



Desarrollo de Aplicaciones Web II

ÍNDICE

Presentación	5
Red de contenidos	6
Gestión de la capa de controlador mediante <i>framework Spring</i>	
SEMANA 1 : Arquitectura del <i>framework Spring</i>	7
SEMANA 2 : Implementación de la capa controlador mediante el <i>framework Spring</i>	19
Gestión de la capa vista mediante <i>AJAX</i>	
SEMANA 3 : Los objetos DOM (<i>Document Object Model</i>) del lenguaje <i>Java Script</i>	27
SEMANA 4 : XML y AJAX	35
Gestión de la capa modelo datos mediante el <i>framework Ibatis</i>	
SEMANA 5 : <i>Insert, Update, Delete</i> mediante el <i>framework Ibatis</i>	49
SEMANA 6 : Consultas a través del <i>framework Ibatis</i>	59
SEMANA 7 : Examen parcial de teoría	
SEMANA 8 : Examen parcial de laboratorio	
SEMANA 9 : Consultas dinámicas	67
SEMANA 10 : Manejo del patrón DAO mediante el <i>framework Ibatis</i>	75
SEMANA 11 : Paginación mediante el <i>framework Ibatis</i>	87
SEMANA 12 : Transacciones mediante el <i>framework Ibatis</i>	93
SEMANA 13 : Manejo de <i>Stored procedure</i> mediante el <i>framework Ibatis</i>	93
SEMANA 14 : Manejo del campo <i>BLOB</i> y <i>CLOC</i> mediante <i>framework Ibatis</i>	99
SEMANA 15 : Evaluación técnica	
SEMANA 16 : Sustentación comercial de sistemas	
SEMANA 17 : Examen final de teoría	

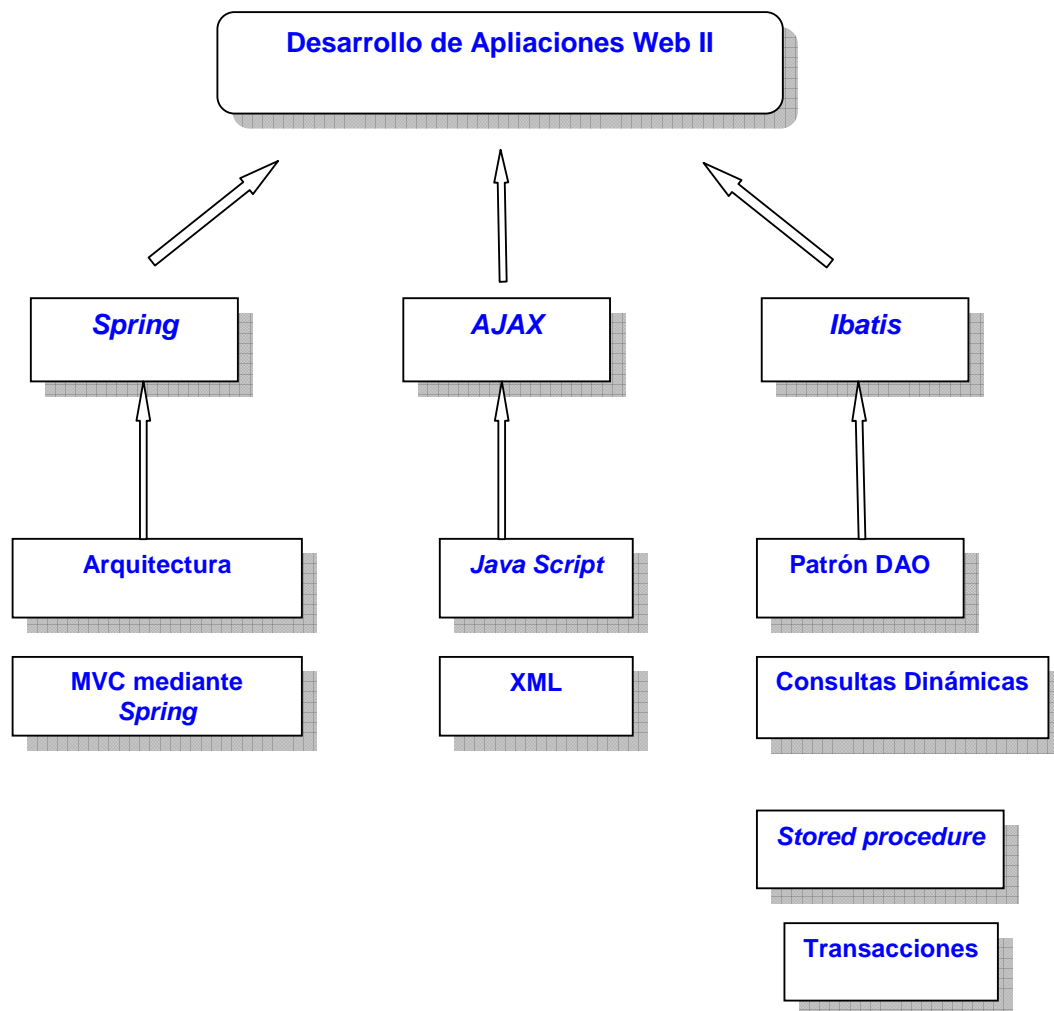
PRESENTACIÓN

Desarrollo de Aplicaciones WEB II pertenece a la línea de cursos de programación y se dicta en la carrera de Computación e Informática. El curso brinda un conjunto de herramientas del lenguaje *Java* y *frameworks* como *Ibatis* y *Spring* que permite a los alumnos utilizar para la implementación de aplicaciones web en *Java*.

El manual para el curso ha sido diseñado bajo la modalidad de unidades de aprendizaje, las que se desarrollan durante semanas determinadas. En cada una de ellas, hallará los logros, que debe alcanzar al final de la unidad; el tema tratado, el cual será ampliamente desarrollado; y los contenidos, que debe desarrollar, es decir, los subtemas. Por último, encontrará las actividades que deberá desarrollar en cada sesión, que le permitirán reforzar lo aprendido en la clase.

El curso es eminentemente práctico: consiste en desarrollar una aplicación web. En primer lugar, se inicia con crear programas utilizando el *framework* de *Spring*. Continúa con la presentación de nuevas tecnologías en la capa Vista utilizando la tecnología *AJAX*. Por último, se desarrolla el *framework* de *Ibatis* que facilitará el manejo de la capa del Modelo de Datos.

RED DE CONTENIDOS



UNIDAD DE
APRENDIZAJE**1**

SEMANA

1

GESTIÓN DE LA CAPA DE CONTROLADOR MEDIANTE *FRAMEWORK SPRING*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, al finalizar la sesión de clase, utilizando el controlador de *framework Spring* implementarán sus aplicaciones WEB.

TEMARIO

- Arquitectura del *framework* de *Spring*
- Características
- Inversión de Control
- Inyección de Dependencias

ACTIVIDADES PROPUESTAS

- Los alumnos configuran su aplicación utilizando el Patrón *Model View Controller* mediante el *framework* de *Spring*.

1. *Spring*

El *framework Spring* (también conocido simplemente como *Spring*) es un *framework* de código abierto de desarrollo de aplicaciones para la plataforma Java. La primera versión fue escrita por Rod Johnson, quien lo lanzó primero con la publicación de su libro *Expert One-on-One Java EE Design and Development* (Wrox Press, octubre 2002). También, hay una versión para la plataforma *.NET*, *Spring.net*.

El *framework* fue lanzado inicialmente bajo *Apache 2.0 License* en junio de 2003. El primer gran lanzamiento hito fue la versión 1.0, que apareció en marzo de 2004 y fue seguida por otros hitos en septiembre de 2004 y marzo de 2005.

A pesar de que el *framework Spring* no obliga a usar un modelo de programación en particular, se ha popularizado en la comunidad de programadores en Java al considerársele una alternativa y sustituto del modelo de *Enterprise JavaBean*. Por su diseño, el *framework* ofrece mucha libertad a los desarrolladores en Java y soluciones muy bien documentadas y fáciles de usar para las prácticas comunes en la industria.

Mientras que las características fundamentales de este *framework* pueden emplearse en cualquier aplicación hecha en Java, existen muchas extensiones y mejoras para construir aplicaciones basadas en web por encima de la plataforma empresarial de Java (*Java Enterprise Platform*).

Un *framework* mejorado basado en las ideas detrás de *Spring* es *Google Guice*.

Nota: a partir del 2009, las actualizaciones del producto (en su forma binaria) estarán disponibles únicamente para la última versión publicada del *framework*.

Para acceder a las actualizaciones en forma binaria para versiones anteriores habrá que pagar una suscripción. Sin embargo, estas actualizaciones estarán disponibles libremente (y gratuitamente) en forma de código fuente en los repositorios públicos del proyecto. No está planteado un cambio de licencia.

1.1 Características del *framework Spring*

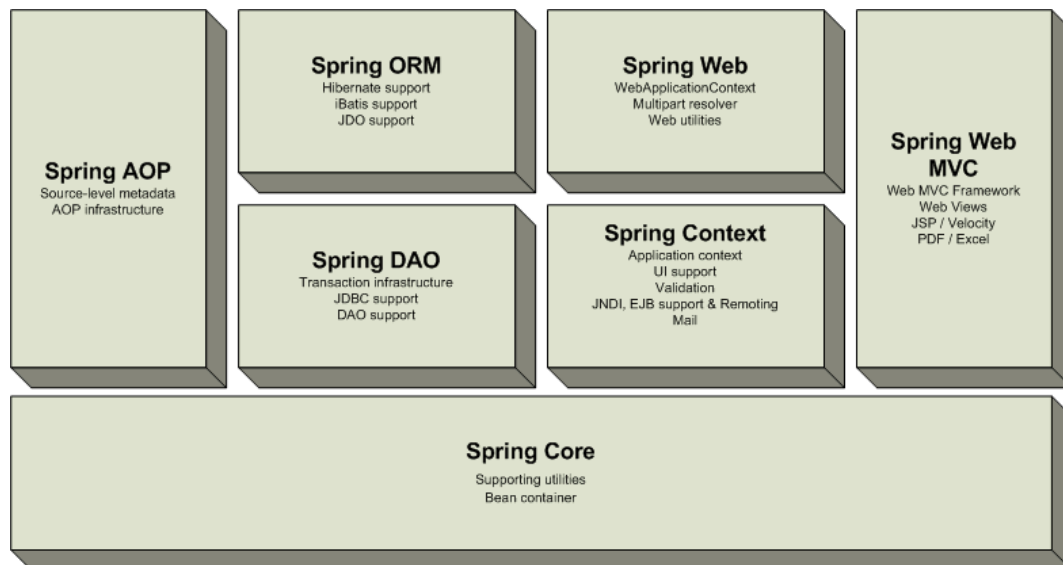
- La inicial motivación era facilitar el desarrollo de aplicaciones J2EE, promoviendo buenas prácticas de diseño y programación. En concreto, se trata de manejar patrones de diseño como *Factory*, *Abstract Factory*, *Builder*, *Decorator*, *Service Locator*, etc; que son ampliamente reconocidos dentro de la industria del desarrollo de *software*.
- Es código abierto.
- Enfoque en el manejo de objetos de negocio, dentro de una arquitectura en capas
- Una ventaja de *Spring* es su modularidad, pudiendo usar algunos de los módulos sin comprometerse con el uso del resto:
 - El *Core Container* o Contenedor de Inversión de Control (*Inversion of Control*, IoC) es el núcleo del sistema, responsable de la creación y configuración de los objetos.

- *Aspect-Oriented Programming Framework*, que trabaja con soluciones que son utilizadas en numerosos lugares de una aplicación, lo que se conoce como asuntos transversales (*cross-cutting concerns*).
- *Data Access Framework*, que facilita el trabajo de usar un API con JDBC, *Hibernate*, etc.
- *Transaction Management Framework*
- *Remote Access framework*. Facilita la existencia de objetos en el servidor que son exportados para ser usados como servicios remotos.
- *Spring Web MVC*. Maneja la asignación de peticiones a controladores y desde estos a las vistas. Implica el manejo y validación de formularios.
- *Spring Web Flow*
- *Spring Web Services*
- Una característica de *Spring* es que puede actuar integrador entre diferentes APIs (JDBC, JNDI, etc.) y *frameworks* (por ejemplo, entre *Struts* e *Ibatis*).

Spring proporciona lo siguiente:

- Una potente gestión de configuración basada en *JavaBeans*, aplicando los principios de Inversión de Control (IoC). Ello hace que la configuración de aplicaciones sea rápida y sencilla. Ya no es necesario tener *singletons* ni archivos de configuración, una aproximación consistente y elegante. Esta factoría de *beans* puede ser usada en cualquier entorno, desde *applets* hasta contenedores J2EE. Estas definiciones de *beans* se realizan en lo que se llama el contexto de aplicación.
- Una capa genérica de abstracción para la gestión de transacciones, lo que permite gestores de transacción enchufables (*pluggables*), y hace sencilla la demarcación de transacciones sin tratarlas a bajo nivel. Se incluyen estrategias genéricas para JTA y un único JDBC *DataSource*. En contraste con el JTA simple o EJB CMT, el soporte de transacciones de *Spring* no está atado a entornos J2EE.
- Una capa de abstracción JDBC que ofrece una significativa jerarquía de excepciones (evitando la necesidad de obtener de *SQLException* los códigos que cada gestor de base de datos asigna a los errores), simplifica el manejo de errores y reduce considerablemente la cantidad de código necesario.
- Integración con *Hibernate*, JDO e *Ibatis* SQL *Maps* en términos de soporte a implementaciones DAO y estrategias con transacciones. Especial soporte a *Hibernate* añade convenientes características de IoC, y soluciona muchos de los comunes problemas de integración de *Hibernate*. Todo ello cumple con las transacciones genéricas de *Spring* y la jerarquía de excepciones DAO.

- Funcionalidad AOP, totalmente integrada en la gestión de configuración de *Spring*. Se puede aplicar AOP a cualquier objeto gestionado por *Spring*, añadiendo aspectos como gestión de transacciones declarativa. Con *Spring* se puede tener gestión de transacciones declarativa sin EJB, incluso sin JTA, si se utiliza una única base de datos en un contenedor web sin soporte JTA.
- Un *framework* MVC (*Model-View-Controller*), construido sobre el núcleo de *Spring*. Este *framework* es altamente configurable vía interfaz y permite el uso de múltiples tecnologías para la capa vista como pueden ser JSP, *Velocity*, *Tiles*, *iText* o POI. De cualquier forma, una capa modelo realizada con Spring puede ser fácilmente utilizada con una capa web basada en cualquier otro *framework* MVC, como *Struts*, *WebWork* o *Tapestry*.



1.2 Historia

Los primeros componentes de lo que se ha convertido en *Spring Framework* fueron escritos por Rod Johnson en el año 2000, mientras trabajaba como consultor independiente para sus clientes en la industria financiera en Londres. Mientras escribía el libro *Expert One-on-one J2EE Design And Development (Programmer to programmer)*, Rod amplió su código para sintetizar su visión acerca de cómo las aplicaciones que trabajan con varias partes de la plataforma J2EE podían llegar a ser más simples y más consistentes que aquellas que los desarrolladores y compañías estaban usando por aquel entonces.

En el año 2001, los modelos dominantes de programación para aplicaciones basadas en web eran ofrecidas por el API *Java Servlet* y los *Enterprise JavaBeans*, ambas especificaciones creadas por Sun Microsystems en colaboración con otros distribuidores y partes interesadas que disfrutaban de gran popularidad en la comunidad *Java*. Las aplicaciones que no eran basadas en web, como las aplicaciones basadas en cliente o aplicaciones en

batch, podían ser escritas con base en herramientas y proyectos de código abierto o comercial que proveerán las características requeridas para aquellos desarrollos.

Rod Johnson es reconocido por crear un *framework* que está basado en las mejores prácticas aceptadas, y ello las hizo disponibles para todo tipo de aplicaciones, no sólo aquellas basadas en web. Estas ideas también estaban plasmadas en su libro y, tras la publicación, sus lectores le solicitaron que el código que acompañaba al libro fuera liberado bajo una licencia *open source*.

Se formó un pequeño equipo de desarrolladores que esperaba trabajar en extender el *framework* y un proyecto fue creado en *Sourceforge* en febrero de 2003. Después de trabajar en su desarrollo durante más de un año, lanzaron una primera versión (1.0) en marzo de 2004. Después de este lanzamiento, *Spring* ganó mucha popularidad en la comunidad *Java*, debido en parte al uso de *Javadoc* y de una documentación de referencia por encima del promedio de un proyecto de código abierto.

Sin embargo, el *framework Spring* también fue duramente criticado en el año 2004 y sigue siendo el tema de acalorados debates. Al tiempo en que se daba su primer gran lanzamiento, muchos desarrolladores y líderes de opinión vieron a *Spring* como un gran paso con respecto al modelo de programación tradicional; esto era especialmente cierto con respecto a *Enterprise JavaBeans*. Una de las metas de diseño del *framework Spring* es su facilidad de integración con los estándares J2EE y herramientas comerciales existentes. Esto quita en parte la necesidad de definir sus características en un documento de especificación elaborado por un comité oficial y que podría ser criticado.

El *framework Spring* hizo que aquellas técnicas que resultaban desconocidas para la mayoría de programadores se volvieran populares en un periodo muy corto de tiempo. El ejemplo más notable es la inversión de control. En el año 2004, *Spring* disfrutó de unas altísimas tasas de adopción y, al ofrecer su propio *framework* de programación orientada a aspectos (*aspect-oriented programming*, AOP), consiguió hacer más popular su paradigma de programación en la comunidad *Java*.

En 2005, *Spring* superó las tasas de adopción del año anterior como resultado de nuevos lanzamientos y más características fueron añadidas. El foro de la comunidad formada alrededor del *framework Spring* (*The Spring Forum*), que se inició a finales de 2004, también ayudó a incrementar la popularidad del *framework* y, desde entonces, ha crecido hasta llegar a ser la más importante fuente de información y ayuda para sus usuarios.

En el mismo año, los desarrolladores del proyecto abrieron su propia compañía para ofrecer soporte comercial y establecieron una alianza con BEA. En diciembre de 2005, la primera conferencia de *Spring* fue realizada en Miami y reunió a 300 desarrolladores en el transcurso de tres días, seguida por una conferencia en Amberes en junio de 2006, donde se concentraron más de 400 personas.

