Capítulo 3 : Spring, un *framework* de aplicación

En este capítulo se dan a conocer todos los aspectos técnicos de Spring, los conceptos básicos, como sus creadores, sus características esenciales, las partes de su arquitectura, así como los componentes que lo conforman. En algunas partes se dan pedazos de código con el fin de ejemplificar mejor cierta situación. Este capítulo es un análisis un poco más profundo de las diferentes partes que conforman la arquitectura de Spring. Con el fin de ejemplificar mejor la situación de la aplicación mencionada en el capítulo siguiente se profundizó en la parte de Web MVC y JDBC.

3.1 Introducción e historia

Spring es un *framework* de aplicación desarrollado por la compañía Interface 21, para aplicaciones escritas en el lenguaje de programación Java. Fue creado gracias a la colaboración de grandes programadores, entre ellos se encuentran como principales partícipes y líderes de este proyecto Rod Johnson y Jürgen Höller. Estos dos desarrolladores, además de otros colaboradores que juntando toda su experiencia en el desarrollo de aplicaciones J2EE *(Java 2 Enterprise Editions*), incluyendo EJB (*Enterprise JavaBeans*), Servlets y JSP (*Java Server Pages*), lograron combinar dichas herramientas y otras más en un sólo paquete, para brindar una estructura más sólida y un mejor soporte para este tipo de aplicaciones.

Además se considera a Spring un *framework lightweight*, es decir liviano o ligero, ya que no es una aplicación que requiera de muchos recursos para su ejecución, además el *framework* completo puede ser distribuido en un archivo .jar de alrededor de 1 MB, lo cual

representa muy poco espacio, y para la cantidad de servicios que ofrece es relativamente insignificante su tamaño.

Este *framework* se encuentra actualmente en su versión 1.2.5, aunque es una versión temprana, está adquiriendo gran auge y una gran popularidad. Una de las características que ayuda a este éxito, es que es una aplicación *open source*, lo cual implica que no tiene ningún costo, ni se necesita una licencia para utilizarlo, por lo tanto da la libertad a muchas empresas y desarrolladores a incursionar en la utilización de esta aplicación. Además de que está disponible todo el código fuente de este *framework* en el paquete de instalación.

Spring no intenta "reinventar la rueda" sino integrar las diferentes tecnologías existentes, en un sólo *framework* para el desarrollo más sencillo y eficaz de aplicaciones J2EE portables entre servidores de aplicación [Johnson, 2005].

Otro de los principales enfoques de Spring y por el cual está ganando dicha popularidad es que simplifica el desarrollo de aplicaciones J2EE, al intentar evitar el uso de EJB, ya que como menciona Craig Walls en su libro Spring in Action, "En su estado actual, EJB es complicado. Es complicado porque EJB fue creado para resolver cosas complicadas, como objetos distribuidos y transacciones remotas." [Walls, 2005]. Y muchas veces aunque el proyecto no es lo suficientemente complejo, se utiliza EJB, contenedores de alto peso y otras herramientas que soportan un grado mayor de complejidad, como una solución a un proyecto. "Con Spring, la complejidad de tu aplicación es proporcional a la complejidad del problema que se está resolviendo." [Walls, 2005]. Esto sin embargo no le quita crédito a EJB, ya que también ofrece a los desarrolladores servicios valiosos y útiles para resolver ciertas tareas, la diferencia radica en que Spring intenta brindar los mismos servicios pero simplificando el modelo de programación.

Spring fue creado basado en los siguientes principios:

- El buen diseño es más importante que la tecnología subyacente.
- Los JavaBeans ligados de una manera más libre entre interfaces es un buen modelo.
- El código debe ser fácil de probar.

3.2 Arquitectura de Spring

Spring es un *framework* modular que cuenta con una arquitectura divida en siete capas o módulos, como se muestra en la Figura 3.1, lo cual permite tomar y ocupar únicamente las partes que interesen para el proyecto y juntarlas con gran libertad.

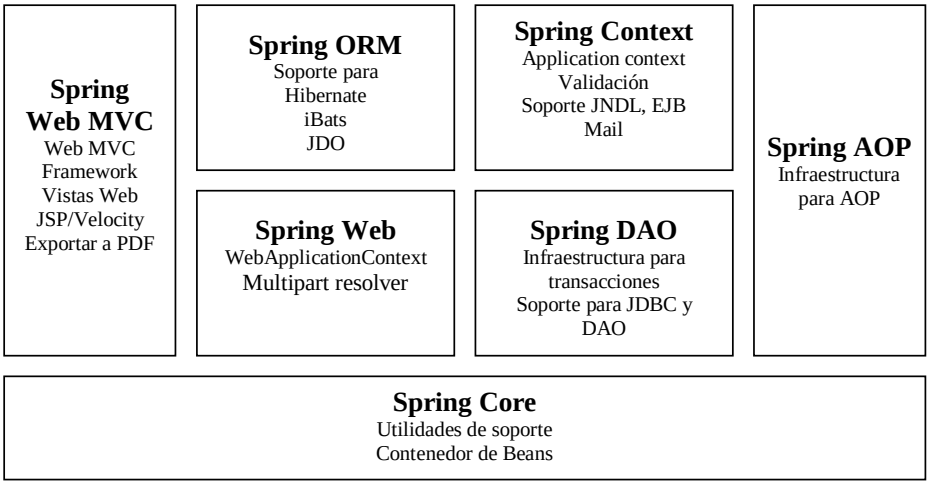

Figura 3.1: Arquitectura de Spring [Balani, 2005]

3.2.1 Spring Core

Esta parte es la que provee la funcionalidad esencial del *framework*, está compuesta por el BeanFactory, el cual utiliza el patrón de Inversión de Control (*Inversion of Control*) y configura los objetos a través de Inyección de Dependencia (*Dependency Injection*). El

núcleo de Spring es el paquete `org.springframework.beans` el cual está diseñado para trabajar con JavaBeans [Johnson, 2005].

