

## 41. XML Schema-based configuration

### 41.1 Introduction

This appendix details the XML Schema-based configuration introduced in Spring 2.0 and enhanced and extended in Spring 2.5 and 3.0.

#### DTD support?

Authoring Spring configuration files using the older DTD style is still fully supported.

Nothing will break if you forego the use of the new XML Schema-based approach to authoring Spring XML configuration files. All that you lose out on is the opportunity to have more succinct and clearer configuration. Regardless of whether the XML configuration is DTD- or Schema-based, in the end it all boils down to the same object model in the container (namely one or more `BeanDefinition` instances).

The central motivation for moving to XML Schema based configuration files was to make Spring XML configuration easier. The '*classic*' `<bean/>`-based approach is good, but its generic-nature comes with a price in terms of configuration overhead.

From the Spring IoC containers point-of-view, *everything* is a bean. That's great news for the Spring IoC container, because if everything is a bean then everything can be treated in the exact same fashion. The same, however, is not true from a developer's point-of-view. The objects defined in a Spring XML configuration file are not all generic, vanilla beans. Usually, each bean requires some degree of specific configuration.

Spring 2.0's new XML Schema-based configuration addresses this issue. The `<bean/>` element is still present, and if you wanted to, you could continue to write the *exact same* style of Spring XML configuration using only `<bean/>` elements. The new XML Schema-based configuration does, however, make Spring XML configuration files substantially clearer to read. In addition, it allows you to express the intent of a bean definition.

The key thing to remember is that the new custom tags work best for infrastructure or integration beans: for example, AOP, collections, transactions, integration with 3rd-party frameworks such as Mule, etc., while the existing bean tags are best suited to application-specific beans, such as DAOs, service layer objects, validators, etc.

The examples included below will hopefully convince you that the inclusion of XML Schema support in Spring 2.0 was a good idea. The reception in the community has been encouraging; also, please note the fact that this new configuration mechanism is totally customisable and extensible. This means you can write your own domain-specific configuration tags that would better represent your application's domain; the process involved in doing so is covered in the appendix entitled Chapter 42, *Extensible XML authoring*.

### 41.2 XML Schema-based configuration

#### Referencing the schemas

To switch over from the DTD-style to the new XML Schema-style, you need to make the following change.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<!-- bean definitions here -->

</beans>
```

The equivalent file in the XML Schema-style would be...

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
        beans.xsd">

    <!-- bean definitions here -->

</beans>
```

### Note

The 'xsi:schemaLocation' fragment is not actually required, but can be included to reference a local copy of a schema (which can be useful during development).

The above Spring XML configuration fragment is boilerplate that you can copy and paste (!) and then plug `<bean/>` definitions into like you have always done. However, the entire point of switching over is to take advantage of the new Spring 2.0 XML tags since they make configuration easier. The section entitled the section called “the util schema” demonstrates how you can start immediately by using some of the more common utility tags.

The rest of this chapter is devoted to showing examples of the new Spring XML Schema based configuration, with at least one example for every new tag. The format follows a before and after style, with a *before* snippet of XML showing the old (but still 100% legal and supported) style, followed immediately by an *after* example showing the equivalent in the new XML Schema-based style.

## the util schema

First up is coverage of the `util` tags. As the name implies, the `util` tags deal with common, *utility* configuration issues, such as configuring collections, referencing constants, and suchlike.

To use the tags in the `util` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the snippet below references the correct schema so that the tags in the `util` namespace are available to you.

```
<?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:util="http://www.springframework.org/schema/util" xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
        beans.xsd
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-
        util.xsd"> <!-- bean definitions here -->

</beans>
```

**<util:constant/>**

Before...

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
          class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `FieldRetrievingFactoryBean`, to set the value of the `isolation` property on a bean to the value of the `java.sql.Connection.TRANSACTION_SERIALIZABLE` constant. This is all well and good, but it is a tad verbose and (unnecessarily) exposes Spring's internal plumbing to the end user.

