tareas para aplicaciones web y contiene soporte para la integración con Struts.

- **MVC:** Spring también provee su propio framework basado en MVC (similar a Struts) que ayuda a la integración de este patrón de diseño en la aplicación.

### 2.7.1 Inyección de Dependencias

Uno de los retos a los que se debe enfrentar todo desarrollador de software es que el nivel de acoplamiento entre los objetos involucrados sea el más pobre posible. Una solución muy eficaz se encuentra en el uso de interfaces y en el concepto fundamental de Spring: La inyección de dependencias.

A través del siguiente caso de una clase que se encargará de sumar 2 números se mostrará cómo se logra tal nivel de acoplamiento usando Spring:

```
1   public class Suma{
2
3      public void operar(double num1, double num2) {
4          double resultado = num1 + num2;
5          System.out.println("El resultado es: " + resultado);
6      }
7   }
```

Código 2.16: Suma de 2 números

```
1    public class Calculo {
2
3       public void calcular() {
4           Suma suma=new Suma();
5           operación.operar(3.21,5.24);
6       }
7   }
```

Código 2.16 (*continuación*)

```
1    public class Main {
2
3       public static void main(String[] args) {
4            Calculo calculo = new Calculo();
5            calculo.calcular();
6        }
7    }
```

Código 2.16 (*continuación)*

En la clase `Calculo` se manda a llamar la clase concreta encargada de llevar a cabo la suma por medio de su método `operar()` que recibe los números a ser sumados. Sin embargo, el problema que se tiene es que la clase `Calculo` se encuentra muy acoplada a `Suma`. Debido a la instanciación directa de esta última en `Calculo` (línea 4).

Si en algún momento del desarrollo se pide que la aplicación no sume, sino que reste, se debe crear la nueva clase encargada de restar, así como modificar el tipo de objeto a instanciar, de `Suma` a `Resta`. Lo mismo tendría que suceder si en un futuro el requerimiento de la aplicación vuelve a cambiar y se pide multiplicación o división de 2 números.

Para solucionar este problema, se puede delegar a otro componente la implementación concreta de dicha clase.

```
1   public interface Operacion {
2       public void operar(double num1, double num2);
3   }
```

Código 2.17: Suma de 2 números usando Interfaces

```
1   public class Suma implements Operacion{
2       public void operar(double num1, double num2) {
3          double resultado = num1 + num2;
4          System.out.println("El resultado es: " + resultado);
5       }
6   }
```

Código 2.17 (*continuación)*

```
1   public class Calculo {
2       Operacion operacion;
3       public void setOperacion(Operacion operacion) {
4           this.operacion = operacion;
5       }
6       public void calcular() {
7           operacion.operar(1.23, 4.32);
8       }
9   }
```

Código 2.17 (*continuación)*

```
1   public class Main {
2
3       public static void main(String[] args) {
4           Operacion suma = new Suma();
5           Calculo calculo = new Calculo();
6           calculo.setOperacion(suma);
7           calculo.calcular();
8       }
9   }
```

Código 2.17(*continuación)*

La clase `Main` ahora será la encargada de instanciar a `Suma`, pero a través de la interface `Operacion` y será referida hacia `Calculo` por medio del método `setOperacion` definido en este el cuál recibirá la implementación de la interface que debe ser utilizada. De esta manera, `Calculo` sólo sabrá que debe hacer una operación, pero sin saber cuál. Pese a esta mejora, el código de `Main` sigue estando acoplado a la clase concreta de la operación a realizar y es aquí donde Spring comienza su trabajo.

El rol fundamental de Spring es la creación de los Beans que se encuentren en el archivo de configuración que define el contexto de la aplicación (`ApplicationContext`). Si se aprovecha esta funcionalidad, se puede delegar a Spring la creación de los distintos beans requeridos en esta aplicación.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns=http://www.springframework.org/schema/beans>
3
4       <bean id="suma" class="pruebas.Suma"/>
5       <bean id="calculo" class="pruebas.Calculo">
6           <property name="operacion"  ref="suma"/>
7       </bean>
8   </beans>
```

Código 2.18: Suma de 2 números en Spring

Los beans son declarados por medio de la etiqueta `bean` y sus atributos: `id` que recibe el nombre utilizado para identificar el bean y `class` que define la clase que lo contiene.

Los 2 objetos principales de esta aplicación son: `suma` y `calculo` por lo que son declarados en el XML de Spring en la línea 4 y 5. `calculo`, puede tener diferentes implementaciones de `Operacion` (en este caso suma) y a través del método `setOperacion` definido en él se indica cuál será. Para que Spring pueda realizar esto, se debe definir el objeto que contiene tan implemetanción. En la línea 6 con la etiqueta `property` se indica con el atributo `name` la propiedad que será referida a este por medio del método setter y la instancia que será utilizada con el atributo `ref`. En este caso: En el objeto `calculo` se define una instancia `suma` del atributo `operacion`. (`operacion` se encuentra definido en `Calculo` como una variable de clase).