3.2.1.1 *Bean Factory*

Es uno de los componentes principales del núcleo de Spring. Es una implementación del patrón Factory, pero a diferencia de las demás implementaciones de este patrón, que muchas veces sólo producen un tipo de objeto, BeanFactory es de propósito general, ya que puede crear muchos tipos diferentes de *Beans* [Walls, 2005]. Los *Beans* pueden ser llamados por nombre y se encargan de manejar las relaciones entre objetos.

Todas las *Bean Factories* implementan la interfase `org.springframework.beans.factory.BeanFactory`, con instancias que pueden ser accesadas a través de esta interfaz. Además también soportan objetos de dos modos diferentes [Johnson. 2005]:

- *Singleton*: Existe únicamente una instancia compartida de un objeto con un nombre particular, que puede ser regresado o llamado cada vez que se necesite. Este es el método más común y el más usado. Este modo está basado en el patrón de diseño que lleva el mismo nombre.
- *Prototype*: también conocido como *non-singleton*, en este método cada vez que se realiza un regreso o una llamada, se crea un nuevo objeto independiente.

La implementación de BeanFactory más usada es `org.springframework.beans.factory.xml.XmlBeanFactory` que se encarga de cargar la definición de cada *Bean* que se encuentra guardada en un archivo XML y que consta de:

id (que será el nombre que el que se le conocerá en las clases), clase (tipo de *Bean*), *Singleton* o *Prototype* (modos del *Bean* antes mencionados), propiedades, con sus atributos *name*, *value* y *ref*, además argumentos del constructor, método de inicialización y método de destrucción. A continuación se muestra un ejemplo de un *Bean*:

```
<beans>
      <bean id=”exampleBean” class=”eg.ExampleBean” singleton=”true”/>
      <property name="driverClassName" value="com.mysql.jdbc.Driver" />
</beans>
```

Como se muestra en el ejemplo la base de este documento XML es el elemento `<beans>`, y adentro puede contener uno o más elementos de tipo `<bean>`, para cada una de las diferentes funciones que se requieran.

Para cargar dicho XML se le manda un `InputStream` al constructor de `XmlBeanFactory` de la siguiente manera:

```
BeanFactory fact = new XmlBeanFactory(new FileInputStream(“bean.xml”));
```

Esto sin embargo no quiere decir que sea instanciado directamente, una vez que la definición es cargada, únicamente cuando se necesite el *Bean* se creará una instancia dependiendo de sus propiedades. Para tomar un *Bean* de un *Factory* simplemente se usa el método `getBean()`, mandándole el nombre del *Bean* que se desea obtener [Walls, 2005]:

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

3.2.1.2 *Inversion of Control*

“*Inversion of Control* se encuentra en el corazón de Spring” [Walls, 2005]. `BeanFactory` utiliza el patrón de Inversión de Control o como se le conoce *IoC*, que es una de las funcionalidades más importantes de Spring. Esta parte se encarga de separar del

código de la aplicación que se está desarrollando, los aspectos de configuración y las especificaciones de dependencia del *framework*. Todo esto configurando los objetos a través de Inyección de Dependencia o *Dependency Injection*, que se explicará más adelante.

Una forma sencilla de explicar el concepto de *IoC* es el "principio Hollywood": "No me llames, yo te llamaré a ti". Traduciendo este principio a términos de este trabajo, en lugar de que el código de la aplicación llame a una clase de una librería, un *framework* que utiliza *IoC* llama al código. Es por esto que se le llama "Inversión", ya que invierte la acción de llamada a alguna librería externa [Johnson, 2005].

3.2.1.3 *Dependency Injection*

Es una forma de Inversión de Control, que está basada en constructores de Java, en vez de usar interfaces específicas del *framework*. Con este principio en lugar de que el código de la aplicación utilicé el API del *framework* para resolver las dependencias como: parámetros de configuración y objetos colaborativos, las clases de la aplicación exponen o muestran sus dependencias a través de métodos o constructores que el *framework* puede llamar con el valor apropiado en tiempo de ejecución, basado en la configuración [Johnson, 2005].

Todo esto se puede ver de una forma de *push* y *pop*, el contenedor hace un *push* de las dependencias para ponerlas dentro de los objetos de la aplicación, esto ocurre en tiempo de ejecución. La forma contraria es tipo *pull*, en donde los objetos de la aplicación jalan las dependencias del ambiente. Además los objetos de la Inyección de Dependencia nunca cargan las propiedades ni la configuración, el *framework* es totalmente responsable de leer la configuración.

Spring soporta varios tipos de Inyección de Dependencia, pero en si estos son los dos más utilizados:

- *Setter Injection*: en este tipo la Inyección de Dependencia es aplicada por medio de métodos *JavaBeans setters*, que a la vez tiene un *getter* respectivo.
- *Constructor Injection*: esta Inyección es a través de los argumentos del constructor.

A continuación se muestra un ejemplo tomado del libro de *Java Development with Spring framework* del autor Rob Johnson, en el cual se muestra como un objeto es configurado a través de Inyección de Dependencia. Se tiene una interfaz `Service` y su implementación `ServiceImpl`. Supóngase que la `ServiceImpl` tiene dos dependencias: un `int` que tiene configura un `timeout` y un DAO (*Data Access Object*). Con el método `SetterInjection` se puede configurar `ServiceImpl` utilizando las propiedades de un *JavaBean* para satisfacer estas 2 dependencias.

```
public class ServiceImpl implements Service {
      private int timeout;
      private AccountDao accountDao;

public void setTimeout(int timeout) {
this.timeout = timeout;
}

public void setAccountDao(AccountDao accountDao) {
      this.accountDao = accountDao;
}
```

Con *Constructor Injection* se le da las dos propiedades al constructor:

```
public class ServiceImpl implements Service {
      private int timeout;
```

```
        private AccountDao accountDao;

public ServiceImpl(int timeout, AccountDao accountDao) {
this.timeout = timeout;
this.accountDao = accountDao;
}
```

"La clave de la innovación de la Inyección de Dependencia es que trabaja con sintaxis pura de Java: no es necesaria la dependencia del API del contenedor" [Johnson, 2005].