The following XML Schema-based version is more concise and clearly expresses the developer's intent (*'inject this constant value'*), and it just reads better.

```
<bean id="..." class="...">
  <property name="isolation">
    <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
  </property>
</bean>
```

**Setting a bean property or constructor arg from a field value**

[FieldRetrievingFactoryBean](#) is a `FactoryBean` which retrieves a static or non-static field value. It is typically used for retrieving public static final constants, which may then be used to set a property value or constructor arg for another bean.

Find below an example which shows how a static field is exposed, by using the [staticField](#) property:

```
<bean id="myField"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
  <property name="staticField" value="java.sql.Connection.TRANSACTION_SERIALIZABLE" />
</bean>
```

There is also a convenience usage form where the `static` field is specified as the bean name:

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
```

This does mean that there is no longer any choice in what the bean id is (so any other bean that refers to it will also have to use this longer name), but this form is very concise to define, and very convenient to use as an inner bean since the id doesn't have to be specified for the bean reference:

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
          class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </property>
</bean>
```

It is also possible to access a non-static (instance) field of another bean, as described in the API documentation for the [FieldRetrievingFactoryBean](#) class.

Injecting enum values into beans as either property or constructor arguments is very easy to do in Spring, in that you don't actually have to *do* anything or know anything about the Spring internals (or

even about classes such as the `FieldRetrievingFactoryBean`). Let's look at an example to see how easy injecting an enum value is; consider this JDK 5 enum:

```
package javax.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED

}
```

Now consider a setter of type `PersistenceContextType`:

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }

}
```

a. and the corresponding bean definition:

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION" />
</bean>
```

This works for classic type-safe emulated enums (on JDK 1.4 and JDK 1.3) as well; Spring will automatically attempt to match the string property value to a constant on the enum class.

### <util:property-path/>

Before...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

The above configuration uses a Spring `FactoryBean` implementation, the `PropertyPathFactoryBean`, to create a bean (of type `int`) called `testBean.age` that has a value equal to the `age` property of the `testBean` bean.

After...

```

<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>

```

The value of the `path` attribute of the `<property-path/>` tag follows the form `beanName.beanProperty`.

### Using `<util:property-path/>` to set a bean property or constructor-argument

`PropertyPathFactoryBean` is a `FactoryBean` that evaluates a property path on a given target object. The target object can be specified directly or via a bean name. This value may then be used in another bean definition as a property value or constructor argument.

Here's an example where a path is used against another bean, by name:

```

// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'
<bean id="theAge"
  class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetBeanName" value="person"/>
  <property name="propertyPath" value="spouse.age"/>
</bean>

```

In this example, a path is evaluated against an inner bean:

```

<!-- will result in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
  class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetObject">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="12"/>
    </bean>
  </property>
  <property name="propertyPath" value="age"/>
</bean>

```

There is also a shortcut form, where the bean name is the property path.

```

<!-- will result in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
  class="org.springframework.beans.factory.config.PropertyPathFactoryBean">

```

This form does mean that there is no choice in the name of the bean. Any reference to it will also have to use the same id, which is the path. Of course, if used as an inner bean, there is no need to refer to it at all:

```

<bean id="..." class="...">
  <property name="age">
    <bean id="person.age"
          class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
    </property>
  </bean>
</bean>

```

The result type may be specifically set in the actual definition. This is not necessary for most use cases, but can be of use for some. Please see the Javadocs for more info on this feature.

### <util:properties/>

Before...

```

<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>

```

The above configuration uses a Spring FactoryBean implementation, the PropertiesFactoryBean, to instantiate a java.util.Properties instance with values loaded from the supplied [Resource](#) location).

After...

```

<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>

```

### <util:list/>

Before...

```

<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
    <list>
      <value>pechorin@hero.org</value>
      <value>raskolnikov@slums.org</value>
      <value>stavrogin@gov.org</value>
      <value>porfiry@gov.org</value>
    </list>
  </property>
</bean>

```

The above configuration uses a Spring FactoryBean implementation, the ListFactoryBean, to create a java.util.List instance initialized with values taken from the supplied sourceList.

