SPRING (core)

VÍCTOR CUSTODIO

Introducción a SPRING

Orígenes

Surgió a finales de 2002, comienzos de 2003.

Dentro de todos los frameworks para desarrollar aplicaciones J2EE, es el más empleado.

¿Por qué Spring?

- Spring se centra en proporcionar mecanismos de gestión de los objetos de negocio.
- Esta estructurado en capas, puede introducirse en proyectos de forma gradual, usando las capas que nos interesen, permaneciendo toda la arquitectura consistente.
- Spring es un framework idóneo para proyectos creados desde cero y orientados a pruebas unitarias

Módulos de Spring Framework

Estructura del Framework

Core: Proporciona el contenedor de Inversión de Control y la Inyección de dependencias. Su principal concepto es la BeanFactory.

Context: Proporciona métodos de acceso a los objetos del modo tradicional en otros frameworks (EJB). Permite internacionalización, propagación de eventos, recarga de recursos y la creación transparente de contextos.

DAO: Proporciona una capa de abstracción a JDBC, que elimina la necesidad de manejar código asociado a JDBC y con los errores específicos de cada base de datos. Permite implementar gestión transaccional programática y declarativa, para todos los POJOS.

Estructura del Framework

ORM: Proporciona una capa de integración con frameworks de mapeo objeto-relación, como Hibernate, JPA, JDO, …

AOP: Proporciona una implementación de programación orientada a aspectos, compatible con la Alianza AOP, para definir puntos de acceso, interceptores, proxies dinámicos, etc..

Web: Proporciona funcionalidades básicas de integración web, como upload de ficheros en varias partes, inicialización del contenedor IoC mediante servlet listeners y un contexto de aplicación web. Este es el paquete necesario para integrar Spring con Struts o WebWork.

Estructura del Framework

MVC: Proporciona una implementación del patrón Modelo- Vista-Controlador, para el desarrollo de aplicaciones web. Separa el código del modelo y los formularios web, permitiendo enlazar con el resto de funcionalidades de Spring.

Además, para potenciar el desarrollo de aplicaciones web, los desarrolladores de Spring han desarrollado Spring Web Flow, un diseñador de flujos de navegación de alto nivel para aplicaciones web, integrándose por debajo con frameworks de MVC, como el propio Spring MVC o Struts.

Conceptos Fundamentales de Spring

Contenedor de Inversión de Control

- **Concepto**: **delegar la responsabilidad al framework de que las cosas ocurran, no a la aplicación.**

Spring emplea el contenedor de IoC en TODA su arquitectura.

- Está orientado a trabajar con JavaBeans, y permite configurar esos beans (POJOS) de nuestra lógica de negocio, como se instanciarán, que propiedades tendrán, así como las relaciones entre ellos: habilita pooling de instancias, hot swapping, etc...
- SPRING es una factoría, que nos devolverá instancias por nombre, gestionando sus propiedades y relaciones

Conceptos Fundamentales de Spring

Inyección de Dependencias

El contenedor de IoC emplea la inyección de dependencias para eliminar el acoplamiento de nuestros objetos de negocio con las APIs del framework.

Mediante la configuración, el contenedor es capaz de instanciar las dependencias que tiene un objeto definidas, para devolverlo ante una petición totalmente configurado, en tiempo de ejecución.

Conceptos Fundamentales de Spring

Inyección de Dependencias

Un ejemplo:

```
public class Persona{
private String nombre;
private Direccion direccion;
//Getters y setters
}
public class Direccion{
private String cp;
//Getters y setters
}
```

Conceptos Fundamentales de Spring

Inyección de Dependencias

Un ejemplo:

Ahora intentamos crear una persona. Necesitamos previamente crear una dirección. Por tanto, la persona y la dirección mantienen una dependencia.

```
public static void main(String [] args){
Direccion dir = new Direccion();
dir.setCp("28905");
Persona persona = new Persona();
persona.setNombre("Luis");
persona.setDireccion(dir); //Inyección de dependencias por setter.
}
```

Conceptos Fundamentales de Spring (RESUMEN)

En definitiva Spring es un contenedor ligero de POJOS que se encarga de la creación de beans (**Factoría de Beans**) mediante la **Inversión de control** y la **inyección de dependencias**.