3.2.2 Spring Context

"El módulo BeanFactory del núcleo de Spring es lo que lo hace un contenedor, y el módulo de contexto es lo que hace un *framework*" [Walls, 2005].

En sí *Spring Context* es un archivo de configuración que provee de información contextual al *framework* general. Además provee servicios *enterprise* como JNDI, EJB, e-mail, validación y funcionalidad de agenda.

3.2.2.1 Application Context

`ApplicationContext` es una subinterfaz de `BeanFactory`, ya que `org.springframework.context.ApplicationContext` es una subclase de `BeanFactory`. En si todo lo que puede realizar una `BeanFactory` también lo puede realizar `ApplicationContext`. En sí agrega información de la aplicación que puede ser utilizada por todos los componentes.

Además brinda las siguientes funcionalidades extra:

- Localización y reconocimiento automático de las definiciones de los *Beans*
- Cargar múltiples contextos
- Contextos de herencia
- Búsqueda de mensajes para encontrar su origen.
- Acceso a recursos
- Propagación de eventos, para permitir que los objetos de la aplicación puedan publicar y opcionalmente registrarse para ser notificados de los eventos.
- Agrega soporte para internacionalización (i18n)

En algunos casos es mejor utilizar *ApplicationContext* ya que obtienes más funciones a un costo muy bajo, en cuanto a recursos se refiere.

Ejemplo de un ApplicationContext:

```
ApplicationContext ct =
   new FileSystemXmlApplicationContext (“c:\bean.xml”);
ExampleBean eb = (ExampleBean) ct.getBean(“exampleBean”);
```

3.2.3 Spring AOP

Aspect-oriented programming, o AOP, es una técnica que permite a los programadores modularizar ya sea las preocupaciones *crosscutting*, o el comportamiento que corta a través de las divisiones de responsabilidad, como *logging*, y manejo de transacciones. El núcleo de construcción es el *aspect*, que encapsula comportamiento que afectan a diferentes clases, en módulos que pueden ser reutilizados [Johnson, 2005].

AOP se puede utilizar para:

- Persistencia
- Manejo de transacciones
- Seguridad
- *Logging*
- *Debugging*

AOP es un enfoque diferente y un poco más complicado de acostumbrarse en comparación con OOP (*Object Oriented Programming*). Rob Johnson prefiere referirse a AOP como un complemento en lugar de como un rival o un conflicto.

Spring AOP es portable entre servidores de aplicación, funciona tanto en servidores Web como en contenedores EJB.

Spring AOP soporta las siguientes funcionalidades:

- *Intercepción*: se puede insertar comportamiento personalizado antes o después de invocar a un método en cualquier clase o interfaz.
- *Introducción*: Especificando que un *advice* (acción tomada en un punto particular durante la ejecución de un programa) debe causar que un objeto implemente interfaces adicionales.
- *Pointcuts dinámicos y estáticos*: para especificar los puntos en la ejecución del programa donde debe de haber intercepción.

Spring implementa AOP utilizando *proxies* dinámicos. Además se integra transparentemente con los BeanFactory que existen. En el ejemplo siguiente se muestra como definir un *proxy* AOP [Johnson, 2005]:

```
<bean id="myTest"
      class="org.springframework.aop.Framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
            <value>org.springframework.beans.ITestBean</value>
      </property>
      <property name="interceptorNames">
            <list>
                  <value>txInterceptor</value>
                  <value>target</value>
            </list>
      </property>
</bean>
```

3.2.4 Spring ORM

En lugar de que Spring proponga su propio módulo ORM (*Object-Relational Mapping*), para los usuarios que no se sientan confiados en utilizar simplemente JDBC, propone un módulo que soporta los *frameworks* ORM más populares del mercado, entre ellos [Johnson, 2005]:

- Hibernate (2.1 y 3.0): es una herramienta de mapeo O/R *open source* muy popular, que utiliza su propio lenguaje de *query* llamada HQL.
- iBATIS SQL Maps (1.3 y 2.0). una solución sencilla pero poderosa para hacer externas las declaraciones de SQL en archivos XML.
- Apache OJB (1.0): plataforma de mapeo O/R con múltiples APIs para clientes.
- Entre otros como JDO (1.0 y 2.0) y Oracle TopLink.

Todo esto se puede utilizar en conjunto con las transacciones estándar del *framework*. Spring y Hibernate es una combinación muy popular. Algunas de las ventajas que brinda Spring al combinarse con alguna herramienta ORM son:

- *Manejo de sesión*: Spring hace de una forma más eficiente, sencilla y segura la forma en que se manejan las sesiones de cualquier herramienta ORM que se quiera utilizar.
- *Manejo de recursos*: se puede manejar la localización y configuración de los SessionFactories de Hibernate o las fuentes de datos de JDBC por ejemplo. Haciendo que estos valores sean más fáciles de modificar.
- *Manejo de transacciones integrado*: se puede utilizar una plantilla de Spring el para las diferentes transacciones ORM.
- *Envolver excepciones*: con esta opción se pueden envolver todas las excepciones para evitar las molestas declaraciones y los catch en cada segmento de código necesarios.
- *Evita limitarse a un solo producto*: Si se desea migrar o actualizar a otra versión de un ORM distinto o del mismo, Spring trata de no crear una dependencia entre la herramienta ORM, el mismo Spring y el código de la aplicación, para que cuando sea necesario migrar a un nuevo ORM no sea necesario realizar tantos cambios.
- *Facilidad de prueba*: Spring trata de crear pequeños pedazos que se puedan aislar y probar por separado, ya sean sesiones o una fuente de datos (*datasource)*.

