Spring

ÍNDICE

1. INTRODUCCIÓN

Spring es un framework Open Source creado para abordar las complejidades de los desarrollos de las aplicaciones Empresariales Web.

A diferencia de otros frameworks como Struts o JSF, diseñados para simplificar y mejorar el desarrollo de las capas Vista y Controlador en una aplicación Web MVC, Spring resulta ser un framework de propósito general que puede ser utilizado con todos los tipos de desarrollos en Java y en todas las capas en las que se organiza la aplicación.

Spring está organizado en una serie de módulos que pueden ser utilizados individualmente en una aplicación. Cada módulo ofrece una serie de utilidades que permiten simplificar la implementación de una determinada funcionalidad o capa dentro de una aplicación.

1.1. BENEFICIOS DEL USO DE SPRING

Entre los beneficios que podemos encontrar a la hora de utilizar Spring en el desarrollo de aplicaciones Java, encontramos:

- **Integración entre capas**. Una de las características de Spring es la posibilidad de derivar en el entorno de ejecución la instanciación de los objetos y la gestión de su ciclo de vida, facilitando de esta manera la integración entre las capas de una aplicación.

- **Utilización de clases POJO**. El código de una aplicación Spring puede ser definido en clases POJO, lo que reduce la dependencia con el API de Spring. Esto, además de simplificar el desarrollo de las aplicaciones, las hace más flexibles ante cambios.

- **Simplificación de APIs de Java estándar y enterprise**. Spring cuenta con APIs específicos, construidos sobre determinadas clases del Java estándar y enterprise, que simplificar la realización de las tareas propias de dichas clases. En este sentido, se incluye un API de acceso a datos que encapsula la funcionalidad JDBC y JPA, permitiendo al desarrollador abstraerse de gran cantidad de detalles durante la creación del código de acceso a los datos.

- **Ligero**. Spring es un framework muy ligero tanto en términos de tamaño (el framework completo se distribuye en un único archivo .jar con tamaño de alrededor de 1 MB) y de sobrecarga. La penalización de rendimiento en las aplicaciones que supone el uso de Spring es prácticamente inapreciable.

- **Alternativa al uso de EJB**. A la hora de desarrollar la capa de negocio de una aplicación, Spring proporciona alternativas al uso de EJB que hace más flexible y cómoda la realización de esta capa, como por ejemplo las técnicas de orientación a aspectos (AOP).

1.2. MÓDULOS SPRING

Como ya hemos indicado, Spring se encuentra organizado en varios módulos funcionales que pueden ser utilizados de manera independiente. En la figura 1 tenemos una imagen que nos muestra los principales módulos que componen éste framework.

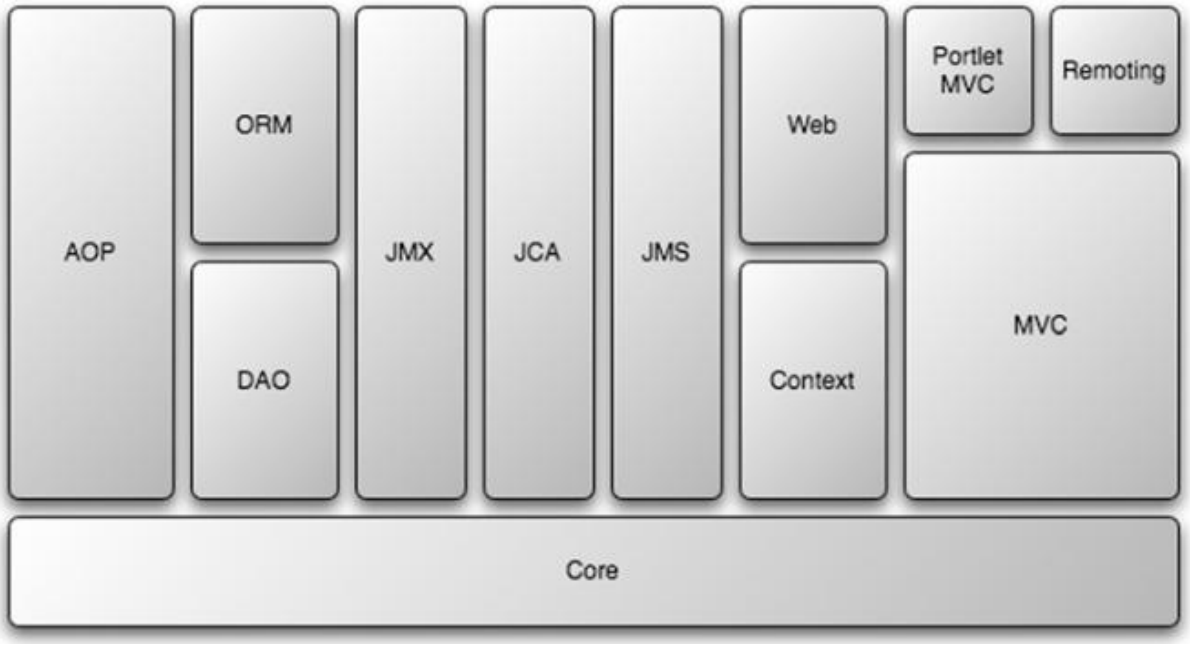

Figura. 1.

Seguidamente, comentaremos brevemente las características de algunos de ellos:

- **Core**. Constituye la capa base sobre la que están construidos el resto de módulos de Spring. La posibilidad de configurar objetos y definir las dependencias entre los mismos en archivos de configuración XML, es una de las principales características de Spring y es proporcionada precisamente por este módulo. Esta característica, constituye la base de lo que se conoce como Inversión de Control (IoC).

- **AOP**. Otra de las principales funcionalidades ofrecidas por Spring, la Programación Orientada a Aspectos, es proporcionada por este módulo.

- **DAO**. A fin de simplificar las operaciones de acceso a datos en una aplicación, Spring cuenta con el módulo DAO, el cual incluye una serie de clases que encapsulan parte de las instrucciones de acceso a datos que habitualmente se incluyen en la capa de negocio de una aplicación, tanto vía JDBC como JPA.

- **ORM**. Como complemento al anterior, Spring incluye un módulo ORM para poder establecer las reglas de mapeo-relacional y poder utilizar así una capa de persistencia en la aplicación. Este módulo ORM se integra con la mayoría de los frameworks de persistencia existentes, como JPA, Hibernate o iBatis.

- **Web MVC**. Este módulo representa todo un framework MVC completo para facilitar al programador la construcción de aplicaciones Web siguiendo este patrón. Aunque Spring puede integrarse con otros frameworks MVC como Struts, este framework propietario puede aprovecharse de las funcionalidades proporcionadas por el Core como la IoC.

2. INVERSIÓN DE CONTROL

La Inversión de Control (IoC) es, sin duda alguna, la principal funcionalidad que proporciona el framework Spring. Dado que dicha funcionalidad está incluida en el Core, puede ser aprovechada en cualquier tipo de aplicación.

La IoC se basa fundamentalmente en la posibilidad de obtener referencias a objetos dentro de un programa, sin necesidad de especificar en el código los datos de localización de de estos objetos, lo que proporciona un total desacoplamiento entre las capas de una aplicación.

Por ejemplo, si desde la capa Web necesitamos hacer uso de un objeto de negocio para realizar una determinada operación de lógica implementada por dicho objeto, utilizando Spring la capa web no tendrá que preocuparse de los detalles tales como que tipo de objeto es, donde está localizado, etc., simplemente tendrá que solicitar al contenedor de Spring (localizado en el módulo Core) que le proporcione el objeto, encargándose dicho contenedor de su localización.

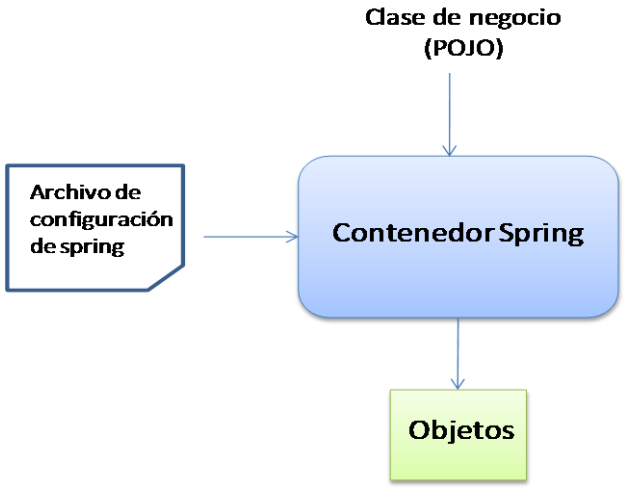

Figura. 2.

Lógicamente, para que esto sea posible, será necesario definir en algún lugar los detalles necesarios para que el contenedor IoC pueda localizar y configurar este objeto, este lugar es el **archivo de configuración de Spring**.

Seguidamente, vamos a ver los diferentes actores que intervienen en una aplicación Spring de estas características.

2.1. CREACIÓN DE LA CLASE BEAN

Aunque existen diferentes posibilidades en cuanto los tipos de objetos que pueda crear un contenedor IoC, uno de los principios básicos es que la clase a instanciar debe cumplir con los condicionamientos de un JavaBean, es decir, deberá tener un constructor con parámetros y métodos set/get para poder configurarlo a través de propiedades. Este último no es un condicionante estricto pues, como veremos más adelante, Spring también permite configurar instancias a través de constructores con parámetros.

Como primer ejemplo de aplicación Spring, supongamos que tenemos una clase que realiza una serie de operaciones sobre una determinada cadena de caracteres que se suministra a través de un método *set*. Dichas operaciones consisten en el cálculo de la longitud de caracteres de la cadena y del número de vocales de la misma. Ha aquí el código de dicha clase:

```
package beans;

public class OperaCadenas {
```

```
    private String texto;
    public String getTexto() {
        return texto;
    }
    public void setTexto(String texto) {
        this.texto = texto;
    }
    public int contarCaracteres(){
        return texto.length();
    }
    public int contarVocales(){
        int total=0;
        for(int i=0;i<texto.length();i++){
            switch(texto.toLowerCase().charAt(i)){
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                    total++;
            }
        }
        return total;
    }
}
```

Como vemos, la clase incluye un constructor sin parámetros (en este caso, es el constructor por defecto) y una pareja de métodos set/get para configurar la propiedad “texto” que representa la cadena de caracteres sobre la que se va a operar, por tanto, cumple los requerimientos para poder ser instanciada y configurada por el contenedor de Spring.

2.2. EL ARCHIVO DE CONFIGURACIÓN DE BEANS

En el archivo de configuración de beans es donde indicamos al contenedor las instancias que queremos crear, las propiedades que deben ser configuradas en cada una, dependencias con otras instancias, etc.

Este archivo puede tener cualquier nombre, y deberá estar localizado en alguno de los paquetes de la aplicación. Su estructura se muestra en el siguiente listado:

```
<?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.xsd">

  <bean >

      //configuración bean 1

      :

  </bean>

  <bean >

      //configuración bean 2

      :

  </bean>

</beans>
```

Dentro del elemento principal <beans> se definirán los diferentes beans a instanciar. Cada elemento <bean> establecerá los datos para poder instanciar una clase, datos que serán suministrados a través de los atributos del elemento.

Entre los principales atributos de <bean> tenemos:

- **id**. Representa el identificador de la instancia. Este identificador es necesario para poder obtener una referencia a la instancia desde el código de la aplicación.
- **class**. Nombre cualificado de la clase a instanciar.
- **scope**. Modo de instanciación del bean. Aunque existen otros posibles valores a utilizar (solo válidos en aplicaciones Web), los principales valores de este atributo serán:
    - **Singleton**. Crea una única instancia del bean, que será compartida por todas las solicitudes del bean. Este es el valor por defecto del atributo.
    - **Prototype**. Cada petición del bean tendrá su propia instancia.

De cara a configurar la instancia, le tenemos que indicar a Spring los valores que debe establecer en las diferentes propiedades del bean, una vez que ésta se haya creado. Para ello utilizaremos el elemento <property>, uno por cada propiedad a configurar. Cada elemento deberá definir, en el caso más sencillo, los siguientes atributos:

- **name**. Nombre de la propiedad a establecer. A la hora de asignar el valor a la propiedad del objeto, Spring buscará en la instancia un método cuyo nombre sea "set", seguido del valor definido en este atributo con la primera letra en mayúscula.

- **value**. Valor que se establecerá en la propiedad, es decir, el que se pasará como parámetro en la llamada al método *set*.

En el caso de la clase de ejemplo "OperaCadenas", la manera en la que podríamos crear y configurar la instancia sería la siguiente:

```
<bean id="oper" class="beans.OperaCadenas" >

    <property name="texto" value="Cadena de prueba"/>

</bean>
```

Donde "oper" es el identificador que le hemos asignado a la instancia y "Cadena de prueba" es el valor inicial que el contenedor a la propiedad "texto".

Más adelante analizaremos las diferentes posibilidades que ofrece Spring a la hora de configurar un bean.

2.3. RECUPERACIÓN DE LA INSTANCIA POR LA APLICACIÓN.

La interfaz **BeanFactory**, localizada en el paquete org.springframework.beans.factory, representa el elemento principal de conexión con el motor de Spring y proporciona los métodos necesarios para acceder a las instancias manejadas por el framework. Además de esta, tenemos otra interfaz llamada ApplicationContext que extiende a la anterior e incluye más métodos que proporcionan características adicionales.

De cara a poder comunicarnos con el Core de Spring desde la aplicación, necesitamos una implementación de alguna de las interfaces anteriores. La clase **ClassPathXmlApplicationContext**, incluida en org.springframework.context.support, representa una implementación de ApplicationContext. Esta clase puede ser instanciada indicando en su constructor la localización del archivo de configuración de beans; a partir de ahí, podemos utilizar el método *getBean()* definido en la interfaz BeanFactory para obtener una referencia a cualquiera de las instancias definidas en dicho archivo.

El siguiente programa obtendría la instancia "texto" del ejemplo anterior, definida en el archivo de configuración "ejemplo1Config.xml", para después llamar a los métodos de operación implementados en la clase:

```
import org.springframework.beans.factory.BeanFactory;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import beans.*;

public class Prueba {

   public static void main(String[] args) {
```

```
        BeanFactory factory= (BeanFactory)
            new ClassPathXmlApplicationContext("ejemplo1Config.xml");
        OperaCadenas op=(OperaCadenas)factory.getBean("oper");
        System.out.println("Total caracteres: "+op.contarCaracteres());
        System.out.println("Total vocales: "+op.contarVocales());
    }
}
```

En el ejemplo anterior vemos como, a partir del identificador del bean definido en el atributo id del elemento <bean> correspondiente, podemos obtener una referencia al mismo utilizando el método *getBean()*.

Aunque el anterior es un ejemplo sencillo, podemos apreciar **como la creación del objeto y la inicialización de sus propiedades es algo que no se realiza desde el código de la aplicación**, tan solo tenemos que preocuparnos de pedirle al contenedor la instancia cuyo identificador se especifica y ya se encargará Spring de todo el proceso de instanciación y configuración a partir de la información suministrada en el archivo de configuración de beans.

Lo interesante del código anterior es que, si la creación del objeto OperaCadenas fuera un proceso más complejo, en el que hubiera que crear otros objetos dependientes o definir toda una serie de propiedades de configuración/localización, el código de la aplicación no cambiaría en absoluto, pues todo este proceso quedaría definido dentro del archivo de configuración.

Este puede ser, por ejemplo, el caso de los DataSource, cuya creación implica la definición de toda una serie de propiedades de configuración. Más adelante, veremos un ejemplo de este tipo.

3. UTILIZACIÓN DE SPRING EN UNA APLICACIÓN JAVA CON NETBEANS

Antes de avanzar en el estudio de las capacidades que nos ofrece el Core de Spring, veamos cómo podemos hacer uso de este framework en uno de los entornos de desarrollo más comúnmente utilizado en la creación de aplicaciones Java: NetBeans. Las siguientes pantallas pueden variar en función de la versión que tengas del framework, aunque el procedimiento no cambia.

Si en una aplicación Java de cualquier tipo creada con NetBeans queremos hacer uso del Core de Spring, el procedimiento a seguir resulta bastante sencillo.

Lo primero será incluir el archivo de configuración de Spring en el proyecto, para lo cual nos situaremos sobre el icono de la aplicación en el explorador de proyectos y elegimos la opción *New -> Other* en el menú contextual. A continuación, en el cuadro de diálogo “New File” seleccionamos la opción “Spring XML Configuration File” dentro de la categoría “Other”, tal y como se ilustra en la figura 3.

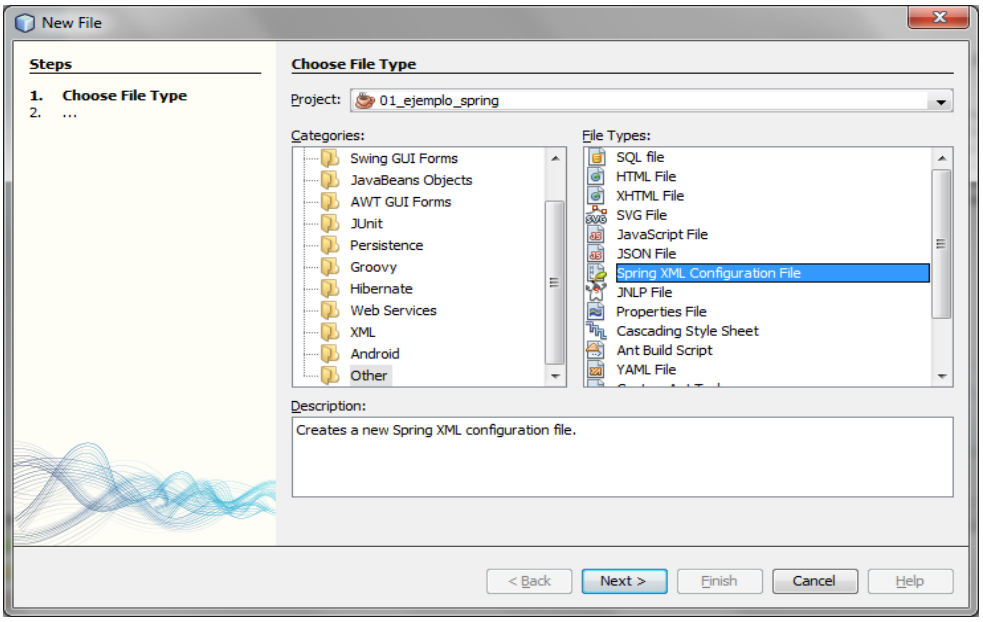

Figura. 3.