Además proporciona:

- Soporte para Aspectos e integración con AspectJ
- Manejo de Transaccionalidad
- Clases que recubren el uso de JDBC, Hibernate, iBatis, JPA, etc.
- Manejo de Webservices, JMX, JMS, JavaMail
- Capa Web
  - Implementación del MVC
  - Flujos Web ->WebflowAPI

SPRING CORE

APPLICATION CONTEXT

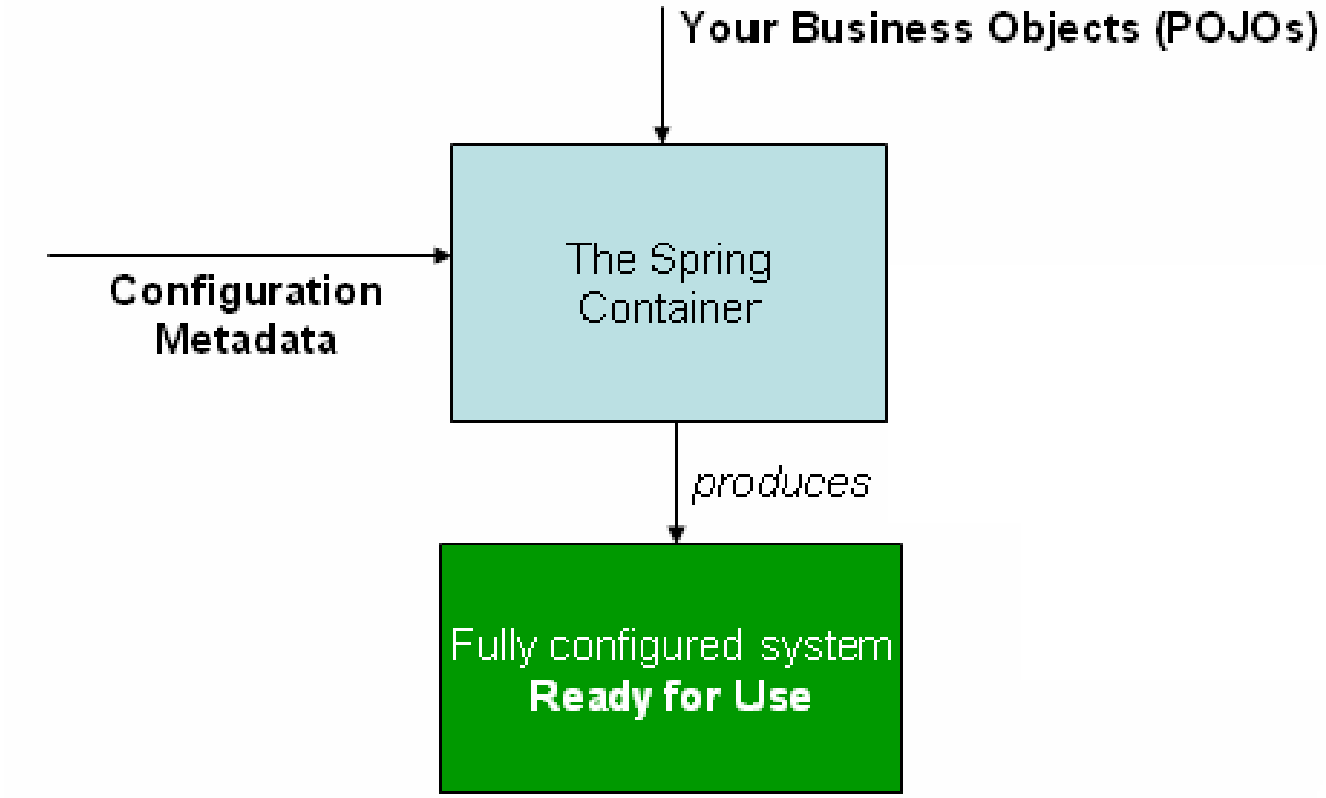
ApplicationContext

Se encuentra en el paquete org.springframework.context, añade la interfaz ApplicationContext.

Proporciona:

1. **Factoría de beans, previamente configurados. Realizando la inyección de dependencias.**
2. **MessageSource, para internacionalización con i18n**.
3. Acceso a recursos, como URLs y ficheros.
4. Propagación de eventos, para las beans que implementen ApplicationListener.
5. Carga de múltiples contextos en jerarquía, permitiendo enfocar cada uno en cada capa.