3.2.5 Spring DAO

El patrón DAO (*Data Access Object*) es uno de los patrones más importantes y usados en aplicaciones J2EE, y la arquitectura de acceso a los datos de Spring provee un buen soporte para este patrón [Johnson, 2005].

3.2.5.1 DAO y JDBC

Existen dos opciones para llevar a cabo el acceso, conexión y manejo de bases de datos: utilizar alguna herramienta ORM o utilizar el *template* de JDBC (*Java Database Connectivity*) que brinda Spring. La elección de una de estas dos herramientas es totalmente libre y en lo que se debe basar el desarrollador para elegir es en la complejidad de la aplicación. En caso de ser una aplicación sencilla en la cual únicamente una clase hará dicha conexión, entonces la mejor opción sería el Spring JDBC o en caso contrario que se requiera un mayor soporte y sea más robusta la aplicación se recomienda utilizar una herramienta ORM.

El uso de JDBC muchas veces lleva a repetir el mismo código en distintos lugares, al crear la conexión, buscar información, procesar los resultados y cerrar la conexión. El uso de las dos tecnologías mencionadas anteriormente ayuda a mantener simple este código y evitar que sea tan repetitivo, además minimiza los errores al intentar cerrar la conexión con alguna base datos [Walls, 2005]. Este es un ejemplo de como se hace la inserción de datos en una base de datos utilizando JDBC tradicional:

```
public void insertPerson(Person person) throws SQLException {
//Se declaran recursos
Connection conn = null;
PreparedStatement stmt = null;
try {
//Se Abre una conexión
```

```
      conn = dataSource.getConnection();
      //Se crea la declaraciòn
      stmt = conn.prepareStatement("insert into person (" +
      "id, firstName, lastName) values (?, ?, ?)");
      //Se Introducen los parámetros
      stmt.setInt(0, person.getId().intValue());
      stmt.setString(1, person.getFirstName());
      stmt.setString(2, person.getLastName());
      //Se ejecuta la instrucciòn
      stmt.executeUpdate();
      }
//Se atrapan las excepciones
catch(SQLException e) {
      LOGGER.error(e);
}
finally {
            //Se limpian los recursos
      try { if (stmt != null) stmt.close(); }
      catch(SQLException e) { LOGGER.warn(e); }
      try { if (conn != null) conn.close(); }
      catch(SQLException e) { LOGGER.warn(e); }
      }
}
```

De todo este código únicamente el 20% es para un propósito específico, el otro 80% es de propósito general en cada una de las diferentes operaciones de acceso a una base de datos. Esto no quiere decir que no sea importante si no que al contrario es lo que le da la estabilidad y robustez al guardado y recuperación de información.

Las clases bases que Spring provee para la utilización de los DAO son abstractas y brindan un fácil acceso a recursos comunes de base de datos. Existen diferentes implementaciones para cada una de las tecnologías de acceso a datos que soporta Spring.

Para JDBC existe la clase `JdbcDaoSupport` que provee métodos para accesar al `DataSource` y al template pre-configurado que se menciono anteriormente: `JdbcTemplate`.

Únicamente se extiende la clase `JdbcDaoSupport` y se le da una referencia al `DataSource` actual.

"En el ejemplo que se muestra a continuación la clase base es JdbcTemplate, que es la clase central que maneja la comunicación con la base de datos y el manejo de excepciones." [Johnson, 2005]. Esta clase provee varios métodos que realizan la carga pesada y resultan bastante convenientes, como el método `execute()` que aparece en el ejemplo, el cual recibe un comando de SQL como su único parámetro. Para este ejemplo se supone que se tiene una tabla ya creada dentro de una base de datos utilizando el siguiente comando SQL:

```
create table mytable (id integer, name varchar(100))
```

Ahora se quiere agregar algunos datos de prueba:

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

public class MinimalTest extendí TestCase{
      private DriverManagerDataSource dataSource;

      public void setup(){
      dataSource = new DriverManagerDataSource();
      dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
      dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
      dataSource.setUsername("sa");
      dataSource.setPassword("");

      JdbcTemplate Jt = new JdbcTemplate(dataSource);
      Jt.execute("delete from mytable");
      Jt.execute("insert into mytable (id,name) values(1,`John´)");
Jt.execute("insert into mytable (id,name) values(2,`Jane´)");
}
```

