Spring Modules

Reference Documentation

Version 0.9

Rob Harrop, Steven Devijver, Costin Leau, Jan Machacek, Thierry Templier, Thomas Risberg, Alex Ruiz, Uri Boness, Gurwinder Singh, Sergio Bossa, Omar Irbouh, Juergen Hoeller, Dave Syer

Copyright Spring Modules Team (c) 2005 - 2007

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

Preface

This document provides a reference guide to Spring Modules' features. Spring Modules contains various add-ons for the core Spring Framework and as such this document assumes that you are already familiar with Spring itself. Since this document is still a work-in-progress, if you have any requests, or comments, please post them on the user mailing list or on the Spring Modules forum [http://forum.springframework.org/forumdisplay.php?f=37]. If you want to report a bug please use the Spring Modules issue tracking [http://opensource2.atlassian.com/projects/spring/browse/MOD].

Before we go on, a few words of gratitude: Chris Bauer (of the Hibernate team) prepared and adapted the DocBook-XSL software in order to be able to create Hibernate's reference guide, also allowing us to create this one.

Chapter 1. Introduction

Spring Modules is a collection of tools, add-ons and modules to extend the Spring Framework [http://www.springframework.org]. The core goal of Spring Modules is to facilitate integration between Spring and other projects without cluttering or expanding the Spring core.

Chapter 2. Ant Integration

2.1. Introduction

This module provides custom Ant artifacts that expose Spring beans into an Ant project in various ways. This is a very poweful idiom for adding rich behaviour to Ant, for example in a code generation step during a build. Can also be used to provide a convenient framework for scripting and automating tasks that require Spring services. More information about Ant can be found at: http://ant.apache.org.

The source code for the examples here is in CVS under `src/etc/test-resources`. They can be run from the ant subdirectory of Springmodules projects using

```
$ ant examples
```

Springmodules Ant is shipped with explicit dependencies on Spring 2.0, but it all works just as well with 1.2.8.

2.2. Setting up Spring Configuration

The first step before using any of the features of ant integration is to configure a Spring BeanFactory with the beans (e.g. services) that you need.

The basic mechanism is provided by the Spring SingletonBeanFactoryLocator. This involves setting up a master BeanFactory which contains beans that are themselves BeanFactory instances. The default search path for the master BeanFactory is `classpath*:beanRefContext.xml`, which means that all files on the classpath called `beanRefContext.xml` will be included.

Inside the master BeanFactory are one or more BeanFactory instances. The active BeanFactory for the custom Ant elements in this package can be chosen by specifying the bean id with the `factoryKey` attribute.

2.2.1. Overriding the BeanFactory locations

The location of the master BeanFactory can be overridden with the `contextRef` attribute of the custom Ant elements provided by this package.

2.2.2. Example BeanFactory Configuration

An example `beanRefContext.xml`:

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

<beans>

  <bean id="test.bootstrap" lazy-init="true"
    class="org.springframework.context.support.ClassPathXmlApplicationContext">
    <!-- bootstrap from bean definitions in this file -->
    <constructor-arg><value>classpath:bootstrapContext.xml</value></constructor-arg>
    <!-- this one refreshes by default, but it is quite lightweight, so OK -->
  </bean>

   <bean id="test.child" lazy-init="true"
```

```
     class="org.springframework.context.support.ClassPathXmlApplicationContext">
     <!-- bootstrap from bean definitions in this file -->
     <constructor-arg>
       <list>
         <value>classpath:childContext.xml</value>
       </list>
     </constructor-arg>
     <constructor-arg value="false"/><!-- do not refresh -->
     <property name="parent" ref="test.bootstrap"/>
  </bean>

</beans>
```

2.3. Exposing a Spring Bean to Ant

The most flexible way to use Spring in an Ant project is to expose a Spring bean as a project reference, and then use it in a normal Ant script target. For this we use the Ant custom type `<springbean>`. The bean is referred to by name, and copied to an Ant project reference with the given id. Example:

```
<project name="test">

  <!-- set up tasks and classpaths first -->

  <target name="script" depends="setup"
    description="Demonstrate exposing a bean as a project reference.">
    <!-- We can pull the bean out in a script by using the project reference -->
    <springbean name="properties" factoryKey="test.bootstrap"
      id="test.bean" />
    <script language="ruby">
      $project.log("test.bean="+$project.getReference("test.bean").toString())
    </script>
    <!-- If we use a valid Java identifier as the id, then it gets translated into a script variable -->
    <springbean name="properties" factoryKey="test.bootstrap"
      id="testBean" />
    <script language="ruby">
      $project.log("testBean['test.name']="+$testBean.value.get("test.name"))
    </script>
  </target>

</project>
```

2.4. Evaluating an Expression on a Spring Bean in Ant

As a simple alternative to writing a script, when the desired operation on the Spring bean is something simple like a method call, we can simply evaluate an expression on the bean using the custom task `<springexec>`. The language used is OGNL [http://www.ognl.org] and the bean is the root of the expression. The expression context also contains references to the Ant project (can be referred to in the expression as `#project`). Again the bean is referred to by name. Example:

```
<project name="test">

  <!-- set up tasks and classpaths first -->

  <target name="expression" depends="setup"
    description="Demonstrate evaluating an expression on a bean.">
    <!-- We can evaluate an OGNL expression with a bean as its root.
    In the example the bean is a Map, so we can call put. -->
    <springexec name="properties" factoryKey="test.bootstrap"
      expression="put('foo','bar')" />
    <!-- The Ant project is exposed in the expression context.
    The bean itself is the root of the expression (#this). -->
    <springexec name="properties" factoryKey="test.bootstrap"
      expression="#project.log(#this)" />
  </target>

</project>
```

2.5. Dependency Injection into a Custom Ant Task

The `<springinject>` task is useful if you want to take advantage of Ant features (e.g. file globbing) or prefer for other reasons to write an Ant Task, but need it to be injected with services that Ant does not know about. You can autowire a task by type or by name (the default) by changing the `autowire` attribute (legal values are "byName" and "byType"). Example:

```

<project name="test">

  <!-- set up tasks and classpaths first -->

  <target name="depend" depends="setup"
    description="Demonstrate autowire dependency injection into a custom task.">
    <taskdef name="test"
      classname="org.springmodules.ant.task.TestTask"
      classpathref="ant.test.classpath" />
    <!-- Inject properties autowire by name into the test task -->
    <springinject taskref="test" factoryKey="test.bootstrap" />
    <!--  Expect to see properties logged -->
    <test id="test" />
  </target>

</project>
```

2.6. Configuring Ant

2.6.1. Definitions

The Ant elements provided by this project are defined in the jar file for this project in a file called `org/springmodules/ant/antlib.xml`.

2.6.2. Classpath

All the custom elements require Spring to be on the classpath (spring-core?). The SpringBeanTask also requires OGNL. The relevant jar files can be added to your `.ant/lib` directory (the standard way of extending the ant classpath), or they can be added using an additional custom task (`springextend`) provided as part of this package.

2.6.3. Example

```

<project name="test" default="script">

  <path id="ant.test.classpath">
    <pathelement location="${target.classes.dir}" />
    <pathelement location="${target.testclasses.dir}" />
    <pathelement location="${target.genclasses.dir}" />
    <path refid="test.classpath" />
  </path>

  <target name="setup" unless="${setup.complete}">
    <taskdef name="springextend"
      classname="org.springmodules.ant.task.ExtendClasspathTask"
      classpath="${basedir}/target/classes" />
    <springextend>
      <path refid="ant.test.classpath" />
    </springextend>
    <taskdef resource="org/springmodules/ant/antlib.xml" />
    <property name="setup.complete" value="true" />
  </target>

</project>
```

Chapter 3. Caching

3.1. Introduction

The *Caching Module* provides a consistent abstraction for performing caching, delivering the following benefits.

- Provides a consistent programming model across different caching APIs such as EHCache [http://ehcache.sourceforge.net/], JBoss Cache [http://www.jboss.com/products/jbosscache], Java Caching System (JCS) [http://jakarta.apache.org/jcs/] and OSCache [http://www.opensymphony.com/oscache].
- Provides a unified, simpler, easier to use, API for programmatic use of caching services than most of these previously mentioned APIs.
- Supports different strategies for declarative caching services.
- The *Caching Module* may be easily extended to support additional cache providers.

3.2. Uses

Caching is frequently used to improve application performance. A good example is the caching of data retrieved from a database. Even though ORM frameworks such as iBATIS [http://www.ibatis.com/] and Hibernate [http://www.hibernate.org/] already provide built-in caching, the *Caching Module* can be useful when executing methods that perform heavy calculations, are time consuming, and/or are resource hungry.

Caching can be added to frameworks without inherent caching support, such as JDBC or Spring JDBC [http://www.springframework.org/docs/reference/jdbc.html].

The *Caching Module* may be used to have more control over your caching provider.

3.3. Configuration

Caching and cache-flushing can be easily configured by following these steps.

1. **Set up the cache provider.** Instead of imposing the use of a particular cache implementation, the *Caching Module* lets you choose a cache provider that best suites the needs of your project.
2. **Enable the caching services.** The *Caching Module* provides two ways to enable caching services.

   a. Declarative caching services.

      - `CacheProxyFactoryBean`.
      - Source-level metadata attributes using Commons-Attributes [http://jakarta.apache.org/commons/attributes/] or JDK 1.5+ Annotations [http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html].
      - AutoProxy with `MethodMapCachingInterceptor` and `MethodMapFlushingInterceptor`.

b. Programmatic use (via a single interface, `org.springmodules.cache.provider.CacheProviderFacade`).

3.4. Cache Provider

The *Caching Module* provides a common interface that centralizes the interactions with the underlying cache provider. Each facade must implement the interface `org.springmodules.cache.provider.CacheProviderFacade` or subclass the template `org.springmodules.cache.provider.AbstractCacheProviderFacade`.

Each strategy has the following properties.

1. **`cacheManager` (required)**

   A cache manager administrates the cache. In general, a cache manager should be able to:

   - Store objects in the cache.
   - Retrieve objects from the cache.
   - Remove objects from the cache.
   - Flush or invalidate one or more regions of the cache, or the whole cache (depending on the cache provider.)

   The *Caching Module* provides factories that allow setting up cache managers and ensure that the created cache managers are properly released and destroyed before the Spring application context is closed.

   - `org.springmodules.cache.provider.jboss.JbossCacheManagerFactoryBean`
   - `org.springmodules.cache.provider.jcs.JcsManagerFactoryBean`
   - `org.springmodules.cache.provider.oscache.OsCacheManagerFactoryBean`

   These factories have a common, **optional** property, **`configLocation`**, which can be any resource used for configuration of the cache manager, such as a file or class path resource.

2. **`failQuietlyEnabled` (optional)**

   If `true`, any exception thrown at runtime by the cache manager will not be rethrown, allowing applications to continue running even if the caching services fail. The default value is `false`: any exception thrown by the cache manager will be propagated and eventually will stop the execution of the application.

3. **`serializableFactory` (optional)**

   Some cache providers, like EHCache and JCS, can only store objects that implement the `java.io.Serializable` interface, which may be necessary when storing objects in the file system or replicating changes in the cache to different nodes in a cluster.

   Such requirement imposes a problem when we need to store in the cache objects that are not

`Serializable` and we do not have control of, for example objects generated by JAXB.

A possible solution could be to "force" serialization on such objects. This can be achieved with a `org.springmodules.cache.serializable.SerializableFactory`. The *Caching Module* currently provides one strategy, `org.springmodules.cache.serializable.XStreamSerializableFactory`, which uses XStream [http://xstream.codehaus.org/] to

- Serialize objects to XML before they are stored in the cache.
- Create objects back from XML after being retrieved from the cache.

This feature is disabled by default (the value of `serializableFactory` is `null`.)

3.4.1. EHCache

EHCache [http://ehcache.sourceforge.net/] can be used as cache provider through the facade `org.springmodules.cache.provider.ehcache.EhCacheFacade`. It must have a `net.sf.ehcache.CacheManager` as the underlying cache manager.