Al pulsar el botón “Next” el asistente nos pedirá el nombre del fichero y su ubicación, que puede ser cualquiera de los paquetes de clases de la aplicación dentro del directorio *src*. Podemos llamarle “ejemplo1Config.xml” como en el ejemplo anterior.

En el último paso (figura 4), el asistente nos pregunta que referencias a espacios de nombres de la especificación Spring queremos incluir en el archivo de configuración. Si solo vamos a utilizar el Core, en principio no es necesario incluir ninguno de los espacios de nombres propuestos.

También aparece en este último paso un botón titulado “Add Spring Framework” que al ser pulsado provocará que se incluyan en la carpeta libraries del proyecto todas las referencias a los archivos .jar en los que está organizado el framework Spring, por lo que directamente podremos tener acceso desde código a todos los paquetes de clases.

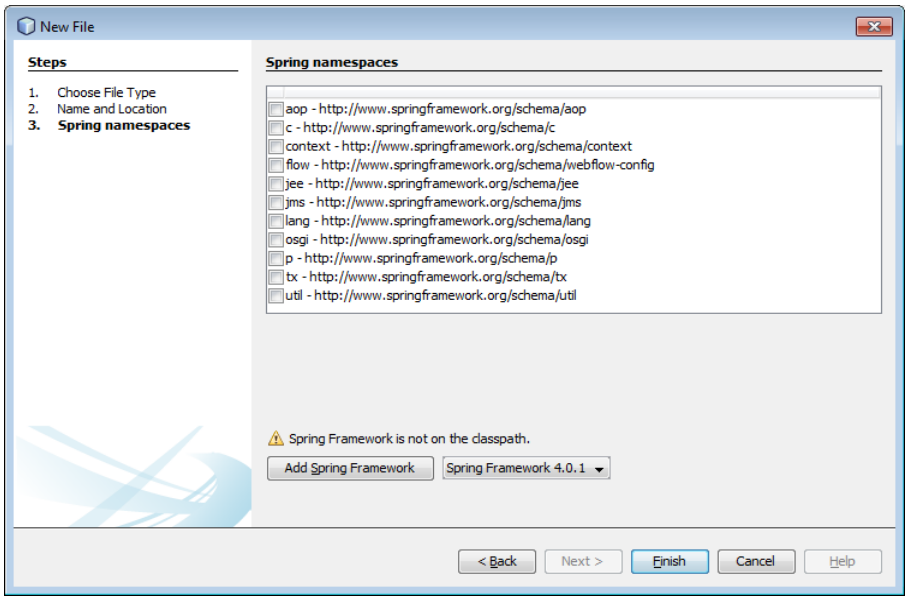

Figura. 4.

Pulsaremos el botón “Finish” y ya podremos hacer uso del core de Spring en nuestra aplicación.

Para probar el funcionamiento de Spring, podemos añadir al archivo de configuración las instrucciones de ejemplo que indicamos anteriormente:

```
<bean id="oper" class="beans.OperaCadenas" >

    <property name="texto" value="Cadena de prueba"/>

</bean>
```

Si añadimos al proyecto las clases OperaCadenas.java y Prueba.java con el código indicado en los listados anteriores, tras ejecutar la clase Prueba se deberá mostrar en la ventana de salida de netbeans lo siguiente:

Total caracteres: 16

Total vocales: 7

4. EL MÓDULO SPRING CORE

A lo largo de este capítulo vamos a estudiar las funcionalidades que nos ofrece el módulo principal de Spring, también conocido como Spring Core. Dichas funcionalidades podremos utilizarlas de forma independiente en las aplicaciones, o en combinación con alguno de los otros módulos de Spring para desarrollos en algún entorno específico.

4.1. CONFIGURACIÓN DE BEANS

Anteriormente hemos visto como configurar de forma básica un bean en el archivo de configuración de Spring de cara a poder ser instanciado desde el contenedor. No obstante, se pueden dar diferentes posibilidades a la hora de configurar un bean que vamos a analizar en este apartado.

4.1.1. Configuración a través de propiedades

Constituye el tipo de configuración que hemos utilizado en los ejemplos anteriores, donde a través del elemento <property> asignamos valores a los atributos de un bean mediante llamadas a los métodos *set*.

Los elementos <property> se indican en el interior de <bean> y **serán procesados por el framework inmediatamente después de la creación de la instancia**.

Además de la utilización del elemento <property>, desde la versión 2 Spring nos permite configurar una propiedad de un bean a través de **atributos del elemento <bean>**, y es que, utilizando el espacio de nombres p, las propiedades de los bean se exponen como atributos del elemento. Por ejemplo, partiendo de la configuración del bean OperaCadenas utilizado en los ejemplos anteriores:

```
<bean id="oper" class="beans.OperaCadenas" >

    <property name="texto" value="Cadena de prueba"/>
```

```
</bean>
```

Si empleamos el espacio de nombres p, el bean se podría configurar también de la siguiente forma:

```
<bean id="oper" class="beans.OperaCadenas" p:texto=”Cadena de prueba”/>
```

Como vemos, cada propiedad es expuesta como un atributo de <bean>, siendo el valor asignado al atributo el valor que se establecerá a la propiedad. Para poder utilizar el espacio de nombres p, será necesario añadir el siguiente atributo en la declaración del elemento raíz del documento:

```
xmlns:p="http://www.springframework.org/schema/p"
```

4.1.2. Configuración a través de constructores

Además de poder establecer valores en el bean a través de propiedades, se puede indicar al contenedor de Spring que realice llamadas a los constructores de la clase durante el proceso de creación de la instancia. Para ello, se deberá utilizar un elemento <constructor-arg> en el interior del elemento <bean> por cada parámetro del constructor, indicando en el atributo *value* de cada elemento el valor que se pasará al parámetro en la llamada.

Si nuestra clase OperaCadenas tuviera el siguiente constructor:

```
public OperaCadenas(String texto){
      this.texto=texto;
 }
```

podría ser configurada en el archivo de configuración de la siguiente manera:

```
<bean id="oper" class="beans.OperaCadenas" >
    <constructor-arg value="Cadena de prueba"/>
 </bean>
```

Supongamos ahora que tenemos la siguiente clase:

```
public class Operaciones{
        private int operando1, operando2;
        private String operador;
        public Operaciones(int op1, String oper){
                this.operando1=op1;
                this.operador=oper;
        }
        public Operaciones(int op1, int op2){
                this.operando1=op1;
```

```
            this.operando2=op2;
        }
        :
    }
```

Si en el archivo de configuración declaramos el bean de la siguiente manera:

```
<bean id=”operaciones” class=”beans.Operaciones”>
        <constructor-arg value=”100”/>
        <constructor-arg value=”50”/>
</bean>
```

Lo esperable es que a la hora de instanciar el bean, el contenedor utilice el segundo de los constructores definidos en la clase, sin embargo no es así, será ejecutado el primer constructor. El motivo es que, al no haber definido ningún tipo para los argumentos del constructor, **Spring convierte todos los valores a String** y a partir de ahí comienza a buscar una coincidencia; al no existir ningún constructor con dos parámetros String, intenta convertir cada valor en el tipo del parámetro requerido por el constructor y en el primer constructor en el que encuentra una coincidencia es en el primero.

Si quisiéramos ejecutar el segundo constructor o cualquier otro cuyos tipos de parámetro coincidan con los valores pasados en la llamada, deberíamos definir explícitamente el tipo de los argumentos en la definición del bean empleando el atributo *type* de cada elemento <constructor-arg>. En nuestro ejemplo sería:

```
<bean id="operaciones" class="beans.Operaciones">
        <constructor-arg value="100" type="int"/>
        <constructor-arg value="50" type="int"/>
 </bean>
```

4.1.3. Argumentos de tipo colección

En algunas ocasiones nos podemos encontrar con que algunos de los atributos que debemos inicializar a través del constructor o de métodos de propiedad son de tipo colección. Para poder definir valores en este tipo de datos a través del archivo de configuración, Spring cuenta con los elementos <list>, <set> y <map>, según el tipo de colección.

A la hora de definir los valores de una determinada propiedad, estos elementos se incluirán como subelementos de <property>. En el caso de propiedades de tipo lista o conjunto, cada <list> o <set> incluirá una serie de elementos <value> para establecer cada uno de los valores de la colección.

Supongamos que la clase Operaciones del ejemplo anterior incluye una propiedad “sumandos” de tipo lista de enteros:

```
public class Operaciones{
   private List<Integer> sumandos;
```

```
    public List<Integer> getSumandos () {
        return sumandos;
    }
    public void setSumandos (List<Integer> sumandos) {
        this. sumandos = sumandos;
    }
        :
}
```

Si queremos inicializar esta propiedad con una serie de valores enteros desde el archivo de configuración, tendríamos que incluir en el mismo la siguiente entrada dentro del elemento <bean>:

```
<property name="sumandos">
        <list>
            <value>25</value>
            <value>38</value>
            <value>90</value>
        </list>
 </property>
```

Tras la creación de la instancia y la llamada al constructor coincidente con los argumentos definidos, el contenedor llamará al método *setSumandos()* del bean pasándole como parámetro un objeto List compuesto por los valores indicados en <list>.

En caso de tratarse de una colección de tipo Set, la definición de valores a asignar en la colección sería similar al anterior:

```
<property name="sumandos">
        <set>
            <value>25</value>
            <value>38</value>
            <value>90</value>
        </set>
 </property>
```

Si estuviéramos ante una propiedad de tipo Map, dado que cada elemento cuenta con una clave y un valor, el contenido de cada uno de estos elementos se establecería a través de un subelemento <entry>, dentro del cual habría que indicar el <key> y <value> asociados.

Por ejemplo, si quisiéramos inicializar la propiedad “empleados” de un bean, donde cada empleado está caracterizado por un nombre y un código de empleado que lo identifica, deberíamos incluir algo similar a esto:

```
<property name=”empleados”>
    <map>
        <entry>
            <key>3260</key>
            <value>Manuel García</value>
        </entry>
        <entry>
            <key>6230</key>
            <value>Laura Jiménez</value>
        </entry>
    </map>
</property>
```

Lo comentado es aplicable también para constructores con argumentos de tipo colección. Si en la clase Operaciones tuviéramos este constructor:

```
public Operaciones(List<Integer> sumandos){
    this.sumandos=sumandos;
}
```

La ejecución del mismo a través del archivo de configuración de Spring sería:

```
<constructor-arg>
    <List>
       <value>25</value>
       <value>38</value>
       <value>90</value>
    </List>
 </property>
```

4.2. INYECCIÓN DE DEPENDENCIA

La inyección de dependencia es una técnica utilizada por algunos entornos de ejecución y frameworks, mediante la cual éstos se encargan de localizar ciertos tipos de objetos que la aplicación necesita y depositarlos (inyectarlos) en variables ya preparadas por ésta, evitando que el programador tenga que preocuparse de estos detalles de localización de los objetos.

Una de las principales aplicaciones de la IoC en Spring es precisamente la utilización de esta técnica para “inyectar” en determinadas propiedades de un bean otros beans que éste necesita para realizar su función. Para ello, el bean que representa al objeto principal (el que requiere una referencia a un segundo objeto), deberá declarar una propiedad del tipo de objeto requerido, donde Spring inyectará dicho objeto.

Para analizar el funcionamiento de la inyección de dependencia en Spring vamos a utilizar el siguiente ejemplo: supongamos que tenemos una clase Fichero que nos permite escribir datos en un determinado fichero y recuperar también la información almacenada en el mismo. El código de dicha clase podría ser el que se muestra en el siguiente listado:

```
package beans;

import java.util.List;

import java.io.*;

import java.util.ArrayList;

public class Fichero {

   private String ruta;

   public Fichero(String ruta){

      this.ruta=ruta;

   }

   public void escribir(String dato){

      try{

         FileWriter out=new FileWriter(ruta, true);

         //escribe la cadena en el fichero cuya

         //ruta se suministra al constructor

         out.write(dato+"\n");

         out.close();

      }

      catch(IOException ex){ex.printStackTrace();}

   }

   public List<String> recuperar(){
```

```
        ArrayList<String> todas=new ArrayList<String>();
        try{
            BufferedReader reader= new BufferedReader(new FileReader(ruta));
            String s;
            //recupera las cadenas almacenadas y las
            //guarda en un arraylist
            while((s=reader.readLine())!=null){
                todas.add(s);
            }
            reader.close();
        }
        catch(IOException ex){ex.printStackTrace();}
        return todas;
    }
}
```

Ahora, queremos implementar una clase que, haciendo uso de un objeto Fichero, pueda solicitar datos al usuario para guardarlos en un archivo y mostrarle por pantalla la información almacenada. Dado que esta clase, a la que llamaremos Operaciones, necesitará hacer uso de un objeto Fichero, declararemos un atributo de este tipo, con sus correspondientes métodos set/get, para que el contenedor pueda inyectar una referencia a dicho objeto:

```
package beans;
import java.util.List;
import java.util.Scanner;
public class Operaciones {
    //variable en la que el contendor Spring
    //inyectará el objeto Fichero
    private Fichero f;
    public Fichero getFichero() {
        return f;
    }
    public void setFichero(Fichero f) {
```

```
        this.f = f;
    }
    public void escribirCadena(){
        Scanner sc=new Scanner(System.in);
        System.out.println("Introduzca cadena");
        f.escribir(sc.nextLine());
    }
    public void mostrarCadenas(){
        List<String> cadenas=f.recuperar();
        for(String s:cadenas){
            System.out.println(s);
        }
    }
}
```

La clase donde obtendremos un objeto Operaciones y probaremos sus métodos se muestra en el siguiente listado:

```
package inicio;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import beans.*;

public class Prueba {
    public static void main(String[] args) {
        BeanFactory factory= (BeanFactory)
                    new ClassPathXmlApplicationContext("ejemplo3Config.xml");
        Operaciones op=(Operaciones)factory.getBean("operaciones");
        op.escribirCadena();
        op.escribirCadena();
        op.mostrarCadenas();
```

```
    }
}
```

Finalmente, será en el archivo de configuración de Spring donde definiremos la dependencia de objetos a fin de que el contenedor pueda aplicar la técnica de inyección:

```
<?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-2.5.xsd">

    <bean id="fichero" class="beans.Fichero">

        <constructor-arg value="c:\testFile.txt" />

    </bean>

    <bean id="operaciones" class="beans.Operaciones">

        <property name="fichero" ref="fichero"/>

    </bean>

</beans>
```

Según vemos en el listado anterior, se utilizará el atributo *ref* para indicar el identificador del objeto que debe ser inyectado en la propiedad. Este atributo también puede ser utilizado en <constructor-arg> para inyectar objetos a través del constructor.

4.2.1. Auto-cableado de beans

A fin de simplificar las tareas de configuración de beans, Spring permite inyectar referencias a beans en propiedades de tipo objeto de forma automática. Para realizar esta tarea utilizaremos el atributo **autowire** del elemento beans, cuyos posibles valores pueden ser los siguientes:

- **byName**. Todas aquellas propiedades de tipo bean del objeto, que no hayan sido establecidas explícitamente, serán inyectadas con instancias cuyos identificadores coincidan con el nombre de la propiedad. En el ejemplo anterior podemos configurar el bean operaciones de la siguiente forma:

```
<bean id=”operaciones” class=”beans.Operaciones” autowired=”byName”/>
```

  De esta manera, la propiedad *fichero* será inicializada con el bean *fichero* definido en el archivo de configuración.

- **byType**. En este caso, Spring inyectará en las propiedades de tipo bean objetos que sean del mismo tipo que la propiedad. Si existe más de un objeto en el contenedor de Spring que cumple la condición, no será posible

determinar cuál de ellos hay que inyectar, por lo que se lanzará una excepción UnsatisfiedDependencyException.

- **constructor**. Con este valor en el atributo autowired se aplicará el auto-cableado en los constructores del bean en función del tipo. Si la clase Operaciones tuviera un constructor que recibiera como parámetro el objeto fichero, bastaría con definir el bean de la siguiente forma para que el objeto fichero se inyectara en el constructor del objeto operaciones durante la creación del mismo:

<bean id="operaciones" class="beans.Operaciones" **autowired="constructor"**/>

Igual que en el caso de *byType*, se debe procurar evitar ambigüedades que impidan al contenedor realizar la inyección de dependencias correctamente.

4.2.2. La anotación @Autowired

El atributo autowired del elemento <bean> permite inicializar todas las propiedades o parámetros de constructor que sean de tipo bean, pero no permite hacerlo sobre una propiedad o constructor en concreto.

Desde la versión Spring 2.5, tenemos a nuestra disposición la anotación **@Autowired**. @Autowired puede colocarse delante del método set asociado a la propiedad que queramos auto-inyectar, de modo que durante la inicialización del bean, el contenedor buscará un objeto compatible con el tipo de objeto definido en el método set y lo inyectará en el mismo. Si definimos el método *setFichero()* de la clase Operaciones de la siguiente manera, la propiedad quedará inicializada automáticamente con el objeto fichero durante la creación del bean:

```
@Autowired

public void setFichero(Fichero f){

        this.f=f;

}
```

También podemos utilizar @Autowired para anotar el constructor de la clase si la inyección se realiza a través de este:

```
@Autowired

public Operaciones(Fichero f){

        this.f=f;

}
```

En la definición del bean operaciones ya no será necesario configurar explícitamente la propiedad ni indicar el atributo autowired:

```
<bean id="operaciones" class="beans.Operaciones"/>
```

La anotación @Autowired se basa en la inyección de dependencias por tipo, de podo que pueden producirse ambigüedades si existe más de un bean coincidente con el tipo de la propiedad. Para evitar estas ambigüedades, puede utilizarse conjuntamente con @Autowired la anotación **@Qualifier**, que permite especificar el identificador del bean que se quiere inyectar.