```
1   public class Main {
2
3      public static void main(String[] args) {
4        ApplicationContext ac = new
5              ClassPathXmlApplicationContext("applicationContext.xml");
6         Calculo calculo = ac.getBean("calculo");
7         calculo.calcular();
8      }
9   }
```

Código 2.18 (*continuación*)

Finalmente en `Main` únicamente se manda a llamar el `applicationContext.xml` en la línea 4, y a partir de este, el bean `calculo` y como se observa, desaparece del código la definición de la operación concreta:

```
Operacion suma = new Suma();
Calculo calculo = new Calculo();
calculo.setOperacion(suma);
```

es equivalente a la siguiente delegación de Spring

```
<bean id="suma" class="pruebas.Suma"/>
<bean id="calculo" class="pruebas.Calculo">
    <property name="operacion"  ref="suma"/>
</bean>
```

} Inyección de Dependencias (DI)

Con esta última versión, se puede apreciar la ventaja que ofrece la programación con interfaces y más aún, si las implementaciones de estas son delegadas a Spring. Así si en algún momento el requerimiento de la operación cambia, sólo se debe modificar su implementación en el XML sin tener que recurrir al código para llevar a cabo la actualización.

### 2.7.2 Integrando Spring con Struts

La manera en que Struts trabaja con los objetos `Action`, por medio del archivo de configuración, permite que el rendimiento de la aplicación sea óptimo, debido a que son almacenados en una colección que contiene las referencias hacia estos y cuando una nueva petición es interceptada, Struts revisa que el `Action` solicitado se encuentre instanciado, de ser así, lo recupera y utiliza. En caso contrario, lo instancia y guarda una referencia de este para futuras solicitudes. No obstante, dicho rendimiento podría ser mejorado si se aprovecha el contenedor de Spring.

Cada `Action` invoca cierta cantidad de objetos necesarios para cumplir con la acción solicitada, es aquí donde el concepto de DI juega un papel importante en el desempeño de la aplicación, pues lo que se busca es que los `Action` sean manejados como beans e instanciados desde el momento que el contenedor arranca con sus respectivas dependencias de los objetos de negocio necesarios. Para lograr esta integración, se necesita llevar a cabo los siguientes puntos:

1. Registrar el `ContextLoaderPlugIn` en el archivo de configuración de Struts:
   Este plug-in es el responsable de cargar el archivo de configuración de Spring cuya ubicación se encuentra definida en el atributo `value` de la propiedad `contextConfigLocation`.

```
1   <plug-in
2        className="org.springframework.web.struts.ContextLoaderPlugIn">
3           <set-property property="contextConfigLocation"
4           value="/WEB-INF/applicationContext.xml"/>
5   </plug-in>
```

Código 2.19: Integración de Struts con Spring

2. Delegar el control de `Action`s a Spring:

   La clase `DelegatingActionProxy` será la responsable de cargar el plug-in de Spring para buscar en el `applicationContext` el `Action` que deberá procesar la petición.

```
1    <action-mappings>
2        <action name="UniformeForm" path="/UniformeAction"
3           type="org.springframework.web.struts.DelegatingActionProxy"
4           scope="request">
5            <forward name="verUniformes" path="/verUniformes2.jsp"/>
6        </action>
7     </action-mappings>
```

Código 2.19 (*continuación*)

   Originalmente en `struts-config.xml` se registraba, en el atributo `type`, la clase de acción correspondiente a la ruta que originaba la ejecución de esta. Con Spring, debido a que cada solicitud recibida deberá pasar por el proxy que la encaminará al `Action` correspondiente, este es el que debe ser registrado en cada etiqueta `action`.

3. Registrar las clases `Action` en el `applicationContext`:

   Para que `DelegatingActionProxy` sepa a qué `Action` ceder el trabajo de la petición, este último debe estar configurado en el `applicationContext` como un bean con sus respectivas dependencias y sus métodos `set` dentro de la clase `Action`.