1.3 Inversión de Control

Es un concepto junto a unas técnicas de programación en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales, en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (*procedure calls*) o funciones.

Tradicionalmente, el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones. En su lugar, en la inversión de control, se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

En cierto modo, es una implementación del Principio de Hollywood, una metodología de diseño de *software*, cuyo nombre proviene de las típicas respuestas que se les dan a los actores *amateurs* en las audiciones que tienen lugar en la meca del cine: no nos llames a nosotros, nosotros te llamaremos a ti.

Es el principio subyacente a la técnica de Inyección de Dependencias, por lo que son términos frecuentemente confundidos.

El flujo habitual se da cuando es el código del usuario quien invoca a un procedimiento de una librería. La inversión de control sucede cuando es la librería la que invoca el código del usuario. Típicamente sucede cuando la librería es la que implementa las estructuras de alto nivel y es el código del usuario el que implementa las tareas de bajo nivel.

El uso de la interfaz y la aparición de los *frameworks* han popularizado este término. De hecho, es el concepto central del *framework* de *Spring*, ya que implementa un "Contenedor" que se encarga de gestionar los objetos (así como sus creaciones y destrucciones) de los objetos del usuario. Por tanto, las aplicaciones que utilicen el *framework* de *Spring* (no *Spring* propiamente dicho) utilizarán Inversión de Control.

1.4 Inyección de Dependencias

En Informática, Inyección de Dependencias (en inglés *Dependency Injection*, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto.

La forma habitual de implementar este patrón es mediante un "Contenedor DI" y objetos POJO. El contenedor inyecta a cada objeto los objetos necesarios según las relaciones plasmadas en un archivo de configuración.

Típicamente, este contenedor es implementado por un *framework* externo a la aplicación (como *Spring* o uno propio de la empresa), por lo cual en la aplicación también se utilizará Inversión de Control al ser el contenedor (almacenado en una biblioteca) quien invoque el código de la aplicación. Esta es la razón por la que los términos de Inversión de Control e Inyección de

Dependencias se confundan habitualmente entre sí, incluso se piense que son lo mismo.

Ejemplo 01

Dentro del modo de programación imperativa estamos acostumbrados a pensar en un flujo de control predefinido por el programador. Sin embargo, hay contextos (el caso típico son la moderna interfaz gráfica de usuario) en los que la aplicación cede el control a un API o sistema externo. Dicho sistema es el que determina los eventos que incidirán en el ciclo de vida de nuestra aplicación. Un ejemplo de inversión de control se encuentra en el modelo de programación orientado a eventos de *Swing* o AWT. De manera informal, la IoC viene representada por la frase "No me llames, yo te llamaré". Es el sistema externo el que llama a nuestra aplicación. Veamos diferentes tipos de IoC:

- Elevación de dependencia (*Dependency Lookup*)
- Inyección de dependencia (*Dependency Injection*)

Un ejemplo de "*Dependency lookup*" puede ser el que sucede en RMI.

```
String url = "rmi://localhost/";  
String nombreObjetoRemoto = "ob1"
```

```
ClaseRemota ob = (ClaseRemota) Naming.lookup (url +  
nombreObjetoRemoto);  
System.out.println( ob.getString() );
```

El objeto cliente, para acceder al servicio, primero lo identifica y después lo localiza por medio de una referencia (en el ejemplo, la referencia es "ob"); a continuación, hace la llamada al procedimiento. El inconveniente es el acoplamiento entre la capa cliente (*front*), la capa de acceso al servicio y la implementación del servicio.

Spring se identifica con una forma de IoC denominada Inyección de Dependencia, en la que el servicio se identifica y localiza por medio de mecanismos no programáticos, externos al código, como por ejemplo un archivo XML. Las dependencias con respecto a los servicios son explícitas y no están en el código. Con ello, se gana en facilidad de *test* y mantenimiento.

Algunas formas de inyección de dependencia son las siguientes:

- Inyección por medio de métodos *Setter*. En el ejemplo anterior de *Spring*, la clase *SpringappController* tiene un método *setLibreria()*, que es el que utiliza el *framework* para inyectar la propiedad "libreria".
- Inyección por medio de constructor. Por ejemplo, el *bean* "prod_02" tiene un constructor que admite dos argumentos:

```
<bean id="prod_02" class="org.ejemplo.Producto">  
  <constructor-arg value="AK" />  
  <constructor-arg value="600" />  
</bean>
```

Ejemplo 02

Core es una especie fábrica de *beans* sofisticada que permite quitar del código *java* la creación de objetos, inicialización y establecer dependencias entre *beans*. Toda esta información se guarda en un archivo XML y la fábrica de *beans* se encarga automáticamente de crear objetos, inicializar los *beans* y pasar unos a otros, de forma que se "vean".

Veamos un pequeño ejemplo con esta fábrica de *beans*. Primero, necesitamos hacer una clase *java* que sea un *bean*. Por ejemplo, la siguiente:

```
package com.chuidiang.pruebas.spring;

/**
 * Bean con los atributos correspondientes a los campos de la tabla de
 * base
 * de datos.
 * @author chuidiang
 */
public class Persona {
    int id;
    String nombre;
    String fechaNacimiento;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getFechaNacimiento() {
        return fechaNacimiento;
    }
    public void setFechaNacimiento(String fechaNacimiento) {
        this.fechaNacimiento = fechaNacimiento;
    }

    /** Para poder escribir el bean en pantalla de forma rápida */
    public String toString()
    {
        return ""+id+ " "+nombre+ " "+fechaNacimiento;
    }
}
```

Simplemente, se tienen tres atributos: un entero *id*, un *String* *nombre* y un *String* *fechaNacimiento*. No ponemos *Date* porque complicaría el ejemplo innecesariamente, ya que la factoría de *beans* necesitará convertir este *Date* a un *String* y viceversa. Esta conversión no está soportada por defecto y deberíamos implementarla.

Ahora creamos el archivo XML que leerá la factoría de *beans* para saber qué tiene que hacer. Este archivo XML, para este solo *bean*, es bastante simple y lo llamaremos *beans.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean id="Juan" class="com.chuidiang.pruebas.spring.Persona">
    <property name="id" value="1"/>
    <property name="nombre" value="Juan"/>
    <property name="fechaNacimiento" value="11/12/1943"/>
  </bean>

  <bean id="Pedro" class="com.chuidiang.pruebas.spring.Persona">
    <property name="id" value="2"/>
    <property name="nombre" value="Pedro"/>
    <property name="fechaNacimiento" value="1/1/1999"/>
  </bean>

</beans>
```

Cada *tag* *<bean>* se convertirá en un objeto de un *bean*. El *id* nos permite identificar ese objeto concreto. El *class* nos dice qué clase es el *bean*. Con este archivo de ejemplo, se harán dos *new* de *com.chuidiang.pruebas.spring.Persona*. A uno lo identificaremos como Juan y al otro como Pedro. Los atributos de cada uno de ellos se rellenarán con los *<property>* que se indican dentro de cada *tag <bean>*.

Se puede ver que las *fechaNacimiento* se rellenan con un *String* del estilo "11/12/1943". Esta es la conversión que necesita hacer la factoría de *beans*, que comentábamos antes y que no sabe hacer. Otras conversiones más sencillas, como "1" a *int*, sí sabe hacerlas.

El código *Java* que tenemos que hacer para crear objetos de la factoría de *beans* y obtener estos dos *beans* es el siguiente:

```
package com.chuidiang.pruebas.spring;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
```

```
public class PruebaBeanContainer {  
  
    public static void main(String[] args) {  
        Resource resource = new  
        FileSystemResource("beans.xml");  
        BeanFactory factory = new XmlBeanFactory(resource);  
        System.out.println(factory.getBean("Juan"));  
        System.out.println(factory.getBean("Pedro"));  
    }  
}
```

Primero, se crea un objeto `FileSystemResource` pasándole el archivo `beans.xml`. Por supuesto, se debe poner el *path* si no está en el directorio actual de ejecución. Luego, se crea un objeto de la factoría de *beans* `XmlBeanFactory`.

Pedimos y escribimos en pantalla ambos *beans*, "Juan" y "Pedro". Por supuesto, esto es un ejemplo muy sencillo. En el archivo `beans.xml`, podemos invocar constructores con parámetros para los *beans*, hacer que unos *beans* tengan referencias a otros, indicar que sólo puede haber un único objeto de un *bean* -patrón *Singleton*-, etc.

Resumen

📖 *Spring* soporta inyección de dependencias a través del constructor y a través de métodos *set*, pero se aconseja hacerlo a través de métodos *set*. Vamos a ver qué habría que hacer para utilizar inyección de dependencias con *Spring* a través de métodos *set*.

```
public class Clase {  
    private Clase2 atributo1;  
    public Clase(){  
    }  
    public void setAtributo1(Clase2 atributo1){  
        this.atributo1 = atributo1;  
    }  
}  
  
public class Clase2 {  
  
}
```

Declaramos nuestros *beans* en el contexto de la aplicación.

```
< bean id="atributo1" class="org.mipaquete.bean.Clase2">  
< /bean>  
< bean id="atributo" class="org.mipaquete.bean.Clase">  
< property name=" atributo1" ref="atributo1" / >  
< /bean>
```

📖 Inyección de Dependencias (en inglés *Dependency Injection*, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto.

📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🔗 <http://www.springframework.org/>

Aquí hallará las descargas del *framework Spring*.

**UNIDAD DE
APRENDIZAJE****1****SEMANA****2**

GESTIÓN DE LA CAPA DE CONTROLADOR MEDIANTE *FRAMEWORK SPRING*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, al finalizar la segunda sesión de clase, utilizando el controlador del *framework Spring*, implementarán sus aplicaciones web.

TEMARIO

- El Patrón de diseño MVC basado en el *framework Spring*

ACTIVIDAD PROPUESTA

- Los alumnos realizarán una aplicación con el patrón MVC haciendo uso del *framework Spring*.

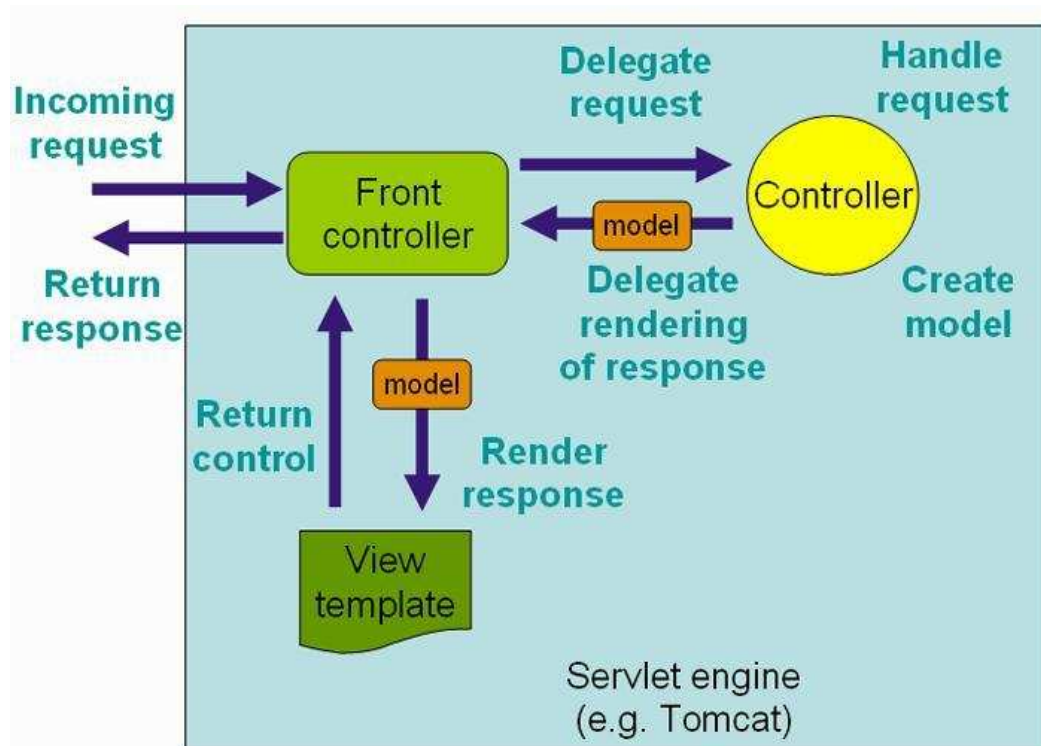
1. El *framework Spring MVC*

Para aplicar el patrón MVC se utiliza lo siguiente:

- Una capa de vista, formada de jsp, html, css, etiquetas personalizadas, etc.
- Una capa de modelo, que cuenta con las subcapas de servicios, persistencia (daos, etc.) y dominio (beans). Se forma mediante clases e interfaz Java.

Sin embargo, se necesita una capa de control, que se compone de lo siguiente:

- `DispatcherServlet`: es el controlador frontal, que recibe y gestiona todas las peticiones (request). Resulta oculto al programador y es creado por *Spring*.
- *Interface* `HandlerMapping`: analiza cada petición y determina el controlador que la gestiona. Podemos contar con varios manejadores, en función de las diversas estrategias de "mapeo" (basado en *cookies*, variables de sesión, etc.). En la mayor parte de los casos, nos sirve el manejador por defecto del *framework Spring*: *BeanNameUrlHandleMapping*.
- Controladores: manejan las peticiones de cada página. Cada controlador recibe las peticiones de su página correspondiente, delega en el dominio y recoge los resultados. Lo que hace es devolver un modelo a la vista que ha seleccionado (por medio del controlador frontal).



Lo que devuelve cada controlador es un objeto del tipo *ModelAndView*. Este objeto se compone de lo siguiente:

- Una referencia a la vista destino
- El modelo: un conjunto de objetos se utilizan para componer (*render*) la vista destino; por ejemplo, un *bean* Cliente o una lista de *beans* (clientes) que se ha obtenido de un DAO.

Configuración del archivo web.xml

Empezaremos con el clásico web.xml en el que se empieza definiendo el *Listener* que, ante el evento *contextInitialized*, cargará el contexto de aplicación.

```
<listener>
  <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

La localización de los archivos de definición de contexto de aplicación se puede hacer por medio del parámetro *contextConfigLocation*. Si no se indica nada, el *framework Spring* buscará un archivo con nombre *applicationContext.xml* en la carpeta WEB-INF.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>WEB-INF/xyz.xml, WEB-INF/abc.xml</param-value>
</context-param>
```

Lo siguiente es señalar el *servlet* que actuará como controlador frontal:

```
<servlet>
  <servlet-name>spring21</servlet-name>
  <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>spring21</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Dos aspectos importantes de la configuración del *DispatcherServlet*:

- El nombre que se da al servlet-mapping no es casual. El *framework Spring* buscará el archivo `spring21-servlet.xml`, que sirve para configurar el resto de controladores, *viewResolvers*, *urlMappings*, etc.
- El `url-pattern` indica los tipos de peticiones que aceptará, en nuestro ejemplo con las extensiones `.do`.

Configuración del archivo `spring-servlet.xml`

En este ejemplo, se utiliza `spring21-servlet.xml` en la carpeta `WEB-INF`, ya que el `servlet-name` de `web.xml` es `spring21`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<!-- Definición de contexto de aplicación para Controlador frontal y resto de controladores -->
<beans>
    <!-- Controlador de index (consulta, insert, etc) -->
    <bean id="indexController" class="com.controlador.IndexController">
        <property name="servicioCliente" ref="servicioCliente" />
    </bean>
    <!-- Controlador de index (borrar) -->
    <bean id="borrarController" class="com.controlador.BorrarController">
        <property name="servicioCliente" ref="servicioCliente" />
    </bean>

    <!-- Indico que las vistas se toman de /WEB-INF/jsp/ y que tendrán extensión jsp --
    >
    <bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <!-- Las llamadas a .do se dirigen a su controlador correspondiente -->
    <bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /inicio.do=indexController
                /borrar.do=borrarController
            </value>
        </property>
    </bean>
</beans>
```

En el ejemplo, tenemos dos controladores: uno que hace las inserciones, actualizaciones y consultas; otro que hace el borrado. Esto se verá más adelante. Normalmente, un formulario está asociado a un controlador, señalando la asociación en el *action* del formulario. En nuestro ejemplo, cada controlador tiene como atributo un *bean* servicioCliente, que es definido en el applicationContext.xml.

El viewResolver indica dónde están las vistas (normalmente JSPs) que son invocadas por el controlador frontal. En el ejemplo, las vistas se toman de la carpeta /WEB-INF/jsp y tienen la extensión jsp.

El *bean* urlMapping indica el mapeo de peticiones y controladores. En nuestro ejemplo, cada petición .do tiene su correspondiente *bean* controlador definido más arriba. Obsérvese que esto debe ser coherente con los url-mapping del servlet frontal en el web.xml. Podríamos evidentemente usar otras extensiones, como *.html o *.form, o indicar un directorio (app/*.do) o varios (*.do).

Configuración del archivo applicationContext.xml

Para terminar con la configuración, es necesario tener en cuenta el tradicional application context de *framework* Spring. En nuestro ejemplo, se llama applicationContext.xml y está en WEB-INF; esto hace que no sea necesario utilizar contextConfigLocation en el web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans SYSTEM "spring-beans.dtd">

<!-- CUANDO HAYA CONEXION A INTERNET: -->
<!--
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:p="http://www.springframework.org/schema/p"
xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:context="http://www.springframework.org/schema/context"
xmlns:jee="http://www.springframework.org/schema/jee"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
          http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
          http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd
          http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
          http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
-->
<beans>

    <!-- Las propiedades del dataSource tienen como valor properties -->
```

```

    <bean id="dataSource" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource"
    destroy-method="close">
        <property name="driverClassName" value="{jdbcdriverClassName}"/>
        <property name="url" value="{jdbcurl}"/>
        <property name="username" value="{jdbcusername}"/>
        <property name="password" value="{jdcpasswd}"/>
    </bean>
    <!-- El DAO que tiene como atributo el dataSource -->
    <bean id="DAOCliente" class="com.persistencia.DAOClienteSpring">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- El Servicio de Cliente tiene como atributo el DAO -->
    <bean id="servicioCliente" class="com.servicio.ServicioCliente">
        <property name="dao" ref="DAOCliente"/>
    </bean>

    <!-- Las propiedades para el dataSource -->
    <bean id="propertyConfigurer"

    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>WEB-INF/configuracion.properties</value>
            </list>
        </property>
    </bean>
</beans>

```

Algunos puntos que se deben destacar son los siguientes:

- Se ha puesto en comentario la referencia al XML Schema, por si falla la conexión a Internet, ya que si no hay conexión nos devolverá un mensaje de error que indica que no encuentra sentido a la etiqueta beans. Cuando hay estos fallos, lo mejor es usar el dtd spring-beans.dtd y ponerlo en el mismo directorio donde está el applicationContext.xml.
- La organización de los beans en spring21-servlet.xml y applicationContext.xml refleja la arquitectura de nuestro proyecto.
 - En spring21-servlet.xml, los controladores tenían como atributo el bean servicioCliente. Este bean es un intermediario entre el controlador y los DAOs. El controlador llama al servicio y este delega en el DAO.
 - En applicationContext.xml, se puede ver que el servicioCliente tiene como atributo el DAO.
 - El DAO tiene como atributo el dataSource.
 - El dataSource tiene como atributos datos que son definidos por un archivo properties. Por ejemplo, jdbc.url se refiere a jdbc:mysql://localhost:3306/proactiv_prueba?autoReconnect=true. En


el archivo configuracon.properties de la carpeta WEB-INF, se encuentra la siguiente definición:

```
jdbc.driverClassName=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/proactiv_prueba?autoReconnect=true  
jdbc.username=usuario  
jdbc.password=password
```

En resumen:

- Los controladores (en spring21-servlet.xml) hacen referencia al servicio servicioCliente.
- En applicationContext.xml, se puede ver que el servicioCliente tiene como atributo el DAO.
- El DAO tiene como atributo el dataSource, que se configura por un archivo properties.


