

Spring Schema Extension eller Spring Domain Specific Languages

Forfatter: Jeppe Cramon – Cramon Consulting (<http://www.cramon.dk>)

Dato: 23/7-2005

Efter at have arbejdet med Spring i lang tid, har jeg lagt mærke til at mine Spring XML konfigurations filer ofte ligner hinanden. Der ud over har de en tendens til at indeholde store mængder XML. Det lugter som en mulighed for abstraktion og genbrug. Springs standard XML format er ret godt til dets formål, nemlig at styre afhængigheder mellem dine komponenter, services, resourcer og lign. Springs XML konfigurations sprog er meget lowlevel, hvilket er påkrævet for at kunne tilbyde den grad af fleksibilitet der er nødvendig.

Som udgangspunkt for mit eksempel tager vi en Spring MVC web applikation, som i stor stil bruger Springs XML konfigurations fil(er). Hensigten, sammenhængen mellem Controllers og Url mappinger mistes hurtigt i de utallige `<bean>` definitioner. Spring-MVC er et meget fleksibel mht. hvordan dine web applikationer kan skrues sammen. Den fleksibilitet har dog den pris, at dine konfigurations filer bliver større og mere komplicerede. Når et projekt først har besluttet sig for hvordan f.eks. en web applikationerne skal konfigureres, bliver man hurtig træt af de omfangsrige konfigurations filer, hvor man nemt kan miste overblikket. Det er min holdning, at det er bedre at konfigurere en MVC applikation i et Domæne Specifikt Sprog (DSL). Jeg vil ikke gå ind i en diskussion om, hvor vidt en domæne specifik XML konfigurations fil kan kaldes et Domæne Specifikt Sprog eller ej. Det er ikke formålet med dette blog entry.

Et eksempel på et mere Domæne Specifikt Sprog for konfiguration af MVC applikationer kan findes i Struts. Her finder du XML konfigurations tags/elementer, der klart mapper til nogle objekter/koncepter i Struts.

```
<form-bean name="accountForm"
type="org.springframework.samples.jspetstore.web.struts.AccountActionForm"/>
```

eller

```
<action path="/shop/addItemToCart"
type="org.springframework.samples.jspetstore.web.struts.AddItemToCartAction"
    name="cartForm" scope="session" validate="false">
    <forward name="success" path="/WEB-INF/jsp/struts/Cart.jsp"/>
</action>
```

Det er efter min mening mere sigende end dette Spring MVC eksempel (som blot er en ud af mange måder at konfigurere URL til Controller mapping i Spring-MVC).

- Først er der indikeret en viewResolver (hvordan bliver view/forward værdien omsat til en navigation til en JSP eller anden controller). Dette er ikke strengt nødvendigt og skal kun ske een gang. Den er medtaget her for at gøre eksemplet mere komplet. At man kan angive en ViewResolver giver Spring-MVC en stor grad af fleksibilitet.
- Der næst er der angivet en Url til controller mapping. Der findes mange forskellige måder at gøre dette på. Et andet eksempel der bruger bean's navn som Url mapping er vist her:

```
<bean id="defaultHandlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/
>

<bean name="/shop/addItemToCart.do"
class="org.springframework.samples.jspetstore.web.spring.AddItemToCartController"
```

>

- Tilsidst er selve controlleren vist, med angivelse af formView og successView (dette afhænger igen af hvilken slags Controller du vælger at implementere. De har forskellige features og krav).

```
</bean>
    ....
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property
name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></p
roperty>
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>

<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/shop/addToItemCart.do">addItemToCartController</prop>
            ...
        </props>
    </property>
</bean>

<bean id="addItemToCartController"
class="org.springframework.samples.jspetstore.web.spring.AddItemToCartController"
>
    <property name="successView" value="Cart"/>
    <property name="formView" value="Cart"/>
    <property name="petStore" ref="petStore"/>
</bean>
```

Da Spring-MVC er så fleksibelt, gør det det også sværere at lave en simpel DSL som dækker alle mulige konfigurations muligheder, så derfor synes jeg også at deres valg at lave konfigurationen i bean XML formatet indtil videre er fornuftigt. Har jeres firma eller jeres projekt i stedet besluttet sig for en konkret måde at konfigurere Spring-MVC, kan I med fordel definere jeres egen Spring-MVC DSL, som derefter kan støtte udviklerne i konfigurationen af applikationerne.

