

Spring Bean Definition Templates

using a BeanFactoryPostProcessor

Sven Ehrke

(sven.ehrke@web.de)

21.05.2006

Table of contents

Introduction.....	I
Solution.....	I
Summary.....	3
Resources.....	3

Introduction

Very often I had the need to configure beans using Spring in a predefined way with only a few details modified. Here is an example of what I wanted to achieve:

```
<bean id="template" class="org.svenehrke.Action"
    abstract="true">
    <property name="controller">
        <bean class="org.svenehrke.Controller">
            <property name="name"
                value="Controller number ${id}"/>
        </bean>
    </property>
</bean>

<bean id="action1" parent="template">
    <property name="name" value="fooAction"/>
</bean>
```

The shown configuration contains an abstract template bean for so called action beans. The example defines one action bean named `action1` which inherits from a template defining the standard configuration for actions. The placeholder `{id}` used in the definition of the template needs to be replaced by a value defined by `action1`. It seemed to me using a `BeanFactoryPostProcessor` is the right tool for this job. So I thought let's see if `PropertyPlaceholderConfigurer` can be reused. It turned out that it can't. First because it is working on all beans defined and I wanted it to work only on certain beans. The other reason is that I wanted it to work on bean definitions which inherited properties with placeholders from the parent.

Solution

Looking at the source code of `PropertyPlaceholderConfigurer` I learned how to start with my own `BeanFactoryPostProcessor` to do the job and gave it the name `TemplateBeanFactoryPostProcessor`.

It has an attribute called `pattern` which contains a regular expression for the names of the beans it is supposed to work on. So if you want it to work on all beans named „`action1`“, „`action2`“ set it's value to „`^action.*`“.

The thing it needed to do is to loop through all the bean definitions and start to replace to placeholders when the pattern matches the beanname. We get all the names of the bean definition like this:

```
String[] beanNames =
    aBeanFactory.getBeanDefinitionNames();
```

where `aBeanFactory` is of type `ConfigurableListableBeanFactory`.

Now we can loop through all the names and if one matches the pattern we retrieve the definition of the bean:

```
String bn = beanNames[i];
BeanDefinition bd =
    aBeanFactory.getBeanDefinition(bn);
```

With the help of a `BeanDefinitionVisitor` all the placeholder handling is done:

```
BeanDefinitionVisitor visitor = new
    TemplateBeanDefinitionVisitor(aBeanFactory, bd, bn);
visitor.visitBeanDefinition(bd);
```

A `BeanDefinitionVisitor` contains a routine which is called whenever a value of a `BeanDefinition` is handled:

```
protected String resolveStringValueInternal(
    String aValue) throws BeansException
```

This is the perfect hook for our placeholder replacement. We get in the value „`#{id}`“ for example and return a different value like „571“ for it which then will be used by Spring as if it was configured like that in the configuration file originally.

But where do we get the values from? In my first attempt I put them in as properties into the same `beandefinitions` I wanted to configure (“`action1`“... in the example) I have been configuring:

```
<bean id="action1" parent="template">
    <property name="id" value="571"/>
    <property name="name" value="fooAction"/>
</bean>
```

But this did not feel right since the placeholders are only supposed to ease configuration and not change the target pojo. Imagine you have several of these „`meta`“ properties mixed with the real business properties in your pojo. It will become a mess.

So I decided to have meta `beandefinitions` which are only there for the placeholders. This is the result:

```
<bean id=":action1" abstract="true">
    <property name="id" value="571"/>
</bean>

<bean id="action1" parent="template">
    <property name="name" value="fooAction"/>
</bean>
```

The preprocessor recognizes meta-beandefinitions from the leading colon in the beanname¹. So for each bean it is about to handle it will look for an associated meta beandefinition and uses it's values to replace the placeholders in the beandefinitions of the target pojos. And if you declare them as abstract then you don't even need to create a class for it.

One difficulty still needed to be resolved. When the routine `getPropertyValues()` is called on a `BeanDefinition` only those properties are returned defined in that `BeanDefinition`. The ones for it's parents are not included. To get them as well we recursively step up the inheritance tree and add all the properties of the parents if not already defined in the child².

Summary

This article explained how a `BeanFactoryPostProcessor` can be used to create bean definitions as templates for other bean definitions. This is especially useful if you have many very similar bean definitions which differ only in a few aspects from each other. Using a bean definition template as shown reduces the configuration overhead a lot.

The shown method works well for simple cases. It has it's limits however. For example you cannot use meta data to configure the classname in a bean definition because Spring directly uses the value from the attribute `class` and thus the post processor does not get a chance to modify it's value. This issue will be addressed in one of my next articles using a technique independent of these restrictions.

Resources

1. This article:
http://www.sven-ehrke.de/articles/sbdeftplpp/spring_bean_deftpl_bfpp.pdf
2. Source code:
<http://www.sven-ehrke.de/articles/sbdeftplpp/source.zip>

¹ I wanted to use an at sign (“@action1”). Unfortunately this is not possible for id attributes in XML. A solution is to use the attribute `name` instead since it does not have this restriction.

² This reusable functionality is implemented in class `BeanDefinitionFlattener`