```
1    <bean name="/UniformeAction" class="Action.UniformeAction">
2        <property name="LUniformes" ref="LUniformes">
3        </property>
4    </bean>
```

Código 2.19 (*continuación*)

### 2.7.3 Integrando Spring con Hibernate

Spring, en su módulo de ORM, ofrece un *template* que facilita y optimiza el uso de objetos `Session` de hibernate. Las ventajas que este ofrece son:

- Manejo transparente de excepciones y liberación de recursos.
- Uso automático de transacciones.
- Evita instanciar objetos `Session` con la implicación de abrir y cerrar conexiones.
- Reducción (aún más) del código de acceso a datos.

La clase `HibernateTemplate` es la encargada de llevar a cabo estas acciones de Hibernate a través de Spring. La clase de acceso a datos debe heredar su funcionalidad de `HibernateDaoSupport` para que le suministre un objeto `HibernateTemplate` por medio de su método `getHibernateTemplate()`. La forma en que se integra a la aplicación es la siguiente:

1. Configurar la fuente de datos:

   Spring elimina la necesidad del archivo de configuración de Hibernate debido a que desde el `applicationContext.xml` pueden ser declarados todos aquellos parámetros necesarios para la conexión con la base de datos. Como primer instancia se necesita configurar el bean del Data Source con sus valores de conexión.

```
1    <bean id="dataSource"
2    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
3        <property name="driverClassName" value="oracle.jdbc.OracleDriver"/>
4        <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:XE"/>
5        <property name="username" value="hugo"/>
6        <property name="password" value="hugo"/>
7    </bean>
```

Código 2.20: Integración de Hibernate con Spring

La clase `DriverManagerDataSource` de la línea 2 es la encargada de gestionar la conexión por medio de JDBC.

2. Registrar el `SessionFactory`:

   Spring provee de una clase capaz de crear el `SessionFactory` sin tener que hacer uso del `HibernateUtil` (código 2.10): `LocalSessionFactoryBean` para aquellas aplicaciones que realizan su mapeo por medio de archivos XML o bien, una subclase de esta, `AnnotationSessionFactoryBean` para las aplicaciones que los realizan a través de anotaciones. La clase indicada deberá ser configurada en el `applicationContext.xml` inyectándole el dataSource, el mapeo de las clases persistentes y de ser necesario, algunos parámetros extra propios de Hibernate (por ejemplo el dialecto al que se deben traducir todas las peticiones de Hibernate a la base de dato, y la opción de poder mostrar las secuencias sql que Hibernate genera y envía al servidor).

```
1    <bean id="sessionFactory"
2    class="org.springframework.orm.hibernate3.annotation.
3    AnnotationSessionFactoryBean">
4        <property name="dataSource" ref="dataSource"/>
5        <property name="annotatedClasses">
6            <list>
7                <!-- Mapeos -->
8            </list>
9        </property>
10       <property name="hibernateProperties">
11          <props>
12              <prop key="hibernate.dialect">
13                     org.hibernate.dialect.OracleDialect
14              </prop>
15              <prop key="hibernate.show_sql">true</prop>
16          </props>
17       </property>
18    </bean>
```

Código 2.20 (*continuación*)

En los códigos 2.12 y 2.13 se mostró cómo se persisten los datos de y hacia la base de datos respectivamente haciendo uso de Hibernate, a continuación, se vuelven a presentar estos dos escenarios con la integración de Hibernate y Spring.

```
1    public List<Proveedores> mostrarProveedores() {
2            List<Proveedores> proveedores;
3            proveedores = getHibernateTemplate().find("from Proveedores");
4            return proveedores;
5     }
```

Código 2.21: Persistencia de los objetos `Proveedores` desde la base de datos con Hibernate y Spring

El método `find()` del `HibernateTemplate` sustituye al `createQuery()` del objeto `Session`, y ahora este *template* es el encargado de instanciar estos objetos.