```
public void testSomething(){
// Aquí va el codigo de prueba que se puede aplicar
}
```

Como se pudo apreciar en el ejemplo anterior no existe manejo de excepciones, ya que el `JdbcTemplate` se encarga de atrapar todas las `SQLExceptions` y las convierte a una subclase de `DataAccessException`, que es la clase que se encuentra en el nivel superior de la jerarquía de las excepciones de acceso a datos. También el `JdbcTemplate` se encarga de controlar la conexión con la base de datos. Y se puede evitar configurar el `DataSource` en DAO, únicamente se tiene que establecer a través de la inyección de dependencia.

3.2.6 Spring Web

El módulo web de Spring se encuentra en la parte superior del módulo de contexto, y provee el contexto para las aplicaciones web. Este módulo también provee el soporte necesario para la integración con el *framework* Struts de Yakarta.

Este módulo también se encarga de diversas operaciones web como por ejemplo: las peticiones multi-parte que puedan ocurrir al realizar cargas de archivos y la relación de los parámetros de las peticiones con los objetos correspondientes (*domain objetcs o business objects*).

3.2.7 Spring Web MVC

Spring brinda un MVC (*Model View Controller*) para web bastante flexible y altamente configurable, pero esta flexibilidad no le quita sencillez, ya que se pueden desarrollar aplicaciones sencillas sin tener que configurar muchas opciones.

Para esto se puede utilizar muchas tecnologías ya que Spring brinda soporte para JSP, Struts, Velocity, entre otros.

El MVC de Spring presenta una arquitectura Tipo 2, para mayor información consultar el Capítulo 2: Marco Teórico de este documento de tesis.

El Web MVC de Spring presenta algunas similitudes con otros *frameworks* para web que existen en el mercado, pero son algunas características que lo vuelven único:

- Spring hace una clara división entre controladores, modelos de JavaBeans y vistas.
- El MVC de Spring está basado en interfaces y es bastante flexible.
- Provee interceptores (*interceptors*) al igual que controladores.
- Spring no obliga a utilizar JSP como única tecnología View también se puede utilizar otras.
- Los Controladores son configurados de la misma manera que los demás objetos en Spring, a través de IoC.
- Los web *tiers* son más sencillos de probar que en otros *frameworks*.
- El web *tiers* se vuelve una pequeña capa delgada que se encuentra encima de la capa de *business objects*.

La arquitectura básica de Spring MVC está ilustrada en la Figura 3.2.

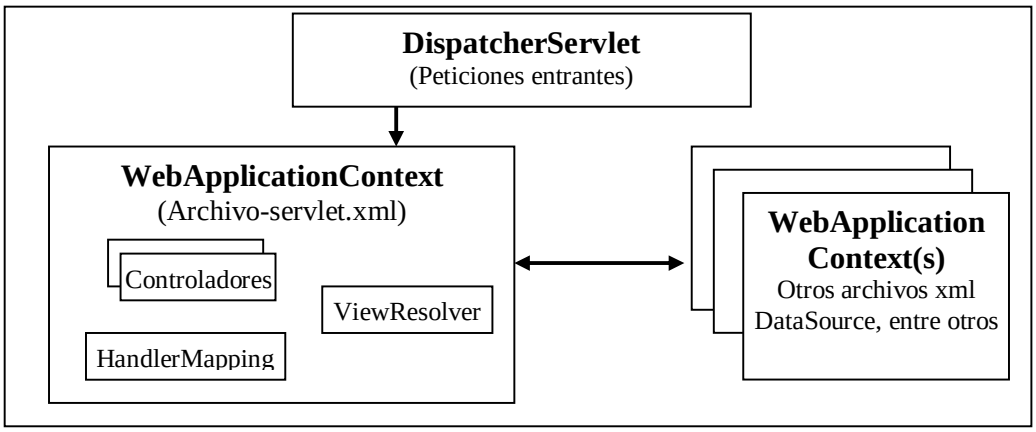

Figura 3.2: Arquitectura básica del Web MVC de Spring [Johnson, 2005]

Para intentar comprender cada parte de la arquitectura del Web MVC de Spring se presenta en la Figura 3.3, el ciclo de vida de una petición o *request* [Walls, 2005]:

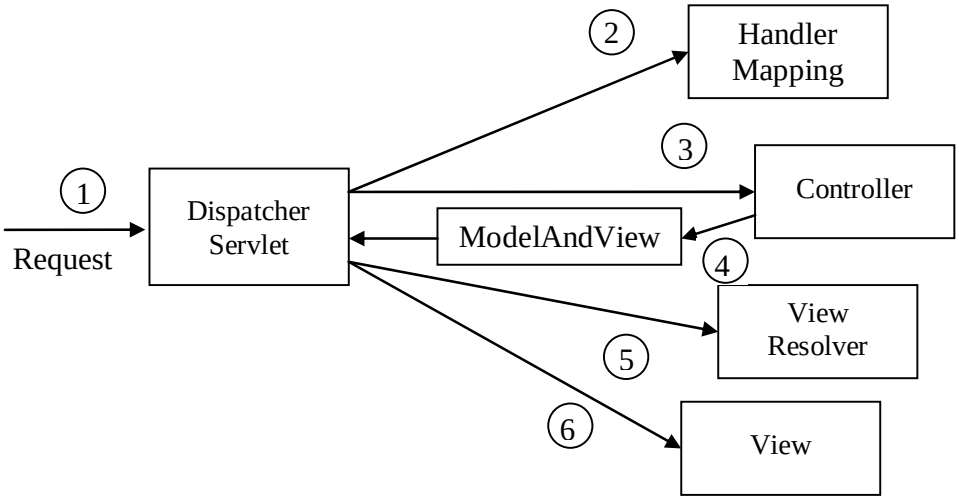

Figura 3.3: Ciclo de vida de un *request* [Walls, 2005]

1. El navegador manda un *request* y lo recibe un DispatcherServlet.
2. Se debe escoger que Controller manejará el *request*, para esto el HandlerMapping mapea los diferentes patrones de URL hacia los controladores, y se le regresa al DispatcherServlet el Controller elegido.

3. El Controller elegido toma el *request* y ejecuta la tarea.
4. El Controller regresa un ModelAndView al DispatcherServlet.
5. Si el ModelAndView contiene un nombre lógico de un View se tiene que utilizar un ViewResolver para buscar ese objeto View que representará el *request* modificado.
6. Finalmente el DispatcherServlet despacha el *request* al View.

Spring cuenta con una gran cantidad de controladores de los cuales se puede elegir dependiendo de la tarea, entre los más populares se encuentra: Controller y AbstractController para tareas sencillas; el SimpleFormController ayuda a controlar formularios y el envió de los mismos, MultiActionController ayuda a tener varios métodos dentro un solo controlador a través del cual se podrán mapear las diferentes peticiones a cada uno de los métodos correspondientes.

3.2.7.1 Dispatcher Servlet

Para configurar el `DispatcherServlet` como el *servlet* central, se tiene que hacer como cualquier *servlet* normal de una aplicación web, en el archivo de configuración web.xml (*Deployment Descriptor*).

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

