Container extension points

Summary

Description

The IoC component of the Spring Framework is designed with the consideration of extension. Generally, the application developer does not need to inherit various BeanFactory or ApplicationContext implementation class. The Spring IoC Container can be extended by plugging in implementations of special integrated interface.

Customizing beans using BeanPostProcessors

The BeanPostProcessors interface defines many number of callback methods. The application developer can implement these methods to provide their own instantiation logic and dependency-resolution logic.

The org.springframework.beans.factory.config.BeanPostProcessor interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container (see below for how this registration is effected), for each bean instance that is created by the container, the post-processor will get a callback from the container both *before* any container initialization methods (such as *afterPropertiesSet* and any declared init method) are called.

It is important to know that a BeanFactory treats bean post-processors slightly differently than an ApplicationContext. An ApplicationContext will *automatically detect* any beans which are defined in the configuration metadata which is supplied to it that implement the BeanPostProcessor interface, and register them as post-processors, to be then called appropriately by the container on bean creation. the other hand, when using a BeanFactory implementation, bean post-processors explicitly have to be registered, with code like this:

```
ConfigurableBeanFactory factory = new XmlBeanFactory(...);

// now register any needed BeanPostProcessor instances
MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();
factory.addBeanPostProcessor(postProcessor);

// now start using the factory
```

This explicit registration step is not convenient, and this is one of the reasons why the various ApplicationContext implementations are preferred above plain BeanFactory implementations in the vast majority of Spring-backed applications

Example: Hello World, BeanPostProcessor-style

Although this is not a proper example, it shows the basic method.

```
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        return bean; // we could potentially return any object reference here...
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException
{
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
```

```
        return bean;
    }
}
<?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:lang="http://www.springframework.org/schema/lang"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.5.xsd">

    <lang:groovy id="messenger"
          script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
    </lang:groovy>

    <!--
        when the above bean ('messenger') is instantiated, this custom
        BeanPostProcessor implementation will output the fact to the system console
     -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

Notice how the InstantiationTracingBeanPostProcessor is simply defined; it doesn't even have a name, and because it is a bean it can be dependency injected just like any other bean.

Customizing configuration metadata with BeanFactoryPostProcessors

The org.springframework.beans.factory.config.BeanFactoryPostProcessor is similar to the BeanPostProcessor in its meaning. However, one of the biggest differences is that the BeanFactoryPostProcessors processes the bean configuration metadata. the Spring IoC container will allow BeanFactoryPostProcessors to read the configuration metadata and potentially change it *before* the container has actually instantiated any other beans.

A bean factory post-processor is executed manually (in the case of a BeanFactory) or automatically (in the case of an ApplicationContext)

In a BeanFactory, the process of applying a BeanFactoryPostProcessor is manual, and will be similar to this:

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

Example: the PropertyPlaceholderConfigurer

The PropertyPlaceholderConfigurer is used to externalize property values from a BeanFactory definition, into another separate file in the standard Java Properties format. This is useful to allow the person deploying an application to customize environment-specific properties (for example database URLs, usernames and passwords), without the complexity or risk of modifying the main XML definition file or files for the container.

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <value>classpath:com/foo/jdbc.properties</value>
    </property>
</bean>

<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

The actual values come from another file in the standard Java Properties format:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

With the context namespace introduced in Spring 2.5, it is possible to configure property placeholders with a dedicated configuration element.

```
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>
```

The PropertyPlaceholderConfigurer doesn't only look for properties in the Properties file you specify, but also checks against the Java System properties if it cannot find a property you are trying to use. This behavior can be customized by setting the systemPropertiesMode property of the configurer.

Example: the PropertyOverrideConfigurer

The PropertyOverrideConfigurer, another bean factory post-processor, is similar to the PropertyPlaceholderConfigurer, but in contrast to the latter, the original definitions can have default values or no values at all for bean properties. If an overriding Properties file does not have an entry for a certain bean property, the default context definition is used.

Properties file configuration lines are expected to be in the format:

```
beanName.property=value
```

With the context namespace introduced in Spring 2.5, it is possible to configure property overriding with a dedicated configuration element:

```
<context:property-override location="classpath:override.properties"/>
```

Customizing instantiation logic using FactoryBeans

The org.springframework.beans.factory.FactoryBean interface is to be implemented by objects that are themselves factories.

The FactoryBean interface is a point of pluggability into the Spring IoC containers instantiation logic. If you have some complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own FactoryBean, write the complex initialization inside that class, and then plug your custom FactoryBean into the container.

The FactoryBean interface provides three methods:

- Object getObject(): has to return an instance of the object this factory creates. The instance can possibly be shared (depending on whether this factory returns singletons or prototypes).
- boolean isSingleton(): has to return true if this FactoryBean returns singletons, false otherwise
- Class getObjectType(): has to return either the object type returned by the getObject() method or null if the type isn't known in advance.

Reference

- Spring Framework - Reference Document / 3.7. Container extension points