```
<!--
  The created cache manager is an instance of net.sf.ehcache.CacheManager
-->
<bean id="cacheManager"
  class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
</bean>

<bean id="cacheProviderFacade"
  class="org.springmodules.cache.provider.ehcache.EhCacheFacade">
  <property name="cacheManager" ref="cacheManager" />
</bean>
```

For more details about using EHCache with Spring, please refer to this excellent article [http://opensource.atlassian.com/confluence/spring/display/DISC/Caching+the+result+of+methods+using+Spring+and+EH by Omar Irbouh.

3.4.2. JBoss Cache

JBoss Cache [http://www.jboss.com/products/jbosscache] can be used as cache provider through the facade `org.springmodules.cache.provider.jboss.JbossCacheFacade`. It must have a `org.jboss.cache.TreeCache` as the underlying cache manager.

```
<!--
  The created cache manager is a singleton instance of org.jboss.cache.TreeCache
-->
<bean id="cacheManager"
  class="org.springmodules.cache.provider.jboss.JbossCacheManagerFactoryBean">
  <!-- Optional properties -->
  <property name="configLocation" value="classpath:org/springmodules/samples/cache-service.xml" />
</bean>

<bean id="cacheProviderFacade"
  class="org.springmodules.cache.provider.jboss.JbossCacheFacade">
  <property name="cacheManager" ref="cacheManager" />
</bean>
```

3.4.3. Java Caching System (JCS)

JCS [http://jakarta.apache.org/jcs/] can be used as cache provider through the facade `org.springmodules.cache.provider.jcs.JcsFacade`. It must have a `org.apache.jcs.engine.control.CompositeCacheManager` as the underlying cache manager.

```
<!--
  The created cache manager is a singleton instance of org.apache.jcs.engine.control.CompositeCacheManager
-->
<bean id="cacheManager"
  class="org.springmodules.cache.provider.jcs.JcsManagerFactoryBean">
  <!-- Optional properties -->
  <property name="configLocation" value="classpath:org/springmodules/samples/jcs-config.properties" />
</bean>

<bean id="cacheProviderFacade"
  class="org.springmodules.cache.provider.jcs.JcsFacade">
  <property name="cacheManager" ref="cacheManager" />
</bean>
```

3.4.4. OSCache

OSCache [http://www.opensymphony.com/oscache] can be used as cache provider through the facade `org.springmodules.cache.provider.oscache.OsCacheFacade`. It must have a `com.opensymphony.oscache.general.GeneralCacheAdministrator` as the underlying cache manager

```
<!--
  The created cache manager is a singleton instance of
  com.opensymphony.oscache.general.GeneralCacheAdministrator
-->
<bean id="cacheManager"
  class="org.springmodules.cache.provider.oscache.OsCacheManagerFactoryBean">
  <!-- Optional properties -->
  <property name="configLocation" value="classpath:org/springmodules/samples/oscache-config.properties" />
</bean>

<bean id="cacheProviderFacade"
  class="org.springmodules.cache.provider.oscache.OsCacheFacade">
  <property name="cacheManager" ref="cacheManager" />
</bean>
```

Rob Harrop posted an article [http://opensource.atlassian.com/confluence/spring/display/INTEGRATE/OSCache] explaining how to set up OSCache in Spring without using any factory.

3.5. Declarative Caching Services

The *Caching Module* offers declarative caching services powered by Spring AOP [http://www.springframework.org/docs/reference/aop.html].

Declarative caching services offers a non-invasive solution, eliminating any dependencies on any cache implementation from your Java code.

The following sections describe the internal components common to the different strategies for declarative caching services.

3.5.1. Caching Advice

A caching advice applies caching to the return value of advised methods. It first checks that a value returned from a method call, with the same method arguments, is already stored in the cache. If a value is found, it will skip the method call and return the cached value. On the other hand, if the advice cannot find a cached value, it will proceed with the method call, store the return value of the call in the cache and finally return the new cached value.

Methods that do not have a return value (return value is `void`) are ignored, even if they were registered for aspect weaving.

3.5.2. Caching Models

Caching models encapsulate the rules to be followed by the *caching advice* when accessing the cache for object storage or retrieval. The *Caching Module* provides caching models for each of the supported cache providers:

- `org.springmodules.cache.provider.ehcache.EhCacheCachingModel` specifies the name of the cache to use.
- `org.springmodules.cache.provider.jboss.JbossCacheCachingModel` specifies the *fully qualified name* (FQN) of the node of the `TreeCache` to use.
- `org.springmodules.cache.provider.jcs.JcsCachingModel` specifies the name of the cache and (optionally) the group to use.
- `org.springmodules.cache.provider.oscache.OsCacheCachingModel` specifies the names of the groups to use, the cron expression to use to invalidate cache entries and the number of seconds that the object can stay in cache. All these properties are optional.

Caching advices can be configured to have caching models in a `java.util.Map` having each entry defined using standard Spring configuration:

```
<-- property of some caching advice -->
<property name="cachingModels">
  <map>
    <entry key="get*">
      <bean class="org.springmodules.cache.provider.jcs.JcsCachingModel">
        <property name="cacheName" value="someCache" />
        <property name="group" value="someGroup" />
      </bean>
    </entry>
  </map>
</property>
```

The type of caching model must match the chosen cache implementation: the example above must use Java Caching System (JCS) [http://jakarta.apache.org/jcs/] as the cache provider.

Caching advices can also have caching models as `java.util.Properties` resulting in a less verbose configuration:

```
<-- property of some caching advice -->
<property name="cachingModels">
  <props>
    <prop key="get*">cacheName=someCache;group=someGroup</prop>
  </props>
</property>
```

The caching model has been defined as a String in the format *propertyName1=propertyValue1;propertyName2=propertyValue2* which the *Caching Module* will automatically convert into a caching model using a `PropertyEditor` provided by the `CacheProviderFacade`.

The key of each entry is different for each declarative caching service strategy. More details will be provided in further sections.

3.5.3. Caching Listeners

An implementation of the interface `org.springmodules.cache.interceptor.caching.CachingListener`. A listener is notified when an object is stored in the cache. The *Caching Module* does not provide any implementation of this interface.

3.5.4. Key Generator

An implementation of `org.springmodules.cache.key.CacheKeyGenerator`. Generates the keys under which objects are stored in the cache. Only one implementation is provided, `org.springmodules.cache.key.HashCodeCacheKeyGenerator`, which creates keys based on the hash code of the object to store and a unique identifier.

3.5.5. Flushing Advice

A flushing advice flushes one or more groups of the cache, or the whole cache (depending on the cache provider) *before* or *after* an advised method is executed.

3.5.6. Flushing Models

Similar to caching models. Flushing models encapsulate the rules to be followed by the *flushing advice* when accessing the cache for invalidation or flushing. The *Caching Module* provides flushing models for each of the supported cache providers:

- `org.springmodules.cache.provider.ehcache.EhCacheFlushingModel` specifies which caches should be flushed.
- `org.springmodules.cache.provider.jboss.JbossCacheFlushingModel` specifies the FQN of the nodes to be removed from the `TreeCache`.
- `org.springmodules.cache.provider.jcs.JcsFlushingModel` specifies which groups in which caches should be flushed. If the a cache is specified without groups, the whole cache is flushed.
- `org.springmodules.cache.provider.oscache.OsCacheFlushingModel` specifies which groups should be flushed. If none is specified, the whole cache is flushed.

Like caching advices, *flushing advices* can be configured to have flushing models in a `java.util.Map`:

```
<-- property of some Flushing advice -->
<property name="flushingModels">
  <map>
    <entry key="update*">
      <bean class="org.springmodules.cache.provider.jcs.JcsFlushingModel">
        <property name="cacheStructs">
          <list>
            <bean class="org.springmodules.cache.provider.jcs.JcsFlushingModel$CacheStruct">
```

```
              <property name="cacheName" value="someCache" />
              <property name="groups" value="group1,group2" />
            </bean>
          </list>
        </property>
      </bean>
    </entry>
  </map>
</property>
```

The type of flushing model must match the chosen cache implementation: the example above must use Java Caching System (JCS) [http://jakarta.apache.org/jcs/] as the cache provider.

Flushing advices can also have flushing models as `java.util.Properties` resulting in a less verbose configuration:

```

<-- property of some flushing advice -->
<property name="flushingModels">
  <props>
    <prop key="update*">cacheName=someCache;group=group1,group2</prop>
  </props>
</property>
```

The flushing model has been defined as a String in the format *propertyName1=propertyValue1;propertyName2=propertyValue2* which the *Caching Module* will automatically convert into a flushing model using a `PropertyEditor` provided by the `CacheProviderFacade`.

The key of each entry is different for each declarative caching service strategy. More details will be provided in further sections.

3.6. Strategies for Declarative Caching Services

The following sections describe the different strategies for declarative caching services provided by the *Caching Module*.

3.6.1. CacheProxyFactoryBean

A `CacheProxyFactoryBean` applies caching services to a single bean definition, performing aspect weaving using a `NameMatchCachingInterceptor` as *caching advice* and a `NameMatchFlushingInterceptor` as *flushing advice*.