Supongamos que tenemos dos beans de tipo de fichero definidos en el archivo de configuración de Spring:

```
<bean id="fichero" class="beans.Fichero">

    <constructor-arg value="c:\testFile.txt" />

  </bean>

<bean id="fichero2" class="beans.Fichero">

    <constructor-arg value="c:\newText.txt" />

</bean>
```

Si queremos inicializar la propiedad fichero del bean operaciones con el objeto fichero2, deberíamos definir el método setFichero de la clase Operaciones de la siguiente manera:

```
@Autowired

public void setFichero(@Qualifier("fichero2") Fichero f){

        this.f=f;

}
```

La anotación @Autowired puede aplicarse también a constructores y directamente sobre atributos. En este último caso, podemos prescindir incluso de la necesidad de disponer de métodos set/get para acceder al atributo, pues Spring empleará reflexión para inyectar la referencia al bean en el atributo.

Para que la anotación @Autowired sea reconocida por Spring, **es necesario incluir en el archivo de configuración de Spring el elemento annotation-config** del espacio de nombres context:

```
<context:annotation-config />
```

EJERCICIO 1

Aunque en el capítulo de acceso a datos tendremos oportunidad de estudiar todas las facilidades que nos ofrece Spring en este área, vamos a desarrollar en este ejercicio la lógica de negocio de una aplicación que realiza la autenticación de usuarios contra una base de datos, base de datos que llamaremos "ejemplo" y, como mínimo, tendrá una tabla "clientes" con dos campos "usuario" y "password"

La aplicación utilizará un DataSource para obtener conexiones a la base de datos, la implementación de DataSource que emplearemos será una de las proporcionadas por el API de Spring y a través de la inyección de dependencia, inyectaremos un objeto de esta clase en la lógica de negocio de la aplicación.

La aplicación será creada como un proyecto de aplicación Java estándar, siguiendo los pasos descritos en el punto anterior para la inclusión de un archivo de configuración de Spring. No debemos olvidarnos de incluir en la sección *libraries* del proyecto, tanto la librería de Spring como el driver para MySQL.

Una vez creado el proyecto, crearemos una interfaz que define los métodos de la lógica de negocio de la aplicación:

```
package modelo;

public interface GestionClientes {

    boolean login(String user, String password);

}
```

La clase que implementa la interfaz de negocio quedará:

```
package modelo;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

public class GestionClientesImpl implements GestionClientes {

    @Autowired

    @Qualifier("data")

    private DataSource ds;

    @Override

    public boolean login(String user, String password){

        Boolean res=false;

        Connection cn=null;

        try{

            cn=ds.getConnection();

            String sql="Select * from clientes where usuario='"+user;

            sql+="' and password='"+password+"'";

            Statement st=cn.createStatement();

            ResultSet rs=st.executeQuery(sql);

            res=rs.next();
```

```
            }
            catch(Exception ex){
                ex.printStackTrace();
            }
            finally{
                try {
                    if(cn!=null){
                        cn.close();
                    }
                } catch (SQLException ex) {
                    ex.printStackTrace();
                }
            }
            return res;
        }
    }
```

Destacar el uso de las anotaciones @Autowired y @Qualifier para la inyección del DataSource en el atributo.

Por su parte, el archivo de configuración de Spring ejercicio1Config.xml quedará de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
       <!--datasource de spring-->
       <bean id="data"
             class="org.springframework.jdbc.datasource.DriverManagerDataSource">
          <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
```

```
            <property name="url" value="jdbc:mysql://localhost:3306/ejemplo"/>
            <property name="username" value="root"/>
            <property name="password" value="root"/>
        </bean>
        <!--clase del modelo-->
        <bean id="gestion" class="modelo.GestionClientesImpl"/>
        <context:annotation-config />
    </beans>
```

Podemos probar el funcionamiento de la aplicación desde el método *main* de la clase principal. Partiendo de un usuario existente en la base de datos (test1/test1), nos debería aparecer el menaje “resgistrado” al ejecutar el siguiente código:

```
package principal;
import modelo.GestionClientes;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Ejercicio1 {
   public static void main(String[] args) {
      BeanFactory factory=
      (BeanFactory)new ClassPathXmlApplicationContext("ejercicio1Config.xml");
      GestionClientes model=(GestionClientes)factory.getBean("gestion");
      if(model.login("test1", "test1")){
         System.out.println("registrado");
      }
      else{
         System.out.println("no registrado");
      }
   }
}
```

4.3. CONFIGURACIÓN A TRAVÉS DE ANOTACIONES

Una de las principales novedades que se introdujeron a partir de la versión 3 de Spring es la posibilidad de utilizar anotaciones para configurar los beans, evitando así el uso de

archivos de configuración XML, una tendencia cada más extendida entre las últimas versiones de los frameworks Java más utilizados.

4.3.1. Implementación de la clase de configuración

Para configurar los beans con esta técnica, necesitamos definir una clase adicional en la que se le indicará al contenedor IoC de Spring la manera en la que tiene que crear y configurar las instancias. Esta clase deberá estar definida con la anotación **@Configuration** del paquete org.springframework.context.annotation:

```
@Configuration

public class ClaseConfiguracion{



}
```

Esta clase podrá tener cualquier nombre que elijamos y, por cada bean que tenga que crear el contenedor, implementaremos un método que definirá la manera en la que debe crearse la instancia. El método deberá devolver la instancia creada y deberá estar declarado con la anotación **@Bean**, localizada en el mismo paquete que la anterior. El formato de este método será el siguiente:

```
@Bean

public ClaseBean metodo(){

        //instrucciones de creación de la instancia

}
```

Por ejemplo, la configuración del bean OperaCadenas que utilizamos en el primer ejemplo presentado, quedaría de la siguiente manera:

```
@Configuration

public class Configurador{

        @Bean

        public OperaCadenas operaCadenas(){

                OperaCadenas op=new OperaCadenas();

                op.setTexto("cadena ejemplo");

                return op;

        }

}
```

Según vemos en el ejemplo anterior, suele ser habitual que el nombre del método coincida con el del bean, con la primera letra minúscula. El motivo de esto es que, como veremos a continuación, **el nombre del método se utiliza como identificador de la instancia a la hora de recuperarla desde la aplicación**.

4.3.2. Obtención de la instancia.

Para poder obtener la instancia del bean configurado de esta manera, no podemos utilizar la clase ClassPathXmlApplicationContext como hasta el momento, sino otra implementación de BeanInterfaz que permita obtener los datos de configuración de la clase anotada con @Configuration, esta clase es **AnnotationConfigApplicationContext** y está definida en el paquete org.springframework.context.annotation. El constructor de esta clase recibe como parámetro un objeto Class que representa la clase de configuración.

Como indicamos anteriormente, el nombre del método de configuración se utilizará como identificador a la hora de obtener el bean:

```
public class Prueba {

   public static void main(String[] args) {

      BeanFactory factory=

        (BeanFactory)new AnnotationConfigApplicationContext(Configurador.class);

      //operaCadenas es el nombre del método de configuración

      OperaCadenas op=(OperaCadenas)factory.getBean("operaCadenas");

      System.out.println("Total caracteres: "+op.contarCaracteres());

      System.out.println("Total vocales: "+op.contarVocales());

   }

}
```

4.3.3. Las librerías ASM

Para poder ejecutar una aplicación basada en anotaciones, el entorno de ejecución de Spring se apoya en ASM. ASM es una librería Java para el análisis y manipulación de bytecodes y es utilizada por el Core de Spring para modificar dinámicamente los bytecodes y generar el nuevo código en tiempo de ejecución a partir de la información de configuración de las anotaciones.

Estas librerías no forman parte de paquete de distribución de Spring, por lo que tendremos que descargarlas de manera independiente desde la dirección http://asm.ow2.org/download/index.html. Una vez descargado y descomprimido el archivo .zip, encontraremos una serie de archivos de librería .jar en la carpeta \lib que habrá que añadir a la sección “libraries” del proyecto netbeans, tal y como se indica en la imagen de la figura 5.

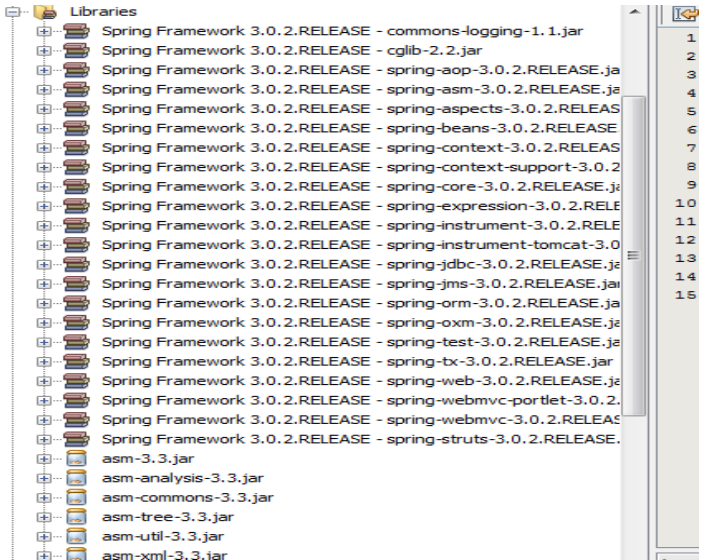

Figura. 5.

EJERCICIO 2

En este ejercicio vamos a desarrollar una nueva versión de la aplicación de ejemplo presentada en el punto 4.2, consistente en un programa que realiza lecturas y escrituras de cadenas en un fichero de texto.

En esta ocasión, se trata de realizar la configuración del sistema y, por tanto, la inyección de dependencias de objetos, a través de anotaciones. Las clases principales, Operaciones y Fichero, no sufrirán ningún cambio respecto al código ya presentado.

La parte más importante de este ejercicio será la clase de configuración, que quedará como se indica en el siguiente listado:

```
package config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import beans.*;

@Configuration

public class Configurador {

    @Bean

    public Fichero fichero() {

        //instancia el bean Fichero

        Fichero t=new Fichero("ejemplo.txt");

        return t;

    }

    @Bean

    public Operaciones operaciones() {

        Operaciones op=new Operaciones();

        //inyecta el bean Fichero
```

```
            op.setFichero(fichero());
            return op;
        }
    }
```

A la hora de obtener una instancia del bean operaciones desde código, utilizaremos la clase AnnotationConfigApplicationContext, que es más apropiada para el caso en que la configuración se realice a través de anotaciones en vez desde XML:

```
package ejercicio2;

import beans.Operaciones;
import config.Configurador;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Test {
   public static void main(String[] args) {
      ApplicationContext context =
            new AnnotationConfigApplicationContext(Configurador.class);
      Operaciones op=(Operaciones)context.getBean("operaciones");
      op.escribirCadena();
      op.escribirCadena();
      op.mostrarCadenas();
   }
}
```

En esta nueva versión, ya no necesitamos ningún archivo de configuración XML.

5. PROGRAMACIÓN ORIENTADA A ASPECTOS

La programación orientada a aspectos es una técnica de programación proporcionada por Spring que consiste en definir determinadas operaciones en clases independientes, denominadas aspectos, de manera que puedan aplicarse o inyectarse en aplicaciones ya existentes, sin necesidad de modificar el código de estas.

Un ejemplo de aspecto puede ser el registro de log de una aplicación o el control de políticas de seguridad. Dicha funcionalidad puede definirse fuera de la lógica de las aplicaciones e inyectarse después en aquellas aplicaciones en las que se desee utilizar.

5.1. TERMINOLOGÍA

Antes de analizar la forma de poner en práctica la AOP en Spring, es necesario definir una serie de términos clave en este tipo de programación:

- **Aspecto**. Tal y como se ha indicado, un aspecto representa la funcionalidad transversal, que se define en clases independientes y que puede ser aplicada a diferentes aplicaciones.

- **Advice**. Un *advice* constituye la implementación actual de un aspecto, más concretamente, representa el módulo de código que define la funcionalidad del aspecto.

- **Target**. Se conoce como *target* al objeto de la aplicación donde será aplicado el aspecto.

- **JoinPoint**. Un aspecto puede ser aplicado en diferentes partes de un *target*, cada uno de estos posibles puntos de aplicación es conocido como un *joinpoint*. Se puede decir que los aspectos se aplican en las aplicaciones insertando el *advice* dentro de los *joinpoint*.

- **PointCut**. Los *joinpoint* son los posibles puntos donde se puede aplicar un aspecto pero a los puntos concretos donde se aplica el *advice* se les conoce como *pointcut*.

- **Proxy**. Tras aplicar un *advice* en un objeto *target*, se genera un nuevo objeto que incluye la funcionalidad de la aplicación cliente más la del aspecto. A este nuevo objeto se le conoce como proxy.

5.2. UTILIZACIÓN DE ANOTACIONES ASPECTJ

AspectJ es un framework para implementación de aplicaciones orientadas a aspectos, que desde la versión 2.5 viene integrado con Spring. La ventaja de la integración de AspectJ con Spring es que no se necesitan conocer los detalles del mismo para poderlo utilizar, tan solo se requiere definir determinados parámetros de configuración a través de una serie de anotaciones.

Para poder utilizar AspectJ en una aplicación Spring, tendremos que descargar las librerías apropiadas y añadirlas al proyecto netbeans. Concretamente, se trata de los archivos aspectjrt.jar, aspectjtools.jar, aspectjweaver.jar y org.aspectj.matcher.jar. Estos archivos podemos encontrarlos en la dirección

http://www.eclipse.org/aspectj/downloads.php

Si descargamos el archivo aspectj-1.7.2.jar (u otra versión más reciente) y lo descomprimimos, encontraremos en la carpeta \lib los cuatro archivos de librería mencionados.

Además de las librerías de AspectJ, siempre que se implementa una aplicación Spring basada en AOP **será necesario incluir también la librería aopalliance.jar**, librería que no forma parte de los archivos .jar que se incluyen con el entorno de desarrollo. Este archivo puede descargarse desde la dirección http://sourceforge.net/projects/aopalliance/.

5.2.1. Creación de una aplicación Spring basada en AspectJ

Una vez configurado adecuadamente el IDE, ya podemos crear aplicaciones Spring/AspectJ. Hay que tener en cuenta que la implementación del aspecto no condiciona la aplicación principal, es decir, implementamos el target o aplicación principal y, una vez que comprobamos que esta funciona correctamente pasamos a implementar la funcionalidad trasversal o aspecto que queremos aplicar, en una clase o aplicación independiente.

Seguidamente, vamos a describir los pasos que tendremos que seguir para crear una aplicación Spring AOP basada en la utilización de AspectJ. De cara a describir de forma práctica los pasos a realizar, utilizaremos como ejemplo la aplicación de autenticación de usuarios implementada en el ejercicio 1, de modo que vamos a implementar un sencillo aspecto consistente en enviar una mensaje a la consola cada vez que el método *login()* va a ser invocado y otro después de que haya sido ejecutado.

5.2.1.1. Implementación del advice

Lo primero que haremos será implementar el advice, es decir, la clase en la que codificaremos las instrucciones que definen la funcionalidad a realizar por el aspecto.

Una de las ventajas de utilizar AspectJ es que el advice es una simple clase POJO, no tiene que heredar ninguna clase especial ni implementar ninguna interfaz, tan solo tendrá que estar anotada con la anotación **@Aspect**:

```
package aspectos;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class Test {

:

}
```

Cada aspecto que queramos aplicar será definido en un método independiente dentro de la clase anterior, cada método puede ser llamado como nosotros queramos, aunque tendrá que tener el siguiente formato:

public void nombre_metodo(JoinPoint jp)

Como vemos, el método declara un objeto org.aspectj.lang.JoinPoint que será inyectado por el entorno de ejecución de Spring durante la llamada al método. A través de este objeto, el aspecto puede acceder a información sobre el joinpoint en donde se aplicará el código.

Entre los métodos de la clase JoinPoint destacamos:

- **getArgs()**. Devuelve un array de objetos que representan los argumentos de llamada al *joinpoint*.
- **getSignature()**. Devuelve un objeto Signature que representa el *joinpoint*. A través del método *getName()* de la interfaz Signature podría conocerse el nombre del método en el que será aplicado el aspecto.
- **getTarget()**. Devuelve el objeto target sobre el que el aspecto se ha aplicado.

Cada método deberá incluir una de las siguientes anotaciones, según donde se quiera aplicar el aspecto:

- **@Before**. El aspecto será aplicado antes de la llamada a un método del target, lo que antes hemos definido como pointcut. Este punto de aplicación del aspecto será definido mediante una expresión que se pasará como parámetro a la anotación y que examinaremos seguidamente.

- **@After.** El aspecto será aplicado después de la llamada al método, independientemente de que se haya producido o no una excepción durante su ejecución.

- **@AfterReturning**. El aspecto se aplica después de la llamada al método, siempre y cuando el método se haya ejecutado normalmente.

- **@Around**. El aspecto será aplicado durante la llamada al método. Mediante este aspecto se puede modificar el valor de retorno del método target.

Como hemos indicado al explicar la anotación @Before, se debe indicar como argumento de estas anotaciones una expresión que indique a Spring el método o métodos sobre los que se debe aplicar el aspecto. Dicha expresión se ajustará al siguiente formato:

```
execution(acceso tipo_retorno nombre(parametros))
```

El significado de cada parte es el siguiente:

- **acceso**. Modificador de acceso que deben tener los métodos pointcut. Para cualquier modificador de acceso se indicará "*".

- t**ipo_retorno**. Tipo de retorno de los métodos. Para cualquier tipo de retorno se indicará "*".

- **nombre**. Nombre cualificado del método. Al igual que en el caso anterior podría utilizarse el "*" como comodín. Por ejemplo, la expresión *.*.* representa cualquier método de cualquier clase incluida en cualquiera de los paquetes de la aplicación.

- **parametros**. Lista de parámetros que incluye el método sobre el que se aplicará el aspecto. Para indicar cualquier número y tipo de parámetros utilizaremos la expresión: (..).

El siguiente listado corresponde al advice completo del ejemplo que estamos tratando, donde los aspectos se aplicarían sobre todos los métodos de la clase GestionClientesImpl:

```
import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class Test {

   @Before("execution(* modelo.GestionClientesImpl.*(..))")
```

```
    public void logBefore(JoinPoint jp) throws Throwable {
        System.out.println("va a autenticar el usuario mediante: "+
                jp.getSignature().getName());
    }
    @After("execution(* modelo.GestionClientesImpl.*(..))")
    public void logAfter(JoinPoint jp) throws Throwable {
        System.out.println("ya se ha autenticado mediante: "+
                jp.getSignature().getName());
    }
}
```

5.2.1.2. Registro del aspecto y configuración de AspectJ

Una vez implementado el aspecto, se registrará en el archivo de configuración de Spring como un bean más.

Como resultado de aplicación del aspecto sobre el target, se genera un objeto proxy que incluye el código del aspecto incorporado al de la aplicación. Este proxy se genera automáticamente gracias a la inclusión del elemento **<aspectj-autoproxy>** del espacio de nombres aop en el archivo de configuración de Spring.

Por tanto, el archivo de configuración de Spring en esta nueva versión de la aplicación quedaría como se indica en el siguiente listado, donde se ha marcado de color de fondo diferente las nuevas instrucciones incorporadas:

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

    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
      http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
      http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">
    <!--datasource de spring-->
    <bean id="data"
```

```
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/libros"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
    <!--clase del modelo-->
    <bean id="gestion" class="modelo.GestionClientesImpl"/>
    <!--registro del aspecto-->
    <bean id="aspecto" class="aspecto.Test"/>
    <aop:aspectj-autoproxy/>
    <context:annotation-config />
</beans>
```

Si ejecutamos ahora de nuevo la clase de prueba de la aplicación, se mostrará en pantalla lo siguiente:

```
va a autenticar el usuario!!! login
ya se ha autenticado!!! login
registrado
```

6. ACCESO A DATOS: EL MÓDULO SPRING DAO

Como sabemos, una aplicación Java utiliza el API JDBC para acceder a bases de datos. Este API, aunque sencillo de manejar, resulta bastante engorroso por la gran cantidad de instrucciones a codificar a la hora de realizar cualquier operación.

