Appendix F. Extensible XML authoring

F.1 Introduction

Since version 2.0, Spring has featured a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. This section is devoted to detailing how you would go about writing your own custom XML bean definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring of configuration files using a schema-aware XML editor, Spring's extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring's current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled Appendix E, *XML Schema-based configuration*.

Creating new XML configuration extensions can be done by following these (relatively) simple steps:

1. Authoring an XML schema to describe your custom element(s).
2. Coding a custom `NamespaceHandler` implementation (this is an easy step, don't worry).
3. Coding one or more `BeanDefinitionParser` implementations (this is where the real work is done).
4. Registering the above artifacts with Spring (this too is an easy step).

What follows is a description of each of these steps. For the example, we will create an XML extension (a custom XML element) that allows us to configure objects of the type `SimpleDateFormat` (from the `java.text` package) in an easy manner. When we are done, we will be able to define bean definitions of type `SimpleDateFormat` like this:

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
```

```
    lenient="true"/>
```

(Don't worry about the fact that this example is very simple; much more detailed examples follow afterwards. The intent in this first simple example is to walk you through the basic steps involved.)

F.2 Authoring the schema

Creating an XML configuration extension for use with Spring's IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we'll use to configure `SimpleDateFormat` objects.

```
<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.com/schema/myns"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:beans="http://www.springframework.org/schema/beans"
    targetNamespace="http://www.mycompany.com/schema/myns"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

   <xsd:import namespace="http://www.springframework.org/schema/beans"/>

   <xsd:element name="dateformat">
      <xsd:complexType>
         <xsd:complexContent>
            <xsd:extension base="beans:identifiedType">
               <xsd:attribute name="lenient" type="xsd:boolean"/>

               <xsd:attribute name="pattern" type="xsd:string" use="required"/>
            </xsd:extension>
         </xsd:complexContent>
      </xsd:complexType>
   </xsd:element>
```

```
</xsd:schema>
```

(The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an `id` attribute that will be used as the bean identifier in the container). We are able to use this attribute because we imported the Spring-provided `'beans'` namespace.)

The above schema will be used to configure `SimpleDateFormat` objects, directly in an XML application context file using the `<myns:dateformat/>` element.

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

Note that after we've created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we're just creating a bean in the container, identified by the name `'dateFormat'` of type `SimpleDateFormat`, with a couple of properties set.

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. Using a properly authored schema, you can use autocompletion to have a user choose between several configuration options defined in the enumeration.

F.3 Coding a `NamespaceHandler`

In addition to the schema, we need a `NamespaceHandler` that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The `NamespaceHandler` should in our case take care of the parsing of the `myns:dateformat` element.

The `NamespaceHandler` interface is pretty simple in that it features just three methods:

- `init()` - allows for initialization of the `NamespaceHandler` and will be called by Spring before the handler is used
- `BeanDefinition parse(Element, ParserContext)` - called when Spring encounters a top-level element (not nested inside a bean definition or a different namespace). This method can register bean definitions itself and/or return a bean definition.
- `BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext)` - called when Spring encounters an attribute or nested element of a different namespace. The decoration of one or more bean definitions is used for example with the out-of-the-box scopes Spring 2.0 supports. We'll start by highlighting a simple example, without using decoration, after which we will show decoration in a somewhat more advanced example.

Although it is perfectly possible to code your own `NamespaceHandler` for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single bean definition (as in our case, where a single `<myns:dateformat/>` element results in a single `SimpleDateFormat` bean definition). Spring features a number of convenience classes that support this scenario. In this example, we'll make use the `NamespaceHandlerSupport` class:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("dateformat", new SimpleDateFormatBeanDefinitionParser());
    }
}
```