En sådan DSL kunne f.eks.se sådan ud:

```
<spring-mvc>
    <view-definition
        viewClass="org.springframework.web.servlet.view.JstlView"
        prefix="/WEB-INF/jsp"
        suffix=".jsp"/>

    <action path="/shop/addItemToCart.do"

class="org.springframework.samples.jspetstore.web.spring.AddItemToCartController"
>
        <forward name="formView">Cart</forward>
        <forward name="successView">Cart</forward>
        <wiring>
            <property name="petStore" ref="petStore"/>
        </wiring>
    </action>

    <action path="/shop/newAccount.do"
```

```

class="org.springframework.samples.jspetstore.web.spring.AccountFormController">
  <forward name="success">index</forward>
  <wiring>
    <property name="petStore" ref="petStore"/>
    <property name="validator" ref="accountValidator"/>
  </wiring>
</action>
</spring-mvc>

```

For at kunne implementere en sådan DSL startede jeg med en meget simpel template baseret løsning der brugte Java og Velocity. Løsningen, der lige så godt kunne have været lavet i XSLT (med bedre matching muligheder tilføje, men også en mere kompliceret mapnings fil), bruger single pass XPath matching (dvs. Velocity template koden kunne ikke foretage matches, dette kunne kun styres fra mapping file) der konverterede input XML'en til standard Spring bean XML format. Det giver nogle simplere, men også kan så effektive mapnings filer.

Mapningen for Spring-MVC DSL'en ovenfor ser sådan ud:

```

< mappings >
  < resultStart >
    <![CDATA[
      <?xml version="1.0" encoding="UTF-8" ?>
      <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
      <beans>
        <!-- For Spring MVC in our company we've decided to go for the
BeanName url mapping handler
        <bean id="defaultHandlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
      ]]>
    </ resultStart >
    < mapping xpath="/spring-mvc/view-definition">
      <![CDATA[
        <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
          <property name="viewClass">
            #set ($viewType = $element.valueOf("@type"))
            #if ($viewType == "JSTL")

              <value>org.springframework.web.servlet.view.JstlView</value>
              #elseif ($viewType == "JSP")

              <value>org.springframework.web.servlet.view.JspView</value>
              #end
            </property>
          </property>
          name="prefix"><value>${element.valueOf("@prefix")}</value></property>
          name="suffix"><value>${element.valueOf("@suffix")}</value></property>
        </bean>
      ]]>
    </ mapping >
    < mapping xpath="/spring-mvc/action">
      <![CDATA[
        <bean name="${element.valueOf("@path")}"
          class="${element.valueOf("@class")}">
          #set ($forwards = $element.selectNodes("forward"))
          #foreach ($forward in $forwards)
            <property
name="${forward.valueOf("@name")}View"><value>$forward.text</value></property>
          #end
          #set ($wiring = $element.selectSingleNode("wiring"))

```

```

        #if ($wiring)
            #foreach ($wiringElement in ${wiring.elements()})
                $wiringElement.asXML()
            #end
        #end
    </bean>
]]>
</mapping>
</resultEnd>
<![CDATA[
</beans>
]]>
</resultEnd>
</mappings>

```

Dette rækker desværre ikke langt nok for mere avancerede behov, hvorfor denne løsning blev droppet til fordel for en totalt Java drevet løsning, som dog også er mere kompleks at arbejde med.

Det jeg ønsker er komplet XML Schema Extension hvor elementer fra et schema kan dekorere eller bruge elementer fra et andet. Dette kan selvfølgelig misbruges til at lave meget komplekse og rodede konfigurations filer, men det kan også bruges fornuftigt til at lave let læselige og effektive konfigurations filer.

Et simpelt eksempel på Schema extension, er f.eks. et `transactional` element der bruges til at dekorere andre objekter (som f.eks. kunne være services, facader eller DAO'er), hvorved disse objekter bliver gjort transaktionelle. I standard Spring gøres dette ved at bruge `TransactionProxyFactoryBean` (eller `Transactional` annotationen som ikke er vist her) til at wrappe target objekterne. Nedenfor er vist hvordan `TransactionProxyFactoryBean` er brugt i Springs PetStore eksempel applikation:

```

<bean id="baseTransactionProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
    abstract="true">
    <property name="transactionManager"><ref
bean="transactionManager"/></property>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>

<bean id="petStore" parent="baseTransactionProxy">
    <property name="target">
        <bean
class="org.springframework.samples.jpetstore.domain.logic.PetStoreImpl">
            <property name="accountDao" ref="accountDao"/>
            <property name="categoryDao" ref="categoryDao"/>
            <property name="productDao" ref="productDao"/>
            <property name="itemDao" ref="itemDao"/>
            <property name="orderDao" ref="orderDao"/>
        </bean>
    </property>
</bean>

```

I dette eksempel bruges både `parent` og `inner beans` for at øge genbruget, men der er stadig meget XML at skrive, specielt hvis du har mere end en facade.