Resumen

 Registro del Controlador del *framework Spring*

```
<beanname="/holamundo.html"  
    class="es.lycka.holamundo.control.HolaMundoController"/>
```

 Resumen de la secuencia de la configuración inicial

- El servidor de aplicaciones ejecuta el evento ContextInited.
- Este evento invoca al ContextLoaderListener (señalado en web.xml) y crea el contexto de aplicación (applicationContext.xml).
- Inicialización del *servlet* frontal (DispatcherServlet) y creación de su contexto (spring21-servlet.xml)
- El controlador central busca e inicializa componentes como ViewResolver o HandlerMapping. Si no los encuentra, inicializa versiones por defecto.

 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

 <http://www.java2s.com/Code/Java/Spring/CatalogSpring.htm>

Aquí hallará proyectos en Spring.

**UNIDAD DE
APRENDIZAJE**

2

SEMANA

3

GESTIÓN DE LA CAPA VISTA MEDIANTE AJAX

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos implementan una aplicación web utilizando AJAX en la capa vista.

TEMARIO

- Objetos DOM (*Document Object Model*) del lenguaje *Java Script*

ACTIVIDADES PROPUESTAS

- Los alumnos realizarán una aplicación con objetos DOM.

1. Document Object Model

El *Document Object Model* (una traducción al español no literal, pero apropiada, podría ser Modelo en Objetos para la representación de Documentos), abreviado DOM, es esencialmente un modelo computacional a través del cual los programas y scripts pueden acceder y modificar dinámicamente el contenido, estructura y estilo de los documentos HTML y XML. Su objetivo es ofrecer un modelo orientado a objetos para el tratamiento y manipulación en tiempo real (o de forma dinámica) a la vez que de manera estática de páginas de internet.

El responsable del DOM es el consorcio W3C (*World Wide Web Consortium*). El DOM es una API para acceder, añadir y cambiar dinámicamente contenido estructurado en documentos con lenguajes como *ECMAScript* (El lenguaje *Java Script*).

2. Desarrollo del DOM

La primera vez que el DOM se utilizó fue con el navegador Netscape en su versión 2.0. Este DOM se conoce también como el modelo básico, o el DOM.Nivel 0. Internet Explorer 3.0 fue el primer navegador de Microsoft que utilizó este nivel. Netscape 3.0 empezó a utilizar *rollovers*. Microsoft empezó a usar *rollovers* en Internet Explorer 4.0. Netscape 4.0 agregó la capacidad de detectar eventos ocurridos en el ratón y el teclado. Una característica de este navegador fue el uso de capas (*layers*). Sin embargo, esta capacidad se ha eliminado en los navegadores creados posteriormente.

En Internet Explorer 4.0 todos los elementos de una página web se empezaron a considerar objetos computacionales con la capacidad de ser modificados. Debido a las diferencias en estos navegadores, el *World Wide Web Consortium* emitió una especificación denominada DOM.Nivel 1 en el mes de octubre de 1998, en la cual se consideraron las características y manipulación de todos los elementos existentes en los archivos HTML y XML. En noviembre del año 2000, se emitió la especificación del DOM.Nivel 2. En esta especificación, se incluyó la manipulación de eventos en el navegador, la capacidad de interacción con CSS, y la manipulación de partes del texto en las páginas de la web. DOM.Nivel 3 se emitió en abril del 2004; utiliza la DTD (Definición del tipo de documento) y la validación de documentos.

Problemas de compatibilidad

La guerra entre navegadores que existe entre el Netscape Navigator y el Internet Explorer de Microsoft con otras compañías crea graves problemas para los programadores de páginas web, ya que, aunque todos los navegadores utilizan el lenguaje *Java Script* como uno de los lenguajes de programación, los objetos no se comportan de la misma forma, lo que obliga con frecuencia a programar las páginas en más de una versión, una para el Netscape, o Firefox, otra para Internet Explorer, otra para Safari, Opera, etc; en suma, no todas las versiones de un mismo navegador se comportan igual.

El W3C, el consorcio encargado de definir los estándares de la web, decidió crear un modelo de objetos único, el DOM, para que todos los fabricantes pudieran adoptarlo, facilitando la compatibilidad plena entre ellos.

No obstante, Microsoft ha añadido su propia extensión al DOM, creando problemas de interoperabilidad para los navegadores web.

Como el navegador de Microsoft, Internet Explorer, es desde el año 2002 el navegador web estándar de facto, esto lleva a un problema real a los

desarrolladores de navegadores más comprometidos con los estándares, como Mozilla. Si adoptan las extensiones de Microsoft al DOM, se arriesgan a perder credibilidad en sus llamadas a que los sitios web respeten el estándar, y si no lo hacen, se arriesgan a alienar a sus usuarios por la pérdida de compatibilidad con casi todos los sitios web que utilizan las extensiones de Microsoft. Los críticos ven esta actitud como otro caso de aplicación de la táctica de Microsoft de "adoptar, extender y extinguir". Esto puede ser considerado irónico, debido a que tanto Microsoft como Netscape fueron responsables de proporcionar extensiones no estándares en la "carrera armamentística" por el control del estándar, y Mozilla surgió como una iniciativa de Netscape.

La opinión general parece indicar que esto cambiará sólo si nuevos navegadores que respeten los estándares ganan una cuota de mercado significativa en la web, de forma que el uso de extensiones no estándares se convierta en un problema comercial para los autores de los sitios web que las usen. Esto ya está pasando con Mozilla Firefox, el cual, desde hace varios años, viene incrementando su utilización en PCs en vez de Internet Explorer de Microsoft.

3. Estableciendo referencias a objetos

El DOM define la manera en que objetos y elementos se relacionan entre sí en el navegador y en el documento. Cualquier lenguaje de programación adecuado para el desarrollo de página web puede ser utilizado. En el caso del lenguaje *JavaScript*, cada objeto tiene un nombre, el cual es exclusivo y único. Cuando existe más de un objeto del mismo tipo en un documento web, estos se organizan en un vector. Algunos nombres de objetos comunes son los siguientes:

Nombre	Traducción
window	ventana
document	documento
body	cuerpo
div	division
p	parrafo

A los objetos, se les puede asignar una identificación, la cual se puede utilizar para hacer referencia a estos, por ejemplo:

Para hacer referencia a elementos del mismo tipo, los mismos que están organizados en un vector, se pueden utilizar puntos decimales de la siguiente manera.

```
document.div[0]  
document.div["Juan"]  
document.div.Juan
```

Donde el elemento "Juan" es el primer elemento del vector de elementos del tipo <div>En forma alternativa se puede únicamente usar el identificador del elemento.

```
Juan
```

También se puede usar la función "getElementById".

```
document.getElementById("Juan")
```

4. Manipulando las propiedades y funciones de objetos

Los objetos computacionales, de la misma forma que cualquier objeto de la vida real, tienen propiedades. Algunos ejemplos de propiedades de objetos de la vida real son dimensiones, color y peso.

```
Objeto.propiedad = valor;  
  
//por ejemplo para el objeto "Juan"  
  
Juan.color = rojo;
```

La manipulación de objetos sigue los mismos principios que en el lenguaje de programación que se esté utilizando. Una de las características de estos objetos es la función para la cual están diseñados, de hecho la mayoría de las ocasiones tienen más de una función. En el lenguaje *Java Script* muchas funciones para cada uno de los objetos, incluyendo el navegador y la ventana que lo contiene, han sido definidas previamente; adicionalmente el usuario puede definir funciones de acuerdo con sus necesidades. Por ejemplo, el código que sigue añade una nueva función al documento utilizado para crear una página web.

```
function comeLaLetraA(Texto){  
  var TextoNuevo = "";  
  while(letras en el Texto recibido){  
    //lee la siguiente letra  
    //si esta letra no es "a" añadela al nuevo texto  
  }  
  return TextoNuevo;  
}
```

5. Manipulando Eventos

Un evento, desde del punto de vista computacional, ocurre cuando alguna situación cambia en la computadora, como la posición del ratón, la opresión de alguna tecla, los contenidos de alguna de las memorias, la condición de la pantalla, etc. En la creación de páginas web, estos eventos representan la interacción de la computadora con el usuario.

Cuando algunos de estos eventos ocurren, como la presión de algún botón del ratón, es deseable que la computadora responda de alguna manera. Esa es la razón por la que existen "*event handlers*" (encargados de manipular eventos) los cuales son objetos que responden a eventos. Una manera de añadir eventos en el DOM utilizando el lenguaje *Java Script* es la siguiente:

```
<element onevent="script">....</element>  
Por ejemplo:  
<div id="midivision" onclick=comeLaLetraA("mamacita apiadate de mi  
ahora")>  
Aquí va otro texto  
</div>
```

Otra forma de manipular eventos en el lenguaje *Java Script* al crear páginas para la web es tratándolos como propiedades de los elementos que forman la página, por ejemplo:

```
object.event = funcion;  
//como puede ser:  
document.mydivision.onclick = hazAlgo;  
// tambien:  
document.getElementById ("mydivision").onclick = hazAlgo;
```

En el DOM se considera que un evento se origina en el exterior de la página web y se propaga de alguna manera hasta los elementos internos de la página. Un posible ejemplo de esta propagación es el siguiente:

```
EVENTO → Ventana → Document → HTML → BODY → DIV → DESTINO  
RESPUESTA → DIV → BODY → HTML → Document → Window → EVENTO
```

Siguiendo esta idea, se establecen tres etapas, captura es cuando el evento se esta trasladando a su destino. Segundo que es cuando llega a su destino. Este destino es el objeto en el cual se va a crear una reacción a este evento. Finalmente, la etapa de burbujeo que es cuando el evento "regresa" a su posición original.

Ciertos objetos pueden estar al pendiente de ciertos de eventos. Para hacer esto, el objeto añade un "oyente de eventos" con la función *addEventListener*. Cuando el evento ocurra, alguna función determinada se lleva a cabo. En este proceso se indica en qué momento la función se lleva a cabo, ya sea en la etapa de captura o

en la etapa de burbujeo. Este momento se indica con la palabra *true* si debe ocurrir en la etapa de captura o *false* si debe ocurrir en la etapa de burbujeo. En el lenguaje *Java Script* se escribe de la siguiente manera:

```
objeto.addEventListener(evento, funcion, momento);

por ejemplo:

document.getElementById("mydivision").addEventListener("click",
hazAlgo, false);
```

6. Acceso directo a los nodos

Los métodos presentados hasta el momento permiten acceder a cualquier nodo del árbol de nodos DOM y a todos sus atributos. Sin embargo, las funciones que proporciona DOM para acceder a un nodo a través de sus padres obligan a acceder al nodo raíz de la página y después a sus nodos hijos y a los nodos hijos de esos hijos, y así sucesivamente hasta el último nodo de la rama terminada por el nodo buscado.

Cuando se trabaja con una página web real, el árbol DOM tiene miles de nodos de todos los tipos. Por este motivo, no es eficiente acceder a un nodo descendiendo a través de todos los ascendentes de ese nodo.

Para solucionar este problema, DOM proporciona una serie de métodos para acceder de forma directa a los nodos deseados. Los métodos disponibles son `getElementsByTagName()`, `getElementByName()` y `getElementById()`.

6.1 Función `getElementsByTagName()`

La función `getElementsByTagName()` obtiene todos los elementos de la página XHTML cuya etiqueta sea igual que el parámetro que se le pasa a la función. El siguiente ejemplo muestra cómo obtener todos los párrafos de una página XHTML:

```
var parrafos = document.getElementsByTagName("p");
```

El valor que devuelve la función es un array con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. En realidad, el valor devuelto no es de tipo array *normal*, sino que es un objeto de tipo *NodeList*. De este modo, el primer párrafo de la página se puede obtener de la siguiente manera:

```
var parrafos = document.getElementsByTagName("p");
var primerParrafo = parrafos[0];
```

De la misma forma, se pueden recorrer todos los párrafos de la página recorriendo el array de nodos devuelto por la función:

```
var parrafos = document.getElementsByTagName("p");
for(var i=0; i<parrafos.length; i++) {
```



```
    var parrafo = parrafos[i];  
  }
```

La función `getElementsByName()` se puede aplicar de forma recursiva sobre cada uno de los nodos devueltos por la función. En el siguiente ejemplo, se obtienen todos los enlaces del primer párrafo de la página:

```
var parrafos = document.getElementsByTagName("p");  
var primerParrafo = parrafos[0];  
var enlaces = primerParrafo.getElementsByTagName("a");
```

6.2 Función `getElementsByName()`

La función `getElementsByName()` obtiene todos los elementos de la página XHTML cuyo atributo *name* coincida con el parámetro que se le pasa a la función. En el siguiente ejemplo, se obtiene directamente el único párrafo de la página que tiene el nombre indicado:

```
var parrafoEspecial = document.getElementsByName("especial");  
  
<p name="prueba">...</p>  
<p name="especial">...</p>  
<p>...</p>
```

Normalmente, el atributo *name* es único para los elementos HTML que lo incluyen, por lo que es un método muy práctico para acceder directamente al nodo deseado. En el caso de los elementos HTML *radiobutton*, el atributo *name* es común a todos los *radiobutton* que están relacionados, por lo que la función devuelve una colección de elementos.

Internet Explorer 7 y sus versiones anteriores no implementan de forma correcta esta función, ya que también devuelven los elementos cuyo atributo *id* sea igual al parámetro de la función.


6.3 Función `getElementById()`


La función `getElementById()` es la función más utilizada cuando se desarrollan aplicaciones web dinámicas. Se trata de la función preferida para acceder directamente a un nodo y para leer o modificar sus propiedades.

La función `getElementById()` devuelve el elemento XHTML cuyo atributo *id* coincide con el parámetro indicado en la función. Como el atributo *id* debe ser único para cada elemento de una misma página, la función devuelve únicamente el nodo deseado.

```
var cabecera = document.getElementById("cabecera");  
  
<div id="cabecera">  
  <a href="/" id="logo">...</a>  
</div>
```

Resumen

 DOM proporciona una serie de métodos para acceder de forma directa a los nodos deseados. Los métodos disponibles son `getElementsByTagName()`, `getElementsByName()` y `getElementById()`.

 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

 <http://es.wikipedia.org/wiki/DOM>

Aquí hallará definiciones sobre *Document Object Model*.

**UNIDAD DE
APRENDIZAJE****2****SEMANA****4**

GESTIÓN DE LA CAPA VISTA MEDIANTE AJAX

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos implementan una aplicación web utilizando AJAX en la capa vista.

TEMARIO

- XML
- AJAX

ACTIVIDADES PROPUESTAS

- Los alumnos implementarán una Aplicación con el uso de AJAX.

1. AJAX

AJAX, acrónimo de *Asynchronous, Java Script* y XML, es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (*Rich Internet Applications*). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma, es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que significa aumentar la interactividad, velocidad y usabilidad en las aplicaciones.

AJAX es una tecnología no sincronizada con el servidor, en el sentido de que los datos adicionales se requieren al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. El lenguaje *Java Script* es el lenguaje interpretado (*scripting language*) en el que normalmente se efectúan las funciones de llamada de AJAX mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido no sincronizado esté formateado en XML.

AJAX es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores dado que está basado en estándares abiertos como el lenguaje *Java Script* y *Document Object Model* (DOM).

1.1 XMLHttpRequest (XHR)

También referida como XMLHTTP (*Extensible Markup Language / Hypertext Transfer Protocol*), es una interfaz empleada para realizar peticiones HTTP y HTTPS a servidores de aplicaciones Web. Para los datos transferidos se usa cualquier codificación basada en texto, incluyendo texto plano, XML, JSON, HTML y codificaciones particulares específicas. La interfaz se presenta como una clase de la que una aplicación cliente puede generar tantos objetos como necesite para manejar el diálogo con el servidor.

El uso más popular, si bien no el único, de esta interfaz es proporcionar contenido dinámico y actualizaciones asíncronas en páginas WEB mediante tecnologías construidas sobre ella, como por ejemplo AJAX.

1.2 Evolución

La primera versión del XMLHttpRequest fue desarrollada por Microsoft, que la introdujo en la versión 5.0 de Internet Explorer. Esta primera versión se publicó utilizando un objeto ActiveX, lo que significa que puede ser utilizada desde cualquier entorno de desarrollo de software con soporte para esta tecnología, es decir, la práctica totalidad de plataformas generalistas de desarrollo para Microsoft Windows. Microsoft ha seguido manteniendo y actualizando esta tecnología incluyendo la funcionalidad dentro del *Microsoft XML Parser* (MSXML) en sus sucesivas versiones. A partir de la versión 7 de Internet Explorer, la interfaz se ofrece de manera integrada. Al ser integrada, el acceso a la interfaz se realiza enteramente con objetos (JavaScript o VBScript) proporcionados por el navegador y no mediante librerías externas.

