Spring 2.0 XML Configuration

Dmitry Sklyut

<beans/> ++

Agenda

- Brief overview of 1.x capabilities
- Motivation for change
- 2.0 XML configuration changes overview
- New Namespaces
- Creating custom extensions
- Summary
- Q&A

1.x Capabilities

- Configuration
  - Code
  - Properties file format
  - DTD based XML format
- XML format
  - Very generic and uniform
  - Everything is a bean or property
  - Uses factories, proxy factories, and lifecycle callbacks to wire up beans
  - Must know FQN of the class and property names.
- AOP interceptors are per bean (proxy) definition.

Motivation

- User requested enhancement
- Minimize verbosity of configuration
- Provide better tooling support
  - Code completion
  - "Compile" time validation
  - Sensible defaults
- Out of the box support for common configuration tasks
- Ability to create domain specific custom configuration syntax
- Better configuration documentation

Spring 2.0 Changes Overview

- 1.x DTD syntax fully supported!!!
  - Including complete interoperability between DTD and schema based syntax.
- 2.x syntax is XML Schema based
- Custom tags for common configuration (jndi, transactions, aop)
- Much less verbose
- Much better tooling support
  - XMLSpy, Oxygen, Stylus Studio, many more
- Ability to create custom configuration syntax tags

Spring 2.0 Namespaces

- beans – same tags as 2.0 version of DTD
- util – define simple collection types
- jee – Java EE related.
  - Look up of jndi object
  - Defining EJB references
- aop – Aop configuration custom tags.
- tx – Transaction configuration.
  - Works in conjunction with Aop namespace
- p – Replacement for <property/>
  - maps custom attributes directly through to bean properties.
- lang – Scripting languages support
- tool –Provides tooling metadata support
  - Experimental

<util:* /> Namespace

- Used to deal with collection types, constants, etc.

- Use the following stanza to enable support

```
<?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
</beans>
```

<util:*/> Namespace

<util:properties /> - Load properties file

- **Before:**

```
<bean id="jdbcConfiguration"
    class="o.s.beans.factory.config.PropertiesFactoryBean">
    <property name="location"
            value="classpath:jdbc-production.properties"/>
</bean>
```

- **After:**

```
<util:properties id="jdbcConfiguration"
            location="classpath:jdbc-production.properties"/>
```

<util:*/> Namespace

- **<util:constant /> -** Replacement for FieldRetrievingFactoryBean

```
<util:constant static-field = “java.lang.Boolean.TRUE”/>
```

- **<util:property-path /> –** Replacement for PropertyPathFactoryBean

```
<util:property-path id=“maxTries"
                                    path=“txManager.maxTries"/>
```

<aop:* /> Namespace

- **All things AOP**
  - **Spring AOP support**
  - **AspectJ AOP support**

- **Uses the following stanza to enable support**

```
<?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:aop="http://www.springframework.org/schema/aop"
</beans>
```

<aop:*/> Namespace

- **Before**

```
<bean id="advice"
    class="com.some.MethodBeforeAdviceImpl"/>
<bean id="advisor"
        class="o.s.aop.support.DefaultPointcutAdvisor">
    <property name="advice"><ref
    local="advice"/></property>
    <property name="pointcut">
        <bean
    class="o.s.aop.support.JdkRegexpMethodPointcut">
            <property name="pattern"
    value="foo.*"/>
        </bean>
    </property>
</bean>
```

- **With (ProxyFactoryBean definitions follows)**

<aop:*/> Namespace

- **After**

```
<aop:config>
   <aop:pointcut id="examplePointCut"
        expression="execution (* foo(..))"/>
  <aop:aspect id="exampleAspect" ref="adviceBean">
        <aop:before method="beforeMethod"
              pointcut-ref="examplePointCut"/>
        <aop:after method="afterMethod" pointcut="…" />
        <aop:around method="aroundMethod" pointcut="…"
              args="account,…"/>
        <aop:after-returning returning="variableName" …/>
        <aop:after-throwing throwing="someException" …/>
  </aop:aspect>
</aop:config>
<bean id="adviceBean" class="com.some.Code"/>
```

<aop:*/> Namespace

- Named, reusable pointcut definitions
- Ability to execute POJO code at joinpoints
- Spring AOP advice is managed with <aop:advisor /> element
- Can use regex expression instead of AspectJ
  - <aop:pointcut id="examplePointCut" type="regex" expression="…"/>
- Can force CGLIB use with
  - <aop:config proxy-target-class="true">

