Spring in the insurance world

Mario Fusco
mario@exmachina.ch

Spring Day 2008, 14 giugno Cagliari
Agenda

- System architecture
  - The big picture
  - How Spring fits all
- Spring features
  - Decoupling with application events
  - Implementing a workflow with jBPM
  - Managing transactions declaratively
Dependency injection is the primary way that Spring promotes loose coupling among application objects ...
Decoupling dependencies

... but it isn’t the only way
Decoupling with applications events

The loosest coupled way for objects to interact with one another is to publish and listen for application events.

- An **event publisher** object can communicate with other objects without even knowing which objects are listening.
- An **event listener** can react to events without knowing which object published the events.
- In Spring, any bean in the container can be either an event listener, a publisher, or both.
To publish an event first it needs to **define the event** itself:

```java
public class CustomEvent extends ApplicationEvent {
    public CustomEvent(Object source) {
        super(source);
    }
}
```

Next you can **publish the event** through the `ApplicationContext`:

```java
applicationContext.publishEvent(new CustomEvent(this));
```

This means that, in order to publish events, your beans will need to have access to the `ApplicationContext`. The easiest way to achieve it is to **hold a static reference** to the context in a singleton that’s made aware of the container by implementing `ApplicationContextAware`
Listening for application events

To allow a bean to listen for application events, it needs to register it within the Spring context and to implement the `ApplicationListener` interface.

```java
public class CustomListener implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof TypedEvent) {
            // Reacts the the event ...
        }
        // Discards the non interesting event
    }
}
```

Unfortunately, this will make your bean to **listen for ALL the events** triggered inside the container and you’ll have to **discard** the non Interesting ones (the biggest part) typically by some `instanceof`
We fixed this issue by replacing the Spring’s default events multicaster with one that notifies only the interested listener.

```xml
<bean id="applicationEventMulticaster" class="ch.exm.util.event.TypedApplicationEventMulticaster"/>
```
public class TypedApplicationEventMulticaster extends SimpleApplicationEventMulticaster {

public void addApplicationListener(ApplicationListener listener) {
    if (listener instanceof TypedApplicationListener) {
        listenerMap.put(listener.getListViewedEventClass(),
                        listener);
    } else super.addApplicationListener(listener);
}

public void multicastEvent(ApplicationEvent event) {
    NotifyListener(listenerMap.get(event.getListViewClass()));
}

private void notifyListener(TypedApplicationListener listener) {
    getTaskExecutor().execute(new Runnable() {
        public void run() { listener.onTypedEvent(event); }
    });
}

indexes the listeners by listened event type

notifies only the interested listeners ... in a separate thread
Listening for a specific event

Through the `TypedApplicationEventMulticaster` your bean can be notified of just a specific class of events by implementing this interface

```java
interface TypedApplicationListener<T extends ApplicationEvent>
{
    void onTypeDefEvent(T event);
    Class<T> getClassOfListenedEvents();
}
```

or even easier by extending the following abstract class

```java
abstract class TypedApplicationListenerAdapter<T>
    implements TypedApplicationListe
{
    public void onApplicationEvent(ApplicationEvent event) {
        onTypeDefEvent((T) event);
    }
}
```

The event multicaster notifies this listener only for the event of type T.
Implementing a workflow with jBPM
Integrating jBPM and Spring
Implementing a workflow with jBPM

jBPM is the JBoss implementation of a BPM (Business Process Management) system. Its main characteristics and feature are:

- It can be configured with any database and it can be deployed on any application server or used a simple java library
- It allows to configure a workflow process via a simple XML file where workflow’s state and transition are defined as it follows

```xml
<start-state name="start">
    <transition name="start" to="prospect"></transition>
</start-state>

<state name="prospect">
    <transition name="request_approval" to="uw_approval"/>
    <transition name="request_quotation" to="retro_quotation"/>
</state>
```

- It has an extensible engine that executes process definitions with tasks, fork/join nodes, events, timers, automated actions, etc.
An insurance policy lifecycle
Integrating jBPM in Spring

