Spring
Framework Core

Rubén Gómez García

Version 1.0.0 2019-03-22

Contenidos

Capítulo 1. Introduccion

El contexto de Spring describe una Factoria de Beans y la relación entre dichos Beans.

Se define con un objeto de tipo **ApplicationContext**.

Proporciona:

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

Capítulo 2. ApplicationContext

ApplicationContext es la interfaz que modela el contexto de Spring, es el contenedor de todos los Beans que gestina Spring, de la cual se proporcionan diferentes implementaciones:

- **ClassPathXmlApplicationContext** → Carga el archivo de configuración desde un archivo XML que se encuentra en el classpath

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

- **FileSystemXmlApplicationContext** → Carga el archivo de configuración desde un archivo en el sistema de ficheros

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

- **AnnotationConfigApplicationContext** → Carga de archivos de configuracion anotados con **@Configuration**

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Configuracion.class);
```

Para el caso de JavaConfig, se pueden registrar nuevos ficheros de configuración en caliente con

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.register(Configuracion.class);
context.refresh();
```

- **XmlWebApplicationContext** → Carga el archivo de configuración desde un XML contenido dentro de una aplicación web

```
<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>
```

Como se ve, el Contexto de Spring hace referencia a uno o varios ficheros de configuración, donde

se definen los Beans que formaran dicho contexto, en general habra dos formas de definir estos ficheros de contexto

- Por XML: Definiendo un fichero XML con los Beans del contexto de Spring.

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

    <!-- Inyección de dependencias por setter -->
    <bean id="plantilla" class="beans.Persona" abstract="true">
        <property name="edad" value="99" />
        <property name="altura" value="1.75" />
    </bean>

    <!-- Inyección de dependencias 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>
</beans>
```

- Por JavaConfig: Definiendo una o varias clases anotadas con **@Configuration**.

```
@Configuration
public class Configuracion {}
```

Capítulo 3. Declaracion de Beans

Dado que será el contexto de Spring quien instancie los Beans, hay que indicarle como lo ha de hacerlo, debido a las distintas naturalezas de las clases, esta construcción se puede realizar de diferentes modos, estando en Spring soportados los siguientes:

- Creación por constructor basico (sin parametros):

```
<bean id="bean1" class="ejemplos.Bean"/>
```

- Creación por contructor con parametros:

```
<bean id="lucia" class="javabeans.Persona">
    <constructor-arg name="nombre" value="Otralucia" />
    <constructor-arg name="edad" value="31" />
    <constructor-arg name="pareja" ref="fernando" />
</bean>
```

- Inyección de propiedades posterior a la contrucción, pero efectiva desde el primer momento que el Bean esta disponible:

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja" ref="lucia"/>
</bean>
```

- **Factoría estática**: Clase con un método estatico, en el ejemplo **createInstance** que retorna un objeto de si mismo.

```
<bean id="bean1" class="ejemplos.Bean" factory-method="createInstance"/>
```

- **Factoría instanciada**: Clase con un método que retorna un objeto de otro tipo.

```
<bean id="myFactoryBean" class="ejemplos.FactoriaDeBeans"/>
<bean id="bean2" factory-bean="myFactoryBean" factory-method="createInstance" class=
"ejemplos.Bean"/>
```

En el caso de necesitar pasar parametros a los métodos de factoria, estos se pasarán con la etiqueta <**constructor-arg**>

Para la definición con JavaConfig, el proceso se simplifica enormemente dado que unicamente hay que codificar las sentencias necesarias en java dentro de métodos anotados con @Bean en la clase de Configuracion, que retornen el Bean que se desea incluir en el contexto.

```
@Bean
public Persona juan(){
    return new Persona("Juan");
}
```

Para la definición con anotaciones, se emplearán alguna de las siguientes

- **@Component** → Beans generales
- **@Repository** → Beans de la capa de persistencia
- **@Controller** → Beans de la capa de control
- **@Service** → Beans de la capa de servicio
- **@Named** (JSR-330)