<aop:*/> Namespace

- **<aop:aspectj-autoproxy /> - Enables support for @AspectJ annotations.**
  - **Automatically creates proxies for advised beans.**
  - **<include name="regex"/> is used to limit @AspectJ beans used in autoproxying.**
- **<aop:scoped-proxy /> - Creates a proxy of a scoped bean, allowing that bean to participate in DI.**

```
<bean id="scopedBean" class="…" scope="session">
    <aop:scoped-proxy />
</bean>
```

<aop:*/> Namespace

- <aop:spring-configured /> - Turns on support for @Configurable annotation which allows non managed objects to benefit from DI (domain objects for example).
  - Needs AspectJ support

<tx:*/> Namespace

- **Declarative transaction support.**

- **Use the following stanza to enable support**

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx=http://www.springframework.org/schema/tx>
</beans>
```

<tx:*/> Namespace

- **<tx:advice> - Defines transactional semantics of AOP advice to be applied**

<tx:*/> Namespace

- **Before:**

```
<bean id="txProxyTemplate" abstract="true"
    class="o.s.t.i.TransactionProxyFactoryBean">
  <property name="transactionManager"
    ref="transactionManager"/>
    <property name="transactionAttributes">
     <props>
          <prop key="save*">
                    PROPAGATION_REQUIRED
          </prop>
     </props>
    </property>
 </bean>
```

<tx:*/> Namespace

```
<bean id="userService" parent="txProxyTemplate">
  <property name="target">
    <bean class="com.some.comp.user.UserService"/>
   </property>
   <property name="transactionAttributes">
    <props><prop key="save*">
    PROPAGATION_REQUIRED, -UserExistsException </prop>
   </props>
   </property>
  </bean>
    <bean id="groupService" parent="txProxyTemplate">
       <property name="target">
          <bean class="com.some.comp.user.GroupService"/>
       </property>
    </bean>
```

<tx:* /> Namespace

- **After**

```
<bean id="userService" class="com.some.comp.user.UserService" />
<bean id="groupService" class="com.some.comp.user.GroupService" />

<tx:advice id="txAdvceUser">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED"
        isolation="DEFAULT" rollback-for="UserExistsException, ..."/>
    </tx:attributes>
</tx:advice>
<tx:advice id="txAdviceDefault">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

<tx:*/> Namespace

```
<aop:config>
    <aop:pointcut id="serviceTxPoincut"
    expression=
    "execution (* com.some.comp..*Service.*(..))"/>
    <aop:advisor advice-ref="txAdviceUser"
                 pointcut-ref="serviceTxPointcut"/>
    <aop:advisor advice-ref="txAdviceDefault"
                 pointcut-ref="serviceTxPointcut"/>
</aop:config>
```

<tx:* /> Namespace

- **<tx:annotation-driven/> - Turn on @Transactional annotation support.**

<jee:* /> Namespace

- **Used to configure jndi references like data sources, EJBs, etc**

- **Use the following stanza to enable support**

```
<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">
</beans>
```

<jee:* /> Namespace

- **<jee:jndi-lookup /> - Exposes jndi bound object to the application context.**

```
<jee:jndi-lookup id="dataSource"
                 jndi-name="jdbc/MyDataSource"/>
```

- **<jee:local-slsb /> – Configure a reference to Local Stateless Session Bean**

```
<jee:local-slsb id="simpleSlsb"
   jndi-name="ejb/RentalServiceBean"
   business-interface="com.foo.service.RentalService"/>
```

<jee:*/> Namespace

- **<jee:remote-slsb /> – Configure a reference to Remote Stateless Session Bean**

```
<jee:remote-slsb id="complexRemoteEjb"
                 jndi-name="ejb/MyRemoteBean"
                 business-
           interface="com.foo.service.RentalService"
                 cache-home="true"
                 lookup-home-on-startup="true"
                 resource-ref="true"
                 home-
           interface="com.foo.service.RentalService"
                 refresh-home-on-connect-failure="true">
```

<p:* /> Namespace

- **Used to map attributes to the bean properties**

- **Use the following stanza to enable support:**

```
<?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:p="http://www.springframework.org/schema/p">
</beans>
```

<p:* /> Namespace

- **Before:**

```
<bean id="id" class="…User">
    <property name="uid" value="admin"/>
    <property name="group" ref="adminGroup"/>
    <property name="name" value="SuperUser" />
</bean>
```

- **After**

```
<bean id="id" class="…User" p:uid="admin"
                            p:group-ref="adminGroup">
    <property name="name" value="SuperUser"/>
</bean>
```

Creating custom extensions

Steps in the process

1. Create XML document
2. Come up with a namespace
3. Create XML Schema
4. Create NamespaceHandler implementation
5. Create BeanDefinitionParser implementation
   (and/or)
6. Create BeanDefinitionDecorator implementation
7. Create META-INF/spring.handlers
8. Create META-INF/spring.schemas

Why we need a namespace?

