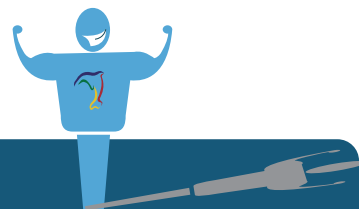


¿Qué ofrece Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

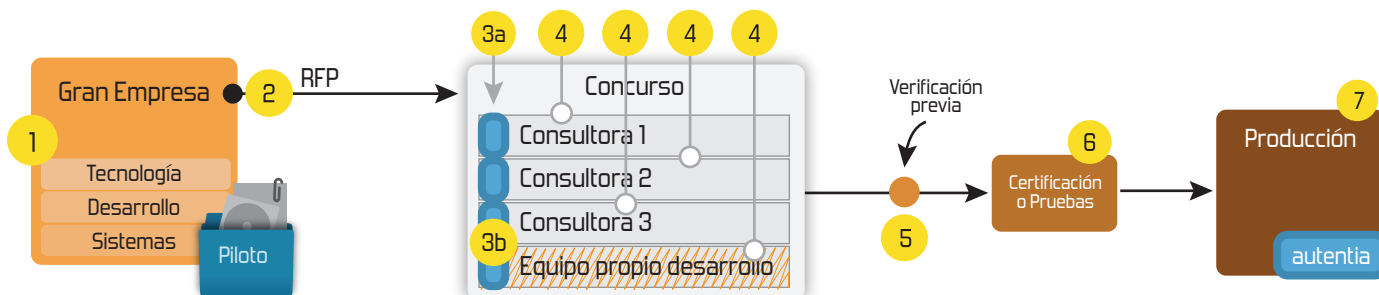
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces,
HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

Control de autenticación y
acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)



Entra en Adictos a través de 

E-mail

Contraseña

Entrar [Deseo registrarme](#)
[Olvidé mi contraseña](#)

[Inicio](#) [Quiénes somos](#) [Formación](#) [Comparador de salarios](#) [Nuestros libros](#) [Más](#)

» Estás en: [Inicio](#) [Tutoriales](#) [Spring BeanPostProcessor](#)



Juan Diego Reyes Ponce

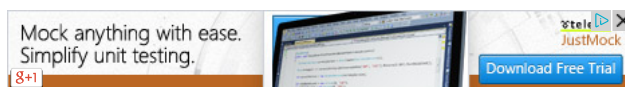
Consultor tecnológico de desarrollo de proyectos informáticos.

Ingeniero en Informática

Puedes encontrarme en [Autentia](#): Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

[Ver todos los tutoriales del autor](#)



Fecha de publicación del tutorial: 2014-01-07

Tutorial visitado 18 veces [Descargar en PDF](#)

Spring BeanPostProcessor

Índice de contenidos

1. Introducción
2. Entorno
3. Creando un BeanPostProcessor
4. Probando la solución
5. BeanFactoryPostProcessor s
6. Referencias

1. Introducción

En este tutorial se va a explicar cómo extender la funcionalidad del contenedor IoC (Inversion of Control) de Spring mediante la definición de BeanPostProcessor s. Si no sabéis qué es el contenedor IoC, podéis consultar la documentación de Spring [aquí](#).

Muchos desarrolladores no conocen el concepto de BeanPostProcessor, y sin embargo los están usando en sus aplicaciones basadas en Spring, sin tener constancia de ello. Es muy importante conocer dichos beans, no sólo para para tener más claro el ciclo de vida de los beans en Spring, sino porque ofrece múltiples ventajas. Definamos primero qué es un BeanPostProcessor:

Un BeanPostProcessor es un bean de Spring que permite extender la funcionalidad del contenedor IoC para agregar lógica de instanciación, lógica de resolución de dependencias (beans colaboradores), etc. A nivel de implementación, BeanPostProcessor no es más que una interfaz que define dos métodos:

```
/** Recibe como parámetro el objeto bean, y su nombre */  
  
public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;  
  
/** Recibe como parámetro el objeto bean, y su nombre */  
  
public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;
```

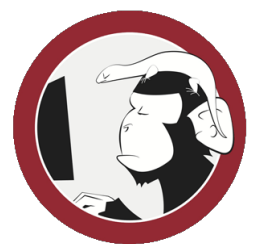
Éstos métodos *callback* serán llamados por el contenedor IoC antes (*postProcessBeforeInitialization*) y después (*postProcessAfterInitialization*) de la instanciación, inicialización y configuración de **cada uno de los beans del contenedor**. Esto ocurre para cada uno de los BeanPostProcessor s definidos en el contexto de Spring.