Para que estas anotaciones se puedan interpretar, habra que activarlas en el contexto, para ello:

- Si el contexto se define con xml

```
<context:component-scan base-package="com.curso.spring"/>
```

- Si el contexto se define con JavaConfig, en alguna de las clases de configuracion se debe añadir la anotación @ConponentScan

```
@Configuration
@ComponentScan(basePackages={ "controllers" })
```

3.1. FactoryBean

Spring ofrece una interface **FactoryBean**, que permite definir factorias de Beans delegadas, es decir en vez de ser Spring el que instancia con la configuración, es la Factoria, instanciandose esta por Spring.

```
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<T> getObjectType();
    boolean isSingleton();
}
```

Estas factorias unicamente han de proporcionar el objeto a construir, su tipo para que Spring sea capaz de discriminar a que factoria ha de pedir el objeto y si el objeto es unico, lo que permite a Spring cachearlo cuando se construye y no volver a emplear la factoria, este último escenario no es habitual.

Los APIs de Spring lo emplean ofreciendo algunas clases que implementan dicha interface como

- JndiFactoryBean
- LocalSessionFactoryBean
- LocalContainerEntityManagerFactoryBean

La particularidad de estas Factorias delegadas, es que al definir el API de Spring su forma, se puede emplear el identificador de la Factoria al hacer referencia a los Bean que crea, esto significa que definiendo un Bean de Spring de tipo Factoria de coches, con id **car**

```
<bean class = "a.b.c.MyCarFactoryBean" id = "car">
    <property name = "make" value ="Honda"/>
    <property name = "year" value ="1984"/>
</bean>
```

Se puede referenciar a los coches que fabrica con el id **car**

```
<bean class = "a.b.c.Person" id = "josh">
    <property name = "car" ref = "car"/>
</bean>
```

Capítulo 4. Ambitos (Scope)

El ambito de un Bean, representa el tiempo en ejecución en el cual el Bean esta disponible.

Se definen cuatro tipos de ambitos para una Bean

- **Singleton**: Una instancia única por JVM (por defecto), es decir el Bean esta diponible siempre.

```
<bean id="bean1" class="ejemplos.Bean"/>
```

- **Prototype**: Una nueva instancia cada vez que se pida el Bean. No tiene una vigencia establecida, depende de lo que la aplicación haga con el, pero para el contexto, una vez que lo contruye y lo proporciona, deja de existir.

```
<bean id="bean1" class="ejemplos.Bean" scope="prototype"/>
```

- **Request**: Ambito disponible en aplicaciones Web, permite asociar la vida de un Bean a una peticion que recibe el servidor, mientras que se pida al contexto el Bean dentro de la misma petición este siempre retornará el mismo Bean, al cambiar de petición, se dará otro.
- **Session**: Ambito disponible en aplicaciones Web, permite asociar la vida de un Bean a una sesión creada en el servidor para un usuario particular, mientras que se pida al contexto el Bean dentro de una petición asociada a una sesión establecida, este siempre retornará el mismo Bean, al cambiar de sesion, se dará otro.

Las beans singleton se instancian una vez que el contexto se carga. Esto se puede cambiar con la propiedad lazy-init="true" que indica al contexto que no cargue la bean hasta que se pida por primera vez.

Capítulo 5. Ciclo de Vida

El ciclo de vida de una bean en el contexto de Spring, permite conocer el momento de la creación y de la destrucción de la bean.

Se pueden definir métodos que se ejecuten en esos dos momentos empleando las propiedades **init-method** y **destroy-method**.

Si se mantiene una homegeneidad en los nombres de los métodos para todas las Bean, se puede definir en la etiqueta <beans/> los nombres de los métodos para facilitar la configuración con las propiedades **default-init-method** y **default-destroy-method**.

Existen algunas interfaces que nos permiten manejar el ciclo de vida sin necesidad de definir las propiedades 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 el bean por el contenedor.