The observant reader will notice that there isn't actually a whole lot of parsing logic in this class. Indeed... the `NamespaceHandlerSupport` class has a built in notion of delegation. It supports the registration of any number of `BeanDefinitionParser` instances, to which it will delegate to when it needs to parse an element in its namespace. This clean separation of concerns allows a `NamespaceHandler` to handle the orchestration of the parsing of *all* of the custom elements in its namespace, while delegating to `BeanDefinitionParsers` to do the grunt work of the XML parsing; this means that each `BeanDefinitionParser` will contain just the logic for parsing a single custom element, as we can see in the next step

F.4 Coding a `BeanDefinitionParser`

A `BeanDefinitionParser` will be used if the `NamespaceHandler` encounters an XML element of the type that has been mapped to the specific bean definition parser (which is `'dateformat'` in this case). In other words, the `BeanDefinitionParser` is responsible for parsing *one* distinct top-level XML element defined in the schema. In the parser, we'll have access to the XML element (and thus its subelements too) so that we can parse our custom XML content, as can be seen in the following example:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;


import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends AbstractSingleBeanDefinitionParser { ❶

   protected Class getBeanClass(Element element) {
      return SimpleDateFormat.class; ❷
   }

   protected void doParse(Element element, BeanDefinitionBuilder bean) {
```

```
        // this will never be null since the schema explicitly requires that a value be supplied
        String pattern = element.getAttribute("pattern");
        bean.addConstructorArg(pattern);

        // this however is an optional property
        String lenient = element.getAttribute("lenient");
        if (StringUtils.hasText(lenient)) {
            bean.addPropertyValue("lenient", Boolean.valueOf(lenient));
        }
    }
}
```

❶ We use the Spring-provided `AbstractSingleBeanDefinitionParser` to handle a lot of the basic grunt work of creating a *single* `BeanDefinition`.

❷ We supply the `AbstractSingleBeanDefinitionParser` superclass with the type that our single `BeanDefinition` will represent.

In this simple case, this is all that we need to do. The creation of our single `BeanDefinition` is handled by the `AbstractSingleBeanDefinitionParser` superclass, as is the extraction and setting of the bean definition's unique identifier.

F.5 Registering the handler and the schema

The coding is finished! All that remains to be done is to somehow make the Spring XML parsing infrastructure aware of our custom element; we do this by registering our custom `namespaceHandler` and custom XSD file in two special purpose properties files. These properties files are both placed in a `'META-INF'` directory in your application, and can, for example, be distributed alongside your binary classes in a JAR file. The Spring XML parsing infrastructure will automatically pick up your new extension by consuming these special properties files, the formats of which are detailed below.

F.5.1 `'META-INF/spring.handlers'`

The properties file called `'spring.handlers'` contains a mapping of XML Schema URIs to namespace handler classes. So for our example, we need to write the following:

```
http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler
```

(The `':'` character is a valid delimiter in the Java properties format, and so the `':'` character in the URI needs to be escaped with a backslash.)

The first part (the key) of the key-value pair is the URI associated with your custom namespace extension, and needs to *match exactly* the value of the `'targetNamespace'` attribute as specified in your custom XSD schema.

F.5.2 `'META-INF/spring.schemas'`

The properties file called `'spring.schemas'` contains a mapping of XML Schema locations (referred to along with the schema declaration in XML files that use the schema as part of the `'xsi:schemaLocation'` attribute) to *classpath* resources. This file is needed to prevent Spring from absolutely having to use a default `EntityResolver` that requires Internet access to retrieve the schema file. If you specify the mapping in this properties file, Spring will search for the schema on the classpath (in this case `'myns.xsd'` in the `'org.springframework.samples.xml'` package):

```
http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd
```

The upshot of this is that you are encouraged to deploy your XSD file(s) right alongside the `NamespaceHandler` and `BeanDefinitionParser` classes on the classpath.

F.6 Using a custom extension in your Spring XML configuration

Using a custom extension that you yourself have implemented is no different from using one of the 'custom' extensions that Spring provides straight out of the box. Find below an example of using the custom `<dateformat/>` element developed in the previous steps in a Spring XML configuration file.

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

   <!-- as a top-level bean -->
   <myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true"/>

   <bean id="jobDetailTemplate" abstract="true">
      <property name="dateFormat">
         <!-- as an inner bean -->
         <myns:dateformat pattern="HH:mm MM-dd-yyyy"/>
      </property>
   </bean>

</beans>
```

F.7 Meatier examples

Find below some much meatier examples of custom XML extensions.