```

<!-- Using a EHCache cache manager -->
<bean id="cacheProviderFacade" class="..." />

<bean id="cacheableServiceTarget"
  class="org.springmodules.cache.integration.CacheableServiceImpl">
  <property name="names">
    <list>
      <value>Luke Skywalker</value>
      <value>Leia Organa</value>
    </list>
  </property>
</bean>

<bean id="cacheableService"
  class="org.springmodules.cache.interceptor.proxy.CacheProxyFactoryBean">
  <property name="cacheProviderFacade" ref="cacheProviderFacade" />
  <property name="cachingModels">
    <props>
      <prop key="get*">cacheName=testCache</prop>
    </props>
  </property>
  <property name="flushingModels">
    <props>
      <prop key="update*">cacheNames=testCache</prop>
    </props>
  </property>
  <property name="cachingListeners">
    <list>
      <ref bean="cachingListener" />
    </list>
  </property>
  <property name="target" ref="cacheableServiceTarget" />
</bean>
```

In the above example, *cacheableServiceTarget* is the *advised* or *proxied* object, i.e. the bean to apply caching services to.

The caching interceptor will use a `NameMatchCachingModelSource` to get the caching models defining the caching rules to be applied to specific methods of the proxied class. In our example, it will apply caching to the methods starting with the text "get."

In a similar way, the flushing interceptor will use a `NameMatchFlushingModelSource` to get the flushing models defining the flushing rules to be applied to specific methods of the proxied class. In our example, it will flush the cache "testCache" after executing the methods starting with the text "update."

3.6.2. Source-level Metadata-driven Autoproxy

Autoproxying is driven by metadata. This produces a similar programming model to Microsoft's .Net `ServicedComponents`. AOP proxies for caching services are created automatically for the beans containing source-level, caching metadata attributes. The *Caching Module* supports metadata provided by Commons-Attributes [http://jakarta.apache.org/commons/attributes/] and JDK 1.5+ Annotations [http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html]. Both approaches are very flexible, because metadata attributes are restricted to describe *whether* caching services should be applied instead of describing *how* caching should occur. The *how* is described in the Spring configuration file.

Setting up *autoproxy [http://www.springframework.org/docs/reference/aop.html#aop-autoproxy]* is quite simple:

```
<bean id="autoproxy"
  class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />
```

3.6.2.1. Jakarta Commons-Attributes

The attributes `org.springmodules.cache.interceptor.caching.Cached` and `org.springmodules.cache.interceptor.flush.FlushCache` are used to indicate that an interface, interface method, class, or class method should be target for caching services.

```
public class CacheableServiceImpl implements CacheableService {

  /**
   * @@org.springmodules.cache.interceptor.caching.Cached(modelId="testCaching")
   */
  public final String getName(int index) {
    // some implementation
  }

  /**
   * @@org.springmodules.cache.interceptor.flush.FlushCache(modelId="testFlushing")
   */
  public final void updateName(int index, String name) {
    // some implementation
  }
}
```

Now we need to tell Spring to apply caching services to the beans having Commons-Attributes metadata:

```

<bean id="attributes"
  class="org.springframework.metadata.commons.CommonsAttributes" />

<bean id="cachingInterceptor"
  class="org.springmodules.cache.interceptor.caching.MetadataCachingInterceptor">
  <property name="attributes" ref="attributes" />
  <property name="cacheProviderFacade" ref="cacheProviderFacade" />
  <property name="cachingListeners">
    <list>
      <ref bean="cachingListener" />
    </list>
  </property>
  <property name="cachingModels">
    <props>
      <prop key="testCaching">cacheName=testCache</prop>
    </props>
  </property>
</bean>

<bean id="cachingAttributeSourceAdvisor"
  class="org.springmodules.cache.interceptor.caching.CachingAttributeSourceAdvisor">
  <constructor-arg ref="cachingInterceptor" />
</bean>

<bean id="flushingInterceptor"
  class="org.springmodules.cache.interceptor.flush.MetadataFlushingInterceptor">
  <property name="attributes" ref="attributes" />
  <property name="cacheProviderFacade" ref="cacheProviderFacade" />
  <property name="flushingModels">
    <props>
      <prop key="testFlushing">cacheNames=testCache</prop>
    </props>
  </property>
</bean>

<bean id="flushingAttributeSourceAdvisor"
  class="org.springmodules.cache.interceptor.flush.FlushingAttributeSourceAdvisor">
  <constructor-arg ref="flushingInterceptor" />
</bean>

<!-- Set up the objects to apply caching to -->
<bean id="cacheableService"
  class="org.springmodules.cache.integration.CacheableServiceImpl">
  <property name="names">
    <list>
      <value>Luke Skywalker</value>
      <value>Leia Organa</value>
    </list>
  </property>
</bean>
```

The property `modelId` of the metadata attribute `Cached` should match the id of a caching model configured in the caching advice (in our example the bean with id *cachingInterceptor*.) This way the caching advice will know which caching model should use and how caching should be applied. In the above example, the caching advice will store in the EHCache *testCache* the return value of the method `getName`.

The same matching mechanism is applied to flushing models. The property `modelId` of the metadata attribute `FlushCache` shoule match the id of a flushing model configured in the flushing advice (the bean with id *flushingInterceptor*.) The flushing advice will know which flushing model to use. In the above example, the EHCache *testCache* will be flushed after executing the method `updateName`.

Usage of Commons-Attributes requires an extra compilation step which generates the code necessary to access metadata attributes. Please refer to its documentation [http://jakarta.apache.org/commons/attributes/] for more details.

3.6.2.2. JDK 1.5+ Annotations

Source-level metadata attributes can be declared using JDK 1.5+ Annotations:

```
public class TigerCacheableService implements CacheableService {

  @Cacheable(modelId = "testCaching")
  public final String getName(int index) {
    // some implementation.
  }

  @CacheFlush(modelId = "testFlushing")
  public final void updateName(int index, String name) {
    // some implementation.
  }
}
```

The annotations `org.springmodules.cache.annotations.Cacheable` and `org.springmodules.cache.annotations.CacheFlush` work exactly the same as their Commons-Attributes counterparts. Configuration in the Spring context is also very similar:

```
<bean id="cachingAttributeSource"
  class="org.springmodules.cache.annotations.AnnotationCachingAttributeSource">
</bean>

<bean id="cachingInterceptor"
  class="org.springmodules.cache.interceptor.caching.MetadataCachingInterceptor">
  <property name="cacheProviderFacade" ref="cacheProviderFacade" />
  <property name="cachingAttributeSource" ref="cachingAttributeSource" />
  <property name="cachingListeners">
    <list>
      <ref bean="cachingListener" />
    </list>
  </property>
  <property name="cachingModels">
    <props>
      <prop key="testCaching">cacheName=testCache</prop>
    </props>
  </property>
</bean>

<bean id="cachingAttributeSourceAdvisor"
  class="org.springmodules.cache.interceptor.caching.CachingAttributeSourceAdvisor">
  <constructor-arg ref="cachingInterceptor" />
</bean>

<bean id="flushingAttributeSource"
  class="org.springmodules.cache.annotations.AnnotationFlushingAttributeSource">
</bean>

<bean id="flushingInterceptor"
  class="org.springmodules.cache.interceptor.flush.MetadataFlushingInterceptor">
  <property name="cacheProviderFacade" ref="cacheProviderFacade" />
  <property name="flushingAttributeSource" ref="flushingAttributeSource" />
  <property name="flushingModels">
    <props>
      <prop key="testFlushing">cacheNames=testCache</prop>
    </props>
  </property>
</bean>

<bean id="flushingAttributeSourceAdvisor"
  class="org.springmodules.cache.interceptor.flush.FlushingAttributeSourceAdvisor">
  <constructor-arg ref="flushingInterceptor" />
</bean>
```

```
<!-- Set up the objects to apply caching to -->
<bean id="cacheableService"
  class="org.springmodules.cache.annotations.TigerCacheableService">
  <property name="names">
    <list>
      <value>Luke Skywalker</value>
      <value>Leia Organa</value>
    </list>
  </property>
</bean>
```

By using JDK 1.5+ Annotations, we don't need the extra compilation step (required by Commons-Attributes.) The only downside is we can not use Annotations with JDK 1.4.

3.6.3. BeanNameAutoProxyCreator

```
<bean id="cachingInterceptor"
  class="org.springmodules.cache.interceptor.caching.MethodMapCachingInterceptor">
  <property name="cacheProviderFacade" ref="cacheProviderFacade" />
  <property name="cachingListeners">
    <list>
      <ref bean="cachingListener" />
    </list>
  </property>
  <property name="cachingModels">
    <props>
      <prop key="org.springmodules.cache.integration.CacheableService.get*">cacheName=testCache</prop>
    </props>
  </property>
</bean>

<bean id="flushingInterceptor"
  class="org.springmodules.cache.interceptor.flush.MethodMapFlushingInterceptor">
  <property name="cacheProviderFacade" ref="cacheProviderFacade" />
  <property name="flushingModels">
    <props>
      <prop key="org.springmodules.cache.integration.CacheableService.update*">cacheNames=testCache</prop>
    </props>
  </property>
</bean>

<bean
  class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames">
    <list>
      <idref local="cacheableService"/>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <value>cachingInterceptor</value>
      <value>flushingInterceptor</value>
    </list>
  </property>
</bean>

<bean id="cacheableService"
  class="org.springmodules.cache.integration.CacheableServiceImpl">
  <property name="names">
    <list>
      <value>Luke Skywalker</value>
      <value>Leia Organa</value>
    </list>
  </property>
</bean>
```

Assuming that we already have a `CacheProviderFacade` instance in our ApplicationContext, the first thing we need to do is create the *caching advice* `MethodMapCachingInterceptor` and the *flushing advice* `MethodMapFlushingInterceptor` to use. AOP proxies are created for the objects which match the given fully qualified class name and method name (which accepts wildcards.)

Once we have the advices, we feed them to a `BeanNameAutoProxyCreator` along with the names of the beans in the ApplicationContext we want to apply caching services to.

3.7. Programmatic Use

First, we need to configure a `org.springmodules.cache.provider.CacheProviderFacade` in the Spring ApplicationContext (please refer to Section 3.4, “Cache Provider” for more details.) Then we need to obtain a reference to it and call any of this methods from our Java code:

```
void cancelCacheUpdate(Serializable key) throws CacheException;

void flushCache(FlushingModel model) throws CacheException;

Object getFromCache(Serializable key, CachingModel model) throws CacheException;

boolean isFailQuietlyEnabled();

void putInCache(Serializable key, CachingModel model, Object obj) throws CacheException;

void removeFromCache(Serializable key, CachingModel model) throws CacheException;
```

Chapter 4. Commons Validator

Warning

This module is deprecated. Starting from version 0.4 this module is integrated with the validation module. All classes in this module where deprecated and in version 0.5 this module will be removed as a whole.

Please refer to the documentation of the validation module (Chapter 9) for information of how to use the commons validator. Not much has changed in general, only the packages names and some bug fixes.

4.1. Introduction

The Commons Validator is a library that allows you to perform validation based on rules specified in XML configuration files.

TODO: Describe the concepts of Commons Validator in more details.

4.2. Configure an Validator Factory

Firstly you need to configure the Validator Factory which is the factory to get Validator instances. To do so, the support provides the class DefaultValidatorFactory in the package org.springmodules.commons.validator.

You need to specify with the property validationConfigLocations the file containing the Commons Validator rules and the file containing the validation rules specific to the application.

The following code shows how to configure this factory.