Ved brug af Spring Xml Schema Extension Parser kan jeg gøre det samme på følgende måde:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/beans"
xmlns:transaction="http://www.springframework.org/transaction"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Spring.xsd"
xsi:schemaLocation= "http://www.springframework.org/transaction
SpringTransaction.xsd">
    <transaction:transactional defaultPropagation="PROPAGATION_REQUIRED"
transactionManager="transactionManager">
        <bean
class="org.springframework.samples.jpetstore.domain.logic.PetStoreImpl">
            <property name="accountDao" ref="accountDao"/>
            <property name="categoryDao" ref="categoryDao"/>
            <property name="productDao" ref="productDao"/>
            <property name="itemDao" ref="itemDao"/>
            <property name="orderDao" ref="orderDao"/>
        </bean>
        <bean
class="org.springframework.samples.jpetstore.domain.logic.OtherFacade">
            ...
        </bean>
    </transaction:transactional>
</beans>

```

Xml Schema Extension Parserens konfiguration er lidt mere kompliceret end standard måden i Spring, men tilgængelig foretager man det ikke så ofte:

```

DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();
XmlNamespaceAwareBeanDefinitionReader beanDefinitionReader = new
XmlNamespaceAwareBeanDefinitionReader(beanFactory);
SchemaExtensionXmlBeanDefinitionParser beanDefinitionParser = new
SchemaExtensionXmlBeanDefinitionParser();

beanDefinitionParser.registerSchemaExtensionParser("http://www.springframework.org/transacti
on", new SpringTransactionParser());
beanDefinitionParser.registerSchemaExtensionParser("http://www.springframework.org/beans"
, new SpringBeansParser());
beanDefinitionReader.setBeanDefinitionParser(beanDefinitionParser);

int numberOfBeansLoaded = beanDefinitionReader.loadBeanDefinitions(new
ClassPathResource("org/springframework/beans/factory/xml/schemaextension/MyConfigFile.xml"));

```

Specifikationen af XML Schemaerne til brug under XML parsning er valgfri og udeladt her.

Parsningen af elementer fra de forskellige namespaces bliver foretaget af Schema Extension Parsers, som styres af en SchemaExtensionXmlBeanDefinitionParser. Hvert namespace har sin egen SchemaExtensionParser. Schema Extension Parseren er event drevet, lidt på samme måde som en SAX parser. Det vil sige at den modtager et metode kald for hver start og slut tag i XML'en. Har du f.eks. et "transactional" tag, vil hoved parseren (SchemaExtensionXmlBeanDefinitionParser) kalde metoderne "startTransactional" og "endTransactional" i den Schema Extension Parser, som er ansvarlig for det namespace "transactional" er defineret i. Start metoden modtager som parameter det XML element der bliver parset, samt et ElementParseContext object der bruges til at dele objekter og bean definitioner. End metode kaldet modtager kun et ElementParseContext objekt.

Her er parseren for transaction namespace:

```

public class SpringTransactionParser implements SchemaExtensionParser {

    public SpringTransactionParser() {
        super();
    }

    public void startTransactional(Element element, ElementParseContext
parseContext) {
        // We tell the parse context that any immediate child contexts
shouldn't register beans in the BeanFactory
        // since we're going to alter them in endTransactional and register
them ourselves.
        // This hack is necessary since Spring doesn't allow you to
remove/unregister beans once they're registered
        parseContext.setChildContextRegisterBeansInFactory(false);

        // Read and store the Transaction Manager and DefaultPropagation for
use in the endTransactional
        String transactionManagerBeanRef =
element.getAttribute("transactionManager");
        parseContext.getAttributes().put("transactionManager",
transactionManagerBeanRef);
        String defaultPropagation =
element.getAttribute("defaultPropagation");
        parseContext.getAttributes().put("defaultPropagation",
defaultPropagation);
    }

    public void endTransactional(ElementParseContext parseContext) {
        String transactionManagerBeanRef = (String)
parseContext.getAttributes().get("transactionManager");
        String defaultPropagation = (String)
parseContext.getAttributes().get("defaultPropagation");

        // None of the beans in the child context have been registered in
the BeanFactory.
        // Here we register them using a new name (the original name
suffixed with "Target").
        // We are then inserting a new Bean into the BeanFactory with the
name of the original bean.
        // This bean will be of type TransactionProxyFactoryBean and it's
target will point
        // to the original bean, effectively creating a Transactional proxy
wrapping the original bean
        for (Iterator beansIterator =
parseContext.getChildParseContext().getRegisteredBeans().iterator();
beansIterator.hasNext();) {
            BeanDefinitionHolder originalBeanDefinitionHolder =
(BeanDefinitionHolder) beansIterator.next();