F.7.1 Nesting custom tags within custom tags

This example illustrates how you might go about writing the various artifacts required to satisfy a target of the following configuration:

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

   <foo:component id="bionic-family" name="Bionic-1">
      <foo:component name="Mother-1">
        <foo:component name="Karate-1"/>
        <foo:component name="Sport-1"/>
      </foo:component>
      <foo:component name="Rock-1"/>
   </foo:component>

</beans>
```

The above configuration actually nests custom extensions within each other. The class that is actually configured by the above `<foo:component/>` element is the `Component` class (shown directly below). Notice how the `Component` class does *not* expose a setter method for the `'components'` property; this makes it hard (or rather impossible) to configure a bean definition for the `Component` class using setter injection.

```
package com.foo;

import java.util.ArrayList;
import java.util.List;

public class Component {

   private String name;
   private List<Component> components = new ArrayList<Component> ();

   // mmm, there is no setter method for the 'components'
   public void addComponent(Component component) {
      this.components.add(component);
   }
```

```
    public List<Component> getComponents() {
        return components;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

The typical solution to this issue is to create a custom `FactoryBean` that exposes a setter property for the `'components'` property.

```
package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.List;

public class ComponentFactoryBean implements FactoryBean<Component> {

    private Component parent;
    private List<Component> children;

    public void setParent(Component parent) {
        this.parent = parent;
    }

    public void setChildren(List<Component> children) {
        this.children = children;
```

```
    }

    public Component getObject() throws Exception {
        if (this.children != null && this.children.size() > 0) {
            for (Component child : children) {
                this.parent.addComponent(child);
            }
        }
        return this.parent;
    }

    public Class<Component> getObjectType() {
        return Component.class;
    }

    public boolean isSingleton() {
        return true;
    }
}
```

This is all very well, and does work nicely, but exposes a lot of Spring plumbing to the end user. What we are going to do is write a custom extension that hides away all of this Spring plumbing. If we stick to the steps described previously, we'll start off by creating the XSD schema to define the structure of our custom tag.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/component"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.com/schema/component"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

    <xsd:element name="component">
        <xsd:complexType>
```

```
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
               <xsd:element ref="component"/>
            </xsd:choice>
            <xsd:attribute name="id" type="xsd:ID"/>
            <xsd:attribute name="name" use="required" type="xsd:string"/>
         </xsd:complexType>
      </xsd:element>

</xsd:schema>
```

We'll then create a custom `NamespaceHandler`.

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

   public void init() {
      registerBeanDefinitionParser("component", new ComponentBeanDefinitionParser());
   }
}
```

Next up is the custom `BeanDefinitionParser`. Remember that what we are creating is a `BeanDefinition` describing a `ComponentFactoryBean`.

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
```

```
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;

import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

   protected AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {
      return parseComponentElement(element);
   }

   private static AbstractBeanDefinition parseComponentElement(Element element) {
      BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);
      factory.addPropertyValue("parent", parseComponent(element));

      List<Element> childElements = DomUtils.getChildElementsByTagName(element, "component");
      if (childElements != null && childElements.size() > 0) {
         parseChildComponents(childElements, factory);
      }

      return factory.getBeanDefinition();
   }

   private static BeanDefinition parseComponent(Element element) {
      BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition(Component.class);
      component.addPropertyValue("name", element.getAttribute("name"));
      return component.getBeanDefinition();
   }

   private static void parseChildComponents(List<Element> childElements, BeanDefinitionBuilder factory) {
      ManagedList<BeanDefinition> children = new ManagedList<BeanDefinition>(childElements.size());

      for (Element element : childElements) {
         children.add(parseComponentElement(element));
      }
```