```
<url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Entonces el *DispatcherServlet* buscará como está indicado por el *tag* `<servlet-name>` el contexto de aplicación (*application Context*) correspondiente con el nombre que se haya puesto dentro de ese *tag* acompañado de la terminación –servlet.xml, en este caso buscará el archivo ejemplo-servlet.xml. En donde se pondrán las definiciones y como su nombre lo indica el contexto de la aplicación dentro de diferentes *beans*, con sus correspondientes propiedades y atributos, para el caso del Web MVC, se pondrán los diferentes *ViewResolver* a utilizar, los controladores y sus propiedades, el HandlerMapping, así como los diferentes *beans* que sean necesarios.

3.2.7.2 Handler Mappings

Existen diversas maneras en que el *DispatcherServlet* pueda determinar y saber que controlador es el encargado de procesar un *request*, y a que *bean* del *Application Context* se lo puede asignar. Esta tarea la lleva a cabo el *Handler Mapping* o el manejador de mapeo, existen 2 tipos principales que son los que mas se usan:

- *BeanNameUrlHandlerMapping*: mapea el URL hacia el controlador en base al nombre del bean del controlador. Ejemplo de cómo se declara:

Esta es la declaración principal del *Handler Mapping:*

```
<bean id="beanNameUrlMapping" class="org.springframework.web.
servlet.handler.BeanNameUrlHandlerMapping"/>
```

A continuación se escribe cada uno de los diferentes URLs que se vayan a utilizar en la aplicación, poniéndolo en el atributo *name* del *bean* y en el atributo *class* la clase del

Controlador que vaya a procesar ese dicho *request*, y finalmente se ponen las diferentes propiedades que utilizará dicho controlador.

```
<bean name="/ejemplo.htm" class="web.ExampleController">
<property name="Servicio">
<ref bean="Servicio"/>
</property>
</bean>
```

Así cuando el usuario entre a una página con el URL /ejemplo.htm el *request* que haga será dirigido hacia el controlador `ExampleController`

- *SimpleUrlHandlerMapping*: mapea el URL hacia el controlador basando en una colección de propiedades declarada en el *applicationContext*. Ejemplo de cómo declarar un HandlerMapping de este tipo:

```
<bean id="simpleUrlMapping" class=
"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
<props>
<prop key="/ejemplo.htm">ExampleController</prop>
<prop key="/login.htm">LoginController</prop>
</props>
</property>
</bean>
```

En este *Handler Mapping* se declara una lista de propiedades en las cuales se pondrá cada uno de los URLs como una propiedad, con el URL como atributo *key* y como valor el nombre del bean del controlador que será responsable de procesar la petición o *request.* Por ejemplo el URL /login.htm será responsabilidad del controlador con el nombre del *bean* LoginController.

Se pueden trabajar con múltiples *Handler Mappings* por si son necesarios en diferentes situaciones, únicamente se le tiene que agregar la propiedad 'order' para poder asignarle en que orden el *DispatcherServlet* va a considerarlas para poder invocarlas.

3.2.7.3 View Resolvers

En el Spring MVC una vista o *View* es un *bean* que transforma los resultados para que sean visibles para el usuario y los pueda interpretar de una mejor forma. En sí un *View Resolver* es cualquier *bean* que implemente la interfaz `org.springframework.web.servlet.ViewResolver`. [Walls, 2005]. Esto quiere decir que un *View Resolver* es el encargado de resolver el nombre lógico que regresa un controlador en un objeto `ModelAndView`, a un nombre de archivo físico que el navegador podrá desplegarle al usuario junto con los resultados procesados.

Spring MVC cuenta con cuatro *View Resolvers* diferentes:

- *InternalResourceViewResolver*: Resuelve los nombres lógicos en un archivo tipo View que es convertido utilizando una plantilla de archivos como JSP, JSTL o Velocity
- *BeanNameViewResolver*: Resuelve los nombres lógicos de las vistas en *beans* de tipo View en el applicationContext del DispatcherServlet
- *ResourceBundleViewResolver*: Resuelve los nombres lógicos de las vistas en objetos de tipo View contenidos en un ResourceBundle o un archivo con extensión .properties.
- *XMLViewResolver*: Resuelve los nombres los lógicos de las vistas que se encuentran en un archivo XML separado.

El *View Resolver* más utilizado es el InternalResourceViewResolver, y se especifica en el *web applicationContext* de nuestra aplicación de la siguiente manera:

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

En este *bean* de `id viewResolver`, se especifica de que clase se quiere que se implemente, para este caso será el tipo de *View Resolver* que se quiere de los 4 mencionados anteriormente. Después vienen tres propiedades:

- viewClass: sirve para especificar que tipo de plantilla se quiere usar para desplegar la vista, en este caso se utilizó JSTL (*Java Standard Tag Library*), que es una de las plantillas más comunes para este tipo de tarea, también se puede seleccionar Velocity o Tiles para desplegar la información
- prefix: el prefijo que antecederá al nombre lógico de la vista, generalmente es la ruta donde se encontrarán los JSP, Ejemplo: /WEB-INF/jsp
- suffix: el sufijo que tendrán nuestras vistas, ya que la mayoría de los nombres físicos que se utilizan son JSPs se le asigna la extensión .jsp.