Se pueden configurar múltiples BeanPostProcessors y se puede controlar el orden en el que éstos son ejecutados, mediante la propiedad 'order' (ya veremos más adelante como hacer que un BeanPostProcessor personalizado sea ordenable). El contenedor de Spring cargará de forma automática los BeanPostProcessor s que encuentre en su contexto.

Es importante hacer notar que los BeanPostProcessor s actúan sobre los beans del ámbito del contenedor. Es decir que si se ha establecido una jerarquía de contextos en una aplicación, sólo actuarán sobre el contexto donde hayan sido definidos (ver [Spring context hierarchies](#) para más información).

Ventajas

Catálogo de servicios Autentia



Síguenos a través de:



Últimas Noticias

» [IX Autentia Cycling Day \(ACTUALIZADO\)](#)

» [Autentia en la carrera de las empresas](#)

» [Spring 4.0 ¿qué hay de nuevo amigo?](#)

» [Torneo de pádel solidario AMEB](#)

» [Próxima charla: Gradle como alternativa a Maven para la construcción de proyectos en Java](#)

[Histórico de noticias](#)

Últimos Tutoriales

» [Cómo montar un raid1 en una máquina corriendo debian.](#)

» [Notificaciones locales en iOS.](#)

» [Configuración básica de seguridad en servidor linux con iptables y fail2ban](#)

» [Desarrollo rápido de aplicaciones CRUD con OpenXava](#)

» [ApacheDS: tests de integración contra un servidor LDAP embebido.](#)

Los BeanPostProcessor s nos ofrecen ventajas como las siguientes:

- Nos permiten extender la funcionalidad del contenedor de IoC para añadir funcionalidad antes y/o después de la configuración de los beans del mismo.
- Sirven de validadores para aquellos casos que no queramos que el contexto se cargue sin cumplir una serie de requisitos
- Combinados con AOP (Aspect Oriented Programming), ofrecen muchas posibilidades de pre y post procesamiento

2. Entorno

Para la realización de este tutorial, se ha usado el siguiente entorno:

- Macbook pro core i7 con 16gb RAM
- SO: Mavericks
- IDE: Spring Tool Suite 3.4.0 RELEASE
- Spring 3.2.6 RELEASE
- Maven 3.0.4

3. Creando un BeanPostProcessor

En este tutorial vamos a crear un par de BeanPostProcessor s. El primero simplemente imprimirá por pantalla el nombre del bean inicializado, y el segundo comprobará si las propiedades de configuración de la aplicación están asignadas. Si no las están, se producirá una excepción que impedirá que el contexto de Spring se siga cargando.

Ambos beans implementarán la interfaz *BeanPostProcessor* y la interfaz *Ordered* para que el contenedor de Spring lo tenga en cuenta para ordenar su ejecución con respecto al resto de BeanPostProcessor s.

HelloWorldBeanPostProcessor.java

```
public class HelloWorldBeanPostProcessor implements BeanPostProcessor, Ordered {

    /**
     * Método que se ejecuta antes de inicializar el bean
     */
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {

        System.out.println(String.format("Initializing bean %s", beanName));
        return bean;
    }

    /**
     * Método que se ejecuta después de inicializar el bean
     */
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {

        System.out.println(String.format("Bean %s initialized successfully", beanName));
        return bean;
    }

    /**
     * Obtiene la ordenación del beanpostprocessor
     */
    public int getOrder() {
        return Integer.MAX_VALUE;
    }
}
```