El proyecto Mozilla incorporó la primera implementación integrada de XMLHttpRequest en la versión 1.0 de la Suite Mozilla en 2002. Esta

implementación sería seguida por Apple a partir de Safari 1.2, Konqueror, Opera Software a partir del Opera 8.0 e iCab desde la versión 3.0b352.

El *World Wide Web Consortium* presentó el 27 de septiembre de 2006 el primer borrador de una especificación estándar de la interfaz. La versión actual de 15 de abril de 2008, etiquetada como borrador final (*last call working draft*), es el resultado de varias revisiones.

Mientras no se alcance una versión definitiva, los desarrolladores de aplicaciones Web deberán tener en cuenta las diferencias entre implementaciones o bien utilizar paquetes o *frameworks* que realicen esta función.

El 25 de febrero de 2008, se publicó la primera versión de la especificación XMLHttpRequest nivel 2. Esta nueva especificación, que se inicia antes de haber publicado la versión definitiva de la interfaz, pretende añadir nuevas funciones como peticiones entre dominios (*cross-site*), eventos de progreso y manejo de flujos de bytes (*streams*) tanto para el envío como para la recepción.

1.3 Implementación

El XMLHttpRequest se presenta encapsulado en una clase. Para utilizarlo, la aplicación cliente debe crear un nuevo objeto mediante el constructor adecuado. Es posible realizar peticiones sincronizadas y no sincronizadas al servidor; en una llamada asíncrona el flujo de proceso no se detiene a esperar la respuesta como se haría en una llamada sincronizada, si no que se define una función que se ejecutará cuando se complete la petición: un manejador de evento.

XMLHttpRequest es una interfaz para realizar llamadas mediante HTTP, por lo que es recomendable un buen conocimiento de este protocolo. Es importante el manejo correcto de la memoria *cache* en el servidor HTTP, en los *proxy cache* intermedios y en el navegador web.

Otro elemento importante es el manejo de juegos de caracteres, la codificación y decodificación de texto y su identificación mediante cabeceras HTTP y MIME. El estándar XMLHttpRequest recomienda UTF-8 para la codificación de cadenas de texto. La codificación particular de los datos transmitidos se determina según el siguiente algoritmo, utilizando la primera opción que corresponda.

Si los datos transmitidos son XML o HTML, y así se identifica mediante la correspondiente cabecera Content-Type de HTTP, la codificación se detectará basándose en las reglas estándar de XML o HTML según corresponda.

Si la cabecera HTTP especifica un tipo MIME mediante Content-Type e identifica un *charset*, se utiliza dicho *charset*.

Es importante tener esto en cuenta en entornos donde se mezclen varias codificaciones; por ejemplo, pueden producirse errores de visualización de caracteres al incorporar funcionalidad AJAX a una página WEB codificada con ISO 8859-1.

1.4 Implementación

Atributo	Descripción
readyState	Devuelve el estado del objeto como sigue. 0 = sin inicializar, 1 = abierto, 2 = cabeceras recibidas, 3 = cargando y 4 = completado.
responseBody	(Nivel 2) Devuelve la respuesta como un array de bytes.
responseText	Devuelve la respuesta como una cadena.
responseXML	Devuelve la respuesta como XML. Esta propiedad devuelve un objeto documento XML, que puede ser examinado usando las propiedades y métodos del árbol del DOM.
Status	Devuelve el estado como un número (por ejemplo 404 para "Not Found" y 200 para "OK").
statusText	Devuelve el estado como una cadena (por ejemplo "Not Found" o "OK").

1.5 Métodos

Método	Descripción
abort()	Cancela la petición en curso.
getAllResponseHeaders()	Devuelve el conjunto de cabeceras HTTP como una cadena.
getResponseHeader(nombreCabecera)	Devuelve el valor de la cabecera HTTP especificada.
open (método, URL [, asíncrono [, nombreUsuario [, clave]]])	<p>Especifica el método, URL y otros atributos opcionales de una petición.</p> <p>El parámetro de método puede tomar los valores "GET", "POST", o "PUT" ("GET" y "POST" son dos formas para solicitar datos, con "GET" los parámetros de la petición se codifican en la URL y con "POST" en las cabeceras de HTTP).</p> <p>El parámetro <i>URL</i> puede ser una URL relativa o completa.</p> <p>El parámetro <i>asíncrono</i> especifica si la petición será gestionada sincronizada o no. Un valor <i>true</i> indica que el proceso del script continúa después del método send(), sin esperar a la respuesta, y <i>false</i> indica que el script se detiene hasta que se complete la operación, tras lo cual se reanuda la ejecución.</p>

	En el caso asíncrono, se especifican manejadores de eventos, que se ejecutan ante cada cambio de estado y permiten tratar los resultados de la consulta una vez que se reciben, o bien gestionar eventuales errores.
<code>send([datos])</code>	Envía la petición.
<code>setRequestHeader(etiqueta, valor)</code>	Añade un par etiqueta/valor a la cabecera HTTP a enviar.

1.6 Eventos

Propiedad	Descripción
<code>onreadystatechange</code>	Evento que se dispara con cada cambio de estado
<code>onabort</code>	(Nivel 2) Evento que se dispara al abortar la operación
<code>onload</code>	(Nivel 2) Evento que se dispara al completar la carga
<code>onloadstart</code>	(Nivel 2) Evento que se dispara al iniciar la carga
<code>onprogress</code>	(Nivel 2) Evento que se dispara periódicamente con información de estado

En el cuadro anterior, se ha marcado como Nivel 2 los elementos que no pertenecen a la especificación estándar pero sí están en la futura especificación Nivel 2.

La propuesta oficial de W3C no incluye propiedades y eventos presentes en otras implementaciones, algunos de ellos ampliamente difundidos, como los eventos *onload*, *onerror*, *onprogress*, *onabort* y *ontimeout*. Algunos de ellos sí son recogidos por la nueva especificación Nivel 2, como puede verse en el cuadro anterior.

1.7 Implementación de un Caso

1.7.1 Archivo JSP

```

<tr>
    <td>
        Ingrese el código de un Vendedor
    </td>
</tr>
<tr>
    <td>
        Teclee aquí:
        <input type="text" id="txtCodigo" name="txtCodigo"
onkeyup="buscaVendedor();" />
    </td>
</tr>
<tr>
    <td>
        Nombre<input type="text" readonly id="txtNombre" />
    </td>
</tr>

```

```

        <td>Edad<input type="text" readonly id="txtEdad"></td>
    </tr>
    <tr>
        <td>Distrito<input type="text" readonly id="txtDistrito"></td>
    </tr>
    <tr>
        <td>Estado Civil<input type="text" readonly id="txtEstado"></td>
    </tr>
</table>

</body>
</html>

```

1.7.2 Archivo *Java Script*

```

function foco(){
    document.getElementById("txtCodigo").focus();
}

function limpiar(){
    document.getElementById("txtCodigo").value="";
}

var req;

function buscaVendedor(){
    var codigo = document.getElementById("txtCodigo");
    var url = "/Leccion_ajax_teoría/AjaxResponseServletConXML?codigo="+
    escape(codigo.value);

    if(window.XMLHttpRequest){
        req = new XMLHttpRequest();
    }else if(window.ActiveXObject){
        req = new ActiveXObject("Microsoft.XMLHTTP");
    }

    req.open("GET", url, true);
    req.onreadystatechange = callback;
    req.send(null);

}

function callback(){
    if(req.readyState == 4){
        if(req.status == 200){

            if (window.XMLHttpRequest){
                nonMSPopulate();
            }else if (window.ActiveXObject){
                msPopulate();
            }

        }
    }
    limpiar();
}

```



```
}

function nonMSPopulate(){
    var resp = req.responseText;
    var parser = new DOMParser();
    var dom = parser.parseFromString(resp,"text/xml");

    nombre = dom.getElementsByTagName("nombre");
    var txtNombre = document.getElementById('txtNombre');
    txtNombre.value=nombre[0].childNodes[0].nodeValue;

    edad = dom.getElementsByTagName("edad");
    var txtEdad = document.getElementById('txtEdad');
    txtEdad.value=edad[0].childNodes[0].nodeValue;

    distrito = dom.getElementsByTagName("distrito");
    var txtDistrito = document.getElementById('txtDistrito');
    txtDistrito.value=distrito[0].childNodes[0].nodeValue;

    estado = dom.getElementsByTagName("estado");
    var txtEstado = document.getElementById('txtEstado');
    txtEstado.value=estado[0].childNodes[0].nodeValue;
}

function msPopulate(){
    var resp = req.responseText;

    var xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async="false";
    xmlDoc.loadXML(resp);

    nombre = xmlDoc.getElementsByTagName("nombre");
    var txtNombre = document.getElementById('txtNombre');
    txtNombre.value=nombre[0].firstChild.data;

    edad = xmlDoc.getElementsByTagName("edad");
    var txtEdad = document.getElementById('txtEdad');
    txtEdad.value=edad[0].firstChild.data;

    distrito = xmlDoc.getElementsByTagName("distrito");
    var txtDistrito = document.getElementById('txtDistrito');
    txtDistrito.value=distrito[0].firstChild.data;

    estado = xmlDoc.getElementsByTagName("estado");
    var txtEstado = document.getElementById('txtEstado');
    txtEstado.value=estado[0].firstChild.data;
}
```

1.7.3 Archivo Servlet

```
package demo.servlets;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import demo.model.service.VendedorService;
import demo.recursos.beans.BeanVendedor;

public class AjaxResponseServletConXML extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public void doGet(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

        VendedorService servicio = new VendedorService();

        try {
            String codigo = request.getParameter("codigo");
            StringBuffer returnXML = new StringBuffer();

            if (codigo != null) {
                BeanVendedor bean =
servicio.obtenerPorPK(Integer.parseInt(codigo));

                returnXML.append("<vendedor>");
                returnXML.append("<nombre>" + bean.getNombre()
+ " " + bean.getApellido()+ "</nombre>");

                returnXML.append("<edad>").append(bean.getEdad()).append("</edad>
");

                returnXML.append("<distrito>").append(bean.getDistrito()).append("</dist
rito>");

                returnXML.append("<estado>").append(bean.getEstadoCivil()).append("
</estado>");

                returnXML.append("</vendedor>");

                System.out.println("VENDEDOR->" +
returnXML.toString());
            }
        }
    }
}
```

```
        response.setContentType("text/xml");
        response.setHeader("Cache-Control", "no-cache");
        response.getWriter().write(returnXML.toString());
    }
    else {
        response.setContentType("text/xml");
        response.setHeader("Cache-Control", "no-cache");
        response.getWriter().write("?");
    }

    } catch (Exception e) {
        System.out.println(e);
    }

}
}
```

1.8 JSON

JSON, acrónimo de "*El lenguaje Java Script Object Notation*", es un formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos del lenguaje *Java Script* que no requiere el uso de XML.

La simplicidad de JSON ha dado lugar a la generalización de su uso, especialmente como alternativa a XML en AJAX. Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos en este contexto es que es mucho más sencillo escribir un analizador semántico de JSON. En el lenguaje *Java Script*, JSON puede ser analizado trivialmente usando el procedimiento `eval()`, lo cual ha sido fundamental para la aceptación de JSON por parte de la comunidad de desarrolladores AJAX, debido a la ubicuidad del lenguaje *Java Script* en casi cualquier navegador web.

En la práctica, los argumentos a favor de la facilidad de desarrollo de analizadores o del rendimiento de los mismos son poco relevantes, debido a las cuestiones de seguridad que plantea el uso de `eval()` y el auge del procesamiento nativo de XML incorporado en los navegadores modernos. Por esa razón, JSON se emplea habitualmente en entornos donde el tamaño del flujo de datos entre cliente y servidor es de vital importancia (de aquí su uso por Yahoo, Google, etc. que atienden a millones de usuarios) cuando la fuente de datos es explícitamente de fiar y donde no es importante el no disponer de procesamiento XSLT para manipular los datos en el cliente.

Si bien es frecuente ver JSON posicionado contra XML, también es frecuente el uso de JSON y XML en la misma aplicación. Por ejemplo, una aplicación de cliente que integra datos de Google Maps con datos meteorológicos en SOAP hace necesario soportar ambos formatos.

Cada vez hay más soporte de JSON mediante el uso de paquetes escritos por terceras partes. La lista de lenguajes soportados incluye *ActionScript*, C, C#, *ColdFusion*, *Common Lisp*, *Delphi*, E, Java, El lenguaje *Java Script*, ML, *Objective CAML*, *Perl*, *PHP*, *Python*, *Rebol*, *Ruby*, y *Lua*.

En diciembre de 2005, Yahoo! comenzó a dar soporte opcional de JSON en algunos de sus servicios web.

El término JSON está altamente difundido en los medios de programación; sin embargo, es un término mal descrito ya que, en realidad, es solo una parte de la definición del estándar ECMA-262 en que está basado el lenguaje *Java Script*. De ahí que ni Yahoo ni Google emplean JSON, sino LJS. Una de las cualidades intrínsecas del lenguaje *Java Script* denominada LJS facilita el flujo de datos e incluso de funciones, para la cual no requiere la función `eval()` si son datos los que se transfieren como en el caso de xml. Todo lo referente a transferencia de datos en todos sus tipos, incluyendo *arrays*, *booleans*, *integers*, etc. no requieren de la función `eval()`, y es precisamente en eso en donde supera por mucho el lenguaje *Java Script* al XML, si se utiliza el LJS y no la incorrecta definición de JSON.

1.9 Uso de JSON

En teoría, es trivial analizar JSON en el lenguaje *Java Script* usando la función `eval()` incorporada en el lenguaje. Por ejemplo:

```
miObjeto = eval('(' + json_datos + ')');
```

En la práctica, las consideraciones de seguridad, por lo general, recomiendan no usar `eval` sobre datos crudos y debería usarse un analizador distinto para garantizar la seguridad. El analizador proporcionado por JSON.org usa `eval()` en su función de análisis, protegiéndola con una expresión regular de forma que la función solo ve expresiones seguras.

```
var http_request = new XMLHttpRequest();
var url = "http://example.net/jsondata.php"; // Esta URL debería
devolver datos JSON

// Descarga los datos JSON del servidor.
http_request.onreadystatechange = handle_json;
http_request.open("GET", url, true);
http_request.send(null);

function handle_json() {
    if (http_request.readyState == 4) {
        if (http_request.status == 200) {
            var json_data = http_request.responseText;
            var the_object = eval("(" + json_data + ")");
        } else {
            alert("Ha habido un problema con la URL.");
        }
        http_request = null;
    }
}
```

Obsérvese que el uso de `XMLHttpRequest` en este ejemplo no es compatible con todos los navegadores, aunque existen variaciones sintácticas para Internet Explorer, Opera, Safari, y navegadores basados en Mozilla.

También, es posible usar elementos `<iframe>` ocultos para solicitar los datos de manera asíncrona, o usar peticiones `<form target="url_to_cgi_script" />`. Estos métodos eran los más habituales antes del advenimiento del uso generalizado de XMLHttpRequest.

Hay una biblioteca para el *framework* .NET que exporta clases .NET con la sintaxis de JSON para la comunicación entre cliente y servidor, en ambos sentidos.

1.10 Ejemplo de JSON

A continuación, se muestra un ejemplo simple de definición de barra de menús usando JSON y XML.

JSON:

```
{ "menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      { "value": "New", "onclick": "CreateNewDoc()" },
      { "value": "Open", "onclick": "OpenDoc()" },
      { "value": "Close", "onclick": "CloseDoc()" }
    ]
  }
}
```

XML:

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Si bien los defensores de JSON a menudo notan que éste es más abreviado que XML, obsérvese que los dos ejemplos tienen unos 190 caracteres cuando se eliminan los espacios en blanco. Además, el uso de compresión GZIP para enviar los datos al navegador puede reducir la diferencia de tamaños entre ambos formatos. De hecho, cuando se usa GZIP sobre los ejemplos anteriores, el ejemplo en XML es más pequeño por 6 bytes. Si bien esto no es concluyente, muestra que es necesario experimentar con el conjunto de datos a tratar para determinar qué formato será más eficiente en términos de tamaño. JSON no es siempre más pequeño que XML.

El beneficio de JSON, entonces, no es que sea más pequeño a la hora de transmitir, sino que representa mejor la estructura de los datos y requiere menos codificación y procesamiento.

1.11 Comparación con XML y otros lenguajes de marcado

XML goza de mayor soporte y ofrece muchas más herramientas de desarrollo (tanto en el lado del cliente como en el lado del servidor). Hay muchos analizadores JSON en el lado del servidor, existiendo al menos un analizador para la mayoría de los entornos. En algunos lenguajes, como Java o PHP, hay diferentes implementaciones donde escoger. En el lenguaje *Java Script*, el análisis es posible de manera nativa con la función `eval()`. Ambos formatos carecen de un mecanismo para representar grandes objetos binarios.

Con independencia de la comparación con XML, JSON puede ser muy compacto y eficiente si se usa de manera efectiva. Por ejemplo, la aplicación DHTML de búsqueda en BarracudaDrive recibe los listados de directorio como JSON desde el servidor. Esta aplicación de búsqueda está permanentemente consultando al servidor por nuevos directorios, y es notablemente rápida, incluso sobre una conexión lenta.

Los entornos en el servidor normalmente requieren que se incorpore una función u objeto analizador de JSON. Algunos programadores, especialmente los familiarizados con el lenguaje C, encuentran JSON más natural que XML, pero otros desarrolladores encuentran su escueta notación algo confusa, especialmente cuando se trata de datos fuertemente jerarquizados o anidados muy profundamente.

YAML es un superconjunto de JSON que trata de superar algunas de las limitaciones de éste. Aunque es significativamente más complejo, aún puede considerarse como ligero. El lenguaje de programación Ruby utiliza YAML como el formato de serialización por defecto. Así pues, es posible manejar JSON con bastante sencillez.

Resumen

📖 XMLHttpRequest es el objeto que permite la petición Asíncrona en *browser* diferentes a Internet Explorer

```
req = new XMLHttpRequest();
```

📖 ActiveXObject es el objeto que permite la petición Asíncrona en el *browser* Internet Explorer

```
req = new ActiveXObject("Microsoft.XMLHTTP");
```

📖 JSON

Acrónimo de "*El lenguaje Java Script Object Notation*", es un formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos del lenguaje Java Script que no requiere el uso de XML.

📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🖱 <http://www.maestrosdelweb.com/editorial/ajax/>

Aquí hallará teoría sobre AJAX.

🖱 <http://es.wikipedia.org/wiki/AJAX>

Aquí hallará teoría sobre AJAX.

**UNIDAD DE
APRENDIZAJE****3****SEMANA****5**

GESTIÓN DE LA CAPA MODELO DATOS MEDIANTE EL *FRAMEWORK IBATIS*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, utilizando el *framework Ibatis*, implementarán una aplicación web que gestiona transacciones de acceso a base de datos y consultas dinámicas obtenidas de archivos de configuración XML del *framework Ibatis*.

TEMARIO

- Introducción al *framework Ibatis*
- Manejo de inserciones, eliminaciones y actualizaciones con el *framework Ibatis*

ACTIVIDADES PROPUESTAS

- Los alumnos implementarán una aplicación mediante el *framework Ibatis*.

1. El *framework* *Ibatis*

Ibatis es un *framework* (método de trabajo) de código abierto basado en capas desarrollado por *Apache Software Foundation*, que se ocupa de la capa de Persistencia (se sitúa entre la lógica de Negocio y la capa de la Base de Datos). Puede ser implementado en Java y .NET (también existe un port para Ruby on Rails llamado RBatis).

Ibatis asocia objetos de modelo (*JavaBeans*) con sentencias SQL o procedimientos almacenados mediante archivos XML, simplificando la utilización de bases de datos.

1.1 Características

Es posible subdividir la capa de Persistencia en tres subcapas:

- La capa de Abstracción será la interfaz con la capa de la lógica de negocio, haciendo las veces de fachada (Patrón *Facade*) entre la aplicación y la persistencia. Se implementa de forma general mediante el patrón *Data Access Object* (DAO) y, particularmente en *Ibatis*, se implementa utilizando su *framework* DAO (*ibatis-dao.jar*).
- La capa de *Framework* de Persistencia será el interfaz con el gestor de Base de Datos ocupándose de la gestión de los datos mediante un API. Normalmente, en Java se utiliza JDBC; *Ibatis* utiliza su *framework* SQL-MAP (*ibatis-sqlmap.jar*).
- La capa de *Driver* se ocupa de la comunicación con la propia base de datos utilizando un *Driver* específico para la misma.

Toda implementación de *Ibatis* incluye los siguientes componentes:

- ***Data Mapper***: proporciona una forma sencilla de interacción de datos entre los objetos Java y .NET y bases de datos relacionales.
- ***Data Access Object***: abstracción que oculta la persistencia de objetos en la aplicación y proporciona un API de acceso a datos al resto de la aplicación.

El marco de trabajo *SQL Maps* es muy tolerante tanto con las malas implementaciones de los modelos de datos, como con las malas implementaciones de los modelos de objetos.

A pesar de ello, es muy recomendable que se usen las mejores prácticas tanto cuando se diseñe la base de datos (normalización apropiada, etc.), como cuando se diseñe el modelo de objetos. Haciendo esto se tendrá garantizado un mejor rendimiento y un diseño más claro.

La forma mas fácil de empezar es analizando con qué se está trabajando. ¿Cuáles son los objetos de negocio? ¿Cuáles son las tablas? ¿Cómo se relacionan? Para el primer ejemplo, se debe considerar la siguiente clase *Person* que sigue las convenciones típicas de los *JavaBeans*.

Person.java

```
package examples.domain;

//imports omitidos....

public class Person {
    private int id;
    private String firstName;
    private String lastName;
    private Date birthDate;
    private double weightInKilograms;
    private double heightInMeters;
    public int getId () {
        return id;
    }
    public void setId (int id) {
        this.id = id;
    }
}

//...asumimos que aquí tendríamos otros métodos get y set....
}
```

1.2 Registro de una Clase en los archivos SQL MAP

SQL Maps no limita a la hora de establecer relaciones del tipo tabla por clase o múltiples tablas por clase o múltiples clases por tabla. Existen muy pocas restricciones dado que se dispone de toda la potencia que ofrece el SQL. Por ejemplo, usemos la siguiente tabla, que podría ser apropiada para una relación tabla por clase:

Person.sql

```
CREATE TABLE PERSON(
PER_ID NUMBER (5, 0) NOT NULL,
PER_FIRST_NAME VARCHAR (40) NOT NULL,
PER_LAST_NAME VARCHAR (40) NOT NULL,
PER_BIRTH_DATE DATETIME ,
PER_WEIGHT_KG NUMBER (4, 2) NOT NULL,
PER_HEIGHT_M NUMBER (4, 2) NOT NULL,
PRIMARY KEY (PER_ID))
```

El archivo donde se registra la sentencia SQL actuará como el maestro para la configuración de nuestra implementación basada en SQL Map.

El archivo de configuración es un archivo XML. Dentro de él configuraremos ciertas propiedades, el DataSource JDBC y los mapeos SQL que utilizaremos en nuestra aplicación.

Este archivo ofrece un lugar central donde configurar el DataSource. El marco de trabajo puede manejar varias implementaciones de DataSources , dentro de los que se incluyen *Ibatis* están SimpleDataSource, Jakarta DBCP (Commons) y cualquier DataSource que se pueda obtener a través de un contexto JNDI (por ejemplo, Un DataSource configurado dentro de un servidor de aplicaciones).

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE sqlMapConfig PUBLIC "-//ibatis.apache.org/DTD SQL Map
Config 2.0//EN" "http://ibatis.apache.org/dtd/sqlmapconfig2.dtd">
```

<¡Asegúrate siempre de usar las cabeceras XML correcta como la de arriba! >

```
<sqlMapConfig>
```

*<!
El archivo de properties especificado aquí es el lugar donde poner las
propiedades (name=value)
que se usarán después en este archivo de configuración donde veamos
elementos de configuración
como por ejemplo "\${driver}". El archivo suele ser relativo al classpath y es
opcional.>*

```
<properties
resource="examples/sqlmap/maps/SqlMapConfigExample.properties" />
```

*<¡Estas propiedades controlan los detalles de configuración de SqlMap,
principalmente las relacionadas con la gestión de transacciones. Todas son
opcionales (ver Guía del Desarrollador para más información).>*

```
<settings
cacheModelsEnabled="true"
enhancementEnabled="true"
lazyLoadingEnabled="true"
maxRequests="32"
maxSessions="10"
maxTransactions="5"
useStatementNamespaces="false"
/>
```

*<¡typeAlias permite usar un nombre corto en referencia a un nombre cualificado
de una clase>*

```
<typeAlias alias="order" type="testdomain.Order"/>
```

*<¡Configura un DataSource que podrá ser usado con SQL Map usando
SimpleDataSource. Date cuenta del uso de las propiedades del archivo de
recursos de arriba. >*

```
<transactionManager type="JDBC" >
<dataSource type="SIMPLE">
<property name="JDBC.Driver" value="${driver}"/>
<property name="JDBC.ConnectionURL" value="${url}"/>
<property name="JDBC.Username" value="${username}"/>
<property name="JDBC.Password" value="${password}"/>
</dataSource>
</transactionManager>
```

*<¡Identifica todos los archivos de ubicación XML usados en SQL Map para cargar
los ubicados SQL. Date cuenta de los paths son relativos al classpath. Por ahora,
solo tenemos uno... >*

```
<sqlMap resource="examples/sqlmap/maps/Person.xml" />
</sqlMapConfig>
```

- Este es un ejemplo de archivo de properties que simplifica la automatización de la configuración del archivo de configuración de SQL Maps (ej. A través de Ant o de herramientas para la integración continua para diferentes entornos... etc.).
- Estos valores pueden ser usados en cualquier valor de propiedad en el archivo de arriba (ej #“\${driver}”). El uso del archivo de propiedades es completamente opcional.

```
driver=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:oracle1
username=jsmith
password=test
```

1.3 Los archivos SQL Map

Ahora que tenemos configurado un DataSource y listo el archivo de configuración central, necesitaremos proporcionar al archivo de SQL Map nuestro código SQL y los mapeos para cada uno de los objetos parámetro y de los objetos resultado (entradas y salidas respectivamente).

Continuando con el ejemplo de arriba, vamos a construir un archivo SQL Map para la clase Person y la tabla PERSON. Empezaremos con la estructura general de un documento SQL y una sencilla sentencia SQL.

Person.xml

```
<?xml version="1.0" encoding="UTF8" ?> <!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org/DTD SQL Map 2.0/EN" "http://ibatis.apache.org/dtd/sql-map2.dtd"> <sqlMap namespace="Person">
```

```
  <select id="getPerson" resultClass="examples.domain.Person">
    SELECT
    PER_ID      as id,
    PER_FIRST_NAME as firstName,
    PER_LAST_NAME as lastName,
    PER_BIRTH_DATE as birthDate,
    PER_WEIGHT_KG as weightInKilograms,
    PER_HEIGHT_M  as heightInMeters
    FROM PERSON
    WHERE PER_ID = #value#
  </select>
</sqlMap>
```

El ejemplo de arriba muestra la forma más sencilla de un mapeo SQL. Usa una característica del marco de trabajo SQL Maps que permite mapear de forma automática columnas de un ResultSet a propiedades de un JavaBean (o keys de un objeto java.util.Map, etc.) basándose en el encaje de los nombres, es decir que encajen los nombres de los campos del ResultSet con los de los atributos de JavaBean. El símbolo #value# es un parámetro de entrada. Más específicamente, el uso de “value” implica que estamos usando un recubrimiento de un tipo primitivo (ej. java.lang.Integer; pero no estamos limitados solo a este).

Aunque es sencillo, hay algunas limitaciones a la hora de usar el enfoque de mapeo mediante autoresultado. No hay forma de especificar el tipo de las columnas de salida (si fuera necesario) o cargar de forma automática datos relacionados (propiedades complejas) y además hay unas leves implicaciones de rendimiento ya que requiere el acceso a `ResultSetMetaData`. Usando `resultmap`, podemos superar todas estas limitaciones. Sin embargo, por ahora, la simplicidad es nuestra meta; además, siempre podemos cambiar a un enfoque diferente más adelante (sin cambiar el código fuente Java).

La mayoría de las aplicaciones de base de datos no solo leen de dicha base de datos, sino que también los tienen que modificar. Ya hemos visto un ejemplo de cómo se mapea una sencilla sentencia `SELECT`, pero ¿cómo hacerlo para `INSERT`, `UPDATE` y `DELETE`. La buena noticia es que no es diferente. Abajo completamos nuestro mapeo SQL para la clase `Person` con más registros que proporciona un conjunto de sentencias para acceder y modificar los datos en la base de datos.

Person.xml

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org/DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sqlmap2.dtd">
<sqlMap namespace="Person">
```

<!--Usa un recubrimiento de un tipo primitivo (ej. java.lang.Integer) como parámetro y permite que los resultados sean auto ubicados a las propiedades del JavaBean Person -->

```
<select id="getPerson" parameterClass="int"
resultClass="examples.domain.Person">
SELECT
PER_ID      as id,
PER_FIRST_NAME as firstName,
PER_LAST_NAME as lastName,
PER_BIRTH_DATE as birthDate,
PER_WEIGHT_KG as weightInKilograms,
PER_HEIGHT_M  as heightInMeters
FROM PERSON
WHERE PER_ID = #value#
</select>
```

<!--Usa las propiedades del JavaBean Person como parámetro para insertar. Cada uno de los parámetros entre los símbolos #almuadilla# es una propiedad del JavaBean. -->

```
<insert id="insertPerson" parameterClass="examples.domain.Person">
INSERT
    INTO PERSON (PER_ID, PER_FIRST_NAME, PER_LAST_NAME,
    PER_BIRTH_DATE, PER_WEIGHT_KG, PER_HEIGHT_M)
    VALUES  (#id#,    #firstName#,    #lastName#,    #birthDate#,
    #weightInKilograms#, #heightInMeters#)
</insert>
```

<!Usa las propiedades del JavaBean person como parámetro para actualizar. Cada uno de los parámetros entre los símbolos #almuadilla# es una propiedad del JavaBean. >

```
<update id="updatePerson" parameterClass="examples.domain.Person">
  UPDATE PERSON SET PER_FIRST_NAME = #firstName#,
  PER_LAST_NAME = #lastName#, PER_BIRTH_DATE = #birthDate#,
  PER_WEIGHT_KG = #weightInKilograms#,
  PER_HEIGHT_M = #heightInMeters#
```

<!Usa las propiedades del JavaBean person como parámetro para borrar. Cada uno de los parámetros entre los símbolos #almuadilla# es una propiedad del JavaBean. >

```
<delete id="deletePerson" parameterClass="examples.domain.Person">
  DELETE PERSON
  WHERE PER_ID = #id#
</delete>
</sqlMap>
```

Programando con el marco de trabajo SQL Map

Ahora que tenemos todo configurado y el registro, lo que necesitamos es programar nuestra aplicación Java. El primer paso es configurar SQL Map. Es simplemente una forma de cargar nuestra configuración del archivo XML de SQL Map que creamos antes. Para cargar el archivo XML, haremos uso de la clase Resources incluida en el marco de trabajo.

```
String resource = "com/ibatis/example/sqlMapconfig.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
```

Un objeto de la clase SqlMapClient es *thread safe* y será un servicio de larga duración. Solo es necesario crear el objeto/configurarlo una vez para todo el ciclo de vida de una aplicación. Esto lo convierte en un buen candidato para ser un miembro estático de una clase base (ejemplo de una clase base DAO), o si se prefiere, mantenerlo configurado de forma más centralizada y disponible de forma más global, se podría crear una *class wrapper* para mayor comodidad. Aquí hay un ejemplo de una clase que podría servir para este propósito:

```
public MyAppSqlConfig {
  private static final SqlMapClient sqlMap;
  static {
    try {
      String resource = "com/ibatis/example/sqlMapconfig.xml";
      Reader reader = Resources.getResourceAsReader(resource);
      sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
    } catch (Exception e) {
      // Si hay un error en este punto, no importa cuál sea. Será un error
      // irrecuperable del cual
      // nos interesará solo estar informados.
      // Deberás registrar el error y reenviar la excepción de forma que se te
      // notifique el
      // problema de forma inmediata.
```

```

    e.printStackTrace();
    throw new RuntimeException ("Error initializing MyAppSqlConfig class.
    Cause: " + e);
  }
}

```

```

public static SqlMapClient getSqlMapInstance () {
    return sqlMap;
}

}

```

Ahora que el objeto de SqlMap está inicializado y es accesible de forma sencilla, podemos hacer uso de ella. Usémosla primero para obtener un objeto Person de la base de datos. (Para este ejemplo, asumiremos que hay 10 registros en la tabla PERSON con rangos de PER_ID del 1 al 10).

Para obtener un objeto Person de la base de datos, simplemente necesitamos el objeto de SqlMap, el nombre de la sentencia a ejecutar y un Person ID. Carguemos, por ejemplo, el objeto Persona cuyo "id" es el número 5.

```

... SqlMapClient sqlMap = MyAppSqlMapConfig.getSqlMapInstance();

```

```

// Como se codificó arriba ... Integer personPk = new Integer(5); Person person =
(Person) sqlMap.queryForObject ("getPerson", personPk); ...

```

Ahora que tenemos un objeto Person de la base de datos, modifiquemos algunos datos. Cambiaremos, por ejemplo, la altura y el peso de la persona.

```

...
person.setHeightInMeters(1.83); // objeto de arriba.
person.setWeightInKilograms(86.36);

```

```

...
sqlMap.update("updatePerson", person);

```

Si queremos borrar el objeto Person, es igual de fácil.

```

...
sqlMap.delete ("deletePerson", person);

```

La inserción de un nuevo objeto Person es similar.
 Person newPerson = new Person();

newPerson.setId(11); // normalmente obtendrás el ID de una secuencia o de una tabla personalizada

```

newPerson.setFirstName("Clinton");
newPerson.setLastName("Begin");
newPerson.setBirthDate (null);
newPerson.setHeightInMeters(1.83);
newPerson.setWeightInKilograms(86.36);

```

```


...
sqlMap.insert ("insertPerson", newPerson);
...

```



1.4 Descripción de la etiquetas de los archivos SQL Map (Archivos donde se describen las sentencias SQL)

Statement Element	Attributes	Child Elements	Methods
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel xmlResultName (Java only)		insert
		All dynamic elements	update delete All query methods
<insert>	id parameterClass parameterMap	All dynamic elements	insert
		<selectKey> <generate> (.NET only)	update delete
<update>	id parameterClass parameterMap	All dynamic elements	insert
		<generate> (.NET only)	update delete
<delete>	id parameterClass parameterMap	All dynamic elements	insert
		<generate> (.NET only)	update delete
<select>	id parameterClass resultClass parameterMap resultMap cacheModel		
		All dynamic elements <generate> (.NET only)	All query methods
<procedure>	id parameterClass resultClass parameterMap resultMap xmlResultName (Java only)		insert
		All dynamic elements	update delete All query methods

Resumen

 Es importante reconocer los elementos en un archivo de *Ibatis*. Se inserta la sentencia (lo de mayúscula) y los valores que vendrán en los atributos del *bean* (lo de minúscula y encerrados entre #).

```
<insert id="insertPerson" parameterClass="examples.domain.Person">
  INSERT
    INTO
      PERSON
        (PER_ID,
         PER_FIRST_NAME,
         PER_LAST_NAME,
         PER_BIRTH_DATE,
         PER_WEIGHT_KG,
         PER_HEIGHT_M)
  VALUES
    (#id#,
    #firstName#,
    #lastName#,
    #birthDate#,
    #weightInKilograms#,
    #heightInMeters#)
</insert>
```

 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

 <http://es.wikipedia.org/wiki/IBATIS>

Aquí hallará definiciones sobre *Ibatis*.

 <http://ibatis.apache.org/>

La página oficial de *Ibatis*

**UNIDAD DE
APRENDIZAJE****3****SEMANA****6**

GESTIÓN DE LA CAPA MODELO DATOS MEDIANTE EL *FRAMEWORK IBATIS*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, utilizando el *framework Ibatis*, implementarán una aplicación web que gestiona transacciones de acceso a base de datos y consultas dinámicas obtenidas de archivos de configuración XML.

TEMARIO

- Consultas con *Ibatis*

ACTIVIDADES PROPUESTAS

- Los alumnos implementarán una aplicación mediante *Ibatis* utilizando consultas dinámicas.