```
<bean id="validatorFactory"
      class="org.springmodules.commons.validator.DefaultValidatorFactory">
  <property name="validationConfigLocations">
    <list>
      <value>/WEB-INF/validator-rules.xml</value>
      <value>/WEB-INF/validation.xml</value>
    </list>
  </property>
</bean>
```

4.3. Use a dedicated validation-rules.xml

The file *validation-rules.xml* must contain Commons Validator elements based on classes provided by the support of this framework in Spring Modules.

For example, the configuration of the entities "required" and "requiredif" must be now in the *validation-rules.xml* file.

```
<validator name="required"
          classname="org.springmodules.commons.validator.FieldChecks"
          method="validateRequired"
          methodParams="java.lang.Object,
                        org.apache.commons.validator.ValidatorAction,
                        org.apache.commons.validator.Field,
                        org.springframework.validation.Errors"
          msg="errors.required">

  <javascript><![CDATA[
    (...)
  ]]></javascript>
</validator>

<validator name="requiredif"
           classname="org.springmodules.commons.validator.FieldChecks"
           method="validateRequiredIf"
           methodParams="java.lang.Object,
                         org.apache.commons.validator.ValidatorAction,
                         org.apache.commons.validator.Field,
                         org.springframework.validation.Errors,
                         org.apache.commons.validator.Validator"
          msg="errors.required">
</validator>
```

The validation sample of the distribution provides a complete *validation-rules.xml* based on the classes of the support.

You must note that the support of *validwhen* is not provided at the moment in the support. However, some codes are provides in JIRA. For more informations, see the issues MOD-38 [http://opensource2.atlassian.com/projects/spring/browse/MOD-38] and MOD-49 [http://opensource2.atlassian.com/projects/spring/browse/MOD-49].

4.4. Configure a Commons Validator

Then you need to configure the Validator itself basing the previous Validator Factory. It corresponds to an adapter in order to hide Commons Validator behind a Spring Validator.

The following code shows how to configure this validator.

```
<bean id="beanValidator" class="org.springmodules.commons.validator.DefaultBeanValidator">
  <property name="validatorFactory" ref="validatorFactory"/>
</bean>
```

4.5. Server side validation

Spring MVC provides the implementation *SimpleFormController* of the interface *Controller* in order to process HTML forms. It allows a validation of informations processing by the controller by using the property v*alidator* of the controller. In the case of Commons Validator, this property must be set with the bean *beanValidator* previously configured.

The following code shows how to configure a controller which validates a form on the server side using the support of Commons Validator.

```
<bean id="myFormController" class="org.springmodules.sample.MyFormController">
  (...)
  <property name="validator" ref="beanValidator"/>
  <property name="commandName" value="myForm"/>
  <property name="commandClass" value="org.springmodules.sample.MyForm"/>
  (...)
</bean>
```

The *beanValidator* bean uses the value of the property *commandClass* of the controller to select the name of the form tag in the *validation.xml* file. The configuration is not based on the *commandName* property. For example, with the class name *org.springmodules.sample.MyForm*, Commons Validator must contain a form tag with *myForm* as value of the name property. The following code shows the contents of this file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons Validator Rules Configuration 1.1//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_1.dtd">

<form-validation>
  <formset>
    <form name="myForm">
      <field property="field1" depends="required">
        <arg0 key="error.field1" />
      </field>
      <field property="field2" depends="email">
        <arg0 key="error.field2" />
      </field>
    </form>
  </formset>
</form-validation>
```

4.6. Client side validation

The support of Commons Validator in Spring Modules provides too the possibility to use a client side validation. It provides a dedicated taglib to generate the validation javascript code. To use this taglib, we firstly need to declare it at the beginnig of JSP files as following.

```
<%@ tglib uri="http://www.springmodules.org/tags/commons-validator" prefix="validator" %>
```

You need then to include the generated javascript code in the JSP file as following by using the *javascript* tag.

```
<validator:javascript formName="account"
    staticJavascript="false" xhtml="true" cdata="false"/>
```

At last, you need to set the *onSubmit* attribute on the *form* tag in order to trigger the validation on the submission of the form.

```
<form method="post" action="(...)" onsubmit="return validateMyForm(this)">
```

Chapter 5. db4o

5.1. Introduction

db4o module facilitates integration between Spring framework and db4o [http://www.db4o.com/about/productinformation/], allowing easier resource management, DAO implementation support and transaction strategies. In many respects, this modules is similar in structure, naming and functionality to Spring core modules for Hibernate, JPA or JDO - users familiar with Spring data access packages should feel right at home when using db4o spring integration.

As samples, a web application named *Recipe Manager* and some examples 'converted' from db4o distribution (mainly chapter 1) are available.

5.2. Configuration

Before being used, db4o has to be configured. db4o module makes it easy to externalize db4o configuration (be it client or server) from the application into Spring application context xml files, reducing the code base and allowing decoupling of application from the environment it runs in. The core class for creating db4o's `ObjectContainer` is the `ObjectContainerFactoryBean` . Based on the various parameter passed to it, the objectcontainer can be created from a db4o database file, from an `ObjectServer` or based on a `Configuration` object.

5.2.1. Configuring an ObjectContainer

The FactoryBean will create `ObjectContainers` based on its properties, using the algorithm below:

1. if the databaseFile is set, a local file based client will be created
2. if memoryFile is set, a local memory based client will be instantiated
3. if a server property is set, a client `ObjectContainer` will be created within the VM using the given server object
4. if all the above fail, a connection to a (possibly) remote machine will be opened using the hostName, port, user and password properties.

For example in order to create a memory based file `ObjectContainer` , the following configuration can be used:

```
<bean id="memoryContainer" class="org.db4ospring.ObjectContainerFactoryBean">
 <property name="memoryFile">
     <bean class="com.db4o.ext.MemoryFile"/>
 </property>
</bean>
```

For an `ObjectContainer` connected to a (remote) server:

```
<bean id="remoteServerContainer" class="org.db4ospring.ObjectContainerFactoryBean">
 <property name="hostName" value="localhost"/>
 <property name="port" value="123"/>
 <property name="user" value="foo"/>
 <property name="password" value="bar"/>
```

```
</bean>
```

While creating a database file based, local `ObjectContainer` can be achieved using a bean definition such as:

```
<bean id="fileContainer" class="org.db4ospring.ObjectContainerFactoryBean">
 <property name="databaseFile" value="classpath:db4o-file.db"/>
</bean>
```

For local configurations, it is possible to pass a db4o `Configuration` object (if no configuration is given, as in the examples above, the JVM global configuration is being used):

```
<bean id="myContainer">
  <property name="configuration" ref="customizedConfiguration"/>
  ...
</bean>
```

See the db4o configuration section for more information on defining and using a `Configuration` object.

5.2.2. Configuring an ObjectServer

ObjectServerFactoryBean can be used for creating and configuring an `ObjectServer` :

```
<bean id="server" class="org.db4ospring.ObjectServerFactoryBean">
 <property name="userAccessLocation" value="user-access.properties"/>
 <property name="databaseFile" value="file://./db4o.db"/>
 <property name="port" value="123"/>
</bean>
```

Note the *userAccessLocation* property which specifies the location of a `Properties` file that will be used for user acess - the properties file keys will be considered the user names while the values as their passwords.

5.2.3. Using db4o's Configuration object

When a complex configuration is required, `ConfigurationFactoryBean` offers an extensive list of db4o parameters which can be used to customize db4o `ObjectContainers` . The `FactoryBean` can work with the global JVM db4o configuration, a cloned configuration from the global one or a newly created (which ignored the settings on the global) based on the `configurationCreationMode` parameter:

```
<bean id="configurationObject" class="org.db4ospring.ConfigurationFactoryBean">
 <property name="messageLevel" value="2"/>
 <property name="activationDepth" value="3"/>
 <!-- possible values are NEW, CLONED and GLOBAL -->
 <property name="configurationCreationMode value="NEW"/>
</bean>
```

5.3. Inversion of Control: Template and Callback

The core classes of db4o module that are used in practice, are `Db4oTemplate` and `Db4oCallback` . The template translates db4o exceptions into Spring Data Access exception hierarchy (making it easy to integrate db4o with other persistence frameworks supported by Spring) and maps most of db4o `ObjectContainer` and `ExtObjectContainer` interface methods, allowing one-liners:

```
db4oTemplate.activate(personObject, 4); // or
db4oTemplate.releaseSemaphore("myLock");
```

5.4. Transaction Management

db4o module provides integration with Spring's excellent transaction support through `Db4oTransactionManager` class. Since db4o statements are always executed inside a transaction, Spring transaction demarcation can be used for commiting or rolling back the running transaction at certain points during the execution flow.

Consider the following example (using Spring 2.0 transactional namespace):

```
<?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"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="
http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

 <!-- this is the service object that we want to make transactional -->

 <bean id="fooService" class="x.y.service.DefaultFooService"/>

 <!-- the transactional advice (i.e. what 'happens'; see the <aop:advisor/> bean below) -->
 <tx:advice id="txAdvice" transaction-manager="txManager">
 <!-- the transactional semantics... -->
   <tx:attributes>
   <!-- all methods starting with 'get' are read-only -->
     <tx:method name="get*" read-only="true"/>
   <!-- other methods use the default transaction settings (see below) -->
     <tx:method name="*"/>
   </tx:attributes>
 </tx:advice>

 <!--
    ensure that the above transactional advice runs for any execution of an
    operation defined by the FooService interface
  -->
 <aop:config>
  <aop:pointcut id="fooServiceOperation" expression="execution(*x.y.service.FooService.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
 </aop:config>

 <bean id="txManager" class="org.db4ospring.Db4oTransactionManager">
  <property name="objectContainer" ref="objectContainer"/>
 </bean>

// more bean definition follow
</beans>
```

5.5. Outside the Spring container

It is important to note that db4o-spring classes rely as much as possible on db4o alone and they work with objects either configured by the developer or by Spring framework. The template as well as the `FactoryBeans` can be instantiated either by Spring or created programatically through Java code.