Capítulo 6. Extension del contexto

Se puede definir el contexto en multiples ficheros puediendose estos referenciar desde el principal.

Con XML

```
<import resource="cicloDeVida.xml"/>
```

Con Javaconfig

```
@Configuration
@Import(ConfiguracionPersistencia.class)
```

Tambien se pueden mezclar las dos formas de definir el contexto, importando desde la clase de configuración un fichero de XML

```
@ImportResource("classpath:/config/seguridad.xml")
```

O bien importando desde un XML una clase de configuracion, para lo unico que hay que hacer es escanear el paquete donde este dicha clase

```
<context:component-scan base-package="com.curso.spring.configuracion" />
```

En las clases de configuración, se puede emplea la anotacion @Autowired para obtener la referencia a un Bean que este definido en otro fichero

Capítulo 7. Inyeccion de Dependencias

Las dependencias pueden ser de tipos complejos, por lo que se hará por referencia

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja" ref="lucia"/>
</bean>
```

O de tipos simples, incluido String, que se hará por valor.

```
<bean id="fernando" class="javabeans.Persona">
    <property name="nombre" value="Fernando"/>
</bean>
```

Se pueden inyectar los Beans por Setter

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja" ref="lucia"/>
</bean>
```

O por contructor

```
<bean id="lucia" class="javabeans.Persona">
    <constructor-arg name="nombre" value="Otralucia" />
    <constructor-arg name="edad" value="31" />
    <constructor-arg name="pareja" ref="fernando" />
</bean>
```

Tambien se pueden inyectar Beans exclusivos para otro Bean, a los que solo este tiene acceso, suelen ser anonimos

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja">
        <bean class="javabeans.Persona">
            <constructor-arg name="nombre" value="Otralucia" />
            <constructor-arg name="edad" value="31" />
            <constructor-arg name="pareja" ref="fernando" />
        </bean>
    </property>
</bean>
```

Se puede indicar de forma explicita que se quiere inyectar un valor NULL con la etiqueta <null/>

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja"><null/></property>
</bean>
```

Capítulo 8. Definición de Colecciones

Se dispone de etiquetas particulares para la definición de colecciones

- List

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

- Set

```
<set>
    <value>just some string</value>
    <ref bean="myDataSource" />
</set>
```

- Map

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

- Properties

```
<props>
    <prop key="adm"> administrator@somecompany.org</prop>
</props>
```

Capítulo 9. Ficheros de propiedades

Se pueden obtener literales de ficheros de propiedades a traves de expresiones EL (SpEl), para ello hay que cargar los ficheros properties como **PropertyPlaceholderConfigurer**

Desde el XML definiendo un bean

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <array>
            <value>configuracion/configuracionBD.properties</value>
            <value>configuracion/configuracionServicio.properties</value>
        </array>
    </property>
</bean>
```

O de forma mas sencilla empleando el espacio de nombres **context**

```
<context:property-placeholder location="db.properties"/>
```

O desde JavaConfig definiendo un bean

```
@Bean
public PropertyPlaceholderConfigurer propertyPlaceholderConfigurer() {

    PropertyPlaceholderConfigurer propertyPlaceholderConfigurer = new
PropertyPlaceholderConfigurer();
    propertyPlaceholderConfigurer.setLocation(new ClassPathResource("db.properties"));

    return propertyPlaceholderConfigurer;
}
```

O empleando la anotacion **PropertySource**

```
@Configuration
@PropertySource("classpath:db.properties")
public class ApplicationConfiguration {}
```

Y posteriormente se pueden refereciar las properties desde el XML

```
<bean class="com.ejemplo.DataSource">
    <property name="password" value="${service.provincias.password}"/>
    <property name="url" value="${service.provincias.url}"/>
    <property name="user" value="${service.provincias.user}"/>
</bean>
```

o desde las clases con la anotacion **@Value**, que puede afectar tanto a propiedades de una clase, como a parametros de un método.