Todo esto es con el afán de tener una programación que no sea tan dependiente, ya que si se quisiera cambiar de carpeta todas las vistas, lo único que se tendría que modificar sería el *prefix* del *viewResolver*.

3.2.7.4 Controladores

"Si el DispatcherServlet es el corazón del Spring MVC los controladores son los cerebros." [Walls, 2005]. Existen varios controladores cada uno especializado para una tarea en particular, claro que como en todos los casos existen algunos que son de uso general. Para poder crear un controlador basta con implementar la interfaz del Controlador deseado, sobrescribir los métodos que sean necesarios para procesar la petición. Esta situación ayuda a poder modularizar la aplicación ya que con la combinación de diversos controladores que sean enfocados a una tarea en particular se puede concentrarse en cada parte de aplicación aislada y tener una mejor estructura que ayudará a detectar más fácilmente las fallas y errores que puedan surgir.

Existe una variedad de controladores, como se muestra en la Figura 3.4, los cuales poseen una jerarquía. Spring brinda libertad al desarrollador de escoger que tipo de controlador desea implementar, no lo limita como en algunos otros *frameworks*.

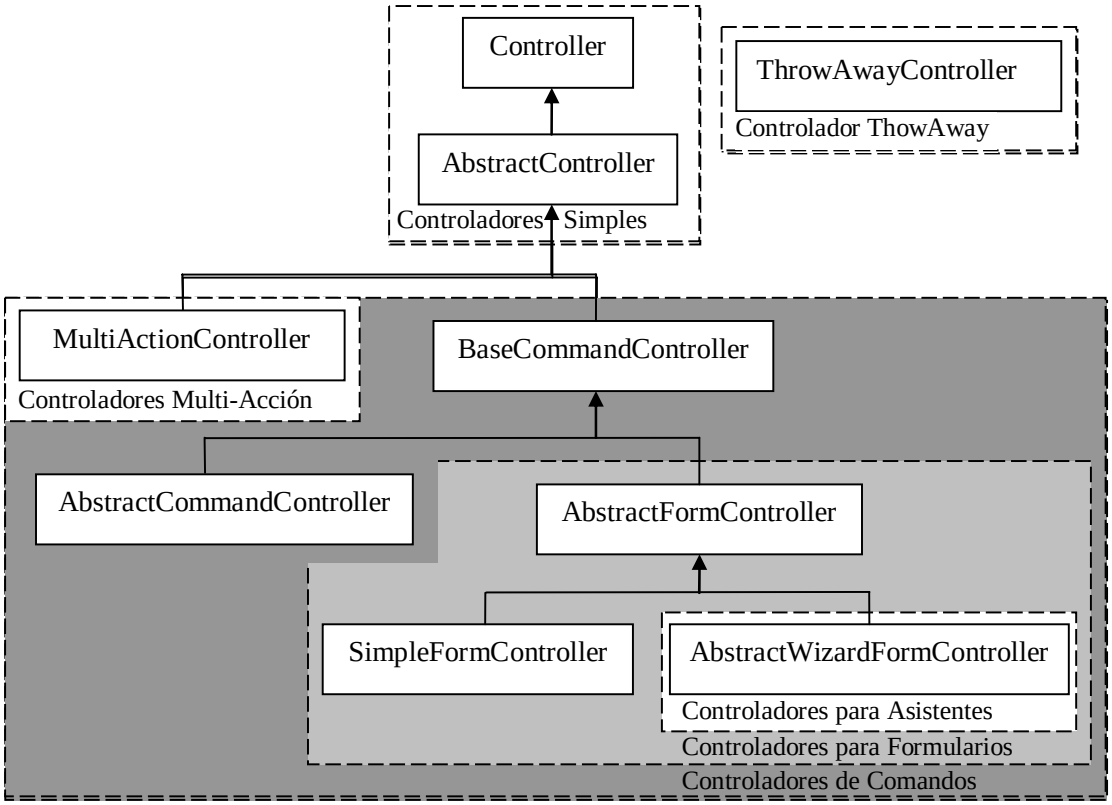

Figura 3.4: Controladores que provee Spring [Johnson, 2005]

La manera en que trabaja cada uno de dichos controladores es similar, cada uno tiene su método específico para procesar las peticiones que haga el usuario, pero todos regresan un objeto tipo ModelAndView, que es como su nombre lo dice el modelo y la vista, ya que se está compuesto de 2 o más atributos. El principal es el nombre de la vista a la que se va a regresar el modelo para que sea desplegado, y los demás atributos pueden ser parámetros que se le agregan por medio del método `.addObject("nombre_parámetro",valor_parámetro)`. Además el Modelo puede estar formado por un solo objeto o por un Map de Objetos, los cuales se identificarán en la vista con el mismo nombre que se haya mandado dentro del Modelo que se le dio al objeto ModelAndView que se este regresando para ser procesado.

El siguiente ejemplo muestra el código básico de un controlador.

```
public class MoviesController extends AbstractController {
//Método que controla el Request
public ModelAndView handleRequestInternal(
HttpServletRequest request, HttpServletResponse response)
throws Exception {
      //Se recupera una lista de todas las peliculas
      List movies = MovieService.getAllMovies();
      //Se manda a la vista movieList el objeto movies, con el nombre
      lógico //movies
      return new ModelAndView("movieList", "movies", movies);
      }
}
```

El objeto ModelAndView contiene el nombre lógico de la vista, el cual el *framework* se encarga a través del *View Resolver* de convertir ese nombre lógico al nombre físico que es el que buscará el servidor web.