There's a Spring Module that makes it easy to wire jBPM with Spring. It allows jBPM's underlying Hibernate `SessionFactory` to be configured through Spring and jBPM actions to access Spring's context ...

```xml
<bean id="jbpmConfiguration" class="org.springframework.
    workflow.jbpm31.LocalJbpmConfigurationFact,
        <property name="sessionFactory" ref="sessionFactory"/>
        <property name="configuration" value="classpath:jbpm.cfg.xml"/>
    </bean>

... and offers convenient ways of working directly with process definitions as well as jBPM API through the `JbpmTemplate`

```xml
<bean id="jbpmTemplate" class="org.springframework.
    workflow.
        <property name="jbpmConfiguration" ref="jbpmConfiguration"/>
    </bean>
```
Calling jBPM API with JbpmTemplate

JbpmTemplate eases to work with the jBPM API taking care of handling exceptions, the underlying Hibernate session and the jBPM context.

For instance to execute a workflow transition in a transactional way:

```java
jbpmTemplate.signal(processInstance, transactionId);
```

It’s also possible, as with every Spring-style template, to directly access to the native JbpmContext through the JbpmCallback:

```java
public ProcessInstance createProcessInstance() {
  return (ProcessInstance) jbpmTemplate.execute(new JbpmCallback(){
    public Object doInJbp(JbpmContext context) {
      GraphSession g = context.getGraphSession();
      ProcessDefinition def =
        g.findLatestProcessDefinition();
      ProcessInstance instance =
        def.createProcessInstance();
      jbpmContext.save(instance);
      return instance;
    }
  });
}
```
Transactions in Spring
Transactions in Spring

The big picture
Transaction’s attributes

In Spring declarative transactions are implemented through its AOP framework and are defined with the following attributes:

- **Propagation behavior** defines the boundaries of the transaction
- **Rollback rules** define what exception prompt a rollback (by default only the runtime ones) and which ones do not
- **Isolation level** defines how much a transaction may be impacted by the activities of other concurrent transactions

- **Read-only**
- **Timeout**

---

**Read-only & Timeout**

Transaction A - **REQUIRES_NEW**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Propagation</th>
<th>Isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction A</td>
<td>REQUIRED</td>
<td>REPEATABLE_READ</td>
</tr>
</tbody>
</table>

---

Exception? → Commit → Rollback
Choosing a transaction manager

Spring supports transactions **programmatically** and even **declaratively** by proxying beans with AOP. But unlike EJB, that’s coupled with a JTA (Java Transaction API) implementation, it **employs a callback mechanism** that abstracts the actual transaction implementation from the transactional code.

Spring **does not directly manage transactions** but, it comes with a set of transaction managers that **delegate responsibility** for transaction management to a platform-specific transaction implementation.

For example if your application’s persistence is handled by Hibernate then you’ll choose to delegate responsibility for transaction management to an **org.hibernate.Transaction** object with the following manager:

```xml
<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
  <property name="dataSource" ref="dataSource"/>
</bean>
```
Managing transactions declaratively

Spring 2.0 adds **two new kinds** of declarative transactions that can be configured through the elements in the tx namespace:

```
xmlns:tx=http://www.springframework.org/schema/tx
```

by adding the spring-tx-2.0.xsd schema to the Spring configuration file:

```
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
```

The first way to declare transactions is with the `<tx:advice>` XML element:

```xml
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:method name="set*" propagation="REQUIRED" />
  <tx:method name="get*" propagation="SUPPORTS" read-only="true"/>
</tx:advice>
```

This is only the **transaction advice**, but in order to define where it will be applied, we need a **pointcut** to indicate which beans should be advised:

```xml
<aop:config>
  <aop:advisor pointcut="execution(* ..MyBean.*(..))" advice-ref="txAdvice"/>
</aop:config>
```
Defining annotation-driven transactions

The second (and easiest) way to declaratively define transactions in Spring 2.0 is through annotations. This mechanism can be enabled as simple as adding the following line of XML to the application context:

```xml
<tx:annotation-driven transaction-manager="transactionManager"/>
```

This configuration element tells Spring to automatically advise, with the transaction advice, all the beans in the application context that are annotated with `@Transactional`, either at the class level or at the method level. This annotation may also be applied to an interface.

The transaction attributes of the advice are defined by parameters of the `@Transactional` annotation. For example our previously illustrated `WorkflowService` performs a workflow transition in a transactional way:

```java
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
void doWkfTransition(WkfObj wkfObj, String transitionName);
```
Mario Fusco
Head of IT Development
mario@exmachina.ch