El objetivo del módulo Spring DAO es proporcionar al programador un conjunto de clases que simplifiquen esta tarea, encapsulando todo el proceso de establecimiento de conexión, control de excepciones, etc. que de otra manera, habría que codificar manualmente.

Además de JDBC, los programadores Java cuentan con otro mecanismo para manipulación de datos en una aplicación: JPA A través de esta tecnología, podemos trabajar en una aplicación con objetos persistentes en vez de hacerlo directamente sobre la base de datos, simplificando el proceso de acceso a la información. Aunque JPA simplifica enormemente el código de tratamiento de los datos, también añade ciertas complejidades a la aplicación, complejidades que se pueden evitar utilizando Spring DAO.

6.1. LA CLASE JDBCTEMPLATE

La clase **JdbcTemplate**, localizada en org.springframework.jdbc.core, proporciona una serie de métodos para realizar operaciones sobre una base de datos a través de JDBC, **sin**

que el programador tenga que hacer uso explícito de este API, ya que todas las instrucciones se encuentran encapsuladas en los métodos de esta clase.

Para crear un objeto de esta clase, es necesario proporcionarle al constructor un objeto DataSource configurado para trabajar con la base de datos con la que se quiere operar. Este objeto DataSource puede ser configurado a través de Spring e inyectado en la clase que va a encapsular la lógica de negocio de la aplicación, tal y como se vio en el ejercicio del punto anterior. Supongamos que la clase de negocio tiene la siguiente estructura:

```
public class Modelo {

   DataSource ds;

   public Modelo(DataSource ds){

      this.ds=ds;

   }

   public void metodoNegocio(){

   //crea un objeto JdbcTemplate para poder

   //realizar las operaciones encomendadas

   JdbcTemplate tmp = new JdbcTemplate(ds);

   :

  }

}
```

Como vemos, cada método de negocio crea un objeto JdbcTemplate local para operar contra la base de datos, a partir del DataSource suministrado a la clase a través del constructor. Este objeto DataSource puede ser suministrado por el contenedor de Spring al Modelo utilizando inyección de dependencia:

```
<bean id="data" class="org.springframework.jdbc.datasource.

                              DriverManagerDataSource">

      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>

      <property name="url" value="jdbc:mysql://localhost:3306/example"/>

   <property name="username" value="root"/>

   <property name="password" value="root"/>

</bean>

<!-- el core Spring inyecta el datasource en un objeto Modelo-->

<bean id=”modelo” class=”logica.Modelo”>

      <constructor-arg ref=”data”/>

</bean>
```

De esta manera el uso de JdbcTemplate en la aplicación a través de los métodos del modelo quedaría:

```
BeanFactory factory=

    (BeanFactory)new ClassPathXmlApplicationContext("configurador.xml");

Modelo model=(Modelo)factory.getBean("modelo");

model.metodoNegocio();
```

6.1.1. Principales métodos de JdbcTemplate

Los métodos de JdbcTemplate, además de simplificar el proceso para operar con la base de datos, evitan al programador la preocupación por la captura de las excepciones SQLException ya que dichos métodos realizan un control por defecto de las mismas.

Entre los métodos proporcionados por esta clase destacamos:

update. Ejecuta una instrucción SQL de tipo Insert, Update o Delete. Tenemos dos versiones de este método, según vayamos a utilizar una instrucción SQL estática o una consulta preparada:

- **void update(String sql)**. Ejecuta la consulta SQL especificada como argumento.
- **void update(String sql, Object…args)**. En esta versión, el primer parámetro representa una cadena SQL parametrizada, mientras que el segundo representa la lista de valores de los parámetros de la consulta:

  ```
  String sql =”Insert into clientes values( ?,?,?);

  Objeto_jdbTemplate.update(sql, “Benito”,”Sanchez”, 5008978);
  ```

query. Ejecuta una instrucción SQL de selección sobre la base de datos. Al igual que anterior, tenemos dos versiones del método para ser ejecutado con instrucciones SQL estáticas y parametrizadas:

- **List<T> query (String sql, RowMapper<T> rowmapper)**. Ejecuta la instrucción de tipo select proporcionada como primer parámetro, devolviendo una colección de objetos con los datos recuperados en la consulta. El parámetro de tipo RowMapper, que comentaremos a continuación, define la manera en la que los registros obtenidos serán mapeados a cada objeto de la colección.
- **List<T> query (String sql, Object…args, RowMapper<T> rowmapper)**. En esta otra versión del método la cadena especificada como primer argumento representa una consulta parametrizada de tipo select, indicándose los valores de los parámetros de la consulta en el conjunto de argumentos args.

execute. Este método lo utilizaremos para ejecutar consultas SQL de tipo DDL. Aunque el método está sobrecargado y existen varias versiones del mismo, la más sencilla es la siguiente:

- **void execute (String sql)**

Donde el parámetro String representa la instrucción SQL que se quiere ejecutar sobre la base de datos. Mediante la siguiente instrucción crearíamos una tabla dentro de la base de datos, utilizando el objeto de tipo JdbcTemplate referenciado por la variable tmp:

```
tmp.execute(“create table data (code integer, text varchar(200))”);
```

6.2. LA INTERFAZ ROWMAPER

A través de esta interfaz, localizada en org.springframework.jdbc.core, podremos indicarle a Spring como tiene que devolvernos los datos a la hora de ejecutar una consulta de tipo Select. La idea es mapear los datos recuperados en campos de un objeto JavaBean definido en la aplicación, mapeo que se realizará dentro de la implementación del método *mapRow()* declarado en esta interfaz. El formato del método es el siguiente:

```
public T mapRow(ResultSet result, int fila) throws SQLException
```

Siendo *result* el objeto ResultSet que hace referencia a los datos de la consulta, y *fila* el número de fila actual que será mapeada. Por su parte, el tipo de devolución, definido como genérico, representa el tipo de objeto que contiene los datos resultantes del mapeo.

Así pues, cuando Spring necesita adaptar los datos resultantes de la ejecución de una instrucción SQL de tipo select, como en el caso del método *query()* de JdbcTemplate mencionado anteriormente, utiliza la implementación que le indiquemos de este método con cada fila de resultados para transformar dicha fila en un objeto de datos.

Supongamos, por ejemplo, que tenemos un JavaBean Persona con la siguiente estructura:

```
public class Persona {

  private String nombre;

  private String direccion;

  private int dni;

  public Persona(){}


  //constructor que permite crear un objeto

  //Persona a partir de sus atributos

  public Persona(String nombre, String direccion, int dni){

    this. nombre = nombre;

    this. direccion = direccion;

    this. dni = dni;

  }

  public void setNombre(String nombre){

    this. nombre = nombre;

  }

  public String getNombre(){

    return this.nombre;

  }
```

```
  public void setDireccion(String direccion){
   this. direccion = direccion;
  }
  public String getDireccion(){
    return this.direccion;
  }
  public void setDni(int dni){
   this. dni = dni;
  }
  public String getDni(){
    return this.dni;
  }
}
```

Si queremos mapear cada fila del ResultSet que apunta a una hipotética tabla de personas en un objeto de tipo Persona, la implementación de RowMapper podría ser algo como esto:

```
public class MapeoPersonas implements RowMapper{
   @Override
   public Persona mapRow(ResultSet rs, int fila) throws SQLException {
      //construye un objeto Persona con los campos
      //de la fila actual
      Persona p=new Persona(rs.getString("nombre"),
                  rs.getString("direccion"),
                  rs.getInt("isbn"));
      return p;
   }
}
```

En la clase anterior, no se ha hecho uso del parámetro *fila*, puesto que todas ellas se van a mapear utilizando el mismo criterio.

A partir de la clase anterior, si quisiéramos ejecutar una instrucción SQL que nos devuelva los datos de todas las personas de la tabla “personas”, utilizando JdbcTemplate, sería tan sencillo como ejecutar la siguiente instrucción:

String sql=”Select * From personas”;

List<Persona> todas = obj_template.query(sql, new MapeoPersonas());

EJERCICIO 3

En este ejercicio desarrollaremos una aplicación Web para el envío y visualización de mensajes entre usuarios de la aplicación.

La estructura de páginas de la aplicación se muestra en el siguiente dibujo:

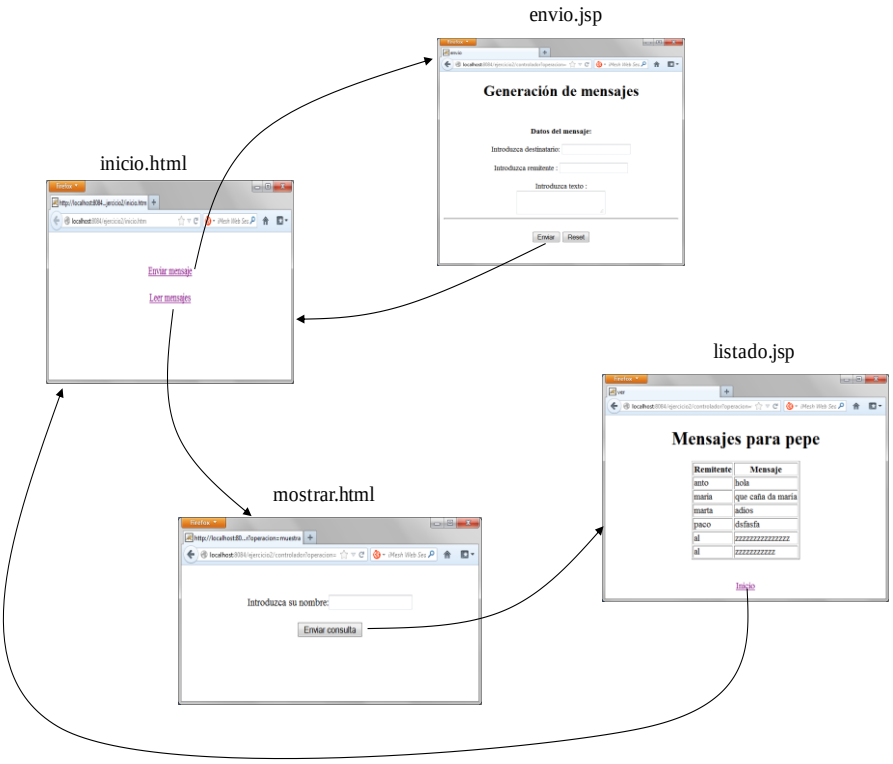

Figura. 6.

Habrá una página inicial con dos enlaces para “Enviar mensaje” y “Visualizar mensajes”. El enlace “Enviar mensaje” nos llevará a una página en la que se solicitarán los datos del mensaje: remitente, destinatario y texto. Tras el envío del mensaje, el programa retornará a la página de inicio.

El enlace “Visualizar mensajes” nos llevará a una página en la que se nos solicitará el nombre de la persona destinataria de mensajes, para llevarnos después a una página en la que se mostrarán los mensajes (remitente y texto), recibidos por dicha persona.

La lógica de negocio de la aplicación se implementará con Spring DAO y trabajaremos contra una base de datos en la que existirá una única tabla donde se guardarán los mensajes. Esta tabla tendrá cuatro campos: idMensaje(campo clave autonumérico), remitente, destinatario y texto.

La aplicación se creará como un proyecto de tipo *Web application* en Netbeans. En primer lugar, crearemos una clase JavaBean llamada Mensaje, que encapsulará los datos de cada mensaje:

```
package javabeans;

public class Mensaje {

  private String remite;

  private String destino;

  private String texto;

  public Mensaje(){}

  //constructor que permite crear un objeto

  //Mensaje a partir de los datos del mismo

  public Mensaje(String remite, String destino, String texto){

    this.remite=remite;

    this.destino=destino;

    this.texto=texto;

  }

  public void setRemite(String remite){

    this.remite=remite;

  }

  public String getRemite(){

    return this.remite;

  }

  public void setDestino(String destino){

    this.destino=destino;

  }

  public String getDestino(){

    return this.destino;

  }

  public void setTexto(String texto){

    this.texto=texto;

  }
```

```
  public String getTexto(){
    return this.texto;
  }
}
```

En cuanto al modelo o lógica de negocio, deberá exponer los métodos definidos en la siguiente interfaz:

```
package modelo;
import java.util.List;
import javabeans.Mensaje;
public interface GestionMensajes {
   void grabaMensaje(Mensaje m);
   List<Mensaje> obtenerMensajes(String destino);
}
```

El método *grabarMensaje()* añade a la base de datos un registro con los datos del mensaje recibido como parámetro, mientras que *obtenerMensajes()* devuelve una colección con los mensajes asociados al destinatario indicado.

Antes de presentar la clase de implementación de esta interfaz, crearemos la clase mapeadora que transforma cada fila de la tabla mensajes en un objeto Mensaje:

```
package modelo;
import java.sql.ResultSet;
import java.sql.SQLException;
import javabeans.Mensaje;
import org.springframework.jdbc.core.RowMapper;
public class Mapeador implements RowMapper{
   @Override
   //reglas de mapeo de ResultSet a JavaBean Mensaje
   public Mensaje mapRow(ResultSet rs, int i) throws SQLException {
      Mensaje m=new Mensaje(rs.getString("remitente"),
                            rs.getString("destinatario"),
                            rs.getString("texto"));
      return m;
   }
```

```
}
```

El siguiente listado corresponde a la clase de implementación del modelo, donde utilizaremos un objeto JdbcTemplate inyectado desde el archivo de configuración de Spring, para la realización de las operaciones de acceso a los datos:

```
package modelo;

import javabeans.*;

import java.util.*;

import org.springframework.jdbc.core.JdbcTemplate;

public class GestionMensajesImpl implements GestionMensajes {

   JdbcTemplate tmp;

   //recibe JdbcTemplate a través de inyección de

   //dependencia

   public GestionMensajesImpl(JdbcTemplate tmp){

      this.tmp=tmp;

   }

   public JdbcTemplate getTemplate(){

      return tmp;

   }

   public void setTemplate(JdbcTemplate tmp){

      this.tmp=tmp;

   }

   @Override

  public List<Mensaje> obtenerMensajes(String destino){

   String tsql="select * from mensajes where destinatario=?";

   List<Mensaje> mensajes=(List<Mensaje>)tmp.query(tsql,

new Object[]{destino}, new Mapeador());

   return mensajes;

  }

   @Override

  public void grabaMensaje(Mensaje m){

   String tsql="Insert into mensajes (remitente, destinatario,texto) values(?,?,?)";
```

```
            //Crea el array de parámetros a partir de
            //los campos del objeto Mensaje
            Object [] parametros=new Object[]{m.getRemite(),m.getDestino(),m.getTexto()};
            tmp.update(tsql, parametros);
          }
        }
```

A continuación presentamos el código del archivo de configuración de Spring:

```
        <?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-3.0.xsd">
            <bean id="datasource" class="
        org.springframework.jdbc.datasource.DriverManagerDataSource">
                        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
                        <property name="url" value="jdbc:mysql://localhost:3306/mensajes"/>
              <property name="username" value="root"/>
              <property name="password" value="root"/>
            </bean>
            <!--objeto JdbcTemplate-->
            <bean id="template" class="org.springframework.jdbc.core.JdbcTemplate">
                  <property name="dataSource" ref="datasource"/>
            </bean>
            <!--inyección del objeto template
               en la lógica de negocio-->
            <bean id="gestion" class="modelo.GestionMensajesImpl" >
               <constructor-arg ref="template"/>
            </bean>
        </beans>
```