AppCfgBeanPostProcessor.java

```
public class AppCfgBeanPostProcessor implements BeanPostProcessor, Ordered {

    /**
     * Método que se ejecuta antes de inicializar el bean
     */
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {

        return bean;
    }

    /**
     * Método que se ejecuta después de inicializar el bean.
     * Comprueba que todas las propiedades de la aplicación han sido rellenadas
     */
    @SuppressWarnings("serial")
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {

        boolean allPropsConfigured = true;

        try {

            if(bean instanceof AppCfg) {

                AppCfg appCfg = (AppCfg)bean;

                Method[] methods = appCfg.getClass().getMethods();

                for(int i=0; i<methods.length && allPropsConfigured; i++) {
```

Últimas ofertas de empleo

2011-09-08
 Comercial - Ventas - MADRID.

2011-09-03
 Comercial - Ventas - VALENCIA.

2011-08-19
 Comercial - Compras - ALICANTE.

2011-07-12
 Otras Sin catalogar - MADRID.

2011-07-06
 Otras Sin catalogar - LUGO.

```

        Method method = methods[i];

        if(method.getName().startsWith("get")) {
            Object ret = method.invoke(appCfg, new Object[0]);

            //Se comprueba que la propiedad no sea nula ni vacía (en caso de ser String)
            if(ret == null ||
                (ret instanceof String && StringUtils.isEmpty((String)ret))) {
                allPropsConfigured = false;
            }
        }
    }

    if(!allPropsConfigured) {
        throw new BeansException("Not all props configured") {};
    }

    } catch(InvocationTargetException e) {
        throw new BeansException("AppCfg class bad configured") {};
    } catch(IllegalAccessException e) {
        throw new BeansException("AppCfg class bad configured") {};
    }

    }

    return bean;
}

/**
 * Obtiene la ordenación del beanpostprocessor
 */
public int getOrder() {
    return Integer.MIN_VALUE;
}
}

```

Como podemos observar, ambas clases implementan la interfaz *BeanPostProcessor*, que la registrará automáticamente el contenedor de Spring como un *BeanPostProcessor*, y *Ordered*, que indicará que el *BeanPostProcessor* es ordenable dentro del contexto de Spring. Si no implementara dicha interfaz, se metería dentro del 'cajón' de beans no ordenables y se ejecutaría en el orden definido en el XML. Hay que tener también en cuenta que si se usa una configuración de Spring basada en Java (Spring 3.0+), la ordenación de los *BeanPostProcessors* se hará por el orden en el que fueron registrados en el contexto.

La clase que va a almacenar los datos de la configuración de la aplicación es la que sigue:

AppCfg.java

```

public class AppCfg {

    private String cfgProperty1;
    private String cfgProperty2;

    public String getCfgProperty1() {
        return cfgProperty1;
    }
    public void setCfgProperty1(String cfgProperty1) {
        this.cfgProperty1 = cfgProperty1;
    }
    public String getCfgProperty2() {
        return cfgProperty2;
    }
    public void setCfgProperty2(String cfgProperty2) {
        this.cfgProperty2 = cfgProperty2;
    }

}

```

Como vemos, es simplemente una clase POJO con dos propiedades de ejemplo. Éstas propiedades las va a inyectar Spring mediante su definición en el archivo de contexto. He optado por usar la solución XML para la configuración de Spring, aunque se puede usar la configuración por Java (Spring 3.0+) y anotaciones *@Autowired* o *@Reference* (JSR-250). En nuestro caso, éstas propiedades serán inyectadas desde un archivo de propiedades que se encuentra en el classpath.

A continuación se muestra el archivo de configuración de contexto 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"
    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-3.2.xsd">

    <context:property-placeholder location="classpath:/config.properties"/>

    <!-- Bean de configuración de la aplicación -->
    <bean name="AppCfg" class="com.autentia.tutoriales.beanpostprocessors.bean.AppCfg">
        <property name="cfgProperty1" value="{app.property1}"/>
        <property name="cfgProperty2" value="{app.property2}"/>
    </bean>

    <!-- BeanPostProcessors -->

```

```

    <bean class="com.autentia.tutoriales.beanpostprocessors.bpp.HelloWorldBeanPostProcessor"/>
    <bean class="com.autentia.tutoriales.beanpostprocessors.bpp.AppCfgBeanPostProcessor"/>