- **An XML namespace is a collection of names, identified by a URI reference.**
- **Namespace is used to differentiate identically named elements/attributes in XML document**

```
<foo:root xmlns:foo="foo">
  <foo:name>bar</foo:name>
  <foo:bean>bean</foo:bean>
</foo:root>
```

```
<bar:root xmlns:bar="bar">
  <bar:name>123</bar:name>
  <bar:bean>XYZ</bar:bean>
</bar:root>
```

PropertyOverrideConfigurer Example
(Step 1 – Create XML Document)

- **Before**

```
<bean id="overrideConfigurer"
    class="o.s.b.factory.config.PropertyOverrideConfigurer">
    <property name="location"
          value="classpath:override.properties" />
</bean>
```

- **After**

```
<utils:properties-override-configurer id="propertiesOverride"
    location="classpath:override.properties" />
```

PropertyOverrideConfigurer Example
(Step 2 – Come up with Namespace)

- **Any URI can be used.**

http://www.chariotsolutions.com/spring/schema/utils

PropertyOverrideConfigurer Example
(Step 3 – Create XML Schema)

```
<xsd:element name="properties-override-configurer">
    <xsd:complexType>
        <xsd:complexContent>
        <xsd:extension base="beans:identifiedType">
                <xsd:attribute name="location"
                               type="xsd:string"
                               use="required">
                </xsd:attribute>
        </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:element>
```

PropertyOverrideConfigurer Example
(Step 4 – Create NamespaceHandler implementation)

```
public class UtilsNamespaceHandler extends
    NamespaceHandlerSupport
{
    public void init()
    {
        registerBeanDefinitionParser(
                "properties-override-configurer",
                 new PropertyOverrideParser());
    }
}
```

PropertyOverrideConfigurer Example
(Step 5 – Create BeanDefinitionParser implementation)

```
public class PropertyOverrideParser extends
    AbstractSimpleBeanDefinitionParser
{
    protected Class getBeanClass(Element element)
    {
        return PropertyOverrideConfigurer.class;
    }
}
```

- AbstractSimpleBeanDefinitionParser – Use when there is one-to-one mapping between properties of a bean and attribute names.

PropertyOverrideConfigurer Example
(Step 7 and 8)

- **Create META-INF/spring.handlers**

http\://www.chariotsolutions.com/spring/schema/utils = com.chariot.spring.schema.UtilsNamespaceHandler

  - **Maps namespace URI to NamespaceHandler.**

- **Create META-INF/spring.schemas**

http\://www.chariotsolutions.com/spring/schema/utils/spring-utils.xsd=config/spring-utils.xsd

  - **Maps XML Schema document to a local copy bypassing internet access.**
  - **Needs xsi:schemaLocation declaration in xml document.**

PropertyOverrideConfigurer Example

```
<?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:utils="http://www.chariotsolutions.com/spring/schema/utils"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.chariotsolutions.com/spring/schema/utils
    http://www.chariotsolutions.com/spring/schema/utils/spring-utils.xsd">

    <utils:properties-override-configurer
            id="propertiesOverride"
            location="classpath:META-INF/config/override.properties" />

    <bean id="someBean" class="com.some.BeanWithOverridableProperties" />
</beans>
```

PropertyPlaceholderConfigurer Example (Step 1 – Create XML Document)

- **Before:**

```
<bean id="placeholderConfigurer"
        class="o.s.beans.factory.config.PropertiesPlaceholderConfigurer">
    <property name="localOverride" value="true" />
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
    <property
    name="ignoreResourceNotFound"><value>true</value></property>
    <property name="systemPropertiesModeName"
    value="SYSTEM_PROPERTIES_MODE_OVERRIDE"/>
    <property name="properties">
        <props>
            <prop key="local.test">test1</prop>
            <prop key="location.test.1">test2</prop>
        </props>
    </property>
    <property name="locations”><list>…</list></property>
</bean>
```

PropertyPlaceholderConfigurer Example (Step 1 – Create XML Document)