En cuanto a la capa Web de la aplicación, tenemos, por un lado, el bloque controlador, formado por un servlet que controla el flujo de peticiones a la aplicación e implementa las dos acciones de la misma: inserción de mensajes y recuperación de mensajes:

```
package servlets;



import javax.servlet.*;



import javax.servlet.annotation.WebServlet;

import javax.servlet.http.*;

import java.io.*;

import java.util.*;

import javabeans.*;

import javax.sql.DataSource;

import modelo.*;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.context.support.ClassPathXmlApplicationContext;

@WebServlet(urlPatterns="/controlador")

public class Controlador extends HttpServlet {

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

   //recuperación del parámetro que determina la

   //operación a realizar

   String op=request.getParameter("operacion");

   //acceso a la pagina de envío de mensajes

   if(op.equals("envio"))

         request.getRequestDispatcher("/envio.jsp").forward(request, response);

   //grabación de un mensaje

   if(op.equals("grabar")){

     //recupera el objeto mensaje generado por la pagina

     Mensaje men=(Mensaje)request.getAttribute("mensa");

     BeanFactory factory= (BeanFactory)
```

```
                new ClassPathXmlApplicationContext("ejercicio3Config.xml");
      GestionMensajes oper=(GestionMensajes)factory.getBean("gestion");
      oper.grabaMensaje(men);
      request.getRequestDispatcher("/inicio.htm").forward(request, response);
    }
    //acceso a la página de solicitud de mensajes
    if(op.equals("muestra"))
      request.getRequestDispatcher("/mostrar.htm").forward(request, response);
    //acceso a la lista de mensajes del usuario
    if(op.equals("listado")){
      BeanFactory factory= (BeanFactory)
                        new ClassPathXmlApplicationContext("ejercicio3Config.xml");
      GestionMensajes oper=(GestionMensajes)factory.getBean("gestion");
      //recupera los mensajes asociados al destinatario
      //indicado y después los deposita en un atributo de
      //petición para, finalmente, invocar a la vista
      //encargada de su presentación
      List<Mensaje> mensajes=oper.obtenerMensajes(request.getParameter("nombre"));
      request.setAttribute("mensajes",mensajes);
      RequestDispatcher rd=request.getRequestDispatcher("/listado.jsp");
      rd.forward(request,response);
    }
  }
}
```

Por último, tenemos las páginas que conforman la vista de la aplicación:

inicio.html

```
<html>
<body>
<center>
   <br/><br/>
```

```
    <a href="controlador?operacion=envio">
        Enviar mensaje
    </a><br/><br/>
    <a href="controlador?operacion=muestra">
        Leer mensajes
    </a>
</center>
</body>
</html>
```

envio.jsp

```
<html>
<head>
<title>envio</title>
</head>
<!--captura de datos e inserción en el Javabean-->
<jsp:useBean id="mensa" scope="request" class="javabeans.Mensaje" />
<jsp:setProperty name="mensa" property="*"/>
<%if(request.getParameter("texto")!=null){%>
  <jsp:forward page="controlador?operacion=grabar"/>
<%}%>

<body>
<center>
  <h1>Generación de mensajes</h1>
  <form method="post">
  <br/><br/>
  <b>Datos del mensaje:</b><br/><br/>
  Introduzca destinatario:  <input type="text" name="destino"><br/>
  <br/>
  Introduzca remitente :  <input type="text" name="remite"><br/>
```

```
  <br/>
  Introduzca texto : <br/>
  <textarea name="texto">
  </textarea>
  <hr/><br/>
  <input type="submit" name="Submit" value="Enviar">
  <input type="reset" value="Reset">
  </form>
</center>
```

mostrar.html

```
<html>
<body>
<center>
   <br/><br/>
   <form action="controlador?operacion=listado" method="post">
   Introduzca su nombre:<input type="text" name="nombre"><br><br>
   <input type="submit">
   </form>
</center>
</body>
</html>
```

listado.jsp

```
<%@ page import="javabeans.*,java.util.*"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>ver</title>
</head>
<body>
<center>
```

```
<h1>
Mensajes para ${param.nombre}
</h1>
<table border=1>
<tr><th>Remitente</th><th>Mensaje</th></tr>
<c:if test="${empty requestScope.mensajes}">
   <jsp:forward page="/nomensajes.jsp"/>
</c:if>
<c:forEach var="m" items="${requestScope.mensajes}">
     <tr><td>${m.remite}</td><td>${m.texto}</td></tr>
</c:forEach>

</table>
<br/><br/>
<a href="inicio.htm">Inicio</a>
</center>
</body>
```

nomensajes.jsp

```
<html>
<head>
<title>nomensajes</title>
</head>
<body>
  <center>
    <h2>
     Lo siento, ${param.nombre} no tiene mensajes
    </h2>
    <br/><br/><br/><br/>
    <a href="inicio.htm">Inicio</a>
```

```
  </center>

</body>

</html>
```

6.3. INTEGRACIÓN DE SPRING CON JPA

Como sabemos, JPA es un framework de persistencia incluido en la especificación Java EE, cuya principal característica es que permite trabajar con cualquier motor de persistencia, como Hibernate, TopLink, etc.

Spring proporciona un módulo JPA que simplifica el trabajo con este framework de persistencia desde cualquier aplicación Java. Gracias a este módulo, tareas como la creación de un objeto EntityManager o la gestión de transacciones se reducen enormemente, centralizándose además todo el proceso en el archivo de configuración del Spring.

6.3.1. Creación y configuración del objeto EntityManagerFactory.

Una aplicación JPA incluye un archivo de configuración persistence.xml en el que se definen los datos de configuración del proveedor o gestor de persistencia. Cuando se trabaja con Spring, todas estas operaciones pueden centralizarse en el archivo de configuración del framework, por lo que persistence.xml se limitará simplemente a registrar las entidades utilizadas por la aplicación.

Además de esto, Spring nos proporciona una implementación de EntityManagerFactory, llamada LocalContainerEntityManagerFactoryBean, que permite crear y configurar este tipo de objetos en el propio archivo de configuración de Spring.

Para configurar un LocalContainerEntityManagerFactoryBean, simplemente tendremos que indicarle el DataSource que contiene los datos de conexión a la base de datos, y el proveedor de persistencia que se desea utilizar.

El siguiente listado corresponde a la configuración de un EntityManager que accede a la base de datos “mensajes” a través de una capa de persistencia, utilizando el proveedor JPA de Hibernate que se incluye en el módulo JPA de Spring:

```
<!—configuración del DataSource-->

<bean id="datasource"

        class="org.springframework.jdbc.datasource.DriverManagerDataSource">

                <property name="driverClassName" value="com.mysql.jdbc.Driver"/>

                <property name="url" value="jdbc:mysql://localhost:3306/mensajes"/>

                <property name="username" value="root"/>

                <property name="password" value="root"/>

  </bean>

  <!—configuración del EntityManagerFactory  -->

        <bean id="entityManagerFactory"
```

```
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="datasource"/>
        <!-- esta propiedad contiene el motor utilizado,
            en este caso es hibernate -->
        <property name="jpaVendorAdapter">
            <bean  class=
        "org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="databasePlatform"
                    value="org.hibernate.dialect.MySQLDialect"/>

            </bean>
        </property>
    </bean>
```

6.3.2. Transaccionalidad

En JPA todas las operaciones que realicen algún tipo de modificación sobre la capa de persistencia deben estar englobadas en una transacción.

El elemento clave para la gestión de transacciones en Spring es la interfaz **PlatformTransactionManager**, pues en ella se definen los métodos para obtener, confirmar y rechazar una transacción. Spring proporciona diferentes implementaciones de esta clase, para JPA podemos utilizar **JpaTransactionManager**.

La creación de un objeto JpaTransactionManager podemos realizarla desde el archivo de configuración de Spring. La única propiedad necesaria para configurar el objeto es EntityManagerFactory, a la que habrá que proporcionar el objeto EntityManagerFactory sobre el que se aplicará la transaccionalidad:

```
<bean id="transactionManager"
     class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

La manera más sencilla de utilizar el objeto transactionManager en el modelo es a través de la anotación **@Transactional**. Todos los métodos definidos con esta anotación serán transaccionales, por lo que no será necesario abrir y confirmar explícitamente la transacción desde código.

Seguidamente profundizaremos en el uso de esta anotación. De cara a completar las operaciones de configuración relativas a la transaccionalidad, será necesario incluir el elemento annotation-driven que habilita el uso de la anotación @Transactional:

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

6.3.2.1. Propiedades de @Transactional

Como hemos indicado, la utilización de @Transaction permite inyectar en los métodos de la capa del modelo las instrucciones necesarias para abrir y confirmar una transacción mediante JpaTransactionManager. Esta anotación soporta una serie de propiedades que permiten configurar su funcionamiento, entre ellas destacamos dos:

- **Propagation**. Establece la manera en la que debe ser propagada la transacción cuando el método es ejecutado. Entre los posibles valores que puede tomar esta propiedad tenemos:
  - PROPAGATION_REQUIRED. El método se debe ejecutar siempre en el contexto de una transacción. Si hay una transacción activa, el objeto participa en ella, pero si no hay ninguna transacción se iniciará una nueva. Es el valor predeterminado
  - PROPAGATION_REQUIRES_NEW. Cada vez que se hace una llamada al método se iniciará una nueva transacción. Si existiera una transacción en curso, sería suspendida hasta completar la ejecución del método y la nueva transacción sea confirmada o rechazada.
  - PROPAGATION_SUPPORTS. El método participa en la transacción activa. Si no existiera transacción activa no se iniciará ninguna.
  - PROPAGATION_NOT_SUPPORTED. El método nunca se ejecutará dentro de una transacción. Si ya existiera una activa al iniciar su ejecución, será suspendida hasta su finalización.

- **readOnly**. Mediante esta propiedad de tipo boolean se puede marcar la transacción como solo lectura, lo que resulta práctico cuando las operaciones a realizar impliquen recuperación de datos. Esto permite optimizar el rendimiento con algunos tipos de proveedores. Su valor por defecto es *false*.

6.3.3. Inyección del objeto EntityManager

Una vez configurado el objeto EntityManagerFactory, tan solo tenemos que utilizar la anotación **@PersistenceContext** para inyectar un objeto EntityManager en una variable atributo definida con ese tipo. Al encontrar esa anotación sobre una variable EntityManager, Spring buscará en el archivo de configuración el EntityManagerFactory, creará una instancia de EntityManager con las propiedades de persistencia definidas y la inyectará en la variable.

El siguiente listado corresponde a la capa modelo de la aplicación de mensajeria implementada mediante JPA, en donde suponemos que “Mensaje” es una entidad asociada a la tabla de mensajes:

```
package modelo;

import java.util.List;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import javax.persistence.Query;
```

```
import org.springframework.transaction.annotation.Transactional;
import entidades.Mensaje;

public class GestionMensajesImpl implements GestionMensajes {
    @PersistenceContext
    EntityManager em;
    @Override
    public List<Mensaje> obtenerMensajes (String destinatario) {
        String jpql=”select m from Mensaje m where “;
        jpql+=”m.destinatario=:dest”
        Query q=em.createNamedQuery(jpql);
        q.setParameter("dest", destinatario);
        return q.getResultList();
    }
    @Transactional
    @Override
    public void grabaMensaje(Mensaje m) {
        em.persist(m);
    }
}
```

El archivo de configuración de Spring al completo se muestra en el siguiente listado:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
```

```
                    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">


        <bean id="datasource"

            class="org.springframework.jdbc.datasource.DriverManagerDataSource">

            <property name="driverClassName" value="com.mysql.jdbc.Driver"/>

            <property name="url" value="jdbc:mysql://localhost:3306/mensajes"/>

            <property name="username" value="root"/>

             <property name="password" value="root"/>

    </bean>

    <bean id="entityManagerFactory" class="

        org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

            <property name="dataSource" ref="datasource"/>

            <property name="jpaVendorAdapter">

                <bean class="

            org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

                    <property name="databasePlatform"

                    value="org.hibernate.dialect.MySQLDialect"/>

                </bean>

            </property>

        </bean>

        <bean id="transactionManager"

            class="org.springframework.orm.jpa.JpaTransactionManager">

            <property name="entityManagerFactory" ref="entityManagerFactory"/>

        </bean>

        <!-- habilita la transaccionalidad a través de @Transactional -->

        <tx:annotation-driven/>

        <!—permite la utilización de @PersistenceContext. -->

    <context:annotation-config />

</beans>
```

Ejercicio 4

Dada una base de datos “cajero”, en la que tenemos una tabla “cuentas” que almacena los siguientes datos de una cuenta bancaria: codigo, cliente y saldo, realizar una aplicación que permita dar de alta cuentas, ingresar y extraer dinero de una cuenta y mostrar los datos de todas las cuentas existentes. Esta aplicación será implementada con Spring y JPA.

Para ello crearemos un proyecto de tipo aplicación Java estándar, crearemos el archivo de configuración de Spring y añadiremos las librerías del framework al proyecto. Además de las librerías de Spring, debemos incluir también la librería de hibernate, el driver JDBC de MySQL y el archivo aopalliance.jar

Una vez creado el proyecto, añadiremos la entidad “Cuenta” (podemos utilizar el asistente de netbeans para JPA), cuyo listado se muestra a continuación:

```
package entidades;


import java.io.Serializable;
import javax.persistence.*;
@Entity
@Table(name="cuentas")
public class Cuenta implements Serializable {
	private static final long serialVersionUID = 1L;
	@Id
	private int codigo;
	private String cliente;
	private double saldo;
	public Cuenta() {
	}
	public Cuenta(int codigo, String cliente, double saldo) {
		super();
		this.codigo = codigo;
		this.cliente = cliente;
		this.saldo = saldo;
	}
	public int getCodigo() {
		return this.codigo;
```

```
		}
		public void setCodigo(int codigo) {
			this.codigo = codigo;
		}
		public String getCliente() {
			return this.cliente;
		}
		public void setCliente(String cliente) {
			this.cliente = cliente;
		}
		public double getSaldo() {
			return this.saldo;
		}
		public void setSaldo(double saldo) {
			this.saldo = saldo;
		}
	}
```

En cuanto al modelo, estará formado por una interfaz Cajero en la que se definirán los métodos de negocio:

```
	package service;
	import java.util.List;
	import entidades.Cuenta;
	public interface Cajero {
	   void nuevaCuenta(Cuenta c);
	   void ingresar(int codigo, double cantidad);
	   void extraer(int codigo, double cantidad);
	   double obtenerSaldo(int codigo);
	   List<Cuenta> recuperarCuentas();
	}
```

Y la correspondiente clase de implementación:

```
package service;
import java.util.List;
import entidades.Cuenta;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import org.springframework.transaction.annotation.Transactional;
public class CajeroImpl implements Cajero {
        @PersistenceContext
        private EntityManager em;
        @Override
    @Transactional
        public void nuevaCuenta(Cuenta c) {
                em.persist(c);
        }
        @Override
    @Transactional
        public void ingresar(int codigo, double cantidad) {
        Cuenta c=em.find(Cuenta.class, codigo);
        if(c!=null){
            c.setSaldo(c.getSaldo()+cantidad);
            em.merge(c);
        }
        }
        @Override
    @Transactional
        public void extraer(int codigo, double cantidad) {
        Cuenta c=em.find(Cuenta.class, codigo);
        if(c!=null){
           c.setSaldo(c.getSaldo()-cantidad);
```

```
            em.merge(c);
        }
        }
        @Override
    @Transactional(readOnly=true)
        public double obtenerSaldo(int codigo) {
        Cuenta c=em.find(Cuenta.class, codigo);
        double saldo=0.0;
        if(c!=null){
            saldo=c.getSaldo();
        }
        return saldo;
        }
        @Override
        public List<Cuenta> recuperarCuentas() {
        Query q=em.createQuery("select c from Cuenta c");
        return q.getResultList();
        }
}
```

El archivo persistence.xml lo dejaremos vacío, ya que toda la configuración de persistencia se centralizará en el archivo de configuración de Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
</persistence>
```

En cuanto al archivo de configuración de Spring, quedará como se indica a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

                xmlns:context="http://www.springframework.org/schema/context"

                xmlns:tx="http://www.springframework.org/schema/tx"

                xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd

                        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.1.xsd

                        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.1.xsd">



                <bean id="datasource"

                    class="org.springframework.jdbc.datasource.DriverManagerDataSource">

                <property name="driverClassName" value="com.mysql.jdbc.Driver"/>

                <property name="url" value="jdbc:mysql://localhost:3306/banco"/>

                <property name="username" value="root"/>

                <property name="password" value="root"/>

            </bean>

                <bean id="emFactory" class=

                    "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

                        <property name="dataSource" ref="datasource"/>

                        <property name="jpaVendorAdapter">

                                <bean class=

                "org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

                                        <property name="databasePlatform"

                                            value="org.hibernate.dialect.MySQLDialect"/>

                                </bean>

                        </property>


                </bean>

                <bean id="jpaTemplate" class="org.springframework.orm.jpa.JpaTemplate">

                        <property name="entityManagerFactory" ref="emFactory"/>
```

```
        </bean>
        <bean id="transactionManager" class=
                "org.springframework.orm.jpa.JpaTransactionManager">
                <property name="entityManagerFactory" ref="emFactory"/>
        </bean>
        <bean id="cajero" class="service.CajeroImpl"/>
        <context:annotation-config/>
        <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

Por último, en el siguiente listado se muestra una apliación de ejemplo basada en consola, que realiza las operaciones sobre el cajero:

```
package ejercicio4;
import entidades.Cuenta;
import java.util.List;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.Cajero;
public class Test {
   public static void main(String[] args) {
      BeanFactory factory=
         (BeanFactory)new ClassPathXmlApplicationContext("ejercicio4Config.xml");
      Cajero cajero=(Cajero)factory.getBean("cajero");
      cajero.nuevaCuenta(new Cuenta(888,"test profe",400));
      cajero.ingresar(888, 100);
      cajero.extraer(888, 50);
      List<Cuenta> cuentas=cajero.recuperarCuentas();
      for(Cuenta c: cuentas){
         System.out.println(c.getCliente());
      }
   }
```