Chapter 6. Flux

6.1. Introduction

Flux is a job scheduler, workflow engine, and business process management (BPM) engine. More information about Flux can be found at: http://www.fluxcorp.com.

6.2. Exposing Flux as a Spring Bean

A Flux Spring bean can be created using one of the following methods:

- Use the following configuration to create a Flux spring bean with the default configuration options

```
<bean id="fluxEngineBean"
    class="org.springmodules.scheduling.flux.EngineBean"
    destroy-method="dispose">
</bean>
```

- Use the following configuration to create a Flux spring bean from the configuration properties that are defined in the "fluxconfig.properties" file.

```
<bean id="fluxEngineBeanFromConfigurationFile"
      class="org.springmodules.scheduling.flux.EngineBean"
      destroy-method="dispose">

  <constructor-arg type="java.lang.String">
     <value>fluxconfig.properties</value>
  </constructor-arg>

</bean>
```

  An XML Engine bean and a Configuration bean can also be created in similar ways. To create these beans, use the "org.springmodules.scheduling.flux.XmlEngineBean" and "org.springmodules.scheduling.flux.ConfigurationBean" classes.

6.3. Getting Help

If you have any questions, feel free to contact our support team.

Email

support@fluxcorp.com

Telephone

+1 (406) 656-7398

Chapter 7. Hivemind Integration

7.1. Introduction

Hivemind is lightweight container providing IoC capabilities similar to Spring. More information about HiveMind can be found at: http://jakarta.apache.org/hivemind [???].

7.2. Configure an Hivemind `Registry`

In HiveMind, the `Registry` is the central location from which your application can gain access to services and configuration data. The `RegistryFactoryBean` allows for a HiveMind `Registry` to be configured and started within the Spring ApplicationContext:

There are two ways configure this `Registry` with the `RegistryFactoryBean` class:

- No configuration location is specified. In this case, Hivemind looks for an XML file named hivemodule.xml in the META-INF directory.
- One or more configuration file locations are specified. In this case, Spring Modules will use these configuration files to configure `Registry` instance.

The code below shows how to configure a `RegistryFactoryBean` that loads `Registry` configuration from a file called configuration.xml:

```
<bean id="hivemindRegistry" class="org.springmodules.hivemind.RegistryFactoryBean">
  <property name="configLocations">
    <value>configuration.xml</value>
  </property>
</bean>
```

The `RegistryFactoryBean` uses Spring's resource abstraction layer allowing you to specify any valid Spring `Resource` path for the configLocations property.

7.3. Exposing HiveMind Services as Spring Beans

Using the `ServiceFactoryBean` it i spossible to expose any service defined in a HiveMind `Registry` to your application as a Spring bean. This can be desirable if you want to make use of features found in both products but you want your application to code only to one.

The ServiceFactoryBean class requires access to a HiveMind `Registry`, and as such, you generally need to configure both a `RegistryFactoryBean` and a `ServiceFactoryBean` as shown below:

```
<bean id="registry" class="org.springmodules.hivemind.RegistryFactoryBean">
  <property name="configLocations">
    <value>configuration.xml</value>
  </property>
</bean>

<bean id="sampleService" class="org.springmodules.hivemind.ServiceFactoryBean">
  <property name="registry">
    <ref local="registry"/>
  </property>
  <property name="serviceInterface">
    <value>org.springmodules.samples.hivemind.service.ISampleService</value>
  </property>
  <property name="serviceName">
    <value>interfaces.SampleService</value>
  </property>
</bean>
```

Whether you define both `serviceInterface` and `serviceName` or just `serviceInterface` depends on how your HiveMind `Registry` is configured. Consult the HiveMind documentation for more details on how HiveMind services are identified and accessed.

Chapter 8. JavaSpaces

8.1. Introduction

JavaSpaces module offers Spring-style services, like transaction management, template, callback and interceptor as well as remoting services to JavaSpaces based environments.

8.2. JavaSpaces configuration

One challenge when dealing with Jini-based environments(like JavaSpaces) is retrieving the appropriate services. JavaSpaces module addresses this problem by providing generic as well as customized classes to work with various JavaSpaces implementations as well as Jini services in a simple and concise manner.

8.2.1. Using specialized classes

JavaSpaces modules offers configuration support out of the box for:

8.2.1.1. Blitz

Blitz [http://www.dancres.org/bjspj/docs/docs/blitz.html] is an open-source implementation of JavaSpaces. JavaSpaces module provides two Blitz-based factory beans:

```
<beans>
...
 <!-- Blitz based localSpace -->
 <bean id="blitzLocalSpace" class="org.springmodules.javaspaces.blitz.LocalSpaceFactoryBean">
  <property name="configuration"
     value="classpath:/org/springmodules/javaspaces/blitz/blitz.config"/>
 </bean>
 <!-- Null space - useful for tests -->
 <bean id="blitzLocalSpace" class="org.springmodules.javaspaces.blitz.NullSpaceFactoryBean"/>
</beans>
```

8.2.1.2. GigaSpaces

GigaSpaces [http://www.gigaspaces.com/] is a commercial JavaSpaces implementation that provides a free Community Edition. See the dedicated documentation for more support information and the online Wiki documentation which is available at http://gigaspaces.com/wiki/display/GS/Spring

```
<beans>
 <!-- GigaSpaces localSpace -->
 <bean id="gigaSpaceLocalSpace"
    class="org.springmodules.javaspaces.gigaspaces.GigaSpacesFactoryBean">
  <property name="urls">
   <list>
    <value>/./myCache?properties=gs&amp;</value>
   </list>
  </property>
 </bean>
</beans>
```

8.2.2. Using a generic Jini service

For generic Jini services (including other JavaSpaces implementations), JiniServiceFactoryBean can be used for retrieval:

```
<beans>
  <bean id="anotherSpace" class="org.springmodules.jini.JiniServiceFactoryBean">
   <property name="serviceClass" value="net.jini.space.JavaSpace"/>
   <property name="serviceName" value="SomeJavaSpacesImplementation"/>
   <property name="timeout" value="10000"/>
  </bean>
</beans>
```

8.3. Inversion of Control: JavaSpaceTemplate and JavaSpaceCallback

JavaSpaceTemplate is one of the core classes of the JavaSpaces module. It allows the user to work directly against the native JavaSpace API in a consistent manner, handling any exceptions that might occur and taking care of the ongoing transaction (if any) as well as converting the exceptions into Spring DAO and Remote exception hierarchy. The template can be constructed either programmatically or declaratively (through Spring's xml) and requires a JavaSpace implementation instance:

```
<beans>
  <bean id="javaSpace" class="...">
  ...
  </bean>
  <bean id="spaceTemplate" class="org.springmodules.javaspaces.JavaSpaceTemplate">
   <property name="space" ref="javaSpace"/>
  </bean>
</beans>
```

Once constructed the *JavaSpaceTemplate* offers shortcut methods to the JavaSpace interface as well as native access:

```
spaceTemplate.execute(new JavaSpaceCallback() {
  public Object doInSpace(JavaSpace js,Transaction transaction)
    throws RemoteException, TransactionException,
           UnusableEntryException, InterruptedException {
      ...
      Entry myEntry = ...;
      js.write(myEntry, transaction, Lease.FOREVER);
      Entry anotherEntry = ...;
      js.read(anotherEntry, transaction, Lease.ANY);
      return null;
     }
    });
```

The advantage of the *JavaSpaceCallback* is that it allows several operations on the JavaSpace API to be grouped and used inside the transaction or with other Jini transactions (for example if using multiple nested *JavaSpaceCallbacks*).

8.4. Transaction Management

One important feature of JavaSpaces module is the *JiniTransactionManager* which integrates Jini transaction API with Spring transaction infrastructure. This allows users for example to use declaratively or programmatically use Jini transactions in the same manner as with JDBC-based transactions - without any code change or API coupling; changing the transaction infrastructure is as easy as changing some configuration lines (see Spring reference documentation [http://static.springframework.org/spring/docs/2.0.x/reference/transaction.html]for more information). Using

JiniTransactionManager is straight forward:

```
<beans>
 <bean id="javaSpace" class="...">

 <bean id="jiniTransactionManager" class="org.springmodules.jini.JiniServiceFactoryBean">
  <property name="serviceClass" value="net.jini.core.transaction.server.TransactionManager"/>
  <property name="timeout" value="10000"/>
 </bean>

 <!-- declaration of jini transaction manager -->
 <bean id="transactionManager" class="org.springmodules.transaction.jini.JiniTransactionManager">
  <property name="transactionManager" ref="jiniTransactionManager"/>
  <property name="transactionalContext" ref="javaSpace"/>
 </bean>
 ...
</beans>
```

JiniTransactionManager requires two parameters:

- transactionManager - an instance of net.jini.core.transaction.server.TransactionManager. In most cases this is provided by Mahalo [http://java.sun.com/products/jini/2.0/doc/api/com/sun/jini/mahalo/package-summary.html] and can be retrieved using the generic JiniServiceFactoryBean as we have discussed.
- transactionalContext - which is a simple object used for detecting the transaction context as Jini transactions can spawn across several contexts.

To some extent, the *JiniTransactionManager* is similar to Spring's JtaTransactionManager [http://static.springframework.org/spring/docs/2.0.x/api/org/springframework/transaction/jta/JtaTransactionManager.html], providing integration with a custom transactional API.

Note that *JiniTransactionManager* is not JavaSpace specific - it can be used on any Jini resource.