3.2.7.5 Procesamiento de formularios

Una de las increíbles facilidades que brinda Spring es la de llenar, recibir y procesar formularios, a través de los 3 diferentes controladores que brinda Spring para el manejo de formas o formularios. Ya que con estos controladores, se facilita el procesamiento, llenado de la forma, almacenamiento (si es necesario) de los datos, así como desplegar errores y confirmaciones. Para el usuario todo este proceso es totalmente transparente ya que el formulario que este llenando tendrá el mismo formato y los mismos elementos que cualquier otro formulario normal.

Existen varios pasos a seguir para poder utilizar un controlador de formularios, uno de ellos rellenar el archivo .jsp con *tags* que provee la librería spring.tld. Esta librería se

debe de combinar de la siguiente manera: En el *Deployment Descriptor* de la aplicación se pone el siguiente *tag*:

```
<taglib>
<taglib-uri>/spring</taglib-uri>
<taglib-location>/WEB-INF/spring.tld</taglib-location>
</taglib>
```

Una vez realizada esta acción se configura el *applicationContext* de la aplicación con el *bean* que describirá toda la información que el controlador necesitará.

Para el control de las formas, se recomienda crear un Objeto que tenga atributos, cada uno de ellos con sus respectivos métodos set y get, para que a través del patrón de Inyección de Dependencia se pueda poner la información dentro de ese objeto y sea más fácil de manipular y almacenar. A este objeto se le conoce como “objeto commando” o *command object*.

Estas son algunas de las propiedades que el controlador necesita, no todas son obligatorias.

- **formView**: nombre lógico de la vista que contiene la forma.
- **successView**: nombre lógico de la vista que se desplegará una vez que la forma sea llenada con éxito.
- **commandClass**: la clase a la que pertenecerá el Objeto comando u objeto base sobre el cual se va a trabajar.
- **commandName**: nombre lógico que se le quiere dar al objeto comando, con el cual se le reconocerá en la vista.

- **validator**: este será el *bean* de referencia de la clase que extiende a la interfaz org.springframework.validation.Validator que se encarga después de que se hizo la vinculación (*binding*) de validar que todos los campos cumplan con los requisitos que le desarrollador desee y regresará los errores y las violaciones, y en que campo fue donde se cometieron, así como el mensaje de error que se debe desplegar.

  Este es un ejemplo de un *bean* de configuración de un `SimpleFormController`:

```
<bean id="registerController" class="web.RegisterController">
<property name="formView">
<value>newStudentForm</value>
</property>
<property name="successView">
<value>studentWelcome</value>
</property>
<property name="commandClass">
<value>business.User</value>
</property>
<property name="commandName">
<value>user</value>
</property>
<property name="validator">
<ref bean="registerValidator"/>
</property>
</bean>
<bean id="registerValidator" class="validators.RegisterValidator"/>
```

Una vez realizada esta configuración se debe crear una clase para el controlador, la cual debe extender a la clase `SimpleFormController`, en la cual se pueden sobrescribir varios métodos como por ejemplo:

- `Map referenceData(HttpServletRequest req, Object command, BindException errors)`: en este método se realiza el llenado de los campos que el desarrollador desee que aparezcan llenos una vez que se cargue el formulario. Estos campos vendrán llenos con los datos que se encuentren en el Map que será regresado. Este se mezclará con el modelo para que la vista los procese y se desplieguen en los campos del formulario necesarios.

- `ModelAndView onSubmit(HttpServletRequest req, HttpServletResponse res, Object command, BindException errors)`: Este método puede tener variaciones en cuanto al número de parámetros que recibe, ya que existen formas más sencillas de este método. Estos métodos son llamados después de que ocurrió la validación y en el objeto `BindException` no hubo ningún error, se ejecuta esté método, en el cual se realiza la acción que se desee una vez que el usuario haya enviado el formulario, ya sea almacenar la información en una base de datos, etc.

- `void doSubmitAction(HttpServletRequest req)`: este método es la forma más simple de los métodos para enviar los formularios ya que no se necesita crear un objeto ModelAndView, el método crea uno nuevo con la instancia successView que se haya dado en la configuración del controlador en el applicationContext. Y se pueden llevar a cabo las mismas operaciones que en cualquier método de envío, por ejemplo almacenar en una base de datos la información del formulario.

Ahora solamente queda utilizar los *tags* dentro del JSP para poder vincular cada uno de los campos a un atributo del Objeto comando que se haya escogido. Para esto se utiliza el *tag* `<spring:bind>` que únicamente tiene el atributo `path`, que es donde se asigna a que

atributo del objeto comando se va a referir ese campo del formulario. Este *tag* trae dentro también los errores de validación que se llegarán a dar al momento de enviar el formulario. Esto se da gracias a un objeto status de tipo `Bind-Status` que a través de 3 parámetros ayudará a vincular [Walls, 2005]:

- expression: La expresión usada para recuperar el nombre del campo, si por ejemplo la propiedad es Movie.name, el valor de la expresión y el nombre del campo será name.
- value: El valor del campo asociado, una vez que fue enviada la forma se utiliza para volver a poner el valor de este campo si es que hubo errores y que no se pierda toda la información que el usuario proporciono.
- errorMessages: Un arreglo de Strings que contiene los errores relacionados con este campo.

Ejemplo del código fuente de un JSP utilizando los *tags* de vinculación `<spring:bind>`.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="spring" uri="/spring" %>
…
<form method="POST" action="/registerUser.htm">
<spring:bind path="user.firstName">
Nombre:
<input type="text"
name="<c:out value="${status.expression}"/>"
value="<c:out value="${status.value}"/>">
<p>Errores:<c:out value=”${status.errorMessages}”/></p>
</spring:bind>
…
</form>
```

Una de las combinaciones más importante en cuanto a las vistas JSP es utilizar la tecnología JSTL para poder realizar ciclos, condiciones, entre otros de una manera más rápida, sencilla y eficaz para el desarrollador.