```
1   public void guardarProveedores (Proveedores pv) {
2         getHibernateTemplate().save(pv);
3   }
```

Código 2.22: Persistencia de los objetos `Proveedores` hacia la base de datos con Hibernate y Spring.

Para persistir el objeto hacia la base de datos únicamente se llama al método `save()` del *template* y será el que confirme la transacción en la base de datos.

### 2.7.4 Manejo de Transacciones

En toda aplicación, ya sea web o de escritorio, es vital que se garantice alta integridad y consistencia en los datos, por tal motivo, es indispensable el uso de transacciones para que esto sea posible.

Una transacción es un grupo de instrucciones relacionadas en un aspecto de la lógica de negocio que deben ser ejecutadas en su totalidad con éxito y en caso de falla, ninguna de estas se verá reflejada en la Base de Datos. El comportamiento esperado de una transacción se describe a partir de sus propiedades:

1. Atomicidad: Todas las instrucciones de la transacción son vistas como una sola que no puede ser dividida. Los cambios surgidos desde la primera hasta la última instrucción deben realizarse con éxito (commit) o bien abortarse en caso de que alguna de estas falle (rollback).
2. Consistencia: Una vez que la transacción haya finalizado, los datos deben permanecer en una manera consistente, esto es que ninguna de sus restricciones de integridad sea violadas.
3. Aislamiento: Las transacciones pueden ser lanzadas en forma paralela y estas no deben ser afectadas por la acción de otras que corran sobre el mismo conjunto de datos. Existen varios niveles de aislamiento: Desde uno completo, en el que 2 ó más transacciones que sean ejecutadas en un mismo dominio de datos provoque el bloqueo de estos y no podrán ser utilizados hasta que la transacción actual haya finalizado, hasta niveles más bajos en los que varias transacciones pueden trabajar sobre un mismo conjunto de datos a la vez.
4. Durabilidad: Una vez que la transacción fue completada de manera exitosa, los datos deben residir de manera permanente en la base de datos.

#### 2.7.4.1 La anotación `@Transactional`

Esta anotación es utilizada para marcar un método como transaccional, puede ser declarada a nivel interface, en donde cada método implementado será ejecutado bajo la definición transaccional o bien a nivel método, en donde específicamente el método marcado con la anotación será transaccional.

**Propiedades:**

1. `Propagation`: Es la asociación de la transacción con el método en ejecución. En otras palabras, determina si en el método actual se crea una nueva transacción, se trabaja sobre una ya existente, o no se requiere transacción. Sus opciones son las siguientes:
   - `PROPAGATION_REQUIRED`: Es el valor por defecto, en caso de no ser especificado, indica que el método debe ser ejecutado como parte de una transacción. Si no existe la crea, de lo contrario corre sobre una ya existente.
   - `PROPAGATION_SUPPORTS`: Indica que el método no requiere una transacción, pero en caso de existir, esta será utilizada.
   - `PROPAGATION_MANDATORY`: Indica que el método requiere necesariamente una transacción y en caso de no existir, lanzará una excepción.
   - `PROPAGATION_REQUIRES_NEW`: Una transacción nueva debe comenzar cada que el método marcado con esta opción sea ejecutado. Si existe una, es suspendida.
   - `PROPAGATION_NOT_SUPPORTED`: Indica que el método no debe ser parte de ninguna transacción, de existir, esta es suspendida y retomada una vez que el método finaliza.
   - `PROPAGATION_NEVER`: Indica que el método no debe ser ejecutado como parte de una transacción, si existe, lanzará una excepción.

2. `Isolation`: Cuando se trabaja con transacciones existen 3 tipos de problema que se pueden encontrar debido a la concurrencia de estas sobre los datos.
   - Lectura sucia: Ocurre cuando una transacción lee datos que no han sido confirmados (commit).
   - Lectura no repetitiva: Ocurre si dentro de una transacción se leen datos diferentes para una misma consulta enviada más de una vez, debido a que otra transacción modificó los datos.
   - Lectura fantasma: Ocurre cuando una transacción envía una consulta más de una vez y las filas retornadas no coinciden debido a que otra transacción insertó nuevos datos que coinciden con el criterio de búsqueda de la consulta.

Para evitar estos problemas, se configura esta propiedad de aislamiento con sus siguientes opciones:

- `ISOLATION_READ_UNCOMMITED`: Es el nivel más bajo de aislamiento en el que los problemas previamente descritos pueden ocurrir. No es recomendable para operaciones críticas, sin embargo, con esta opción de aislamiento el rendimiento de la aplicación es mayor.
- `ISOLATION_READ_COMMITED`: Este nivel no permite que se hagan lecturas de datos a los que no se ha hecho commit, pero permite que estos sean modificados por otras transacciones.
- `ISOLATION_REPETABLE_READ`: Este nivel tiene las características de `READ_COMMITED` y además no permite que los datos manejados por una transacción sean modificados.
- `ISOLATION_SERIALIZABLE`: Es el nivel más alto de aislamiento de una transacción, en el que además de las condiciones de `REPETABLE_READ`, no permite que se haga inserción de datos que cumplan con el criterio de búsqueda de la consulta en una transacción en curso.

En la siguiente tabla se muestran los problemas que pueden ocurren por cada grado de aislamiento

| | READ_UNCOMMITED | READ_COMMITED | REPETABLE_READ | SERIALIZABLE |
|---|---|---|---|---|
| Lectura Sucia | ✓ | × | × | × |
| Lectura no repetitiva | ✓ | ✓ | × | × |
| Lectura fantasma | ✓ | ✓ | ✓ | × |

Tabla 2.18: Comparativa entre los posibles niveles de aislamiento en transacciones

3. `read-only`: Indica si la transacción debe ser manejada como de sólo lectura. Hibernate provee una optimización cuando esta propiedad es marcada como `true`.

4. `rollbackFor`: Indica el tipo de excepción, que al ser lanzada, provocará el rollback de la transacción. Su contraparte es la opción `noRollbackFor`.

### 2.7.4.2 Configuración

Para comenzar la configuración, se debe elegir la estrategia que utilizará Spring para el manejo de las transacciones.

Cada API de acceso a datos en Java contiene su propia definición para el manejo de transacciones. Spring en su API, cuenta con interfaces que abstraen estas definiciones y las implementa para ser utilizadas en los diferentes casos. Para Hibernate 3 se debe utilizar la clase `org.springframework.orm.hibernate3.HibernateTransactionManager`.
El `transactionManager` debe obtener un objeto `sessionFactory` que le permita realizar la transacción. Por consiguiente, en el archivo de configuración de Spring se inyecta una dependencia de este al `transactionManager`.