```
@Value("${nombre}")
private String nombre;

@Bean
public DataSource dataSource(
        @Value("${dbDriver}") String driverClass,
        @Value("${dbUrl}") String jdbcUrl,
        @Value("${dbUsername}") String user,
        @Value("${dbPassword}") String password) {
}
```

Notar que el acceso a las propiedades se hace con la sintaxis ${} y no #{}, esta última esta reservada a SpEL

Otra alternativa, es emplear la clase **Environment**

```
@Autowired
private Environment environment;
```

Leyendo posteriormente las propiedades

```
environment.getProperty("dbDriver")
```

Capítulo 10. Herencia

Se pueden reutilizar configuraciones de Beans a través de la herencia, para ello se dispone de dos propiedades

- **abstract** → Si es **true** indica que la bean declarada es abstracta y por tanto no podrá ser nunca instanciada, es decir no se le puede pedir al contenedor.

```
<bean id="personaGenerica" class="beans.Persona" abstract="true">
    <property name="nombre"><null /></property>
    <property name="cp" value="28001" />
</bean>
```

- **parent** → Indica el id de otra bean que se utilizará como padre. Concepto similar al extends en las clases Java, pero aplicado a los valores de las propiedades de los Beans.

```
<bean id="julio" parent="personaGenerica">
    <property name="nombre" value="Julio" />
</bean>
```

Capítulo 11. Autoinyección (Autowiring)

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

Se proporcionan 4 tipos de autowiring:

- Por nombre (byName) → El contenedor busca un bean cuyo nombre (ID) sea el mismo que el nombre de la propiedad. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia.

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

- Por tipo (byType) → El contenedor busca un único bean cuyo tipo coincida con el tipo de la propiedad a inyectar. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia y si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo **org.springframework.beans.factory.UnsatisfiedDependencyException**

```
<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>
```

- Por constructor (constructor) → El contenedor busca en los Beans disponibles en el contexto, unos que satisfagan los requerimientos del constructor del Bean en construccion, la resolución de las dependencias del constructor se realiza por tipo. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia y si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo **org.springframework.beans.factory.UnsatisfiedDependencyException**

```
<bean id="dir" class="beans.Direccion">
    <property name="cp" value="28900"/>
</bean>
<bean id="persona3" class="beans.Persona" autowire="constructor">
    <property name="nombre" value="persona"/>
</bean>
```

Para el ejemplo la clase Persona debe tener un constructor del tipo:

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

- Autodetectado (autodetect) → Se intenta realizar el autowiring por constructor. Si no se produce intentará realizarlo por tipo. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia y si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo **org.springframework.beans.factory.UnsatisfiedDependencyException**

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

Se puede configurar la autoinyección con anotaciones, empleando la anotacion @Autowired, aplicado al constructor

```
@Component
public class Negocio {

    private Persistencia persistencia;

    @Autowired
    public Negocio(Persistencia persistencia) {
        this.persistencia = persistencia;
    }
}
```

o al método de set.

```
@Component
public class Negocio {

    @Autowired
    private Persistencia persistencia;
}
```

Si no se requiere la inyección, es decir puede valer null, la anotación tiene una propiedad booleana **required**

```
@Component
public class Negocio {

    @Autowired(required=false)
    private Persistencia persistencia;
}
```

Tambien se puede emplear @Inject del estandar JSR-330

```
@Named
public class Negocio {

    @Inject
    private Persistencia persistencia;
}
```

Para el uso de las anotaciones, se ha de configurar el contexto para que sea capaz de interpretarlas, esto se consigue de dos formas con la etiqueta <context:component-scan> o con <context:annotation-config>

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

La autoinyeccion si se define en el atributo o en el método de set, es por tipo, si se define en el constructor es por constructor y si se desea realizar una autoinyección por nombre, se dispone de @Qualifier, que asociado a la propiedad o al parametro del setter o el constructor, permite indicar el Id del bean a inyectar

```
@Component
public class Negocio {

    @Autowired
    @Qualifier("dao")
    private Persistencia persistencia;
}
```