8.5. Remoting: JavaSpaceInterceptor

JavaSpaceInterceptor represents an important feature that allows method calls to be 'published' and retrieved transparently to and from the space (in a manner similar to delegate worker). The calls can be blocking (synchronous) or non-blocking (asynchronous). Consider the following context:

```
<beans>

 <bean id="javaSpace" clas="..."/>

 <bean id="spaceTemplate" class="org.springmodules.javaspaces.JavaSpaceTemplate">
  <property name="space" ref="javaSpace"/>
 </bean>

 <bean id="spaceInterceptor" class="org.springmodules.javaspaces.JavaSpaceInterceptor">
  <property name="javaSpaceTemplate" ref="spaceTemplate"/>
  <property name="timeoutMillis"><value>500</value></property>
 </bean>

 <!-- Client side -->
 <bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <!--
   Definition of the Spring AOP interceptor chain. The spaceInterceptor
   must be the last interceptor as there is no local target to invoke.
   Any number of other interceptors can be added, e.g. to monitor performance,
   add security or other functionality.
  -->
  <property name="interceptorNames">
   <list>
    <value>spaceInterceptor</value>
   </list>
  </property>
  <property name="proxyInterfaces">
   <list>
     <value>org.springmodules.beans.ITestBean</value>
     <value>org.springframework.core.Ordered</value>
   </list>
  </property>
 </bean>

 <!--
  Server side. We start these as threads.
  Note the singleton=false setting. This means that each one of these we obtain from
  Spring will be a distinct instance.
 -->
 <bean id="testBeanWorker" class="org.springmodules.javaspaces.DelegatingWorker" singleton="false">
  <property name="javaSpaceTemplate" ref="spaceTemplate"/>
  <property name="delegate" ref="testBean"/>
  <property name="businessInterface">
   <value>org.springmodules.beans.ITestBean</value>
  </property>
 </bean>
 <!-- method 'consumer' -->
 <bean id="testBean" class="org.springmodules.beans.TestBean" >
  <property name="name"><value>rod</value></property>
  <property name="age"><value>34</value></property>
 </bean>
</beans>
```

There are several important elements inside the context:

- *proxy* - represents the client side -all calls made to it will be delegated to the JavaSpace. The *JavaSpaceInterceptor* will transform all Method Invocations into JavaSpace entries and publish them into the space. Interested parties (which can execute the call) will pickup the entry and the write back the result which is returned to the caller.
- *testBeanWorker* - represents the server side. JavaSpaces Module provides already an implementation through *DelegatingWorker* which watches the JavaSpace and will pick any method calls which it can compute. The call entries are transformed into method invocations which are delegated to the appropriate implementation - in our case *testBean*.

8.6. GigaSpaces Spring Integration

8.6.1. Simplifying Business Logic Abstraction

The GigaSpaces Spring integration plays a major part in GigaSpaces Write Once Scale Anywhere roadmap. It allows you to write your POJO once using Spring and scale it anywhere using GigaSpaces middleware. Spring provides a framework for implementing the application business logic, while GigaSpaces implements the middleware and service framework for executing this business logic efficiently, in a scalable fashion.

8.6.2. Online Wiki Documentation

Please refer to the online Wiki documentation which is available at
http://gigaspaces.com/wiki/display/GS/Spring

GigaSpaces Spring Integration

Shay Hassidim

Gershon Diner

Lior Ben Yizhak

2.1. Introduction – Give Spring Some Space

This chapter describes the integration between GigaSpaces and the Spring Framework (www.springframework.org [http://www.springframework.org/]).

2.1.1. Simplify business logic abstraction using Spring/POJO support

GigaSpaces Spring integration plays a major part in GigaSpaces "Write Once Scale Anywhere" roadmap. It allows you to write your POJO once using Spring and Scale it Anywhere using the GigaSpaces middleware - Spring provides a framework for implementing the application business logic and GigaSpaces implements the middleware and service framework for executing this business logic efficiently in a scalable fashion.

GigaSpaces Spring integration contains two main parts:

Middleware abstraction – DAO, JavaSpace , Transaction, , JDBC, Remoting, Parallel processing , JMS – Enabling a relatively none intrusive approach for implementing the business logic on top of GigaSpaces. With this approach GigaSpaces users can leverage the rich functionality and simplification of the Spring framework and the scalability of GigaSpaces.

Service Abstraction – Enable dynamic deployment of Spring beans into the Grid

The goal of this architecture is to enable end-to-end dynamic scalability of stateful applications across the grid.

The following diagram illustrates the different components the integration includes.

Figure 1.

2.1.1.1. Middleware Abstraction

The middleware abstraction maps specific Spring interfaces into the relevant GigaSpaces middleware component i.e. the Data-Grid, Messaging Grid and Parallel Processing. This allows Spring based application to benefit from the performance, dynamic scalability and clustering capabilities of the GigaSpaces middleware without going through any complex development phase.

The middleware abstraction includes the following common components – these are shared across the different GigaSpaces components:

POJO2Entry Converter – The POJO to entry model is common to all middleware components and is used map an existing POJO into the data grid. The approach taken here is very similar to the O/R mapping approach. Class metadata such as indexes, update mode, serialization mode, persistency mode can be added at the class level or attribute level using Java annotation or using the gs.xml files and Spring XML configuration file.

The POJO-Space support is an enhancement of to the existing JavaSpaces interface. This enhancement adds capabilities to write and read POJO's directly through the Space API. It adds additional behavior required to address specific requirements in the Messaging or Data-Grid world such as oneway operations (aka send and forget), update semantics etc.

Transaction support – Spring provides a transaction abstraction layer that can be used to plug-in different transaction implementation without changing application code. The GigaSpaces transaction provides support through that interface to the Jini Transaction and Local Transaction managers.

2.1.1.1.1. Data Grid Abstraction

2.1.1.1.1.1.

JavaSpace and GigaSpace Templates

The JavaSpacesTM technology designed to help you solve two related problems: distributed persistence and the design of distributed algorithms. JavaSpaces services use RMI and the serialization feature of the Java programming language to accomplish these goals.

See:

http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html

The Spring JavaSpace template used to map existing objects into the space and allow JavaSpace operations to use the Spring transaction abstraction behavior.

The GigaSpace template provides extensions to the JavaSpace template and support batch operations , enhanced notifications options , Pojo support , optimistic locking , update semantics , count , clean , fifo , security and more.

The advantages using this approach are:

Performance – Object can be written into the local space memory and synchronized on the background with a backend Data Base.

Built-In clustering – Data written into the space becomes immediately available to all instances holding DOA reference to this cluster.

Advanced data distribution - Data written into the space can leverage the existing data distribution topologies i.e. partitioning, replication, master/local without changing the code and use choose the appropriate model at deployment.

OO Support – Since the space provide built-in POJO support object can be written directly into the space without going through any O/R mapping. The same objects can be queried using the SQL syntax since the space implements a built-in indexing mechanism. Through the hibernate CacheStore plug-in those object can be stored in any DB with a user defined custom O/R mapping capabilities. With this approach users can benefit from the performance and simplicity of the space model and still use hibernate O/R mapping support to map those objects into an existing database.

2.1.1.1.1.2.
JDBC Template

Since GigaSpaces provides JDBC support users can write their code using the standard SQL syntax and that code will work with other JDBC compliant implementation (NOTE the opposite direction i.e. taking an existing JDBC implementation into this model is not fully supported yet and will require additional manual migration effort).

2.1.1.1.2. Messaging Abstraction

GigaSpaces Spring integration provides messaging abstraction in two forms:

2.1.1.1.2.1.
JMS template

In this case GigaSpaces behaves just like a standard JMS provider through the JMS implementation. Users that are already using JMS in their implementation could benefit from the data-virtualization capabilities GigaSpaces provides and the ability to scalae JMS based application using the partitioned GigaSpaces cluster.

2.1.1.1.2.2.
Remoting

The remoting interface is used to invoke a bean using variety pluggable transport implementation. Spring support Remote Method Invocation (RMI) , Spring's HTTP invoker ,Hessian, Burlap or JAX RPC to be used as the transport implementation in addition to the space based remoting. A space based remoting implementation takes advantage of the space high availability and implicit content based routing semantics to enable scalable communication between different services.

The benefits of this approach are:

Transparency - A call to a space based remoting looks exactly the same to the any other remoting. Moving an implementation from one implementation to a space based approach can be made in a completely seamless manner.

Reliability - The space can ensure the execution of a method in several ways:

Retries

Durability – the request can be sent even if the service is not available.

Transactions – ensures consistency recoverability in case the service failed during the execution of a certain operation.

Fail-over – A request is replicated to a backup space which takes over if the space fails and ensures continues high availability of the system.

Transparent collocation optimization - Through the embedded space topology the service can be co-located or run as remote process. In case of local communication the request goes through local references. When the service is distributed it will go through the network. Since the space is a shared entity both models can co-exist without changing the configuration. i.e. some service instance can be collocated and other can be remote. All this is done transparent to the client application.

Scalability - The same request can be targeted to the multiple services that will compete on serving that request and through that share the load amongst themselves.

The services can scale across the network dynamically by monitoring the backlog (the amount of pending requests).

Partitioning – Request's can be portioned based on class-name, method argument and in this way ensure that requests that have dependency between themselves in terms of execution order will be routed to the same space instance where the order of execution can be guaranteed. In this way parallelism can be achieved on stateful operations and not just stateless ones.

2.1.1.1.3. Parallel Processing Abstraction

A private case for using the remoting interface mentioned above is for parallel processing in a similar way to the master/worker pattern used with the space.

Figure 2.

Figure 3.

In this case each method call is a task and each return value is a return on the task. Tasks can be executed by multiple service instances each can be running on a different machine and thus leverage its CPU power to increase the processing capacity of the for serving that service. From the end user perspective it looks like he's interacting with a single service. The execution balancing achieved through the space pull-model. i.e. the services blocks for requests, if a worker is under load it will simply pull less requests, otherwise it will pull more requests the same is true if the worker is running on a more powerful machine.

2.1.1.2. Service Abstraction -Turns POJO's into distributed services using the Service Grid.

You can select a bean from a Spring bean descriptor file and deploy it onto the grid, scale it dynamically by adding more instances of that service and manage fail-over scenarios i.e. if one instance fails, the Service Grid will automatically detect that and re-deploy it on another Container running at a different machine. It will also automate the deployment procedure and will select the appropriate machine instance that has the appropriate spring support built into it out of the pool of the available machine. If such machine is not available it will postpone that deployment and re-deploy it as soon as it will become available.

Figure 4.

2.2. Integration Components

The GigaSpaces Spring integration includes the following Components:

2.2.1. Common Services

2.2.1.1. Automatic POJO to Entry Translation

Currently, the Jini/JavaSpace specification dictates that all Space operation should be conducted using Java classes that implement the marker interface net.jini.core.entry.Entry [http://www.gigaspaces.com/docs/JiniApi/net/jini/core/entry/Entry.html].

In order to users will be able to use GigaSpaces capabilities without modifying existing POJOs and alleviate migration from existing object stores or caching facilities (Hibernate, OJB, etc.) to GigaSpaces, the API exposed to client allows writing and reading ordinary POJO objects which do not implement the Entry interface. All relevant conversions done internally in transparent manner.

In order to support the conversion, additional meta-data should be supplied via configuration files named *.gs.xml (similar to Hibernate’s *.hbm.xml descriptors) or via using Java annotations. These files describe the POJO’s properties which are related to GigaSpaces’ behavioral aspects of storing and looking objects in the space, for example, indexing, fifo enabled , timetolive , replicatable , persistent , etc.

Client developers are given the option to use a base/support class which is used for writing applicative DAO or service objects which need to access a Space. The DAO support class maintains a 1:1 relationship with the injected template object, which, in turn, accesses the space to which it is holding a reference.