ApplicationContext



ApplicationContext

En el ApplicationContext se encuentran los siguientes métodos disponibles más importantes:

Método	Descripción
Boolean containsBean(String beanName)	Comprobar si existe definida una bean.
Object getBean(String name)	Recuperar una instancia de una bean
Object getBean(String name, Class beanClass)	Recuperar una instancia, con el casting hecho ya a beanClass.
Class getType(String name)	Retorna el Class de la bean.
Boolean isSingleton(String beanName)	Devuelve true si esa bean es un singleton
String[] getAliases(String beanName)	devuelve los alias de la bean con ese nombre

Implementaciones ApplicationContext

ApplicationContext es una interfaz. Se proporcionan diferentes implementaciones que permiten cargar el archivo de configuración de varios formas. Las mas importantes son las siguientes:

- **ClassPathXmlApplicationContext**
  - Carga el archivo de configuración desde un archivo XML que se encuentra en el classpath
- **FileSystemXmlApplicationContext**
  - Carga el archivo de configuración desde un archivo en el sistema de ficheros
- **XmlWebApplicationContext**
  - Carga el archivo de configuración desde un XML contenido dentro de una aplicación web

Implementaciones ApplicationContext

Si no estamos en una aplicación web, se puede usar lo siguiente:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("appContext.xml");
```

que buscará el recurso desde el classpath, o

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("appContext.xml");
```

que buscará el recurso desde el sistema de ficheros, en este caso relativo al directorio actual en el que nos encontremos.

Implementaciones ApplicationContext

En un entorno Web, podremos instanciar el ApplicationContext mediante un listener para Contenedores Web que soporten Servlet 2.4, o mediante un servlet en un filtro, para versiones inferiores.

El más empleado es el listener, ContextLoaderListener, que se configura del siguiente modo:

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/daoContext.xml
                /WEB-INF/applicationContext.xml </param-value>
</context-param>
<listener><listener-class> org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Fichero de Configuración

Dentro de la etiqueta <beans> definiremos una instancia de una clase con <bean>, que tienen los siguientes atributos:

Id: identifcador único de la bean en el contenedor.

Name: para nombrar una bean de varias maneras.

Alias: renombra una definición existente.

Class: la clase que implementará la bean.

Ejemplo HolaMundoSpring

Ejemplo

Fichero de Configuración

Existen 3 formas de instanciación:

- Instanciación directa
- Con factoría estática: (createInstance debe ser un método static)
  - <bean id="exampleBean“ class="examples.ExampleBean2“ factory-method="createInstance"/>
- Con factoría instanciada:
  - <bean id="myFactoryBean" class="..."/>
  - <bean id=example1” factory- bean=”myFactoryBean” factory-method=”createInstance”/>

Dependencias

- Una bean necesitará de otras beans para desempeñar sus funciones --> De ahí surge el concepto de **Dependencia**.
- Las dependencias pueden establecerse por setter o como argumento del constructor. Se recomienda el uso de setters.
- Una dependencia en una clase de negocio, tendría la variable privada de ese tipo y un setter que lo establece.

Las dependencias pueden ser de 2 tipos:

- Dependencia de valor: Se especifica el valor

- Dependencia de otra bean: Se especifica el id de la bean

Dependencias de valor

Existen los siguientes tipos de inyección de dependencias:

- Inyección por valor
  - Se inyecta un valor directamente.
    Ej: persona.setNombre(“pepe”);
- Inyección de una Bean:
  - Inyección por Setter

```
<bean id="plantilla" class="beans.Persona" >
<property name="edad" value="99" />
<property name="altura" value="1.75" /></bean>
```

  - Inyección por Constructor

```
<bean id="pConstructor" class="beans.Persona">
<constructor-arg index="0" value="Pepe"/>
<constructor-arg index="1" value="2"/>
<constructor-arg index="2" value="7.3" /></bean>
```

Dependencias

- Ej. de dependencia de valor:

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

- Ej. de dependencia de otra bean:

```
<bean id="theTargetBean” class="..."/>
<bean id=”theClientBean” class=”...”>
<property name="targetName“ ref=“theTargetBean”>
</bean>
```