```
        factory.addPropertyValue("children", children);
    }
}
```

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/component=com.foo.ComponentNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/component/component.xsd=com/foo/component.xsd
```

F.7.2 Custom attributes on 'normal' elements

Writing your own custom parser and the associated artifacts isn't hard, but sometimes it is not the right thing to do. Consider the scenario where you need to add metadata to already existing bean definitions. In this case you certainly don't want to have to go off and write your own entire custom extension; rather you just want to add an additional attribute to the existing bean definition element.

By way of another example, let's say that the service class that you are defining a bean definition for a service object that will (unknown to it) be accessing a clustered JCache, and you want to ensure that the named JCache instance is eagerly started within the surrounding cluster:

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"
      jcache:cache-name="checking.account">
    <!-- other dependencies here... -->
</bean>
```

What we are going to do here is create another `BeanDefinition` when the `'jcache:cache-name'` attribute is parsed; this `BeanDefinition` will then initialize the named JCache for us. We will also modify the existing `BeanDefinition` for the

`'checkingAccountService'` so that it will have a dependency on this new JCache-initializing `BeanDefinition`.

```
package com.foo;

public class JCacheInitializer {

   private String name;

   public JCacheInitializer(String name) {
      this.name = name;
   }

   public void initialize() {
      // lots of JCache API calls to initialize the named cache...
   }
}
```

Now onto the custom extension. Firstly, the authoring of the XSD schema describing the custom attribute (quite easy in this case).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/jcache"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.foo.com/schema/jcache"
            elementFormDefault="qualified">

   <xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>
```

Next, the associated `NamespaceHandler`.

```
package com.foo;
```

```
import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

   public void init() {
      super.registerBeanDefinitionDecoratorForAttribute("cache-name",
            new JCacheInitializingBeanDefinitionDecorator());
   }
}
```

Next, the parser. Note that in this case, because we are going to be parsing an XML attribute, we write a `BeanDefinitionDecorator` rather than a `BeanDefinitionParser`.

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class JCacheInitializingBeanDefinitionDecorator implements BeanDefinitionDecorator {

   private static final String[] EMPTY_STRING_ARRAY = new String[0];

   public BeanDefinitionHolder decorate(
         Node source, BeanDefinitionHolder holder, ParserContext ctx) {
      String initializerBeanName = registerJCacheInitializer(source, ctx);
```

```
        createDependencyOnJCacheInitializer(holder, initializerBeanName);
        return holder;
    }

    private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder, String initializerBeanName) {
        AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder.getBeanDefinition());
        String[] dependsOn = definition.getDependsOn();
        if (dependsOn == null) {
            dependsOn = new String[]{initializerBeanName};
        } else {
            List dependencies = new ArrayList(Arrays.asList(dependsOn));
            dependencies.add(initializerBeanName);
            dependsOn = (String[]) dependencies.toArray(EMPTY_STRING_ARRAY);
        }
        definition.setDependsOn(dependsOn);
    }

    private String registerJCacheInitializer(Node source, ParserContext ctx) {
        String cacheName = ((Attr) source).getValue();
        String beanName = cacheName + "-initializer";
        if (!ctx.getRegistry().containsBeanDefinition(beanName)) {
            BeanDefinitionBuilder initializer = BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.class)
            initializer.addConstructorArg(cacheName);
            ctx.getRegistry().registerBeanDefinition(beanName, initializer.getBeanDefinition());
        }
        return beanName;
    }
}
```

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/jcache=com.foo.JCacheNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```

F.8 Further Resources

Find below links to further resources concerning XML Schema and the extensible XML support described in this chapter.

- The XML Schema Part 1: Structures Second Edition
- The XML Schema Part 2: Datatypes Second Edition

Prev Up Next

Appendix E. XML Schema-based configuration Home Appendix G. spring.tld