```
}
```

7. INTEGRACIÓN DE SPRING CON JSF

Como hemos estado viendo en el capítulo anterior, Spring dispone de los módulos DAO y ORM que nos simplifican la implementación del modelo en una aplicación Web basada en MVC.

Tal y como vimos en el ejercicio resuelto 3, en la capa controlador de la aplicación tenemos que recurrir a la interfaz BeanFactory y a la clase ClassPathXmlApplicationContext para cargar el contenedor de Spring y pedirle a éste los beans que implementan el modelo de cara a poder llamar a los métodos de negocio en esta capa controlador. Pues bien, si utilizamos JSF como framework MVC es posible ahorarnos estas instrucciones, dado que Spring proporciona un par de componentes que nos permiten, por un lado, cargar automáticamente el contenedor de Spring y, por otro, inyectar los beans del modelo en propiedades de los managed beans de JSF.

7.1. CARGA DEL CONTENEDOR DE SPRING

La clase **ContextLoaderListener** del paquete org.springframework.web.context está implementada como un escuchador del evento de inicio de aplicación Web y tiene como función realizar la carga del contenedor de Spring.

Por tanto, para utilizarla tan solo tendremos que registrarla como escuchador en el archivo de configuración web.xml de nuestra aplicación. A través del parámetro *contextConfigLocation* se le deberá indicar la ruta del archivo de configuración de Spring, ruta que se toma respecto al directorio raíz de la aplicación. Por ejemplo, si nuestro archivo de configuración de Spring se llama beanConfig.xml y se encuentra en el paquete por defecto, el registro de ContextLoaderListener sería:

```
<!--parámetro de contexto -->

<context-param>

    <param-name>contextConfigLocation</param-name>

    <param-value>/WEB-INF/classes/beanConfig.xml</param-value>

  </context-param>

  <!--registro del escuchador -->

  <listener>

    <listener-class>

            org.springframework.web.context.ContextLoaderListener

    </listener-class>

  </listener>
```

7.2. INYECCIÓN DE BEANS EN PROPIEDADES

Los managed beans de JSP disponen de propiedades (atributos con su pareja de métodos set/get) en las que se almacenan datos para ser utilizados por los controladores de acción. Si queremos que spring inyecte automáticamente objetos del modelo en algunos de estas propiedades, los atributos donde se almacenarán los objetos tendrán que estar anotados con @ManagedProperty("{#idbean}"), donde idbean representa el identificador del bean de struts que se quiere inyectar.

Si tenemos un managed bean Usuario con una propiedad gestionUsuarios que representa un bean de Spring que encapsula la lógica de negocio de la aplicación, para inyectar dicho bean en la propiedad deberíamos declarar el atributo:

```
public class Usuario{

        @ManagedProperty("#{gUsuarios}")

        GestionUsuarios gestion;

        :



}
```

Siendo gUsuarios el identificador asignado al bean en el archivo de configuración de Spring:

```
<bean id="gUsuarios" class="modelo.GestionUsuarios">

</bean>
```

La inyección del bean en la propiedad es realizada durante la creación del managed bean por el motor de JSF, pero este motor se apoya para realizar esta tarea en un objeto Spring de la clase SpringBeanFacesELResolver. Este objeto debe ser declarado en el archivo de configuración de JSF a través de la siguiente entrada:

```
<application>

        <el-resolver>

                org.springframework.web.jsf.el.SpringBeanFacesELResolver

        </el-resolver>

  </application>
```

Ejercicio 5

En este ejercicio vamos a desarrollar de nuevo la aplicación de mensajería elaborada en el ejercicio resuelto número 3, solo que en este caso utilizaremos JSF para implementar el controlador y la vista.

El modelo será exactamente el mismo que el de la otra versión, su implementación con Spring es idéntica al anterior, no se verá afectada en absoluto por el hecho de utilizar uno u otro framework MVC.

En cuanto al controlador, será implementado por dos managed beans, uno que gestionará la inserción del mensaje en la base de datos y otro la recuperación de los mensajes:

MensajeBean

```
package managed;

import javabeans.Mensaje;

import javax.faces.bean.*;

import modelo.*;

@ManagedBean(name="mensajeBean")

@RequestScoped

public class MensajeBean {

   private Mensaje mensaje;

   //inyección del bean gestión declarado en ejercicio5Config.xml

   @ManagedProperty("#{gestion}")

   private GestionMensajes gestion;

   public MensajeBean(){

      //importante esto!!! se suele olvidar

      mensaje=new Mensaje();

   }

   public Mensaje getMensaje() {

      return mensaje;

   }

   public void setMensaje(Mensaje mensaje) {

      this.mensaje = mensaje;

   }

   public GestionMensajes getGestion() {

      return gestion;

   }

   public void setGestion(GestionMensajes gestion) {

      this.gestion = gestion;

   }
```

```
    public String doGrabar(){
      gestion.grabaMensaje(mensaje);
      return "inicio";
    }
}
```

RecuperacionBean

```
package managed;
import javax.faces.bean.*;
import javabeans.Mensaje;
import java.util.List;
import modelo.*;

@ManagedBean(name="recuperacionBean")
@RequestScoped
public class RecuperacionBean {
    private String destinatario;
    private List<Mensaje> mensajes;
      //inyección del bean gestión declarado en ejercicio5Config.xml
    @ManagedProperty("#{gestion}")
    private GestionMensajes gestion;
    public String getDestinatario() {
        return destinatario;
    }
    public void setDestinatario(String destinatario) {
        this.destinatario = destinatario;
    }
    public List<Mensaje> getMensajes() {
        return mensajes;
    }
    public GestionMensajes getGestion() {
```

```
        return gestion;
    }
    public void setGestion(GestionMensajes gestion) {
        this.gestion = gestion;
    }
    public String doListado(){
      mensajes=gestion.obtenerMensajes(destinatario);
      if(mensajes==null||mensajes.size()==0){
          return "nomensajes";
      }else{
          return "listado";
      }
    }
}
```

Como pudes ver, hemos utilizado @ManagedProperty para inyectar el bean de Spring en la propiedad “gestion” de ambos beans.

En cuanto a los archivos de configuración de la aplicación Web, aquí tienes los dos:

web.xml

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Controlador</servlet-name>
        <servlet-class>servlets.Controlador</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
```

```
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Controlador</servlet-name>
    <url-pattern>/controlador</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>faces/inicio.jsp
    </welcome-file>
</welcome-file-list>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/classes/ejercicio4Config.xml</param-value>
</context-param>
<!-- carga el contenedor de Spring -->
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

```
</web-app>
```

faces-config.xml

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

<faces-config version="2.2"

    xmlns="http://xmlns.jcp.org/xml/ns/javaee"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">

  <application>

      <el-resolver>

        org.springframework.web.jsf.el.SpringBeanFacesELResolver

      </el-resolver>

    </application>

</faces-config>
```

Por último, te presentamos el código de las páginas de la aplicación:

inicio.jsp

```
<%@page contentType="text/html" pageEncoding="windows-1252"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>

<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

  "http://www.w3.org/TR/html4/loose.dtd">

<f:view>

  <html>

    <head>

      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>

      <title>JSP Page</title>

    </head>

    <body>

      <h:form>

      <h1><h:commandLink action="envio">Enviar mensaje</h:commandLink></h1>
```

```
            <br/><br/>
            <h1><h:commandLink action="mostrar">
                    Mostrar mensajes</h:commandLink></h1>
            </h:form>
        </body>
    </html>
</f:view>
```

envio.jsp

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<f:view>
    <html>
        <head>
            <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
            <title>JSP Page</title>
        </head>
        <body>
            <h:form>
                <h2>Datos del mensaje:</h2><br/><br/>
                Introduzca destinatario:
                 <h:inputText value="#{mensajeBean.mensaje.destino}"/><br/> <br/>
                Introduzca remitente :
                  <h:inputText value="#{mensajeBean.mensaje.remite}"/><br/><br/>
                Introduzca texto : <br/>
                  <h:inputTextarea  value="#{mensajeBean.mensaje.texto}"/><br/><br/>
                <hr/><br/>
                <h:commandButton action="#{mensajeBean.doGrabar}" value="Enviar"/>
            </h:form>
```

```
        </body>
    </html>
</f:view>
```

mostrar.jsp

```
<%@page contentType="text/html" pageEncoding="windows-1252"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
    <html>
        <head>
            <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
            <title>JSP Page</title>
        </head>
        <body>
            <h:form>
                Introduzca destinatario:
                  <h:inputText value="#{recuperacionBean.destinatario}"/><br/>
                <h:commandButton action="#{recuperacionBean.doListado}"
                                 value="Mostrar"/>
            </h:form>
        </body>
    </html>
</f:view>
```

listado.jsp

```
<%@page contentType="text/html" pageEncoding="windows-1252"%>
```

```
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<f:view>
   <html>
      <head>
         <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
         <title>JSP Page</title>
      </head>
      <body>
         <h:dataTable border="1"
               value="#{recuperacionBean.mensajes}" var="mensaje">
            <h:column>
               <f:facet name="header"><h:outputText value="Remitente"/></f:facet>
               <h:outputText value="#{mensaje.remite}"/>
            </h:column>
            <h:column>
               <f:facet name="header"><h:outputText value="Mensaje"/></f:facet>
               <h:outputText value="#{mensaje.texto}"/>
            </h:column>
         </h:dataTable>
      </body>
   </html>
</f:view>
```

nomensajes.jsp

```
<%@page contentType="text/html" pageEncoding="windows-1252"%>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```
    "http://www.w3.org/TR/html4/loose.dtd">
<f:view>
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
      <title>JSP Page</title>
    </head>
    <body>
      <h1>Lo siento, <h:outputText value="#{recuperacionBean.destinatario}"/>
          no tiene mensajes</h1>
      <br/>
      <br/>
      <h:commandLink action="inicio">Inicio</h:commandLink>
    </body>
  </html>
</f:view>
```

8. SPRING MVC

Desde el punto de vista del desarrollo Web, Spring MVC es el módulo más interesante que nos ofrece este framework pues, su utilización en este tipo de aplicaciones simplificará la aplicación del patrón MVC, ofreciéndonos numerosas posibilidades en este contexto.

8.1. EL PATRÓN MVC.

Cuando hablamos de patrón MVC nos estamos refiriendo a la aplicación de una arquitectura de desarrollo consistente en la división de cualquier desarrollo en tres grandes capas: el Modelo, la Vista y el Controlador.

En la figura 7 podemos ver cómo se organiza la capa intermedia de una aplicación Web Java EE siguiendo este esquema de bloques. A continuación analizaremos detalladamente la funcionalidad y características de cada uno de estos bloques

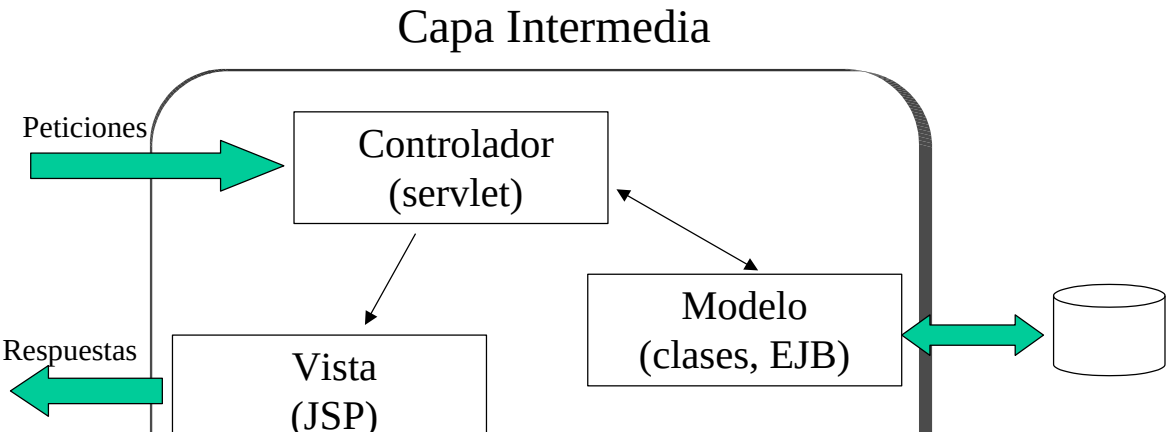

Figura. 7.

- **Controlador**. Se puede decir que el Controlador es el "cerebro" de la aplicación. Constituye el punto de entrada a la aplicación, puesto que todas las peticiones que llegan desde el cliente a la capa intermedia son dirigidas a él cuya misión es determinar las acciones a realizar para cada una de estas peticiones e invocar al resto de los componentes de la aplicación (Modelo y Vista) para que realicen las acciones requeridas en cada caso, encargándose también de la coordinación de todo el proceso.

  Por ejemplo, en el caso de que una petición requiera enviar como respuesta al cliente determinada información existente en una base de datos, el Controlador solicitará los datos necesarios al modelo y, una vez recibidos, se los proporcionará a la Vista para que ésta les aplique el formato de presentación correspondiente y envíe la respuesta al cliente. La función de controlador suele realizada por un servlet central o front controller que recibe todas las peticiones del cliente, apoyado en un grupo de servlets manejadores de acción a los que dirige cada una de las peticiones que le llegan para ser procesadas.

- **Vista**. La Vista es la parte de la aplicación encargada de generar las respuestas (HTML o cualquier otro formato) que deben ser enviadas al cliente. El tipo de componente empleado para la generación de las vistas dependerá de si se trata de una vista estática, en cuyo caso puede ser implementada en un archivo .html, o de una vista dinámica cuyo contenido debe ser generado a partir de la información proporcionada, en cuyo caso su implementación puede ser realizada mediante un servlet o página JSP

  Como se ha indicado, las vistas por regla general no acceden directamente al Modelo; toda la información necesaria para la generación de la respuesta es obtenida de los distintos atributos existentes en ámbitos de aplicación (petición, sesión y aplicación). Los datos obtenidos del Modelo y que deben ser presentados al usuario son depositados por el Controlador en estos atributos, antes de realizar la llamada a la página JSP encargada de presentarlos.

  Para facilitar el intercambio de datos entre Controlador y Modelo y, posteriormente, entre Controlador y Vista, las aplicaciones MVC suelen hacer uso JavaBeans, encapsulando en ellos las entidades de datos manejadas.

- **Modelo**. En la arquitectura MVC la lógica de negocio de la aplicación, incluyendo el acceso a los datos y su manipulación, está encapsulada dentro del modelo. El Modelo lo forman una serie de componentes de negocio independientes del Controlador y la Vista, permitiendo así su reutilización y el desacoplamiento entre las capas.

En una aplicación Java EE el modelo puede ser implementado mediante **clases estándar Java** o a través de **Enterprise JavaBeans**.

Spring MVC aporta elementos que simplificarán la implementación de las capas Controlador y Vista. Pero, aunque no aporta ningún componente específico para el Modelo, las tecnologías utilizadas para la implementación de esta capa pueden hacer uso del Core de Spring y el módulo DAO para acceso a los datos.

8.2. IMPLEMENTACIÓN DEL CONTROLADOR CON SPRING

Un controlador Spring sigue la misma estructura que se indica en la figura 7, donde la función de front-controller es realizada por un servlet de la clase DispatcherServlet.

8.2.1. El servlet controlador DispatcherServlet

DispatcherServlet es una subclase de HttpServlet que forma parte del API de Spring y se encuentra definida en el paquete org.springframework.web.servlet.

Para poder utilizar este servlet, lo único que debemos hacer es registrarlo en el archivo de configuración web.xml de nuestra aplicación. A fin de que todas las peticiones procedentes de la capa cliente sean capturadas por este servlet, se le mapea a un grupo de direcciones que se ajusten a un determinado formato, habitualmente, aquellas que finalicen con la extensión .htm:

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Lo anterior significa que toda petición procedente de la capa cliente que termine con la extensión .htm, será capturada por DispatcherServlet. A partir de ahí, DispatcherServlet se apoyará en una serie de componentes Spring, registrados en el archivo de configuración del framework, para realizar tareas como el mapeo de peticiones a los controladores de acción o la carga de la vista encargada de generar la respuesta al usuario. Utilizará la parte de la dirección que figura delante del punto para determinar el tipo de procesamiento a realizar, en función de algún sistema de mapeo.

8.2.2. El archivo de configuración de Spring

Además del web.xml, las aplicaciones Web creadas con Spring requieren de al menos un archivo de configuración adicional para el registro de los controladores de acción y otros beans utilizados por la aplicación que deben ser gestionados por el Core de Spring.

El escuchador **ContextLoaderListener** se encarga de iniciar el contenedor de Spring, detectando automáticamente el archivo de configuración siempre que se siga un determinado convenio de nombres. En este sentido, ContexLoaderListener es capaz de localizar automáticamente el siguiente fichero de configuración situado en WEB-INF:

- nombre_servlet-servlet.xml. Dado que el nombre del servlet despachador es "dispatcher", el nombre del segundo archivo deberá ser *dispatcher-servlet.xml*. El registro de los beans utilizados por la aplicación puede realizar en cualquiera de estos dos archivos.

Para poder utilizar ContextLoaderListener, simplemente tendremos que registrarlo en el archivo de configuración web.xml:

```
<listener>

    <listener-class>

        org.springframework.web.context.ContextLoaderListener

    </listener-class>

</listener>
```

Se pueden definir archivos de configuración adicionales para registrar, por ejemplo, los beans de la capa del modelo, datasources, etc. Estos archivos pueden estar en cualquier directorio, habitualmente, el WEB-INF, y para que sean cargados por ContextLoaderListener deberá especificarse su dirección en un parámetro contextConfigLocation que deberá ser proporcionado a ContextLoaderListener.

Por ejemplo, si además de dispatcher-servlet.xml queremos incluir un archivo adicional applicationContext.xml, el registro de ContextLoaderListener quedaría:

```
<context-param>

    <param-name>contextConfigLocation</param-name>

    <param-value>/WEB-INF/applicationContext.xml</param-value>

  </context-param>

  <listener>

    <listener-class>

        org.springframework.web.context.ContextLoaderListener

    </listener-class>

  </listener>
```

8.2.3. Controladores de acción: la clase AbstractController

Cada petición recibida por DispatcherServlet debe ser dirigida a un controlador de acción encargado de su procesamiento. Para realizar esta tarea DispatcherServlet necesita hacer uso de algún mecanismo de mapeo. La clase **ControllerClassNameHandlerMapping**, localizada en el paquete org.springframework.web.servlet.mvc.support, proporciona un sencillo mecanismo de mapeo, consistente en, a partir del path de la URL (parte situada a la izquierda de .htm), localizar un controlador cuyo nombre coincida con dicho path, terminado en la palabra Controller.

Por ejemplo, si la URL de la petición termina en /login.htm, se intentará localizar un controlador cuya clase se llame LoginController.

Tanto el controlador como el propio ControllerClassNameHandlerMapping, d**eberán estar registrados como beans en el archivo de configuración de Spring**:

```
<bean class="org.springframework.web.servlet.mvc.support.