1. Consultas mediante el *framework Ibatis*

1.1 Configuraciones

Como motor de base de datos se usará MySQL y en la base de datos "prueba" se tiene la siguiente tabla.

```
mysql> use prueba;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> describe COCHE;
```

Field	Type	Null	Key	Default	Extra
ID_COCHE	int(11)	NO	PRI	NULL	auto_increment
MARCA	varchar(255)	YES		NULL	
MATRICULA	varchar(255)	YES		NULL	
FECHA_MATRICULA	date	YES		NULL	
ID_PROPIETARIO	int(11)	NO			

```
5 rows in set (0.00 sec)
```

La clase java en la que guardaremos los datos de un registro leído de esta tabla o donde guardaremos los datos que queremos insertar o modificar será un *Java bean* como el siguiente.

```
package com.chuidiang.beans;

import java.util.Date;

public class Coche {
    private Integer id;
    private String marca;
    private String matricula;
    private Date fechaMatricula;
    private Integer idPropietario;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getMarca() {
```

```
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    public Date getFechaMatricula() {
        return fechaMatricula;
    }

    public void setFechaMatricula(Date fechaMatricula) {
        this.fechaMatricula = fechaMatricula;
    }

    public Integer getIdPropietario() {
        return idPropietario;
    }

    public void setIdPropietario(Integer idPropietario) {
        this.idPropietario = idPropietario;
    }

    /** Método de conveniencia para escribir rápido en pantalla */
    public String toString() {
        return id + "," + marca + "," + matricula + "," +
            fechaMatricula;
    }
}
```

1.2 El Archivo XML

Ibatis necesita uno o más archivos con las sentencias SQL que usaremos en nuestro programa. El archivo XML con el mapeo, al que llamaremos COCHE.xml, puede ser como este.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
    PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
```

```

"http://ibatis.apache.org/dtd/sql-map-2.dtd" >
<sqlMap >
    <select id="getCoche" resultClass="com.chuidiang.beans.Coche">

SELECT

        ID_COCHE as id,
        MARCA as marca,
        MATRICULA as matricula,
        FECHA_MATRICULA as fechaMatricula,
        ID_PROPIETARIO as idPropietario
    FROM COCHE
        WHERE ID_COCHE = #valor#
    </select>

    <select id="getCoches" resultClass="com.chuidiang.beans.Coche">
SELECT
        ID_COCHE as id,
        MARCA as marca,
        MATRICULA as matricula,
        FECHA_MATRICULA as fechaMatricula,
        ID_PROPIETARIO as idPropietario
    FROM COCHE
    </select>

    <select id="getHashCoche" resultClass="java.util.Hashtable">
SELECT
        ID_COCHE as id,
        MARCA as marca,
        MATRICULA as matricula,
        FECHA_MATRICULA as fechaMatricula,
        ID_PROPIETARIO as idPropietario
    FROM COCHE
        WHERE ID_COCHE = #valor#
    </select>

</sqlMap>

```

La parte del xml version y del DOCTYPE en la cabecera del archivo conviene ponerla para que *Ibatis* pueda verificar que es correcto el resto del archivo. El tag principal es <sqlMap>.

Luego, hay varios tag <select>, <update>, <insert> o <delete> en función del tipo de sentencia SQL que queramos poner. Hay que fijarse que en las sentencias SQL aparecen "variables" entre signos #. *Ibatis*, en tiempo de ejecución, reemplazará estas variables con valores que le pasemos desde código.

Cada uno de estos tags lleva obligatoriamente un atributo id="nombre" en el que debemos ponerle un nombre que distinga unas sentencias SQL de otras. Esos nombres son los que usaremos en el código.

Cada uno de estos *tags* puede llevar además dos atributos *parameterClass* y *resultClass*. Cuando desde código Java intentemos usar uno de estas SQL, usaremos unos métodos que nos proporciona *Ibatis*. Estos métodos admiten como primer parámetro un tipo *String* que es el nombre que identifica la sentencia SQL, y como segundo parámetro un *Object*. *parameterClass*, que es el nombre de la clase que pasaremos como parámetro *Object* en el método *Ibatis*. De ese *parameterObject* *Ibatis* tratará de obtener los valores de las variables entre #. Si *parameterClass*="int" o un tipo simple, en el código deberemos pasar un *Integer* o un tipo simple, e *Ibatis* pondrá directamente ese *Integer* reemplazando a #id#.

Si *parameterClass*="unaClaseJavaBean", en el código pasaremos esa clase como parámetro e *Ibatis* tratará de obtener el id llamando al método *getId()*.

Si *parameterClass*="unHashTable", en el código pasaremos un *Hashtable* como parámetro e *Ibatis* tratará de obtener el id llamando a *get("id")*.

El *parameterClass* no es obligatorio, ya que *Ibatis* tratará de arreglarse con lo que le pasemos. De todas formas, por eficiencia, es mejor ponerlo.

El atributo *resultClass* indica qué clase devolverán los métodos *Ibatis* en el *return*, y es obligatorio si la sentencia SQL tiene que devolver algo -en concreto, para los *SELECT-Ibatis*, igual que hacía con *parameterClass*, creará objetos del tipo indicado en *resultClass* y tratará de rellenar dentro los datos.

Si *resultClass*="unaClaseJavaBean", *Ibatis* hará *new* de unaClaseJavaBean y tratará de rellenar los valores llamando a los métodos *setId()*, *setMarca()*, *setMatricula()*, etc.

Si *resultClass*="unHashTable", *Ibatis* hará *new* de *HashTable* y tratará de rellenar los valores llamando a *put("id", ...)*, *put("marca", ...)*, etc.

1.3 Consultas Con *Ibatis*

Los distintos consultas *SELECT* se registra en el archivo de configuración. El primero es un *SELECT* por id, al que hemos puesto el nombre de "getCoche". Para la cláusula *WHERE* hemos usado #value# y no hemos indicado el *parameterClass*, así que aquí debemos pasar directamente un entero. Si pasamos una clase que no sea un tipo primitivo, *Ibatis* tratará de llamar al método *getId()* o *get("id")*. El código para esta consulta puede ser así.

```
Integer claveCoche = new Integer(1);
Coche coche = (Coche) sqlMap.queryForObject("getCoche", claveCoche);
```

En *queryForObject()* estamos indicando que nos devuelva un solo objeto Coche. Como parámetro pasamos el nombre que le dimos a esta consulta en el archivo COCHE.xml al *SELECT* y el entero que hace de clave. *Ibatis* se encarga de poner el entero en #value#, hacer la consulta, hacer un *new* de la clase Coche y de rellenar todos sus parámetros llamando a los distintos métodos *set()* de la clase Coche. El nombre de los métodos *set()* será el de los AS que hemos puesto en la consulta.

Otro *SELECT* que tenemos es al que hemos llamado "getCoches", que pretendemos que nos devuelva una lista de coches. Este *SELECT* no tiene ningún parámetro de entrada, ya que consulta todos los coches y no hay cláusula *WHERE*. El *resultClass* indica con qué clase se hará cada uno de los registros leídos de base de datos. El código para usar este *SELECT* puede ser como el siguiente.

```
List<Coche> coches = sqlMap.queryForList("getCoches", null);
```

Hemos llamado a *queryForList()* para indicar que queremos una lista. Como parámetro hemos pasado el nombre del *SELECT* que es "getCoches" y un *null*, puesto que no hemos puesto cláusula *WHERE*. *Ibatis* se encarga de hacer la consulta, construir tantas clases *Coche* como registros obtenga y devolvernos una *List* de *Coche*.

Finalmente, otro *SELECT* que hemos puesto es para obtener un *Hashtable* en vez de una clase *Coche*. Al *SELECT* lo hemos llamado "getHashCoche". No hemos indicado un *parameterClass*, pero como hemos puesto en la cláusula *WHERE* un *ID_COCHE=#valor#*, tendremos que pasar un entero como parámetro de entrada. Como *resultClass* hemos puesto un *java.util.Hashtable*, que será donde *Ibatis* nos devuelva el resultado. El código para usar esto puede ser el siguiente.


```
Map hashCoche = (Map) sqlMap.queryForObject("getHashCoche", 3);
```

Ibatis se encarga de hacer la consulta, reemplazando el *#valor#* del *WHERE* por un 3, luego crea un objeto *Hashtable*, lo rellena con el método *put("id",...)*, *put("marca", ...)* y lo devuelve. Los valores que usa como claves para el *Hashtable* son los *AS* que hemos puesto en el *SELECT*.


Resumen

 Para recoger un conjunto de elementos Con List

```
List<Coche> coches = sqlMap.queryForList("getCoches", null)
```

 Para recoger un conjunto de elementos Con Map

```
Map hashCoche = (Map) sqlMap.queryForObject("getHashCoche", 3);
```

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://ibatis.apache.org/javadownloads.cgi>

Aquí hallará documentación sobre *Ibatis*.

**UNIDAD DE
APRENDIZAJE****3****SEMANA****9**

GESTIÓN DE LA CAPA MODELO DATOS MEDIANTE EL *FRAMEWORK IBATIS*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, utilizando el *framework IBATIS*, implementan una aplicación web que gestiona transacciones de acceso a base de datos y consultas dinámicas obtenidas de archivos de configuración XML.

TEMARIO

- Consultas Dinámicas con *Ibatis*

ACTIVIDADES PROPUESTAS

- Los alumnos implementarán una aplicación en *Ibatis* utilizando consultas dinámicas.

1. Consultas Dinámicas con *Ibatis*

Un problema muy común con los que trabajan directamente con DAO es SQL dinámico. Es normalmente muy difícil de trabajar con SQL declaraciones que el cambio no solo los valores de los parámetros, pero que los parámetros y columnas se incluyen a todos. La solución típica es generalmente un desorden de la condicional-si los demás estados y horrible cadena de concatenaciones. El resultado deseado es a menudo una consulta. El *Ibatis DataMapper* API proporciona una solución relativamente elegante que se puede aplicar a cualquier elemento asignado declaración. Aquí está un ejemplo:

Una simple consulta statement dinámico, con dos posibles resultados

```
<select id="dynamicGetAccountList" cacheModel="account-cache"
parameterClass="Account" resultMap="account-result">
  select * from CUENTA
    <isGreaterThan prepend="and" property="Id" compareValue="0">
      donde ACC_ID Id = # #
    </ isGreaterThan>
  por fin ACC_LAST_NAME
</ select>
```

En el ejemplo anterior, existen dos posibles estados que podrían crearse en función de la situación de la propiedad Id del parámetro objeto. Si el ID de parámetro es superior a 0, a continuación, la declaración será creado de la siguiente manera:

```
select * from cuenta, cuando ACC_ID =?
```

Si el ID del parámetro es 0 o menos, la declaración se verá como sigue.

```
select * from CUENTA
```

La utilidad inmediata de este podría no ser aparente hasta una situación más compleja. Por ejemplo, el siguiente es un poco más complejo.

Un complejo y dinámico *SELECT*, con 16 posibles resultados

```
<select id="dynamicGetAccountList" parameterClass="Account" resultMap="account-
result">
  select * from CUENTA
    <dynamic prepend="WHERE">
      <isNotNull prepend="AND" property="FirstName">
        (ACC_FIRST_NAME = # # Nombre
      <isNotNull prepend="OR" property="LastName">
        ACC_LAST_NAME Apellido = # #
      </ isNotNull>
    ),
```

```
</ isNotNull>
```

```
<isNotNull prepend="AND" property="EmailAddress">
```

```
    ACC_EMAIL como EMAILADDRESS # #
</ isNotNull>
<isGreaterThan prepend="AND" property="Id" compareValue="0">
    ACC_ID Id = # #
</ isGreaterThan>
</ dinámica>
por fin ACC_LAST_NAME
</ select>
```

Dependiendo de la situación, no puede haber tantas como 16 diferentes consultas SQL generada a partir de la dinámica por encima de declaración. Para el código de otra persona-si las estructuras y cadena de concatenaciones puede llegar muy desordenado y requiere cientos de líneas de código.

El uso de declaraciones dinámicas es tan simple como la inserción de algunos códigos condicionales en torno a la dinámica de su SQL.

Ejemplo. Creación de una consulta dinámica de declaración con los códigos condicionales

```
<statement id="someName" parameterClass="Account" resultMap="account-result">
  select * from CUENTA
  <dynamic Prepend="where">
    <isGreaterThan prepend="and" property="id" compareValue="0">
      ACC_ID = # ID #
    </ isGreaterThan>
    <isNotNull prepend="and" property="lastName">
      ACC_LAST_NAME lastName = # #
    </ isNotNull>
  </ dinámica>
por fin ACC_LAST_NAME
</ declaración>
```

El elemento dinámico es opcional y ofrece una forma de gestionar una anteponer en los casos en que el anteponer ("WHERE") no deben incluirse a menos que las condiciones que figuran adjuntar a la declaración. La declaración de la sección puede contener cualquier número de elementos condicionales que determinará si la figura de código SQL se incluirá en la declaración. Todos los elementos condicional trabajo basado en el estado de los parámetros objeto pasó a la consulta.. Tanto el elemento dinámico y los elementos han condicionado un "anteponer" atributo. Anteponer el atributo es una parte del código que es libre de ser anulado por el padre del elemento anteponer si es necesario. En el ejemplo anterior, el "where" anteponer anulará la primera condicional. Esto es necesario para asegurar que la instrucción SQL se construye correctamente. For example, in the case of the first true condition, there is no need for the AND, and in fact it would break the statement. Por

ejemplo, en el caso de la primera, no hay necesidad de AND, y de hecho sería pausa la declaración. Las siguientes secciones describen los distintos tipos de elementos, incluidos los binarios condicionales, condicionales unario e iterar.

1.1 Atributos condicional dobles

prepend - SQL Overridable la parte que se a la declaración (opcional)
property - la propiedad que se compara (requerido)
compareProperty - los demás bienes que deben compararse (obligatorio o compareValue)
compareValue - el valor que se compara (obligatorio o compareProperty)

Elemento	Descripción
<isEqual>	<p>Los controles de la igualdad de una propiedad y un valor, o de otra propiedad. Ejemplo de uso:</p> <pre><isEqual anteponer = "Y" propiedad = "status" compareValue = "Y"> CASADO = "TRUE" </ isEqual></pre>
<isNotEqual>	<p>Los controles de la desigualdad de una propiedad y un valor, o de otra propiedad. Ejemplo de uso:</p> <pre><isNotEqual anteponer = "Y" propiedad = "status" compareValue = "N"> CASADO = 'FALSE' </ isNotEqual></pre>
<isGreaterThan>	<p>Comprueba si una propiedad es superior a un valor u otra propiedad. Ejemplo de uso:</p> <pre><isGreaterThan anteponer = "Y" propiedad = "edad" compareValue = "18"> ADOLESCENTES = 'FALSE' </ isGreaterThan></pre>
<isGreaterEqual>	<p>Comprueba si una propiedad es superior o igual a un valor u otra propiedad. Ejemplo de uso:</p> <pre><isGreaterEqual anteponer = "Y" propiedad = "shoeSize" compareValue = "12"> BIGFOOT = 'true' </ isGreaterEqual></pre>
<isLessEqual>	<p>Comprueba si una propiedad es igual o inferior a un valor u otra propiedad. Ejemplo de uso:</p>

Elemento	Descripción
	<pre> <isLessEqual anteponer = "Y" propiedad = "edad" compareValue = "18"> ADOLESCENTES = 'true' </ isLessEqual> </pre>

1.2 Elementos Condicionales Simples

La condicional simple permite comprobar el estado de una propiedad para una condición específica.

1.2.2 Atributos

prepend- La parte sobrescribible de SQL se declara (opcional)

property - la propiedad a ser controlados (requerido)

Elemento	Descripción
<isPropertyAvailable>	<p>Comprueba si una propiedad está disponible (es decir, es una propiedad del parámetro objeto). Ejemplo de uso:</p> <pre> <isPropertyAvailable property="id"> ACCOUNT_ID = # ID # </ isPropertyAvailable> </pre>
<isNotPropertyAvailable>	<p>Comprueba si una propiedad no está disponible (es decir, no una propiedad del parámetro objeto). Ejemplo de uso:</p> <pre> <isNotPropertyAvailable property="age"> STATUS = 'Nuevo' </ isNotEmpty> </pre>
<isNull>	<p>Comprueba si una propiedad es nula. Ejemplo de uso:</p> <pre> <isNull prepend="AND" property="order.id"> ACCOUNT.ACCOUNT_ID = ORDER.ACCOUNT_ID (+) </ isNotEmpty> </pre>
<isNotNull>	<p>Comprueba si una propiedad no es nula. Ejemplo de uso:</p> <pre> <isNotNull prepend="AND" property="order.id"> ORDER.ORDER_ID order.id = # # </ isNotEmpty> </pre>
<isEmpty>	<p>Comprueba si el valor de una colección, cadenas de propiedad es nulo o vacío (" " o el tamaño () <1). Ejemplo de uso:</p> <pre> <isEmpty property="firstName"> LÍMITE 0, 20 </pre>

Elemento	Descripción
	<code></ isEmpty></code>
<code><isEmpty></code>	<p>Comprueba si el valor de una colección, propiedad de cadena no es nulo y no vacío ("" o el tamaño () <1). Ejemplo de uso:</p> <pre> <isEmpty prepend="AND" property="firstName"> First_name LIKE '% \$% \$ Nombre' </ isEmpty> </pre>

1.3 Iterador

Esta etiqueta permite iterar una colección y repetir el contenido para cada elemento de una lista.

1.3.1. Iterate Attributes:

prepend – La parte sobrescribible SQL puede ser unida a la sentencia (opcional)
property- una propiedad de tipo IList que se reiteró más (requerido)
open - la cadena que inicia todo el bloque de iteraciones, utilice parentesis o corchetes(opcional)
close - la cadena que cierra todo el bloque de iteraciones, utilice parentesis o corchetes (opcional),
conjunction - la cadena que se aplicarán en cada iteración, utilice AND y OR (opcional)

Elemento	Descripción
<code><iterate></code>	<p>Reitera más de una propiedad que es de tipo IList. Ejemplo de uso:</p> <pre> <iterate anteponer = "Y" propiedad = "UserNameList" open = "(" close = ")" junto = "O"> nombre de usuario = # UserNameList [] # </ iterate> </pre> <p>Nota: Es muy importante incluir los corchetes [] al final de la lista de nombre de propiedad cuando se utiliza el elemento iterar. Estos corchetes hacen posible recibir un conjunto de datos (List) como un String</p>

1.4 Simple SQL dinámico Elementos

A pesar del poder de la declaración dinámica de sentencias debatido anteriormente, a veces sólo es necesario un simple y pequeño SQL dinámico. Para ello, sentencias SQL pueden contener elementos de SQL dinámico para contribuir a la implementación dinámica de las cláusulas. El concepto funciona mucho como conjunto de parámetros en línea, pero utiliza una sintaxis ligeramente diferente. Considere el siguiente ejemplo:

Un elemento dinámico que cambia el fin de cotejar