2.2.1.2. Transaction Support

GigaSpaces supports 3 types of transactions: Jini transactions (using Jini “Mahalo” Transaction manager), local transactions and JTA/XA transactions. The GigaSpaces Spring integration provides support for the local transaction as well as the Jini distributed transactions. Configuration of the transaction management done via Spring’s configuration file (declaratively), or via coding/annotation (programmatically).

That primarily means that when switching from one transactional model to another, no code changes needed, only configuration modification via the standard Spring beans configuration file.

The Transaction manager is responsible for creating, starting, suspending, resuming, committing and rolling back the transactions which encompass Space resource(s).

The Transaction Manager is injected to Spring’s generic Transaction Interceptor, which intercepts calls to services available on the application context using a proxy, and maintains transactional contexts for these calls, based on configuration details including propagation, isolation, etc. These configuration details may be defined as configuration data in the bean descriptor xml file, using Java 5 annotations in the code, or via any other valid implementation of Spring's TransactionAttributeSource interface.

2.2.2. Data-Grid

In order to utilize GigaSpaces Data-Grid you can use either the JavaSpace Spring template and the JDBC Spring template. See below examples:

2.2.2.1. JavaSpaces Template Example

See below Pojo based JavaSpaces based application code example.

2.2.2.1.1. The BaseSimpleBean POJO

This is a base class we will use as part of the example.

```
public class BaseSimpleBean {

        private String firstName = null;

        public BaseSimpleBean() {}

        public BaseSimpleBean(String test) {

                this.firstName = test;

        }



        public boolean equals(Object other) {

                if(other == null || !(other instanceof SimpleBean))
```

```
				return false;

			else {

				SimpleBean otherBean = (SimpleBean)other;

				return (otherBean.getFirstName().equals(firstName));

			}

		}

		public String getFirstName(){return firstName;}

		public void setFirstName(String test) {this.firstName = test; }

		public String toString(){return "firstName: "+firstName;}

}
```

2.2.2.1.2. The SimpleBean POJO

This Pojo extends the BaseSimpleBean.

```
public class SimpleBean extends BaseSimpleBean{

		private String secondName;

		private Integer age;


		public SimpleBean() {}


		public SimpleBean(String name, Integer age) {

			this.secondName = name;

			this.age = age;

		}


		private Integer getAge()				{ return age; }

		private void setAge(Integer age)	{ this.age = age; }

		private String getSecondName()				{ return secondName; }

		private void setSecondName(String name)	{ this.secondName = name; }
```

```
	public boolean equals(Object other) {
		if(other == null || !(other instanceof SimpleBean))
			return false;
		else {
			SimpleBean otherBean = (SimpleBean)other;
			return ((otherBean.secondName != null &&
otherBean.secondName.equals(secondName) || otherBean.secondName == secondName )) && (otherBean.age
== age)
			&& (otherBean.age == age)
			&& ((otherBean.getFirstName() != null &&
otherBean.getFirstName().equals(getFirstName()) || otherBean.getFirstName() ==getFirstName()));
		}
	}


	public String toString()
	{
		return super.toString()+ ", secondName: "+secondName+", age: "+age;
	}
}
```

2.2.2.1.3. Simple DAO Object used by the application

The following code example demonstrate JavaSpace write and read operations using Spring:

```
public class myMain
{
	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext("gigaspaces.xml");
		GigaSpacesTemplate template =
(GigaSpacesTemplate)context.getBean("gigaspacesTemplate");
		template.clear(null);
		SimpleBean pojo = new SimpleBean("second name", new Integer(32));
		pojo.setFirstName("first name");
    template.write(pojo, Lease.FOREVER);
```

```
        System.out.println("Writing pojo to space...Done!");

      SimpleBean templatePojo = new SimpleBean();

      SimpleBean pojoResult = (SimpleBean)template.read(templatePojo, Long.MAX_VALUE);

              }


}
```

2.2.2.2. JDBC Template Example

The following application code using standard Spring JdbcTemplate to create table , insert , delete and query data from GigaSpaces Data-Grid.

```
public class HelloJdbc

{

              public static void main(String[] args) {

                            {


                                            System.out.println("\nWelcome to GigaSpaces Spring JDBC HelloWorld 
example.");

                                            System.out.println("This example uses Spring JDBCTemplate to write, read and 
"+

                                            "delete entries to space...\n" );

                                            ApplicationContext context = new 
ClassPathXmlApplicationContext("jdbc_gigaspaces.xml");


                                            JdbcTemplate template = (JdbcTemplate) context.getBean("jdbcTemplate");


                                            /* SQL CREATE TABLE statement

                                            *

                                            */

                                            String createSQL = "CREATE TABLE Person(FirstName varchar2 INDEX, " +
"LastName varchar2)";
System.out.println("Create table...");
try {
        template.execute( createSQL );
        System.out.println("Create table... Done!");

} catch (Exception e) {
        System.out.println("\nTable may exist already... ");
        System.out.println("Restart or clean (space-browser) space !");
}

/* SQL INSERT statement
*
*/
int maxRows = 10;
String insertSQL = "INSERT INTO Person VALUES(?,?)";
System.out.println("Insert into table...");
for (int i = 1; i < maxRows; i++) {
        Object[] params = new Object[] {"FirstName" + i,"LastName" + i};
        template.update(insertSQL, params);
        System.out.println("Insert into table... Done!");
}
/* SQL DELETE statement
*
*/
String deleteSQL="DELETE FROM Person WHERE FirstName='FirstName3'";
System.out.println("Delete from table...");
template.execute( deleteSQL );
System.out.print("Delete from table...Done!");
```

```
                        /* SQL SELECT statement
                        *
                        */
                        String selectSQL="SELECT * FROM Person ORDER BY Person.FirstName";
                        System.out.println("Select from table...");
                        template.query( selectSQL, new RowCallbackHandler() {
                              public void processRow(ResultSet rs) throws SQLException
                              {
                                    System.out.println("FirstName : " +
rs.getString("FirstName"));
                                    System.out.println("LastName : "+rs.getString("LastName"));
                              }
                        });
                        System.out.println("Select from table... Done!");
            }
      }
}
```

2.2.2.2.1. Application Context xml - jdbc_gigaspaces.xml

The following file includes the properties to inject into org.springframework.jdbc.datasource.SingleConnectionDataSource and org.springframework.jdbc.core.JdbcTemplate:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">


<beans>
      <!-- Declaration of GigaSpace jdbc driver-->
            <bean id="gigaspaceDataSource"
            class="org.springframework.jdbc.datasource.SingleConnectionDataSource"
destroy-method="destroy"
```

```
                singleton="false">
                <property name="driverClassName"
                        value="com.j_spaces.jdbc.driver.GDriver" />
                <property name="url"
                        value="jdbc:gigaspaces:url:rmi://localhost:10098/./helloJDBCTemplate" />
                <property name="username" value="" />
                <property name="password" value="" />
        </bean>
        <bean id="jdbcTemplate"
                class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource">
                        <ref bean="gigaspaceDataSource" />
                </property>
        </bean>
</beans>
```

2.2.3. Messaging Grid

GigaSpaces JMS Spring Integration allows users to use GigaSpaces middleware with existing JMS based applications.

2.2.3.1. The JMS Spring application Example

Below is are standard JMS Spring based applications - Sender and Receiver using GigaSpaces:

```
public class SenderToQueue
{
        public static void main(String[] args) {

                final int          NUM_MSGS;
                final String        MSG_TEXT = new String("This is a simple message");

                if ( (args.length < 1)) {
```

```
				System.out.println("Usage: java SenderToQueue [<number_of_messages>]");
				System.exit(1);
			}

			ApplicationContext context = new
ClassPathXmlApplicationContext("jms_gigaspaces.xml");

			//get the Spring JMSTemplate (here we use the JMS 102 template
			JmsTemplate102 jmsTemplate102 = (JmsTemplate102)
context.getBean("jmsQueueTemplate");

			if (args.length == 1){
				NUM_MSGS = (new Integer(args[0])).intValue();
			} else {
				NUM_MSGS = 1;
			}
			for (int i = 0; i < NUM_MSGS; i++)
			{
				final String theMessage = MSG_TEXT + " " + (i + 1);
				System.out.println("Sending message: " + theMessage);
				jmsTemplate102.send(new MessageCreator() {
					public Message createMessage(Session session)
					throws JMSException {
						return session.createTextMessage(theMessage);
					}
				});
			}
		}
}
```

```
public class SynchQueueReceiver

{

        public static void main(String[] args) {

                ApplicationContext context = new
ClassPathXmlApplicationContext("jms_gigaspaces.xml");

                //get the Spring JMSTemplate (here we use the JMS 102 template

                JmsTemplate102 jmsTemplate102 = (JmsTemplate102)
context.getBean("jmsQueueTemplate");

                while (true)

                {

                        try{

                                Message msg = jmsTemplate102.receive();



                                if (msg instanceof TextMessage) {

                                        TextMessage textMessage = (TextMessage) msg;

                                        System.out.println("Reading message: " +
textMessage.getText() );

                                } else {

                                        // Non-text control message indicates end of messages.

                                       break;

                                }

                        }catch(Exception e){

                                e.printStackTrace();

                        }

                }

        }

}
```

2.2.3.1.1. Application Context xml - jms_gigaspaces.xml

This file includes the GigaSpaces JMS properties to inject into org.springframework.jndi.JndiTemplate , org.springframework.jms.core.JmsTemplate102 and org.springframework.jndi.JndiObjectFactoryBean:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">


<beans>


        <bean id="jndiTemplate"

                class="org.springframework.jndi.JndiTemplate">

                <property name="environment">

                        <props>

                                <prop
key="java.naming.factory.initial">com.sun.jndi.rmi.registry.RegistryContextFactory</prop>

                                <prop key="java.naming.provider.url">rmi://localhost:10098</prop>

                        </props>

                </property>

        </bean>

        <!-- JMS Queue Template -->

        <bean id="jmsQueueTemplate"

                class="org.springframework.jms.core.JmsTemplate102">

                <property name="connectionFactory">

                        <ref bean="jmsQueueConnectionFactory" />

                </property>

                <property name="defaultDestination">

                        <ref bean="destination" />

                </property>

                <property name="pubSubDomain">

                        <value>false</value>
```

```
			</property>

			<property name="receiveTimeout">

				<value>20000</value>

			</property>

		</bean>



		<!-- JMS Queue Connection Factory -->

		<bean id="jmsQueueConnectionFactory"

			class="org.springframework.jndi.JndiObjectFactoryBean">

			<property name="jndiTemplate">

				<ref bean="jndiTemplate" />

			</property>

			<property name="jndiName">

<value>GigaSpaces;helloJMSTemplate_container;helloJMSTemplate;GSQueueConnectionFactoryImpl</value>

			</property>

		</bean>

		<bean id="destination"

			class="org.springframework.jndi.JndiObjectFactoryBean">

			<property name="jndiTemplate">

				<ref bean="jndiTemplate" />

			</property>

			<property name="jndiName">

<value>GigaSpaces;helloJMSTemplate_container;helloJMSTemplate;jms;destinations;MyQueue</value>

			</property>

		</bean>

</beans>
```

2.2.4. Parallel Processing – Business logic Remote invocation

In order to allow users to utilize GigaSpaces Grid, you can invoke your business logic on remote processes. Proxies to the remote objects are generated automatically. Remoting implemented using JavaSpaces as the transport layer similar to existing Spring remoting technologiessuch as RMI or Web Services.