                                    ControllerClassNameHandlerMapping"/>

 <bean class="controladores.LoginController"/>
```

Centrándonos propiamente en la clase controladora, dicha clase deberá heredar AbstractController, localizada en org.springframework.web.servlet.mvc. Dicha clase cuenta con el método *handleRequestInternal()*, que será llamado por el contenedor de Spring para atender a la petición entrante.

El método *handleRequestInternal()* tiene la misma funcionalidad que el método *service()* de un servlet. Este método tendrá que ser sobrescrito por el programador y definir en él las instrucciones para el procesamiento de la petición. El formato de *handleRequestInternal()* es el siguiente:

```
protected ModelAndView handleRequestInternal(

        HttpServletRequest request,

        HttpServletResponse response) throws Exception{



}
```

Al igual que *service()*, el método *handleRequestInternal()* recibe como parámetros los objetos HttpServletRequest y HttpServletResponse, proporcionados por el servidor de aplicaciones, con los que podemos acceder a los datos de la petición y manipular la respuesta. Pero a diferencia de *service(),* que tiene como tipo de devolución void, *handleRequestInternal()* devuelve un objeto **ModelAndView**.

8.2.4. La clase ModelAndView

A través de un objeto org.springframework.web.servlet.ModelAndView el controlador de acción puede informar al front-controller sobre la vista que debe ser procesada tras su ejecución, para ello, basta con crear un objeto de este tipo utilizando la versión del constructor en la que se proporciona como parámetro el nombre de la vista que se ha de ejecutar:

```
ModelAndView(String nombreVista)
```

La siguiente instrucción provocaría que el front controller transfiriese la petición a la vista “resultado” después de la ejecución del controlador:

return new ModelAndView(“resultado”);

Una vez recuperado el nombre de la vista a través del objeto, el controlador necesita localizar el recurso asociado dicho nombre. Para esta labor se apoya en la clase **InternalResourceViewResolver**, que tiene como función mapear los nombres lógicos asociados a los objetos ModelAndView a páginas JSP.

El sistema de mapeo utilizado por InternalResourceViewResolver consiste en añadir un prefijo y un sufijo al nombre lógico para formar la dirección real del recurso. Los valores del prefijo y sufijo se establecen en la instrucción de configuración del bean InternalResourceViewResolver dentro del archivo de configuración de Spring:

```
<bean id="viewResolver"

    class="org.springframework.web.servlet.view.InternalResourceViewResolver"

    p:prefix="/WEB-INF/jsp/"

    p:suffix=".jsp" />
```

Donde “p” representa el prefijo asociado al espacio de nombres http://www.springframework.org/schema/p que es donde están definidos los atributos *prefix* y *suffix*.

Según este sistema de mapeo, si como en el ejemplo anterior un controlador de acción devuelve al front-controller un objeto ModelAndView asociado a la vista “resultado”, la petición será transferida al recurso resultado.jsp, localizado en el subdirectorio jsp dentro de WEB-INF (figura 8)

Figura. 8.

En caso de que la petición deba ser dirigida a otro controlador de acción en vez de a una página JSP, el nombre virtual indicado en la creación del ModelAndView deberá estar precedido por el prefijo forward. Por ejemplo, si queremos transferir la petición al controlador ResultadoController, utilizaríamos:

return new ModelAndView(“forward:resultado.htm”);

La existencia del prefijo “forward” hará que el front-controller ignore el sistema de mapeo definido por InternalResourceViewResolver y directamente procese la url que figura a continuación de “forward:”.

Además de indicarnos la vista a procesar, los objetos ModelAndView pueden almacenar atributos de petición que luego serán recogidos por la vista o recurso destino. Para este fin la clase ModelAndView dispone del método *addObject()*:

ModelAndView addObject(String nombre, Object objeto)

Los objetos almacenados en el modelo pueden ser recuperados por la vista o controlador de acción al que se dirigirá la petición a través del método *getAttribute()* de HttpServletRequest o bien utilizando el objeto implícito EL *requestScope*. Por ejemplo, si el controlador necesita enviar a la vista un objeto JavaBean de tipo Persona tendríamos que escribir:

```
ModelAndView model = new ModelAndView("resultado");

model.addObject("datos", new Persona("Juan",334920));

return model;
```

De cara a recuperar el objeto desde la vista y mostrar sus datos sería:

```
Nombre: ${requestScope.datos.nombre}<br/>

Teléfono: ${requestScope.datos.telefono}
```

8.3. DESARROLLO DE UNA APLICACIÓN MVC CON NETBEANS

El entorno de desarrollo Netbeans simplifica la creación de aplicaciones Web basadas en Spring MVC. Durante la creación del proyecto Web, podemos indicarle a netbeans que añada al mismo todos los elementos necesarios para utilizar el framework.

A modo de ejercicio de ejemplo, vamos utilizar Netbeans para crear una sencilla aplicación Web basada en Spring MVC, que realice la validación de los credenciales de un usuario y, en función de si se trata de un usuario válido o no, nos dirija a una página de bienvenida o error. El esquema de páginas de la aplicación se muestra en la figura 9.

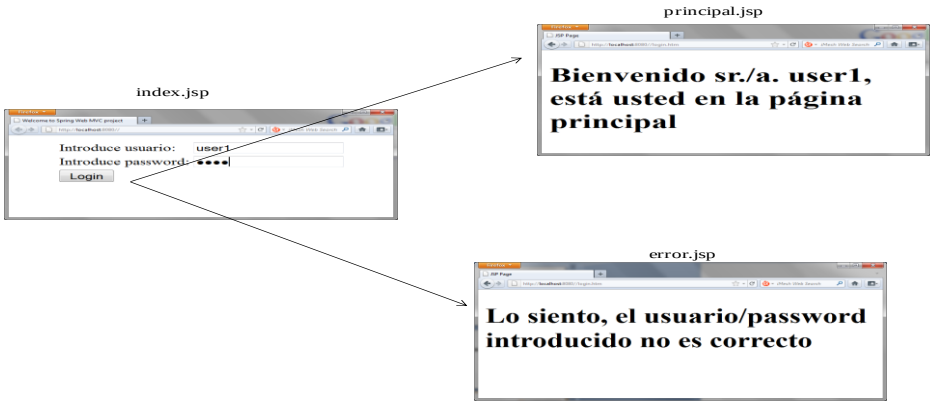

Figura. 9.

En primer lugar, crearemos un proyecto de tipo aplicación Web, proporcionando en el cuadro de diálogo inicial el nombre y ubicación del mismo (figura 10)

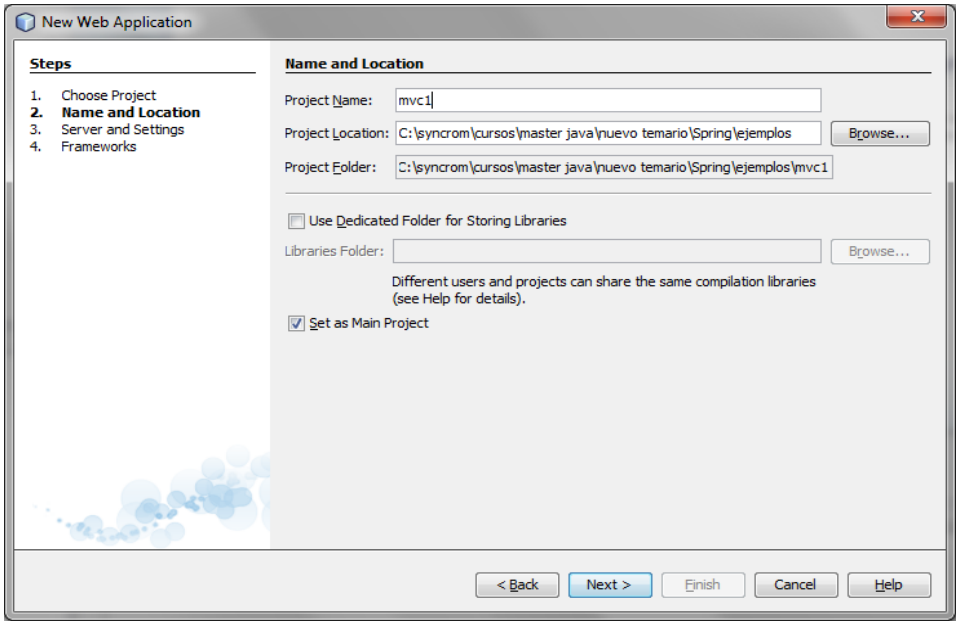

Figura. 10.

En el siguiente paso elegiremos el servidor que queremos utilizar para desplegar la aplicación y la versión JavaEE correspondiente. Spring es soportado tanto por glassfish 3 como por Tomcat 7 y 8, que son los servidores de aplicaciones que habitualmente vienen incorporados con Netbeans, por lo que podemos elegir cualquiera de ellos. En cuanto a la versión de JavaEE, podemos utilizar la 5 o la 6.

En este paso también indicaremos el context-root o dirección Web que queremos asociar a nuestra aplicación.

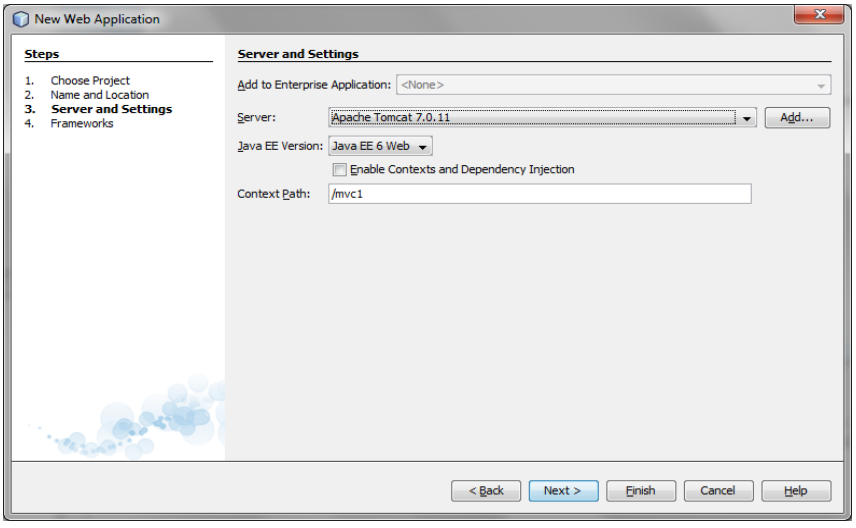

Figura. 11.

En el siguiente y último paso se nos preguntará si queremos utilizar algún framework dentro de nuestra aplicación. Nosotros elegiremos Spring MVC.

Al marcar la opción del framework, se habilitan unas pestañas en la parte inferior para que selecciones algunas opciones adicionales, como la versión a emplear (en nuestro caso la 3 o 4) o, en la pestaña “Configuration”, el nombre que queremos asignar al servlet DispatcherServlet y el formato de direcciones mapeadas al mismo (figura 12)

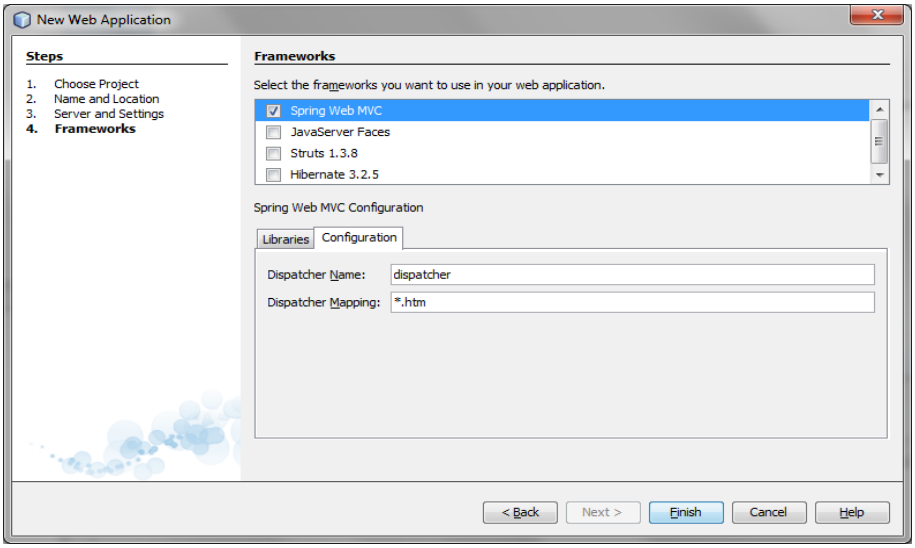

Figura. 12.

Tras pulsar el botón “Finish”, el asistente procede a la construcción del proyecto.

En la ventana de proyecto de Netbeans podemos ver como se han añadido al mismo todas las librerías de Spring para poder hacer uso del framework dentro de nuestra aplicación.

Además de ello, podemos comprobar cómo se han agregado los dos archivos de configuración comentados anteriormente: applicationContext.xml y dispatcher-servlet.xml. El primero de ellos está vacío, pero en el segundo podemos comprobar cómo se encuentran definidos los beans ControllerClassNameHandlerMapping, encargado del mapeo de peticiones a controladores, e InternalResourceViewResolver, que asocia las direcciones virtuales definidas en ModelAndView con las direcciones reales de las vistas a procesar (figura 13).

```
<?xml version="1.0" encoding="UTF-8"?>
<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:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
      http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:prefix="/WEB-INF/jsp/"
        p:suffix=".jsp" />

      :
</beans>
```

Figura. 13.

En este archivo aparecen también otros elementos registrados que no intervienen en la implementación de una aplicación MVC estándar y que hemos omitido en el listado anterior.

Si abrimos el archivo web.xml, observaremos como también el asistente ha registrado por nosotros el escuchador ContextLoaderListener y el servlet DispatcherServlet, al que por defecto se le ha asociado cualquier dirección que finalice en .htm. Por tanto, nuestro trabajo de programación se deberá centrar exclusivamente en la implementación del controlador de acción y las páginas JSP.

8.3.1. Creación del controlador de acción

En el ejercicio que proponemos necesitamos únicamente un controlador de acción, cuya misión será la de comprobar si los credenciales de usuario son correctos y generar el objeto ModelAndView correspondiente.

Netbeans nos proporciona un asistente para simplificar la creación de estas clases. Para acceder al mismo, nos situamos sobre el icono del proyecto y en el menú que aparece al pulsar el botón derecho del ratón seleccionamos *New -> Other*. A continuación, en el cuadro de diálogo *New File* elegimos la opción “Abstract Controller” dentro de la categoría Spring Framework (figura 14)

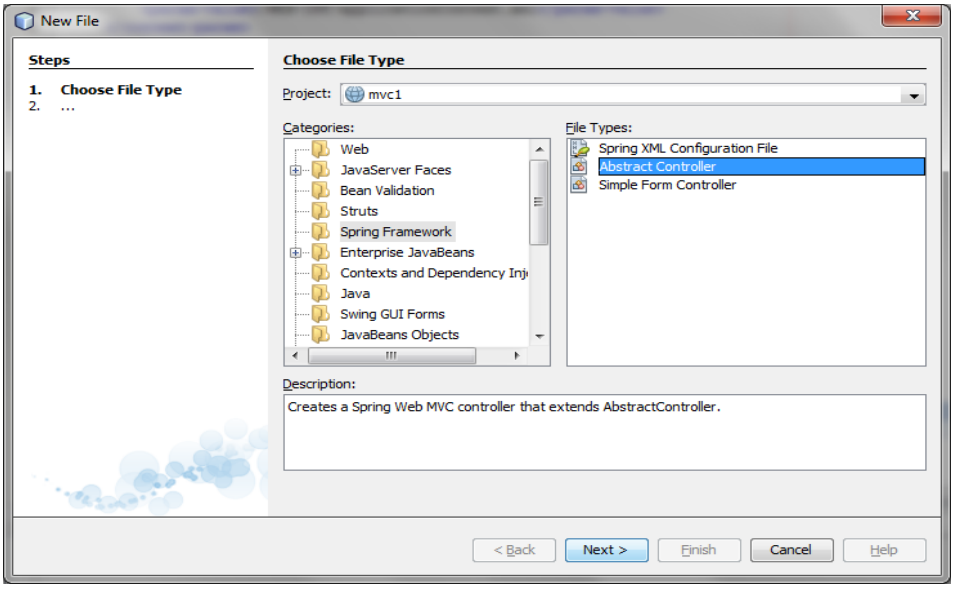

Figura. 14.

En el siguiente y último paso, le indicaremos el nombre que le vamos a dar a la clase (LoginController) y el paquete en el que la queremos guardar.

Al finalizar el asistente, se creará una subclase de AbstractController que incluirá la sobrescritura del método *handleRequestInternal()* dispuesto para ser implementado.

En nuestro ejemplo, la implementación de *handleRequestInternal()* consistirá en comprobar si el usuario y password coinciden con unos valores preestablecidos:

```
package controladores;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;
```

```
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
public class LoginController extends AbstractController {
   protected ModelAndView handleRequestInternal(
       HttpServletRequest request,
       HttpServletResponse response) throws Exception {
      ModelAndView model;
      if(request.getParameter("usuario").equals("user1")&&
                  request.getParameter("password").equals("pwd1")){
         model=new ModelAndView("principal");
      }
      else{
         model=new ModelAndView("error");
      }
      return model;
   }
}
```

Netbeans no registra automáticamente el controlador en el archivo de configuración de Spring, por lo que habrá que hacerlo manualmente. Para ello, añadiremos la siguiente entrada en el dispatcher-servlet.xml:

```
<bean class="controladores.LoginController"/>
```

8.3.2. Creación de las vistas

Las vistas de la aplicación están constituidas por las páginas index.jsp, que es la que contiene el formulario para la solicitud de los datos, principal.jsp y error.jsp. La primera de ellas deberá estar situada en el directorio raíz de la aplicación Web, mientras que las otras dos, que son a las que internamente se redirige al usuario desde el controlador, conviene que se encuentren situadas dentro del directorio privado WEB-INF; en nuestro ejemplo, en el subdirectorio WEB-INF\jsp.

Para crear cada una de las páginas, seleccionaremos la opción JSP de la categoría Web, dentro del cuadro de diálogo de nuevo fichero.

El código de la página **index.jsp** deberá quedar como se indica en el siguiente listado:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```
    "http://www.w3.org/TR/html4/loose.dtd">


<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Welcome to Spring Web MVC project</title>
  </head>

  <body>


    <center>
      <form action="login.htm" method="post">
       <table>
          <tr>
             <td>Introduce usuario:</td><td><input type="text" name="usuario"/></td>
          </tr>
          <tr>
             <td>Introduce password:</td><td>
                <input type="password" name="password"/></td>
          </tr>
          <tr>
             <td colspan="2"><input type="submit" value="Login"></td>
          </tr>
       </table>
      </form>
    </center>
  </body>
</html>
```

Como vemos, el atributo “action” del formulario contiene la dirección “login.htm”, que según el sistema de mapeo definido por **ControllerClassNameHandlerMapping**, hará que la petición sea dirigida a LoginController.