            // Create the TransactionProxyFactoryBean
            AbstractBeanDefinition beanDefinition =
parseContext.createRootBeanDefinition(TransactionProxyFactoryBean.class);

            beanDefinition.getPropertyValues().addPropertyValue("transactionManager",
new RuntimeBeanReference(transactionManagerBeanRef));
            Properties transactionAttributes = new Properties();
            transactionAttributes.setProperty("*", defaultPropagation);

            beanDefinition.getPropertyValues().addPropertyValue("transactionAttributes",
transactionAttributes);
            beanDefinition.getPropertyValues().addPropertyValue("target",
new RuntimeBeanReference(originalBeanDefinitionHolder.getBeanName() +

```

```

"Target"));

        // Register the proxy bean
        BeanDefinitionHolder trxProxyFactoryBeanHolder = new
BeanDefinitionHolder(beanDefinition,
originalBeanDefinitionHolder.getBeanName());
        parseContext.registerBean(trxProxyFactoryBeanHolder);
        // Register the original bean
        BeanDefinitionHolder originalBeanTargetHolder = new
BeanDefinitionHolder(originalBeanDefinitionHolder.getBeanDefinition(),
originalBeanDefinitionHolder.getBeanName() + "Target");
        parseContext.registerBean(originalBeanTargetHolder);
    }
}
}

```

Her er et eksempel på hvordan dele af Parseren for Spring bean namespace er implementeret:

```

public void startBean(Element element, ElementParseContext parseContext) {
    String id = element.getAttribute("id");
    String className = element.getAttribute("class");

    AbstractBeanDefinition beanDefinition =
parseContext.createRootBeanDefinition(className);

    BeanDefinitionHolder beanDefinitionHolder = new
BeanDefinitionHolder(beanDefinition, id);
    parseContext.registerBean(beanDefinitionHolder);
}

public void endBean(ElementParseContext parseContext) {
}

public void startProperty(Element element, ElementParseContext
parseContext) {
    String propertyName = element.getAttribute("name");

    if (element.hasAttribute("value")) {
        String value = element.getAttribute("value");
        BeanDefinitionHolder beanDefinitionHolder =
parseContext.getCurrentBean();

beanDefinitionHolder.getBeanDefinition().getPropertyValues().addPropertyValue(pr
opertyName, value);
    } else {
        PropertyValueHolder propertyValue = new
PropertyValueHolder(propertyName);
        parseContext.getObjectStack().push(propertyValue);
    }
}

public void endProperty(ElementParseContext parseContext) {
    if
(parseContext.getObjectStack().isTopOfType(PropertyValueHolder.class)) {
        PropertyValueHolder propertyValueHolder =
(PropertyValueHolder) parseContext.getObjectStack().pop();
        BeanDefinitionHolder beanDefinitionHolder =
parseContext.getCurrentBean();

beanDefinitionHolder.getBeanDefinition().getPropertyValues().addPropertyValue(pr

```

```
propertyValueHolder.getName(), propertyValueHolder.getValue());
    } else {
        // NOP - All was handled in the startProperty method
    }
}
```

Det spændende med Schema Extension er, at det nu er muligt at genbruge elementer på mange forskellige måder (ikke alle kombinationer giver dog mening). Du kan f.eks. genbruge Springs elementer inde i dine egne, som i dette eksempel:

```
<dao id="customerDao2" class="com.test.MyDao">
  <beans:constructor-arg>
    <beans:ref bean="dataSource"/>
  </beans:constructor-arg>
</dao>
```

Man bliver nød til at afveje fleksibilitet mod kompleksitet. Jo mere specifik din DSL er, desto mindre kan den genbruges. Tilgængæld kan en meget specifik DSL indkapsle en masse funktionalitet, som normalt ville kræve mange beans hvis det samme skulle gøres i standard Spring bean format. Derfor vil en meget specifik DSL også være mere effektiv til den opgave den er designet for.

Med Schema Extension kan du lave project specifikke konfigurations filer, som kun tillader specifikke elementer og kombinationer. På den måde kan man nemmere kontrollere eller guide udviklerne i forbindelse med konfiguration. Da XML Schemaer er brugt som fundament, er vi mere eller mindre sikrede en nem måde at udvide funktionaliteten på, som behovet viser sig. På den måde kan vi lave specifikke og effektive Spring konfigurations filer.

Valget mellem om man skal lave en DSL eller ej afhænger meget af projektet. Hvor stort er det, hvor meget bliver konfigureret, osv.

Koden til XML Schema Extension er submittet til Spring projektet, som er forslag til hvordan denne funktionalitet kan implementeres. Spring 1.3 forventes at tilbyde en eller anden form for Schema Extension. Ønsker du at kigge nærmere på koden så kontakt mig på jeppe@cramon.dk