Dependencias de colecciones

Se pueden indicar usando las etiquetas dentro de las etiquetas <property><property>:

<list/>: reflejado en un tipo List.

```
<list value-type= tipoValor>
    <value>a list element followed by a reference</value>     ó
    <ref bean="myDataSource" />

</list>
```

<set/>: reflejado en un tipo Set.

```
<set value-type= tipoValor>
    <value>just some string</value>    ó
    <ref bean="myDataSource" />

</set>
```

<map/>: reflejado en un tipo Map.

```
<map>
    <entry key=“JUAN” value=“Un valor”/>              ó
    <entry key=“PEPE” value-ref=“miPersona” />
    </map>
```

Dependencias: valores nulos

Inyección de Valores nulos

En una dependencia, para indicar un valor null, debemos especificarlo con la etiqueta **<null/>**

Dependencias con otras beans

Cuando no existe dependencia, pero si se necesita que una bean este instanciada antes que otra, deberemos usar el atributo **depends-on="nombreDeLaBean"**

Dependencias: Herencia

Podemos configurar beans base y utilizarlos a modo de herencia.

La etiqueta <bean> provee 2 atributos para realizarlo:

parent: Indica el id de otra bean que se utilizará como padre. Concepto similar al extends en las clases Java.

abstract: Si es “true” indica que la bean declarada es abstracta, es decir, no podrá ser nunca instanciada.

- **Ejemplos:**
  <bean id=“personaGenerica” class=“beans.Persona” abstract=“true”>
  <property name=“nombre”><null /></property>
  <property name=“cp” value=“28001” />
  </bean>
- **Beans que heredan del anterior:**
  <bean id=“julio” parent=“personaGenerica”>
  <property name=“nombre” value=“Julio” />
  </bean>
- **La bean sobreescribe el valor de la propiedad “nombre”**

Autowiring

Consiste en la inyección de dependencias automática sin necesidad de indicarla en la configuración de una bean.

Spring proporciona 4 tipos de autowiring:

- Por nombre (byName)
- Por tipo (byType)
- Por constructor (constructor)
- Autodetectado (autodetect)

Autowiring : Por Nombre

El contenedor busca un bean cuyo nombre (ID) sea el mismo que el nombre de la propiedad.

```
<bean id="persona" class="beans.Persona" autowire="byName">
    <property name="nombre" value=“Paco"/>
</bean>
<bean id="direccion" class="beans.Direccion">
<property name="cp" value="28900"/>
</bean>
```

Si no se encuentra coincidencia la propiedad se devolverá sin dependencia.

Autowiring : Por Tipo

El contenedor busca un único bean cuyo tipo coincida con el tipo de la propiedad a inyectar.

```
<bean id="dir" class="beans.Direccion">
<property name="cp" value="28900"/>
</bean>
<bean id="persona" class="beans.Persona" autowire="byType">
<property name="nombre" value="persona"/>
</bean>
```

Si no se encuentra coincidencia la propiedad se devolverá sin dependencia.

Si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo

org.springframework.beans.factory.UnsatisfiedDependencyException

Autowiring : Por Constructor

El contenedor busca algún bean que contenga un constructor con el tipo del bean que se quiere inyectar

```
<bean id="persona3" class="beans.Persona" autowire="constructor">
<property name="nombre" value="persona"/>
</bean>
```

- La Persona debe tener un constructor del tipo:

```
public Persona(Direccion dir){
this.direccion= dir;
}
```

Si no se encuentra coincidencia la propiedad se devolverá sin dependencia.

Si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo

org.springframework.beans.factory.UnsatisfiedDependencyException

Autowiring : Resumen

Autowiring parece una poderosa herramienta de configuración pero impone algunas restricciones.