</beans>

```

Aquí se está usando el componente *property-placeholder* de spring-context para cargar el archivo de propiedades y asignarlo en las propiedades del bean AppCfg. Este componente tiene un pequeño secreto, tal y como explicaré más adelante

El fichero de propiedades:

```

app.property1 = property1
app.property2 = property2

```

Por último, se muestra el fichero pom.xml de Maven.

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.autentia.tutoriales</groupId>
    <artifactId>beanpostprocessors</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <spring.version>3.2.6.RELEASE</spring.version>
        <junit.version>4.11</junit.version>
    </properties>

    <dependencies>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>${junit.version}</version>
        </dependency>

    </dependencies>
</project>

```

4. Probando la solución

Siempre tenemos que realizar pruebas unitarias de nuestro desarrollo, por eso es buena práctica hacer TDD, y aquí no iba a ser menos:

BppTest.java

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:/applicationContext-test.xml"})
public class BppTest {

    @Autowired
    ApplicationContext applicationContext;

    @Test
    public void testBpp() {

        Object bean = applicationContext.getBean("AppCfg");
        AppCfg appCfg = (AppCfg)bean;

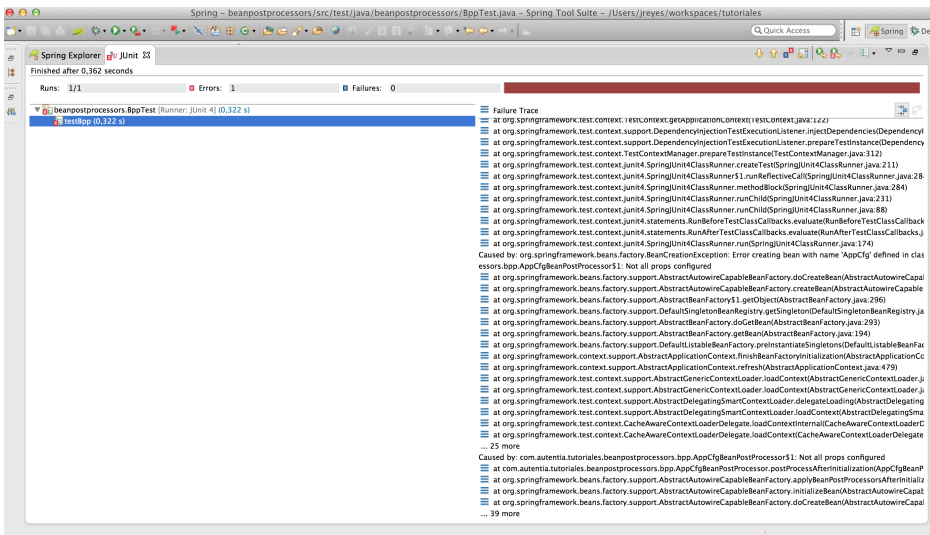
        System.out.println(String.format("Property #1: %s", appCfg.getCfgProperty1()));
        System.out.println(String.format("Property #2: %s", appCfg.getCfgProperty1()));
    }
}

```

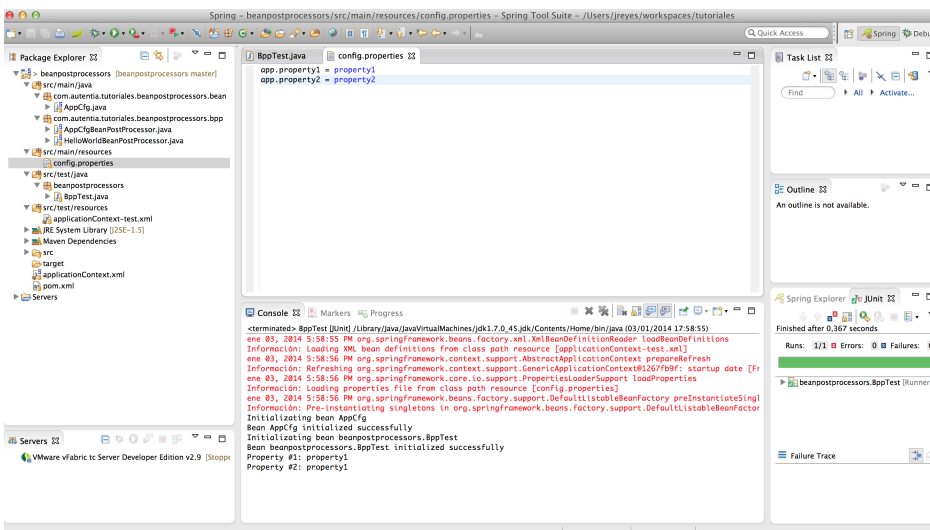
El test es muy sencillo. Primero carga el contexto de Spring (junto con nuestros nuevos BeanPostProcessors). Si no salta ninguna excepción, continúa la ejecución sacando por pantalla las dos propiedades de la aplicación de ejemplo. Es importante fijarse que este test no comprueba funcionalidad más allá del contexto de Spring; toda la funcionalidad la maneja Spring dentro del contenedor de IoC, de forma que si ocurre alguna excepción dentro de nuestros BeanPostProcessors, la ejecución se detendrá y nuestro test fallará.

El contexto de aplicación de Spring usado para el test: *applicationContext-test.xml* es una copia del que he mostrado más arriba.

Ejecutamos el test con sólo la propiedad *app.property1* asignada en el fichero de propiedades:



Lo volvemos a ejecutar con las dos propiedades asignadas:



En el proceso de depuración del test podemos ver que no sólo se ejecutan los métodos de nuestros dos nuevos BeanPostProcessors, sino que además Spring define una serie de BeanPostProcessors que se ejecutan para cualquier contexto: *ApplicationContextAwareProcessor*, *BeanPostProcessorChecker*, etc. No está en el ámbito de este tutorial explicar estos BeanPostProcessor s, aunque sí se pueden consultar en la documentación de Spring.

Podéis descargar el proyecto [aquí](#) 

5. BeanFactoryPostProcessor

Existe en Spring un tipo de BeanPostProcessor especial llamado BeanFactoryPostProcessor. La semántica es la misma, pero la diferencia está en que este bean actúa sobre los metadatos del contexto de Spring, es decir, sobre el mismo archivo de configuración.

La interfaz tiene el siguiente método:

```
void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;
```

Recibe como parámetro el BeanFactory usado en el contexto de la aplicación.

Viendo el fichero de configuración de Spring de más arriba; ¿Recordáis el elemento *property-placeholder*? Ahora bien, la funcionalidad de este elemento es obtener todos los elementos *\${}* y aplicarles el valor de la propiedad del fichero *properties*. ¿Podéis pensar durante un momento cómo lo hace? Efectivamente. El elemento *property-placeholder* se trata de un BeanFactoryPostProcessor, que en su método *postProcessBeanFactory* realiza esta operativa.

Un ejemplo de BeanFactoryPostProcessor sería el siguiente (no implementado en el proyecto ejemplo):

```
public class AppCfgBeanFactoryPostProcessor implements BeanFactoryPostProcessor
{
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException
    {
        PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();

        cfg.setLocation(new FileSystemResource("config.properties"));
        cfg.postProcessBeanFactory(beanFactory);
    }
}
```

6. Referencias

- <http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-factory-extension-bpp>
- <http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-factory-extension-factory-postprocessors>

A continuación puedes evaluarlo:

[Regístrate para evaluarlo](#)

Por favor, vota +1 o compártelo si te pareció interesante

Share |

 0

Anímate y coméntanos lo que pienses sobre este **TUTORIAL**:

» **Regístrate** y accede a esta y otras ventajas «



Esta obra está licenciada bajo licencia [Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)

IMPULSA

Impulsores

Comunidad

[¿Ayuda?](#)

sin clicks

0 personas han traído clicks a esta página

+ + + + + + + +

powered by [karmacracy](#)

Copyright 2003-2014 © All Rights Reserved | [Texto legal y condiciones de uso](#) | [Banners](#) | [Powered by Autentia](#) | [Contacto](#)

 XHTML 1.0

 CSS

 XML RSS

 XML RDF