```
<statement id="getProduct" resultMap="get-product-result">
```



```
select * from producto por orden preferredOrder $ $
</ declaración>
```

En el ejemplo anterior, el elemento dinámico preferredOrder será sustituido por el valor de la propiedad preferredOrder el parámetro de objeto (como un parámetro de ruta). La diferencia es que este es un cambio fundamental a la propia declaración de SQL, que es mucho más grave que la simple fijación de un valor de parámetro. Un error cometido en un elemento dinámico de SQL puede introducir la seguridad, el rendimiento y la estabilidad riesgos. Se debe tener cuidado de hacer un montón de controles redundantes para garantizar que la simple dinámica de SQL elementos están siendo utilizados adecuadamente. Además, se debe ser consciente de su diseño, ya que hay potencial para la base de datos específicos para inmiscuirse en su modelo de objetos de negocio. Por ejemplo, se puede no querer un nombre de columna destinada a un fin de poner fin a la cláusula como una propiedad en su objeto de negocio, o como un valor del campo en su servidor de la página.

Simple elementos dinámicos se pueden incluir dentro de <statements> y vienen en práctica cuando hay una necesidad de modificar la instrucción SQL propia. Ejemplo:

Un elemento dinámico que cambia el operador de comparación

```
<statement id="getProduct" resultMap="get-product-result">
  SELECT * FROM producto
  <dynamic prepend="WHERE">
    <isEmpty property="Description">
      PRD_DESCRIPTION $ operador $ # # Descripción
    </ isEmpty>
  </ dinámica>
</ declaración>
```

En el ejemplo anterior, el operador propiedad del objeto parámetro se utiliza para sustituir al operador \$ \$ token. Por lo tanto, si el operador de propiedad es igual al del producto y la descripción de propiedad es igual al perro%%, entonces el SQL generado sería el siguiente:

```
SELECT * FROM producto, cuando PRD_DESCRIPTION LIKE '%% perro'
```


Resumen

Elemento para manejar búsquedas comparativas

<isEqual>	<p>Los controles de la igualdad de una propiedad y un valor, o de otra propiedad. Ejemplo de uso:</p> <pre><isEqual anteponer = "Y" propiedad = "status" comparevalue = "Y"> CASADO = "TRUE" </ isEqual></pre>
<isNotEqual>	<p>Los controles de la desigualdad de una propiedad y un valor, o de otra propiedad. Ejemplo de uso:</p> <pre><isNotEqual anteponer = "Y" propiedad = "status" comparevalue = "N"> CASADO = 'FALSE' </ isNotEqual></pre>

Elemento para una consulta con una lista de elementos comparativos

<iterate>	<p>Reitera más de una propiedad que es de tipo IList. Ejemplo de uso:</p>
	<pre><iterar anteponer = "Y" propiedad = "UserNameList" open = "(" Close = ")" junto = "O"> nombre de usuario = # UserNameList [] # </ iterar></pre> <p>Nota: Es muy importante incluir los corchetes [] al final de la lista de nombre de propiedad cuando se utiliza el elemento iterar. Estos corchetes hacen posible recibir un conjunto de datos (List) como un String</p>

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://ibatis.apache.org/javadownloads.cgi>

Aquí hallará documentación sobre *Ibatis*.

**UNIDAD DE
APRENDIZAJE****3****SEMANA****10**

GESTIÓN DE LA CAPA MODELO DATOS MEDIANTE EL *FRAMEWORK IBATIS*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, utilizando el *framework Ibatis*, implementan una aplicación web que gestiona transacciones de acceso a base de datos y consultas dinámicas obtenidas de archivos de configuración XML.

TEMARIO

- Manejo del Patrón con *Ibatis*

ACTIVIDADES PROPUESTAS

- Los alumnos implementarán una aplicación en Patrón DAO mediante *Ibatis*.

1. PATRÓN DAO CON *IBATIS*

El patrón DAO nos ayuda a organizar las tareas de persistencia (guardado, búsqueda, recuperación, borrado, etc.) de los objetos y nos permite definir múltiples implementaciones para un mismo objeto mediante la definición de una interfaz. Por ejemplo, la interfaz `UsuarioDao` tiene los métodos `guardarUsuario(Usuario usuario)`, `obtenerUsuario(String loginId)`, etc. y dos implementaciones: `UsuarioMySqlDao` y `UsuarioLdapDao` donde implementamos las operaciones para `MySql` y `Ldap` respectivamente. Con *Ibatis* DAO podremos configurar cuándo usar una u otra implementación sin necesidad de modificar código. Además, agregar un `UsuarioOracleDao` será muy fácil y no implicará modificar código del resto de la aplicación.

Cuando hacemos un análisis y diseño orientado a objetos, obtenemos nuestro modelo de clases y también diseñamos el modelo de datos donde se almacena la información. Y siempre nos queda la tarea de conectarnos con una base de datos, crear statements, construir selects/updates/etc, recorrer resultsets y setear atributos de objetos, etc. para guardar, buscar, recuperar, etc. los valores de los atributos de un objeto. *Ibatis* `sqlMap` simplifica esta tarea resumiéndola a la configuración de archivos XML, con SQL ANSI o propietario y funciona con prácticamente cualquier base de datos con driver JDBC.

1.1 ¿Cuándo utilizar *Ibatis*?

Como toda herramienta, no vale para todo. *Ibatis* `sqlMap` es muy válido en algunos casos.

- Cuando se requiere una curva de aprendizaje rápida y pocos conocimientos previos, y no requiere aprender un lenguaje de consultas como en `Hibernate` o `EJB CMP`.
- Cuando se necesita manipular el SQL (para utilizar SQL propietario, optimizarlo, etc.) o se necesita llamar a procedimientos almacenados.
- Cuando el modelo de datos existe previamente y no está muy normalizado (aunque lógicamente se puede utilizar con modelos nuevos y normalizados)
- Cuando se requiere alto rendimiento y optimización.

1.2 *Ibatis* DAO es válido en las siguientes circunstancias:

- Cuando se sabe que el RDBMS (motor de base de datos) puede cambiar en el futuro.
- Cuando la implementación del acceso a datos puede requerir cambios sustanciales en el futuro.
- Cuando la implementación del acceso a datos puede variar entre un entorno de producción y otro (software comercial en distintos clientes).

1.3 ¿Cómo funciona esto internamente?

Básicamente, funcionan con programación declarativa y extensión del framework. Es decir, se configuran archivos XML y, en *Ibatis* DAO, se extienden clases donde se implementa la interfaz y el comportamiento específico.

1.4 Crear un DAO.XML

```
<!DOCTYPE daoConfig PUBLIC "-//iBatis.com//DTD DAO
Configuration 2.0//EN" "http://www.ibatis.com/dtd/dao-
2.dtd">

<daoConfig>

<context>

    <transactionManager type="SQLMAP">

        <property name="SqlMapConfigResource"

            value="es/dxd/km/dao/sqlMap/sqlMapConfig.xml" />

    </transactionManager>

    <dao interface="es.dxd.km.dao.UsuarioDao"

        implementation="es.dxd.km.dao.sqlMap.UsuarioSqlMapDao" />

</context>

</daoConfig>
```

1.5 Crear una utilidad de configuración tipo DaoConfig

```
public class DaoConfig {

private static final String DAO_XML = "es/dxd/km/dao/dao.xml";

private static final DaoManager daoManager;

static {

    try {

        daoManager = newDaoManager();

    } catch (Exception e) {

        throw new RuntimeException("Description. Cause: " + e, e);

    }

}
```

```
}

public static DaoManager getDaoManager() {

    return daoManager;

}

public static DaoManager newDaoManager() {

    try {

        Reader reader =
            Resources.getResourceAsReader(DAO_XML);

        return DaoManagerBuilder.buildDaoManager(reader, null);

    } catch (Exception e) {

        throw new RuntimeException(

            "Could not initialize DaoConfig. Cause: " + e, e);

    }

}

}
```

1.6 Crear la interfaz del Dao para Usuarios, UsuariosDao:

```
public interface UsuarioDao extends Dao {

    public int updateUsuario(Usuario usuario);

    public int insertUsuario(Usuario usuario);

    public int deleteUsuario(String idUsuario);

    public Usuario getUsuario(String idUsuario);

    public Usuario getUsuarioValidado(String idUsuario, String
password);

    public List getUsuariosByExample(Usuario usuario);

}
```

1.7 Crear su implementación:

```
public class UsuarioSqlMapDao extends SqlMapDaoTemplate implements
es.dxd.km.dao.UsuarioDao {
```

```
public UsuarioSqlMapDao(DaoManager arg0) {  
    super(arg0);  
}  
  
public int updateUsuario(Usuario usuario) {  
    try {  
        return getSqlMapExecutor().update("updateUsuario",  
            usuario);  
    } catch (SQLException e) {  
        throw new DaoException("Error actualizando usuario.  
            Cause: " + e, e);  
    }  
}  
  
public int insertUsuario(Usuario usuario) {  
    try {  
        getSqlMapExecutor().insert("insertUsuario", usuario);  
    } catch (SQLException e) {  
        throw new DaoException("Error insertando usuario.  
            Cause: " + e, e);  
    }  
    return 1;  
}  
  
public int deleteUsuario(String idUsuario) {  
    try {  
        return getSqlMapExecutor().delete("deleteUsuario",  
            idUsuario);  
    } catch (SQLException e) {  
        throw new DaoException("Error actualizando usuario. Cause:  
            " + e, e);  
    }  
}
```

```
}

public Usuario getUsuario(String idUsuario) {

    try {

        Usuario usuario = (Usuario)
        getSqlMapExecutor().queryForObject(

            "getUsuarioById", idUsuario); usuario.setRoles((List)
            getSqlMapExecutor().queryForList("getRolesUsuario",
            usuario));

        return usuario;

    } catch (SQLException e) {

        throw new DaoException("Error recuperando usuario. Cause: "
        + e, e);

    }

}
```

```
public List getUsuariosByExample(Usuario usuario) {

    try {

        Usuario usuarioExample = new Usuario();

        if (usuario.getApellidos() != null) {

            usuarioExample.setApellidos("%"

            + usuario.getApellidos().toLowerCase() + "%");

        }

        if (usuario.getNombre() != null) {

            usuarioExample.setNombre("%"

            + usuario.getNombre().toLowerCase() + "%");

        }

        usuarioExample.setIdUsuario(usuario.getIdUsuario());

        List usuarios = (List) getSqlMapExecutor().queryForList(

        "getUsuariosByExample", usuarioExample);

        // Asignar los roles

        for (Iterator iter = usuarios.iterator(); iter.hasNext();) {
```



```
        Usuario usuario2 = (Usuario) iter.next();

        usuario2.setRoles((List) getSqlMapExecutor().queryForList(
            "getRolesUsuario", usuario2));
    }

    return usuarios;

} catch (SQLException e) {

    throw new DaoException("Error recuperando usuarios. Cause:
    " + e, e);

}

}

public Usuario getUsuarioValidado(String idUsuario, String password) {

    Usuario usuario = getUsuario(idUsuario);

    return usuario.getPassword().equals(password) ? usuario : null;

}

}
```

1.8 Crear la configuración sqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMapConfig

PUBLIC "-//ibatis.com//DTD SQL Map Config 2.0//EN"

"http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>

<properties

resource="properties/SqlMapConfig.properties" />

<settings cacheModelsEnabled="true" enhancementEnabled="true"

lazyLoadingEnabled="true" maxRequests="32" maxSessions="10"

maxTransactions="5" useStatementNamespaces="false" />

<transactionManager type="JDBC">

<dataSource type="SIMPLE">
```

```
<property name="JDBC.Driver" value="${driver}" />
<property name="JDBC.ConnectionURL" value="${url}" />
<property name="JDBC.Username" value="${username}" />
<property name="JDBC.Password" value="${password}" />
</dataSource>

</transactionManager>

<sqlMap
resource="es/dxd/km/dao/sqlMap/Usuario.xml" />
</sqlMapConfig>
```

1.9 Configuración del Archivo Properties

```
driver=org.apache.derby.jdbc.ClientDriver
url=jdbc:derby://localhost:1527/kmdb
username=derby
password=derby
```

1.10 Configuración de los Archivos XML

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMap PUBLIC "-//ibatis.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="Usuario">

    <select id="getUsuarioById" resultClass="es.dxd.km.model.Usuario">

        SELECT rtrim(ID_USUARIO) as idUsuario, rtrim(PASSWORD) as
        password, rtrim(NOMBRE) as nombre, rtrim(APELLIDOS) as apellidos
        FROM APP.USUARIOS WHERE id_usuario = #value#

    </select>

    <select id="getRolesUsuario"

        parameterClass="es.dxd.km.model.Usuario"

        resultClass="es.dxd.km.model.Rol">
```

```
SELECT rtrim(R.ID_ROL) as idRol, rtrim(R.NOMBRE) as nombre,  
rtrim(R.DESCRIPCION) as descripcion FROM APP.ROLES R,  
APP.USUARIOS_ROLES U WHERE R.ID_ROL = U.ID_ROL AND U.ID_USUARIO  
= #idUserio#
```

```
</select>
```

```
<select id="getUsuariosByExample"  
parameterClass="es.dxd.km.model.Usuario"  
resultClass="es.dxd.km.model.Usuario">  
SELECT rtrim(ID_USUARIO) as idUsuario, rtrim(PASSWORD) as  
password, rtrim(NOMBRE) as nombre, rtrim(APELLIDOS) as apellidos  
FROM APP.USUARIOS
```

```
<dynamic prepend="WHERE">
```

```
<isNotNull prepend="AND" property="nombre">
```

```
lower(NOMBRE) like #nombre#
```

```
</isNotNull>
```

```
<isNotNull prepend="AND" property="apellidos">
```

```
lower(APELLIDOS) like #apellidos#
```

```
</isNotNull>
```

```
<isNotNull prepend="AND" property="idUsuario">
```

```
ID_USUARIO = #idUserio#
```

```
</isNotNull>
```

```
</dynamic>
```

```
</select>
```

```
<insert id="insertUsuario"
```

```
parameterClass="es.dxd.km.model.Usuario">
```

```
INSERT INTO APP.USUARIOS (ID_USUARIO, PASSWORD, NOMBRE,  
APELLIDOS) VALUES (#idUserio#, #password#, #nombre#,
```

```

        #apellidos#)

    </insert>

    <update id="updateUsuario"
        parameterClass="es.dxd.km.model.Usuario">

        UPDATE APP.USUARIOS SET PASSWORD = #password#, NOMBRE =
        #nombre#, APELLIDOS = #apellidos# WHERE ID_USUARIO = #idUserario#

    </update>

    <delete id="deleteUsuario" parameterClass="string">

        DELETE FROM APP.USUARIOS WHERE ID_USUARIO = #value#

    </delete>

</sqlMap>

```

Resumen

Ejemplo de Consulta Dinámica

```

<select id="selectQueryUsuario"
    parameterClass="usuario"
    resultClass="usuario">
    SELECT
        A.COD_USU AS idUsuario,
        A.NOM_USU AS nombre,
        A.APE_USU AS apellido,
        A.EMP_USU AS empresa,
        A.CAT_USU AS idCategoria,
        A.LOG_USU AS login,
        A.PAS_USU AS password,
        A.EST_USU AS idEstado,

```

```
        B.NOM_TIP AS strEstado,  
        C.NOM_TIP AS strCategoria  
FROM TB_USUARIO AS A  
    INNER JOIN TB_TIPO AS B ON A.CAT_USU = B.COD_TIP  
    INNER JOIN TB_TIPO AS C ON A.EST_USU = C.COD_TIP  
<dynamic prepend="where">  
    <isNotEmpty property="idUser" prepend="and">  
        A.COD_USU = #idUser#  
    </isNotEmpty>  
    <isNotEmpty property="login" prepend="and">  
        A.LOG_USU = #login#  
    </isNotEmpty>  
    <isNotEmpty property="password" prepend="and">  
        A.PAS_USU = #password#  
    </isNotEmpty>  
</dynamic>  
</select>
```

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://www.javaworld.com/javaworld/jw-07-2008/jw-07-orm-comparison.html>

Aquí hallará comparaciones entre los frameworks de persistencia más utilizados.

**UNIDAD DE
APRENDIZAJE****3****SEMANA****11**

GESTIÓN DE LA CAPA MODELO DATOS MEDIANTE EL *FRAMEWORK IBATIS*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, utilizando el *framework Ibatis*, implementan una aplicación web que gestiona transacciones de acceso a base de datos y consultas dinámicas obtenidas de archivos de configuración XML.

TEMARIO

- Paginación

ACTIVIDADES PROPUESTAS

- Los alumnos implementan aplicaciones con Paginación en *Ibatis*.

1. Paginación

1.1 QueryForPaginatedList

```
PaginatedList QueryForPaginatedList público (string statementName,
                                             parameterObject objeto,
                                             pageSize int);
```

En una base de datos, se consulta a menudo más datos que los usuarios desean ver a la vez, y nuestras necesidades pueden decir que tenemos que ofrecer una larga lista de resultados de una "página" a la vez. Si la consulta devuelve 1000 registros, que puede ser que necesite para presentar los resultados al usuario en grupos de cincuenta, y que ellos mueven hacia adelante y hacia atrás entre los conjuntos, dado que este es un requisito común, el *framework Ibatis* proporciona métodos para esta necesidad.

El *PaginatedList* incluye métodos para navegar a través de las páginas nextPage() previousPage() gotoPage() y también el control de la situación de la página isFirstPage() isMiddlePage() isLastPage() isNextPageAvailable() isPreviousPageAvailable() , getPageIndex() getPageSize(). Aunque el número total de registros disponibles no es accesible desde la PaginatedList, esta debe ser fácilmente realizada por simplemente la ejecución de una segunda declaración que cuenta con los resultados esperados.

1.2 Cache Model

Si desea caché el resultado de una consulta, puede especificar un modelo de caché, como parte de la <sentencia> elemento. El ejemplo muestra <cacheModel> un elemento y un <sentencia> correspondiente.

Ejemplo <cacheModel> elemento con su correspondiente <sentencia>