- Por tipo y nombre obliga a tener un único bean definido que satisfaga el wiring.
- Por nombre tiene problemas de refactorización cambio nombre propiedad en bean.
- Existen propiedades de la bean que en el archivo de configuración no aparecen inicializadas que contendrán valores al aplicar autowiring.
- **En resumen, el autowiring es una poderosa característica que debe usarse con mucha precaución**

Tipos de ámbito de una bean (1)

En Spring existen cinco ámbitos de instanciación:

Scope	Descripcion
Singleton	Solo una instancia de la ben por contenedor
Prototype	Una instancia nueva por petición
Request	Una instancia por petición
Session	Una instancia por sesión http
GlobalSession	Una instancia por sesión http (solo para Portlets)

Para indicar el ámbito, se emplea el atributo scope=“”

Nota: Si una bean de tipo Singleton, tiene una dependencia de otra bean de ámbito session o request, deberá indicarse que esa dependencia se creará a través de un proxy, con la etiqueta <aop:scoped-proxy/>

Tipos de ámbito de una bean: Lazy

Por defecto todos las beans sin scope definido son de tipo singleton.

Spring instancia las beans singleton una vez que el archivo de configuración se carga. Es una buena práctica aunque puede causar problemas con los proxies de EJB, ya que, puede intentar instanciar el proxy antes que la interfaz “home” sea vinculada.

Mediante el atributo **lazy-init=“true”** podemos indicarle al contenedor de Spring que no cargue la bean hasta que se pida por primera vez.

Ciclo de vida de una Bean

Es posible controlar el ciclo de vida de una bean configurada en el contenedor IoC, mediante unos métodos de inicialización y de destrucción de la bean.

Estos métodos **deben existir en la bean**, y puede indicarse al contenedor que los invoque en los momentos indicados, con los atributos init-method=“” y destroy-method=“”.

Se puede definir en la etiqueta <beans/> con default-init- method=“” y default-destroy-method=“” los métodos que se intentará invocar en todas las beans configuradas, si existen dichos métodos.

Cuando no se esté en un contexto Web, la destrucción se deberá registar, invocando el método registerShutdownHook() en AbstractBeanFactory.

Ejemplo:initydestroy.docx

Ciclo de vida de una Bean

Existen algunas interfaces que nos permiten manejar el ciclo de vida de una bean de la misma forma que con init-method y destroy-method:

InizializingBean

Obliga a la clase que la implemente a implementar el método afterPropertiesSet() que será llamado después de que todas las propiedades de la bean hayan sido configuradas.

DisposableBean

Obliga a implementar el método destroy que será llamado justo antes de destruir la bean por el contenedor.

Ejemplo: InitializingBeanand DisposableBean.docx

Internacionalización

ApplicationContext extiende una interfaz llamada MessageSource, que proporciona los métodos:

- **String getMessage(String code, Object[] args, String default, Locale loc):**

Método básico para recuperar un mensaje del MessageSource.

Si no se encuentra un mensaje, se usa el default.

-**String getMessage(String code, Object[] args, Locale loc)**: similar pero sin mensaje por defecto.

-**String getMessage(MessageSourceResolvable resolvable, Locale locale):**

En este caso, MessageSourceResolvable, agrupa los argumentos de los métodos anteriores.

Internacionalización

Cuando un ApplicationContext se carga, automáticamente busca la bean **MessageSource** definida en la configuración.

Esta bean debe llamarse **messageSource**. Si se encuentra, todas las llamadas a los métodos anteriores se delegan sobre ella. Si no lo encuentra, se instancia un StaticMessageSource vacío, para que se encargue de las llamadas.

Spring proporciona dos implementaciones de MessageSource,

ResourceBundleMessageSource y **StaticMessageSource.**

El más usado es el primero, que permite definir las ubicaciones de los ficheros .properties que se van a utilizar.

Ejemplo:

Internacionalización

Ejemplo de configuración de ResourceBundleMessageSource

```
<beans>
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
<property name="basenames">
<list>
<value>format</value>
<value>exceptions</value>
<value>windows</value>
</list>
</property>
</bean>
</beans>
```

En este caso, deberemos tener tres ficheros en el raíz del classpath, con los nombres format.properties, exceptios.properties, windows.properties. Admitiendo el resto de ficheros con los sufijos de locale para internacionalización

Internacionalización

Ejemplo de uso:

```
Locale locale = …; //Locale correspondiente
String text = ctx.getMessage(“usuario.id”, new Object[0], locale);
```

En una JSP, mediante la librería de etiquetas proporcionada por Spring:

```
<spring:message code=“usuario.id” />
```

Ejemplo:Internacionalizacion.docx