After...

```

<!-- creates a java.util.List instance with the supplied values -->
<util:list id="emails">
  <value>pechorin@hero.org</value>
  <value>raskolnikov@slums.org</value>
  <value>stavrogin@gov.org</value>
  <value>porfiry@gov.org</value>
</util:list>

```

You can also explicitly control the exact type of List that will be instantiated and populated via the use of the list-class attribute on the <util:list/> element. For example, if we really need a java.util.LinkedList to be instantiated, we could use the following configuration:

```
<util:list id="emails" list-class="java.util.LinkedList">
  <value>jackshaftoe@vagabond.org</value>
  <value>eliza@thinkingmanscrumpet.org</value>
  <value>vanhoek@pirate.org</value>
  <value>d'Arcachon@nemesis.org</value>
</util:list>
```

If no `list-class` attribute is supplied, a `List` implementation will be chosen by the container.

### **<util:map/>**

Before...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
  <property name="sourceMap">
    <map>
      <entry key="pechorin" value="pechorin@hero.org"/>
      <entry key="raskolnikov" value="raskolnikov@slums.org"/>
      <entry key="stavrogin" value="stavrogin@gov.org"/>
      <entry key="porfiry" value="porfiry@gov.org"/>
    </map>
  </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `MapFactoryBean`, to create a `java.util.Map` instance initialized with key-value pairs taken from the supplied `'sourceMap'`.

After...

```
<!-- creates a java.util.Map instance with the supplied key-value pairs -->
<util:map id="emails">
  <entry key="pechorin" value="pechorin@hero.org"/>
  <entry key="raskolnikov" value="raskolnikov@slums.org"/>
  <entry key="stavrogin" value="stavrogin@gov.org"/>
  <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

You can also explicitly control the exact type of `Map` that will be instantiated and populated via the use of the `'map-class'` attribute on the `<util:map/>` element. For example, if we really need a `java.util.TreeMap` to be instantiated, we could use the following configuration:

```
<util:map id="emails" map-class="java.util.TreeMap">
  <entry key="pechorin" value="pechorin@hero.org"/>
  <entry key="raskolnikov" value="raskolnikov@slums.org"/>
  <entry key="stavrogin" value="stavrogin@gov.org"/>
  <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

If no `'map-class'` attribute is supplied, a `Map` implementation will be chosen by the container.

### **<util:set/>**

Before...

```

<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
  <property name="sourceSet">
    <set>
      <value>pechorin@hero.org</value>
      <value>raskolnikov@slums.org</value>
      <value>stavrogin@gov.org</value>
      <value>porfiry@gov.org</value>
    </set>
  </property>
</bean>

```

The above configuration uses a Spring `FactoryBean` implementation, the `SetFactoryBean`, to create a `java.util.Set` instance initialized with values taken from the supplied 'sourceSet'.

After...

```

<!-- creates a java.util.Set instance with the supplied values -->
<util:set id="emails">
  <value>pechorin@hero.org</value>
  <value>raskolnikov@slums.org</value>
  <value>stavrogin@gov.org</value>
  <value>porfiry@gov.org</value>
</util:set>

```

You can also explicitly control the exact type of `Set` that will be instantiated and populated via the use of the 'set-class' attribute on the `<util:set/>` element. For example, if we really need a `java.util.TreeSet` to be instantiated, we could use the following configuration:

```

<util:set id="emails" set-class="java.util.TreeSet">
  <value>pechorin@hero.org</value>
  <value>raskolnikov@slums.org</value>
  <value>stavrogin@gov.org</value>
  <value>porfiry@gov.org</value>
</util:set>

```

If no 'set-class' attribute is supplied, a `Set` implementation will be chosen by the container.

## the jee schema

The `jee` tags deal with Java EE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

To use the tags in the `jee` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `jee` namespace are available to you.

```

<?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:jee="http://www.springframework.org/schema/jee" xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
    beans.xsd
    http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-
    jee.xsd"> <!-- bean definitions here -->

</beans>

```

### <jee:jndi-lookup/> (simple)

Before...