```
1    <bean id="transactionManager"
2    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
3            <property name="sessionFactory"  ref="sessionFactory"/>
4    </bean>
```

Código 2.23: Configuración de transacciones.

Finalmente, en caso de emplear anotaciones para el manejo de transacciones, se le debe indicar a Spring su uso por medio de la etiqueta `<tx:annotation-driven>` indicándole la referencia del bean que gestiona las transacciones (`transactionManager`).

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Código 2.23 (*continuación*)

La interacción de los objetos tras la integración de los 3 frameworks (Struts, Hibernate y Spring) se muestra en la siguiente figura

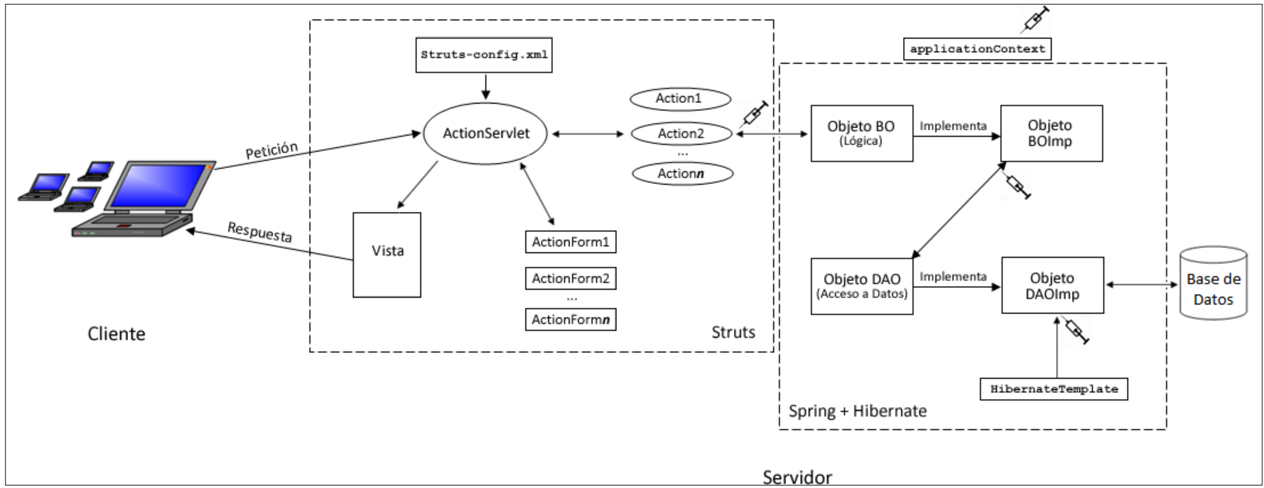


Figura 2.14: Intregración de Struts, Hibernate y Spring.