- **After:**

```
<utils:properties-placeholder-configurer
    id="placeholderConfigurer"  enable-local-override="true"
    ignore-invalid-resource="true" ignore-unresolved="true"
    system-properties-mode="OVERRIDE">
    <utils:local-properties>
        local.test=test1
        location.test.1=test2
    </utils:local-properties>
    <utils:locations>
        classpath:jdbc.properties
        classpath:jpa.properties
    </utils:locations>
</utils:properties-placeholder-configurer>
```

PropertyPlaceholderConfigurer Example (Step 3 – Create XML Schema)

- **XML Schema Definition**

```
<xsd:element name="properties-placeholder-configurer">
    <xsd:complexType>
    <xsd:complexContent>
        <xsd:extension base="beans:identifiedType">
        <xsd:group ref="placeHolderElements" />
        <xsd:attributeGroup ref="placeHolderAttributes" />
        </xsd:extension>
    </xsd:complexContent>
    </xsd:complexType>
</xsd:element>
… (not showing group and attributeGroup definition)
```

PropertyPlaceholderConfigurer Example
(Step 4 – Create NamespaceHandler implementation)

- **Registering BeanDefinitionParser in the same namespace:**

```
registerBeanDefinitionParser(
        "properties-placeholder-configurer",
        new PropertyPlaceholderParser());
```

PropertyPlaceholderConfigurer Example
(Step 5 – Create BeanDefinitionParser implementation)

```
public class PropertyPlaceholderParser extends
    AbstractSingleBeanDefinitionParser
{
    protected Class getBeanClass(Element element)
    {
        return PropertyPlaceholderConfigurer.class;
    }
    protected void doParse(
    Element element, BeanDefinitionBuilder definitionBuilder)
    {
        parseAttributes(element, definitionBuilder);
        parseChildElements(element, definitionBuilder);
    }
```

PropertyPlaceholderConfigurer Example
(Step 5 – Create BeanDefinitionParser implementation)

- **AbstractSingleBeanDefinitionParser – Use when creating a *single* BeanDefinition from relatively complex XML element. That is there is a need to parse inner elements or map XML attributes to corresponding bean properties.**

Other BeanDefinitionParsers

- **AbstractBeanDefinitionParser** – Use this base class when parsing complex XML element into number of BeanDefinitions.
- **BeanDefinitionParser** – can always implement the base interface for the outmost freedom.

Creating BeanDefinitionDecorator

- BeanDefinitonDecorator is used to parse inner custom tags (right under <bean/>) or custom attributes.
- Can transform outer BeanDefinition or return completely different definition.
- Useful to apply custom interceptor chains to single bean (Spring AOP style).
  - Use AbstractInterceptorDrivenBeanDefinitionDecorator for that.

Creating BeanDefinitionDecorator Example

- **Custom XML tag**

```
<bean id="test"
    class="com.comp.common.SimpleCommonBean">
        <utils:synchronized-proxy/>
</bean>
```

- **XML Schema Definition**

```
<xsd:element name="synchronized-proxy">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
</xsd:element>
```

Creating BeanDefinitionDecorator Example

```
public class SynchronizedDecorator extends
    AbstractInterceptorDrivenBeanDefinitionDecorator
{

    protected BeanDefinition
            createInterceptorDefinition(Node node) {
            return new RootBeanDefinition(
                    SynchronizedInterceptor.class);
    }
}
```

Creating BeanDefinitionDecorator Example

- **Register BeanDefinitionDecorator with NamespaceHandler:**

```
registerBeanDefinitionDecorator(
        "synchronized-proxy",
        new SynchronizedDecorator());
```

Summary

If in doubt use DTD format.

Summary

- Minuses
  - Performance hit on start-up
  - One more configuration format to learn
  - Don’t really know what is being created under the hood
  - Limited examples
  - Unproven

Summary

- **Pluses:**
  - **Easier to read, write and understand**
  - **Less verbose**
  - **“Compile” time validation**
  - **Provides better encapsulation**
  - **Highly extensible**
  - **Better platform for documenting configuration (xsd:documentation)**
  - **Tooling support**

Summary

When and Where to use (best practices)

- You are developing a plug-in, library or product based on spring that can be re-used.
- In component based systems – helps to modularize public component services.
- Infrastructure tasks (jmx, jms, etc)
- To minimize repeated copy and paste configurations
- Using spring out of the box tags

Finally

Further reading

- **Spring Reference Documentation -**
  http://static.springframework.org/spring/docs/2.0.x/reference/index.html

Contact

dsklyut@chariotsolutions.com