```
<cacheModel id=" product-cache "implementation="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="size" value="1000" />
</ cacheModel>

<statement id="selectProductList" parameterClass="int" cacheModel=" product-cache
">
  select * from producto que PRD_CAT_ID valor = # #
</ declaración>
```

En el ejemplo, una memoria caché se define para los productos que utiliza una referencia de tipo LRU y se vacía cada 24 horas o cuando los estados de las sentencias se actualizan.

1.1 Archivo XML

```
<sqlMap namespace="Producto">

  <typeAlias alias="producto" type="demo.recursos.beans.BeanProducto"/>

  <select id="selectProducto" parameterClass="producto"
    resultClass="producto" cacheModel="productCache">
    SELECT cod_pro AS idProducto,
           nom_pro AS nombre,
           pre_pro AS precio,
           stk_pro AS stock,
           cat_pro AS categoria,
           fec_reg_pro AS fecha
    FROM TB_PRODUCTO
  </select>
</sqlMap>
```

1.2 La Interfaz

```
package demo.model.patronDAO.interfaces;

import java.util.List;

import com.ibatis.common.util.PaginatedList;

import demo.recursos.beans.BeanProducto;

public interface ProductoDAO {

    public List<BeanProducto> listadoDisplayTag() throws Exception;

    public PaginatedList listadoIbatis() throws Exception;

}
```

1.3 La clase Service

```
public class ProductoService {

    ProductoDAO productoDAO;

    public ProductoService() {
        DaoManager daoMgr = DAOConfig.getDaoManager();
        this.productoDAO = (ProductoDAO)
daoMgr.getDao(ProductoDAO.class);
    }

}
```

```
        public List<BeanProducto> listadoDisplayTag() throws Exception {
            return productoDAO.listadoDisplayTag();
        }

        public PaginatedList listadoIbatis() throws Exception {
            return productoDAO.listadoIbatis();
        }
    }
}
```

1.4 El Controlador

```
public class ActionProducto extends DispatchAction {

    public ActionForward listadoDisplayTag(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception{

        ProductoService servicio = new ProductoService();

        List<BeanProducto> lista = servicio.listadoDisplayTag();
        request.getSession().setAttribute("listadoDisplayTag", lista);

        return mapping.findForward("salida01");
    }

    public ActionForward listadoIbatis(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception{

        ProductoService servicio = new ProductoService();

        PaginatedList lista = servicio.listadoIbatis();
        request.getSession().setAttribute("listadoIbatis", lista);

        return mapping.findForward("salida02");
    }

    public ActionForward nextPage(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception{

        HttpSession session = request.getSession();

        PaginatedList lista = (PaginatedList)session.getAttribute("listadoIbatis");
        if(lista.isNextPageAvailable())
            lista.nextPage();
    }
}
```

```
        session.setAttribute("listadolbatis", lista);

        return mapping.findForward("salida02");
    }

    public ActionForward previousPage(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception{

        HttpSession session = request.getSession();

        PaginatedList lista = (PaginatedList)session.getAttribute("listadolbatis");
        if(lista.isPreviousPageAvailable())
            lista.previousPage();

        session.setAttribute("listadolbatis", lista);

        return mapping.findForward("salida02");
    }

}
```

1.5 La clase Dao

```
package demo.model.patronDAO.daos;

import java.util.List;

import com.ibatis.common.util.PaginatedList;
import com.ibatis.dao.client.DaoManager;

import demo.model.patronDAO.interfaces.ProductoDAO;
import demo.recursos.beans.BeanProducto;

public class ProductoSqlMapDAO extends BaseSqlMapDao implements
ProductoDAO {

    private static final Integer PAG_SIZE = 30;

    public ProductoSqlMapDAO(DaoManager daoManager) {
        super(daoManager);
    }


    @Override @SuppressWarnings("unchecked")
    public List<BeanProducto> listadoDisplayTag() throws Exception {
        return queryForList("selectProducto", null);
    }
}
```

```

    @Override @SuppressWarnings("unchecked")
    public PaginatedList listadolbatis() throws Exception {
        return queryForPaginatedList("selectProducto", null, PAG_SIZE);
    }
}

```

Resumen

 *Ibatis* permite traer datos por bloques realizando la configuración en los Archivos

```
<sqlMap namespace="Producto">
```


```
  <typeAlias alias="producto" type="demo.recursos.beans.BeanProducto"/>
```

```
    <select id="selectProducto" parameterClass="producto" resultClass="producto"
      cacheModel="productCache">
```

```
      SELECT cod_pro AS idProducto,
             nom_pro AS nombre,
             pre_pro AS precio,
             stk_pro AS stock,
             cat_pro AS categoria,
             fec_reg_pro AS fecha
      FROM TB_PRODUCTO
```

```
    </select>
```

```
</sqlMap>
```

 El *PaginatedList* interfaz incluye métodos para navegar a través de las páginas *nextPage()* *previousPage()* *gotoPage()* y también el control de la situación de la página *isFirstPage()* *isMiddlePage()* *isLastPage()* *isNextPageAvailable()* *isPreviousPageAvailable()* , *getPageIndex()* *getPageSize()*.

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://www.javaworld.com/javaworld/jw-07-2008/jw-07-orm-comparison.html>

Aquí hallará comparaciones entre los frameworks de persistencia más utilizados.

**UNIDAD DE
APRENDIZAJE****3****SEMANA****12-13**

GESTIÓN DE LA CAPA MODELO DATOS MEDIANTE EL *FRAMEWORK IBATIS*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, utilizando el *framework Ibatis*, implementan una aplicación web que gestiona transacciones de acceso a base de datos y consultas dinámicas obtenidas de archivos de configuración XML.

TEMARIO

- Transacciones mediante *Ibatis*
- Ejecución de *Stored Procedure* mediante *Ibatis*

ACTIVIDADES PROPUESTAS

- Los alumnos implementaran transacciones mediante *Ibatis*.

1 Transacción (base de datos)

Una transacción en un sistema de gestión de bases de datos (SGBD) es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible o atómica.

Un SGBD se dice transaccional si es capaz de mantener la integridad de los datos, haciendo que estas transacciones no puedan finalizar en un estado intermedio. Cuando por alguna causa el sistema debe cancelar la transacción, empieza a deshacer las órdenes ejecutadas hasta dejar la base de datos en su estado inicial (llamado punto de integridad), como si la orden de la transacción nunca se hubiese realizado.

Para esto, el lenguaje de consulta de datos SQL (*Structured query language*) provee los mecanismos para especificar que un conjunto de acciones deben constituir una transacción.

BEGIN TRAN: Especifica que va a empezar una transacción.

COMMIT TRAN: Le indica al motor que puede considerar la transacción completada con éxito.

ROLLBACK TRAN: Indica que se ha alcanzado un fallo y que debe restablecer la base al punto de integridad.

En un sistema ideal, las transacciones deberían garantizar todas las propiedades ACID; en la práctica, a veces alguna de estas propiedades se simplifica o debilita con vistas a obtener un mejor rendimiento.

1.1 Un ejemplo de transacción

Un ejemplo habitual de transacción es el traspaso de una cantidad de dinero entre cuentas bancarias. Normalmente, se realiza mediante dos operaciones distintas: una en la que se decrementa el saldo de la cuenta origen y otra en la que incrementamos el saldo de la cuenta destino. Para garantizar la consistencia del sistema (en otras palabras, para que no aparezca o desaparezca dinero), las dos operaciones deben ser atómicas, es decir, el sistema debe garantizar que, bajo cualquier circunstancia (incluso una caída del sistema), el resultado final es que, o bien se han realizado las dos operaciones, o bien no se ha realizado ninguna.

2 Transacciones en *Ibatis*

Para realizar la transacción el código deberá tener la siguiente estructura.

```
sqlMapClient.startTransaction();  
sqlMapClient.commitTransaction();  
sqlMapClient.endTransaction();
```

2.1 Configuración del Archivo pedido.xml

```
<sqlMap namespace="Pedido">

    <typeAlias alias="producto" type="demo.recursos.beans.BeanProducto"/>
    <typeAlias alias="pedido" type="demo.recursos.beans.BeanPedido"/>
    <typeAlias alias="detallePedido" type="demo.recursos.beans.BeanDetallePedido"/>

    <insert id="insertPedido" parameterClass="pedido">
        INSERT INTO TB_PEDIDO
        (cod_ped, fec_ped, cod_cli)
        VALUES
        (null, #fecha#, #cliente#)
        <selectKey resultClass="int" >
            SELECT LAST_INSERT_ID()
        </selectKey>
    </insert>

    <insert id="insertDetallePedido" parameterClass="detallePedido">
        INSERT INTO TB_DETALLE_PEDIDO
        (cod_ped, cod_pro, pre_pro, can_pro)
        VALUES
        (#codigoPedido#, #codigoProducto#, #precio#, #cantidad#)
    </insert>

    <update id="updatePrecioProducto" parameterClass="producto">
        UPDATE TB_PRODUCTO SET
            stk_pro = stk_pro - #cantidad#
        WHERE
            cod_pro=#codigo#
    </update>

</sqlMap>
```

2.2 Implementación del DAO

```
public class PedidoSqlMapDAO extends BaseSqlMapDao implements PedidoDAO {

    public PedidoSqlMapDAO(DaoManager daoManager) {
        super(daoManager);
    }

    String resources = "demo/model/patronDAO/sqlmap/sql-map-config.xml";

    public void insertar(BeaPedido bean, Collection<BeanProducto>
detallePedido)
        throws Exception {

    Reader reader = Resources.getResourceAsReader(resources);
```

```

SqlMapClient sqlMapClient = SqlMapClientBuilder.buildSqlMapClient(reader);

    try {
        // 1 Se inserta la cabecera de la nota de pedido
        sqlMapClient.startTransaction();

        Integer codigoPedido = (Integer) insert("insertPedido", bean);

        BeanDetallePedido beanDetallePedido = null;
        for (BeanProducto beanProducto : detallePedido) {

            beanDetallePedido = new BeanDetallePedido();

            beanDetallePedido.setCodigoProducto(beanProducto.getCodigo(
));
            beanDetallePedido.setCodigoPedido(codigoPedido);
            beanDetallePedido.setPrecio(beanProducto.getPrecio());
            beanDetallePedido.setCantidad(beanProducto.getCantidad());

            // 3 Se actualiza el precio
            update("updatePrecioProducto", beanProducto);

            // 3 Se inserta el detalle
            insert("insertDetallePedido", beanDetallePedido);
        }
        sqlMapClient.commitTransaction();
        sqlMapClient.endTransaction();

    } catch (Exception e) {
    }

}

```

3 *Stored procedure en Ibatis*

Los procedimientos almacenados se aplican mediante la declaración <procedure>. El siguiente ejemplo muestra cómo un procedimiento almacenado con parámetros de salida.

procedure.xml

```

<parameterMap id= "swapParameters" class= "map">
    <parameter property= "email1" jdbcType= "VARCHAR" javaType=
"java.lang.String" mode= "INOUT" />
    <parameter property= "email2" jdbcType= "VARCHAR" javaType=
"java.lang.String" mode= "INOUT" />
</ parameterMap>

<procedure id= "swapEmailAddresses" parameterMap= "swapParameters">
    (call swap_email_address (?,?))
</ procedimiento>

```


Invocando al procedimiento que intercambia dos direcciones de correo electrónico entre dos columnas (tabla de base de datos) y de los parametros (Map). El parámetro es solo modificado si el parámetro "mode" se establece *INOUT* o *OUT*. De lo contrario se quedan sin cambios. Obviamente inmutable el parámetro (por ejemplo, String) no puede ser modificado.

Nota: Asegúrese siempre de utilizar el estándar JDBC en la sintaxis del procedimiento almacenado. Consulte la documentación "CallableStatement ODBC" para obtener más información.

3.1 Configuración en el SqlMapClient

Si el procedimiento devuelve un conjunto de resultados (no un conjunto de resultados en un parámetro OUT, pero un conjunto de resultados del procedimiento en sí), entonces utilice ***queryForList ()*** o ***queryForObject ()***. Utilice ***queryForList ()*** si espera más de un resultado objeto, o ***queryForObject ()*** si usted espera un resultado de un objeto.

Si el procedimiento devuelve un conjunto de resultados y actualizaciones de datos, entonces se debe configurar su ***<transactionManager>*** con el atributo ***commitRequired = "true"***.

Si el procedimiento no retorna un conjunto de resultados o solo retorna valores en parámetros de salida, entonces use el método ***update()***

Resumen

📖 Para realizar la transacción el código deberá tener la siguiente estructura.

```
sqlMapClient.startTransaction();
sqlMapClient.commitTransaction();
sqlMapClient.endTransaction();
```

📖 Invocación de un Stored Procedure es mediante el siguiente tag en el archivo de *Ibatis*

```
<procedure id= "swapEmailAddresses" parameterMap= "swapParameters">
    (call swap_email_address (?,?))
</ procedimiento>
```

📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🔗 <http://ibatisnet.sourceforge.net/DevGuide/ar01s03.html>

Aquí hallará temas al manejo del *Ibatis* y la invocación de los *Stored Procedure*.

🔗 <http://www.nabble.com/Mysql-Database-not-showing-the-inserted-record-td14310290.html>

Aquí hallará ejemplos de transacciones.

🔗 <http://opensource.atlassian.com/confluence/oss/display/IBATIS/How+do+I+call+a+stored+procedure>

Manejo de procedimientos almacenados e *Ibatis*.

**UNIDAD DE
APRENDIZAJE****3****SEMANA****14**

GESTIÓN DE LA CAPA MODELO DATOS MEDIANTE EL *FRAMEWORK IBATIS*

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Los alumnos, utilizando el *framework Ibatis*, implementan una aplicación web que gestiona transacciones de acceso a base de datos y consultas dinámicas obtenidas de archivos de configuración XML.

TEMARIO

- Manejo del Campo BLOB y CLOC mediante *Ibatis*

ACTIVIDADES PROPUESTAS

- Los alumnos implementarán aplicaciones con manejo en el Campo BLOB mediante *Ibatis*.

1. MANEJO DEL CAMPO BLOB Y CLOC MEDIANTE *IBATIS*

Se desarrollará un ejemplo de cómo utilizar el tipo de manejador personalizado de *Ibatis* con objetos grandes (LOB), tales como la BLOB (Binary) y la CLOB (de caracteres). A partir de la liberación, *Ibatis* 2.0.9 tiene por defecto los campos BLOB y CLOB. El ejemplo a continuación se ha hecho para Oracle, pero debería funcionar para cualquier base de datos con un bien escrito controlador JDBC. Asegúrese de que usted utiliza el controlador suministrado de Oracle. Se necesita de recurrir a la última versión de ojdbc14.jar.

Report.sql

```
INFORME (
    ID VARCHAR2 (5),
    nombre VARCHAR2 (25),
    descripción VARCHAR2 (1000),
    datos BLOB
)
```

La creación de un objeto de Java (POJO).

Report.java

```
/ *
 * Report.java
 *
 * Creado el 23 de marzo de 2005, 11:00 AM
 * /
reporting.viewer.domain paquete;

/ **
 *
 * @ Author Nathan Maves
 * /
Informe público de clase (

    / **
    * Posee valor de la propiedad id.
    * /
    private string id;
    / **
    * Posee valor de los bienes nombre.
    * /
    private string nombre;
    / **
    * Posee valor de los bienes descripción.
    * /
    private string id;
    / **
    * Posee valor de la propiedad de datos.
    * /
    privado byte [] datos;

    / / Norma de acceso y mutators

    público byte [] GetData () (
```

```
este regreso. datos;
```

```
)
```

```
public void setData (byte [] data) {
    este. datos = datos;
}
)
```

Conectar tanto la base de datos y el uso de POJO junto a *Ibatis*.

Report.xml

```
<typeAlias alias= "Report" type= "reporting.viewer.domain.Report" />

<resultMap class= "Report" id= "ReportResult">
    <result column= "id" property= "id" />
    <result column= "name" property= "name" />
    <result column= "description" property= "description" />
    <result column= "data" property= "data" jdbcType= "BLOB" />
</ resultMap>

<select id= "getReportById" parameterClass= "string" resultMap=
"ReportResult">
    SELECCIONAR
        *
    DE
        INFORME
    DÓNDE
        id = valor # #
</ select>

<insertar id= "insertReport" parameterClass= "Report">
    INSERT INTO
        INFORME (
            id,
            nombre,
            descripción,
            datos
        ),
        valores (
            # # id,
            # # nombre,
            # # descripción,
            # # datos
        ),
</ insert>

<update id= "updateReport" parameterClass= "Report">
    Informe de actualización establecidos
        name = nombre # #,
        descripción = # # descripción,
        datos datos = # #
    DÓNDE
        id = ID # #
</ update>
```


Cuando se trabaja con un CLOB, se tiene que cambiar la propiedad byte [] por java.lang.String.

Tamaño de los datos más grande que el tamaño máximo


Algunos de los más antiguos jdbc para Oracle tienen problemas con las cadenas que son más grandes. El primer paso para corregir este problema es obtener un controlador JDBC de Oracle que es más reciente 10g. Este controlador trabajará con ambas 9i y 10g bases de datos. Además, se puede tratar de establecer al controlador de la propiedad. La propiedad está SetBigStringTryClob = true. Si está utilizando la SimpleDataSource de *Ibatis* en la aplicación, en el archivo de configuración

```
<property name="Driver.SetBigStringTryClob" value="true"/>
```

Resumen

 Para el manejo del campo BLOB, se tiene que realizar lo siguiente en el archivo de *Ibatis*.

```
<resultMap class= "Report" id= "ReportResult">
    <result column= "id" property= "id" />
    <result column= "name" property= "name" />
    <result column= "description" property= "description" />
    <result column= "data" property= "data" jdbcType= "BLOB" />
</ resultMap>
```

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://www.nabble.com/iBATIS-f360.html>

Aquí un foro donde se ventilan varios puntos del manejo de *Ibatis*