La página principal.jsp nos mostrará un mensaje personalizado con el nombre del usuario suministrado:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

   <head>

      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

      <title>JSP Page</title>

   </head>

   <body>

      <h1>Bienvenido sr./a. ${param["usuario"]}, está usted en la página principal</h1>

   </body>

</html>
```

Mientras que error.jsp simplemente visualizará un mensaje de error:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

   <head>

      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

      <title>JSP Page</title>

   </head>

   <body>

      <h1>Lo siento, el usuario/password introducido no es correcto</h1>

   </body>

</html>
```

8.3.3. Definición de controladores como clases POJO

Desde la versión Spring 3, a la hora de definir un controlador de acción no necesitamos que la clase del controlador tenga que heredar AbstractController, puede ser una simple clase normal Java a la que tendremos que aplicar la anotación @Controller del paquete org.springframework.stereotype. En el caso del controlador de acción LoginController presentado en el ejemplo anterior, en la versión 3 de Spring podría definirse de la siguiente manera:

```
@Controller

public class Login{

        :

}
```

8.3.3.1. Métodos controladores de acción

Dado que la clase controladora no tiene que heredar AbstractController, el método o métodos controladores de acción podrán tener cualquier nombre y deberán declararse con la anotación @RequestMapping, mediante la que se establecerá el mapeo de URL al método controlador controlador:

```
@RequestMapping(“/login”)

public String metodoControlador(..){

:

}
```

Como se puede apreciar, a través de @RequestMapping indicamos la URL que provocará la ejecución del método controlador, es decir, ya no se utilizará el nombre de la clase para establecer el mapeo, por lo que ésta podrá llamarse de cualquier manera.

En cuanto al tipo de devolución del método, deberá ser String y servirá para determinar, a través del InternalResourceViewResolver, la vista que se debe enviar al usuario.

Respecto a los argumentos recibidos por los métodos controladores de acción, Spring nos ofrece una gran flexibilidad, ya que se admiten numerosas posibilidades a hora de definir estos. Por ejemplo, entre los posibles argumentos que podemos definir en los controladores de acción tenemos:

- Los objetos HttpServletRequest y HttpServletResponse
- Un objeto Model para el almacenamiento de atributos necesarios para la vista
- Cualquier número de parámetros recibidos en la petición, declarados individualmente con la anotación @RequestParam
- Cualquier número y tipo de atributos, declarados con la anotación @ModelAttribute
- Cualquier cookie existente en la petición, declarada con @CookieValue

Según lo anterior, la clase LoginController definida en el ejemplo anterior podría implementarse de la siguiente forma:

```
@Controller

public class Login {

        @RequestMapping("/login")

        public String control (Model m,
```

```
                @RequestParam("usuario") String user,
                @RequestParam("password") String password){
            if(user.equals("user1")&&password.equals("pwd1")){
                m.addAttribute("texto", "bienvenido a mi pagina ");
                return "principal";
            }
            else{
                m.addAttribute("texto", "el password no es correcto ");
                return "error";
            }
        }
    }
```

En este ejemplo se genera un mensaje para la vista, depositado como un atributo de Model, y que podrá ser recuperado por esta utilizando la expresión {nombreAtributo}.

Además de a una determinada URL, puede asociarse un método controlador de acción a un determinado método de envío. Por ejemplo, sin en una misma clase *controller* queremos definir dos métodos, uno para ser ejecutado con peticiones de tipo POST y otro con tipo GET, se debería implementar de la siguiente manera:

```
@Controller
public class Controlador{
    @RequestMapping(value="/direccion", method=RequestMethod.POST)
    public String metodoPost(..){
        :
    }
    @RequestMapping(value="/direccion", method=RequestMethod.GET)
    public String metodoGet(..){
        :
    }
}
```

Otra forma de implementar la clase equivalente a la anterior sería:

```
@Controller
@RequestMapping("/direccion")
```

```
public class Controlador{
        @RequestMapping(method=RequestMethod.POST)
        public String metodoPost(..){
                :
        }
        @RequestMapping(method=RequestMethod.GET)
        public String metodoGet(..){
                :
        }
}
```

8.3.3.2. Información de configuración

A nivel de archivos de configuración, en el caso del web.xml se deberá mantener el registro de ContextLoaderListener para realizar la carga del contenedor de Spring. Por otro lado, el servlet DispatcherServlet no tendrá que ser mapeado necesariamente a una dirección del tipo *.htm, bastará con asociarle el contexto raíz de la aplicación (/) para que utilice como dirección de mapeo la cadena que aparezca a continuación de la barra:

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Respecto al archivo de configuración de Spring (habitualmente, el dispatcher-servlet.xml), no tendremos que registrar ni el bean ControllerClassNameHandlerMapping, ni el propio objeto controlador, **bastará con incluir el elemento <context:component-scan/> para que Spring se encargue de instanciar automáticamente las clases anotadas con @Controller**. En el atributo base-package se deberá indicar el paquete donde se encuentran las clases controller:

El aspecto que debería tener el archivo dispatcher-servlet.xml para la aplicación de ejemplo anterior sería:

```
<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:context="http://www.springframework.org/schema/context"

        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd

                http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">


    <context:component-scan base-package="controllers"/>

    <!-- el objeto viewResolver debe registrarse igualmente como en el caso anterior-->

    <bean id="viewResolver"

      class="org.springframework.web.servlet.view.InternalResourceViewResolver"

      p:prefix="/" p:suffix=".jsp"/>


</beans>
```

8.4. RECOGIDA DE DATOS DE UN FORMULARIO

A través de los argumentos del método controlador de acción, podemos recoger cada parámetro enviado en una petición cliente de forma individual, o recurrir al objeto HttpServletRequest para recuperarlos uno a uno a través de su método *getParameter()*. No obstante, existe una forma más simple de capturar los datos utilizando los llamados controladores de formulario.

Un controlador de formulario no es más que una clase anotada con @Controller, que puede recibir los parámetros suministrados a través de un formulario encapsulados en un JavaBean.

Para poder recibir los datos encapsulados en un JavaBean, es necesario que el formulario de la página JSP que realiza la captura de los mismos haya sido definida utilizando la librería html de Spring.

Para explicar el funcionamiento de los controladores de formulario, vamos a partir de un ejemplo. Supongamos que queremos implementar la típica página de registro en la que se solicitan unos datos al usuario y la aplicación los graba en una base de datos. No nos interesa la capa de modelo encargada de realizar el registro de los datos, sino la capa vista que realiza la captura de los datos y la capa controlador que los recoge.

En primer lugar, definiremos una clase JavaBean a la que llamaremos Persona, donde serán encapsulados los datos recogidos del formulario. La clase tendrá cinco propiedades, una para cada campo: nombre, apellidos, edad, email y telefono. Este será el código de la clase Persona:

```
public class Persona {

   private String nombre;
```

```
private String apellidos;
private int edad;
private String email;
private long telefono;
public String getApellidos() {
    return apellidos;
}
public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}
public int getEdad() {
    return edad;
}
public void setEdad(int edad) {
    this.edad = edad;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public String getNombre() {
    return nombre;
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}

public long getTelefono() {
```

```
        return telefono;
    }
    public void setTelefono(long telefono) {
        this.telefono = telefono;
    }
}
```

En segundo lugar, presentaremos la página registro.jsp que sería la encargada de capturar los datos de la persona:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!DOCTYPE html>
<html>
    <body>
        <center>
            <h1>Página de registro</h1>
            <form:form action="register" method="post" modelAttribute="persona">
              <br/><br/>
              Nombre:  <form:input path="nombre"/><br/>
              <br/>
              Apellidos :  <form:input path="apellidos"/><br/>
              <br/>
              Edad : <form:input path="edad"/><br/>
              <br/>
              Email : <form:input path="email"/><br/>
              <br/>
              Teléfono : <form:input path="telefono"/><br/>
              <br/>
              <input type="submit" name="Submit" value="Registrar"/>
            </form:form>
        </center>
```

```
    </body>
</html>
```

El formulario se define a través del elemento *form* de la librería de acciones de Spring, que no resulta muy diferente del <form> estándar de HTML. Destacar el atributo *modelAttribute*, en el que se indica el identificador del JavaBean en el que se almacenarán los datos del formulario. Cada campo del formulario utiliza un atributo *path*, en el que se indica la propiedad del bean asociada con el campo.

Por último, presentamos el código del controlador de formulario, que como vemos es una clase POJO anotada con @Controller, que se encargará de procesar la petición *register* enviada desde el formulario:

```
@Controller
@RequestMapping("/register")
@SessionAttributes("persona")
public class RegistroController  {
    @RequestMapping(method=RequestMethod.GET)
    public String init(Model model) {
        //se ejecuta al recibir la petición para acceder a la página
        //de registro. Se encarga de crear el bean y prepararlo
        Persona persona=new Persona();
        model.addAttribute("persona", persona);
        //dirige a la página de registro
        return "registro";
    }
    @RequestMapping(method=RequestMethod.POST)
    public String doSubmitForm(@ModelAttribute("persona") Persona persona) {
        //procesa el objeto persona que incluye los datos
        //del formulario cliente
        //:
        return "registrado";
    }
}
```

Como podemos observar, la clase define dos métodos controladores de acción: *init()*, que está asociado a la petición GET que provoca la llegada a la página, y *doSubmitForm()*, que será ejecutado al producirse el submit del formulario de registro.

El método *init()* tiene como misión preparar el objeto JavaBean, de modo que realiza la instanciación del mismo y lo almacena como un atributo de Model, de ahí que desde la página de registro hagamos referencia al mismo a través de modelAttribute. El método devuelve como valor “registro” a fin de que el usuario será dirigido a dicha página

En cuanto a *doSubmitForm()*, como vemos recibe el objeto persona ya rellenado en el atributo anotado con @ModelAttribute y su misión será invocar a la lógica de negocio.

Destacar también el uso de la anotación **@SessionAttributes**, que permite definir el objeto persona con ámbito de sesión, a fin de que esté disponible con la petición de acceso a la página (GET) y con la de envío de los datos del formulario (POST)

Ejercicio 6

En este último ejercicio del tema vamos a crear una nueva versión de la aplicación de envío y recepción de mensajes desarrolladla en el ejercicio 3, utilizando Spring MVC para la implementación de vista y controlador.

Lo primero que haremos será crear una aplicación Web con Netbeans, y añadiremos como framework MVC a Spring. Con esto tenemos creada la estructura del proyecto.

Dado que el modelo será exactamente igual que el del ejercicio anterior, podemos copiar y pegar directamente en este proyecto las clases Mensaje, GestionMensajes y GestionMensajesImpl creadas en el ejercicio 3. Por su parte, incorporaremos el archivo de configuración de Spring del proyecto antiguo a la carpeta WEB-INF del nuevo proyecto, renombrándolo como modeloConfig.xml. En esta carpeta tendremos los dos archivos de configuración de Spring, el del modelo y el dispatcher-servlet.xml creado automáticamente por Netbeans El contenido de éste último archivo deberá quedar como se indica en el siguiente listado:

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

<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:context="http://www.springframework.org/schema/context"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd

                           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.1.xsd">



    <!-- permite al contenedor de Spring instanciar automáticamente

         las clases controladoras

       anotadas con @Controller -->

       <context:component-scan base-package="controllers"/>
```

```
        <!-- Resuelve los ModelAndView. En este caso, transfiere la petición a páginas jsp
                que estén localizadas en la dirección prefix. El nombre de las páginas
                se obtiene del valor devuelto por los métodos de los controller
         -->
        <bean id="viewResolver"
            class="org.springframework.web.servlet.view.InternalResourceViewResolver"
            p:prefix="/" p:suffix=".jsp"/>
    </beans>
```

En cuanto a web.xml, tendrá que quedar como se indica a continuación:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd">
        <context-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/modeloConfig.xml</param-value>
        </context-param>
        <listener>
            <listener-class>org.springframework.web.context.ContextLoaderListener
                </listener-class>
        </listener>
        <servlet>
            <servlet-name>dispatcher</servlet-name>
            <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
            <load-on-startup>2</load-on-startup>
        </servlet>
        <servlet-mapping>
            <servlet-name>dispatcher</servlet-name>
            <url-pattern>*.htm</url-pattern>
        </servlet-mapping>
```

```
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>inicio.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Obsérvese como en el elemento <context-param> se indica la dirección del archivo de configuración del modelo para que sea cargado por el contexto de Spring durante el arranque de la aplicación.

Seguidamente, implementamos las dos clases controladoras para la gestión de las operaciones de la aplicación: ProcesaMensajeController y RecuperaMensajesController.

```
package controllers;

import javabeans.Mensaje;

import modelo.GestionMensajes;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.SessionAttributes;


@Controller

@RequestMapping("/procesaMensaje")

@SessionAttributes("mensaje")


public class ProcesaMensajeController {

        @Autowired
```

```
	GestionMensajes gestion;;
	@RequestMapping(method=RequestMethod.GET)
	public String setupForm(Model model){
		Mensaje mensa=new Mensaje();
		model.addAttribute("mensaje", mensa);
		return "envio";
	}
	@RequestMapping(method=RequestMethod.POST)
	public String submitForm(@ModelAttribute("mensaje") Mensaje mensaje){
		gestion.grabaMensaje(mensaje);
		return "inicio";
	}
}
```

```
package controllers;

import java.util.List;
import javabeans.Mensaje;
import javax.servlet.http.HttpServletRequest;
import modelo.GestionMensajes;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/recuperaMensajes")
public class RecuperaMensajesController {
	@Autowired
```

```
        GestionMensajes gestion;
        @RequestMapping(method=RequestMethod.POST)
        public String dispatch(Model model, HttpServletRequest request){
                List<Mensaje> mensajes=(List<Mensaje>)
                    gestion.obtenerMensajes(request.getParameter("nombre"));
                model.addAttribute("mensajes", mensajes);
                return "listado";
        }
}
```

Finalmente, indicamos los listados de las vistas:

inicio.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<html>
<body>
<center>
    <br/><br/>

    <a href="procesaMensaje.htm">
        Enviar mensaje
    </a><br/><br/>
    <a href="mostrar.jsp">
        Leer mensajes
    </a>
</center>
</body>
</html>
```

mostrar.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

```
<html>
<body>
<center>
   <br/><br/>
   <form action="recuperaMensajes.htm" method="post">
   Introduzca su nombre:<input type="text" name="nombre"><br><br>
   <input type="submit">
   </form>
</center>
</body>
</html>
```

envio.jsp

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<title>envio</title>
</head>
<body>
<center>
  <h1>Generación de mensajes</h1>
  <form:form action="procesaMensaje.htm" method="post" modelAttribute="mensaje">
  <br/><br/>
  <b>Datos del mensaje:</b><br/><br/>
  Introduzca destinatario:  <form:input path="destino"/><br/>
  <br/>
  Introduzca remitente :  <form:input path="remite"/><br/>
  <br/>
  Introduzca texto : <br/>
  <form:textarea path="texto"/>
  <hr/><br/>
```

```
  <input type="submit" name="Submit" value="Enviar"/>
  </form:form>
</center>
</body>
</html>
```

listado.jsp

```
<%@ page import="javabeans.*,java.util.*"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>ver</title>
</head>
<body>
<center>
<h1>
Mensajes para ${param.nombre}
</h1>
<table border=1>
<tr><th>Remitente</th><th>Mensaje</th></tr>
<c:if test="${empty requestScope.mensajes}">
    <jsp:forward page="/nomensajes.jsp"/>
</c:if>
<c:forEach var="m" items="${requestScope.mensajes}">
      <tr><td>${m.remite}</td><td>${m.texto}</td></tr>
</c:forEach>
</table>
<br/><br/>
<a href="inicio.jsp">Inicio</a>
</center>
</body>
```

```
</html>
```

nomensajes.jsp

```
<html>
<head>
<title>nomensajes</title>
</head>
<body>
  <center>
    <h2>
     Lo siento, ${param.nombre} no tiene mensajes
    </h2>
    <br/><br/><br/><br/>
    <a href="inicio.jsp">Inicio</a>
  </center>
</body>
</html>
```