The Remoting support composed from the following 3 logical units: Taker, worker and delegate. The taker is responsible for accessing the delegate, which, in turn, executes code located on the worker. These 3 units may be co-located within the same VM, or separately deployed on three different nodes/jvm's. The different parties communicate via Task and Result objects.

2.2.4.1. Remoting Example

This section illustrate remoting example. It describe Master , Worker , Task and Result classes implementation and their related classes.

2.2.4.1.1. The Master

The Master executes remote business logic running at the Worker. The ITask implementation is the actual business logic executed at the worker.

The remote worker returns Result object.

```
ITask proxy = (ITask)applicationContext.getBean("proxy");

Result res = proxy.execute("data");
```

2.2.4.1.2. The Worker

The Worker implementation. It is using the generic DelegatingWorker.

```
public class Worker

{

        //member for gigaspaces template

        private GigaSpacesTemplate template;

        //The delegator worker

        private DelegatingWorker iTestBeanWorker;

        private ApplicationContext          applicationContext;


        private Thread itbThread;

        protected void init() throws Exception {
                applicationContext = new
ClassPathXmlApplicationContext("gigaspaces_master_remoting.xml");

                template = (GigaSpacesTemplate)applicationContext.getBean("gigaspacesTemplate");

                iTestBeanWorker = (DelegatingWorker)applicationContext.getBean("testBeanWorker");

        }



        protected void start() {

                itbThread = new Thread(iTestBeanWorker);

                itbThread.start();

        }



        public static void main(String[] args) {

                try {

                        System.out.println("\nWelcome to Spring GigaSpaces Worker remote
Example!\n");

                        Worker worker = new Worker();

                        worker.init();

                        worker.start();

                } catch (Exception ux) {

                        ux.printStackTrace();

                        System.err.println("transError problem..." + ux.getMessage());

                }

        }

}
```

2.2.4.1.3. The ITask

The task interface.

```
public interface ITask extends Serializable{

        public Result execute(String data);

}
```

2.2.4.1.4. The Task

This is the ITask interface implementation used by the worker:

```
public class Task implements ITask{
        private long counter = 0;
  public Task() {
  }

  /**
   * Execute the task
   */
        public Result execute(String data)
        {
                counter++;
                System.out.println("I am doing the task id = "+counter+" with data : "+data);
                Result result = new Result();
                result.setTaskID(counter);
                // do the calc
                result.setAnswer(data);
                return result ;
        }
}
```

2.2.4.1.5. The Result

The Result object used to transport the Answer back to the client caller:

```
public class Result implements Serializable
{
        private long taskID; // task id
        private String answer = null; // result
        public Result() {}
        public String getAnswer() {return answer;   }
        public void setAnswer(String answer){this.answer = answer;}

        public long getTaskID(){   return taskID;}

        public void setTaskID(long taskID){this.taskID = taskID;}

}
```

2.2.4.1.6. gigaspaces_master_remoting.xml

The gigaspaces_master_remoting.xml includes properties injected into the following classes:

Table 1.

Class	Description	Bean Name
org.springmodules.javaspaces.gigaspaces.GigaSpacesUidFactory	Used to generate unique UID for tasks. When using partitioned space the uid hashcode determines the target space of the entry	gigaSpacesUidFactory
org.springframework.spaces.DelegatingWorker	The Generic worker invoking the Task business logic	testBeanWorker
com.gigaspaces.spring.GigaSpacesInterceptor	The Interceptor that pass the task from the client into the worker via the space	javaSpaceInterceptor
org.springframework.aop.framework.ProxyFactoryBean	FactoryBean implementation for use to source AOP proxies from a Spring BeanFactory	proxy
com.gigaspaces.spring.examples.remote.Task	The Remote Task implementation	taskBean

2.2.4.1.7. GigaSpacesUidFactory

To ensure each client will get the relevant result object back from the worker the task and result injected with unique uid generated at the client side.

Table 2.

Property	Description	Type
Space	The space template	Reference

2.2.4.1.7.1. DelegatingWorker

The org.springmodules.javaspaces.gigaspaces.DelegatingWorker configure the worker. This is not a singleton class. The The DelegatingWorker includes the following properties:

Table 3.

Property	Description
javaSpaceTemplate	The GigaSpaces Spring template
delegate	The "Task" bean to be injected into the worker
businessInterface	The "Task" class interface

2.2.4.1.7.2. GigaSpacesInterceptor

The com.gigaspaces.spring.GigaSpacesInterceptor controls the client side Interceptor that submits the task into the space and getting the result back. Getting the result back can be done is synchronous or asynchronous manner allowing the client to wait or continue with its activity before the actual result has been sent back from the worker. The GigaSpacesInterceptor extends the JavaSpaceInterceptor that support UID injection to the task and result objects allowing client to retrieve back the related result for specific task.

Table 4.

Property	Description	Type	Value
javaSpaceTemplate	The JavaSpace template	Reference	
uidFactory	The task includes unique identifier. This ensures that each client will get correct result object in return.	Reference	
synchronous	Should client wait until master returns result before continues. When running in asynchronous mode the and result has	boolean	false/true

	not been sent back from the worker the client will wait specified time as defined as part of the timeoutMillis parameter in case matching result does not exists within the space.		
timeoutMillis	Time in millisecond to wait for matching result to be found within the space (take timeout times).	long	
serializableTarget	Causes this target to be passed to space in a RunnableMethodEntry	Reference	

2.2.4.1.7.3.
ProxyFactoryBean

org.springframework.aop.framework.ProxyFactoryBean – the client-side proxy.

Table 5.

Property	Description	Type	Values
interceptorNames	Definition of the Spring AOP interceptor chain. The spaceInterceptor must be the last interceptor as there is no local target to invoke. Any number of other interceptors can be added, e.g. to monitor performance ,add security or other functionality	list	javaSpaceInterceptor PerformanceMonitorInterceptor
proxyInterfaces		list	com.gigaspaces.spring.examples.remo

2.2.4.1.7.4.

Application context file

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">


<!-- Root application context -->


<beans>
	<!-- Declaration of GigaSpace factory bean -->
	<bean id="gigaspace"
		class="org.springmodules.javaspaces.gigaspaces.GigaSpacesFactoryBean">
		<property name="urls">
			<list>
				<value>jini://*/*/remotingSpace</value>
			</list>
		</property>
	</bean>

	<!-- Declaration of GigaSpaces uid factory -->
	<bean id="gigaSpacesUidFactory"
		class="org.springmodules.javaspaces.gigaspaces.GigaSpacesUidFactory">
		<property name="space" ref="gigaspace"/>
	</bean>

	<!-- Declaration of GigaSpace template-->
	<bean id="gigaspacesTemplate"
		class="org.springmodules.javaspaces.gigaspaces.GigaSpacesTemplate">
		<property name="space" ref="gigaspace" />
	</bean>
```

```
    <bean id="testBeanWorker"

            class=" org.springmodules.javaspaces.DelegatingWorker"

            singleton="false" >

            <property name="javaSpaceTemplate" ref="gigaspacesTemplate"/>

            <property name="delegate"><ref local="taskBean"/></property>

            <property
name="businessInterface"><value>com.gigaspaces.spring.examples.remote.ITask</value></property>

    </bean>

    <bean id="javaSpaceInterceptor"

            class=" org.springmodules.javaspaces.gigaspaces.GigaSpacesInterceptor">

            <property name="javaSpaceTemplate" ref="gigaspacesTemplate"/>

            <property name="uidFactory" ref="gigaSpacesUidFactory"/>

            <property name="synchronous"><value>true</value></property>

            <!--

            The Time for waiting to take the result from the space by the master

            -->

            <property name="timeoutMillis"><value>3000</value></property>



            <!--

                    Comment out this property for "service seeking" behavior where the endpoint is
assumed to host a service to invoke.

            -->

            <!--

            <property name="serializableTarget" ref="taskBean">

            </property>

            -->

    </bean>

    <!--

            This is the client-side proxy.

    -->
```

```
	<bean id="proxy"
		class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="interceptorNames">
			<list>
				<value>javaSpaceInterceptor</value>
				<!--
				<value>PerformanceMonitorInterceptor</value>
				-->
			</list>
		</property>
		<property name="proxyInterfaces">
			<list>
				<value>com.gigaspaces.spring.examples.remote.ITask</value>
			</list>
		</property>
	</bean>
	<!--
		Simple test target
	-->
	<bean id="taskBean" class="com.gigaspaces.spring.examples.remote.Task" >
	</bean>
</beans>
```

2.2.5. Service Grid

The Service Grid allows users to build Pojo or Spring based application as usual and deploy these into the Grid as Services. The Service Grid managing the life cycle of the deployed Service by provisioning , starting and managing it when running at the Service Grid container.

See below simple Hello class implementation and the required steps to deploy it into the Service Grid.

2.2.5.1. Hello Interface

Your Pojo should implement an interface:

```
package example;


import java.rmi.RemoteException;


public interface Hello {
  /**
   * Say hello!
   */
  String sayHello(String greetings) throws RemoteException;
}
```

2.2.5.2. Hello implementation

Here is the Hello class implementation:

```
package example;


public class HelloImpl implements Hello {


   public String sayHello(String greetings) {
      System.out.println("**** Greeter says : "+greetings);
      return("Hello!");
  }


  public HelloImpl()
              {
                  System.out.println("**** Hello Service Started! ****");
              }
}
```

2.2.5.3. The Deployment File

This is the Service Grid deployment file. This should include the example.Hello interface , the Implementation Class and the relevant libraries information:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>

<!DOCTYPE gs-deployment SYSTEM "java://gs-deploy-desc.dtd" [

  <!ENTITY CodeServer.IP SYSTEM

      "java://java.net.InetAddress.getLocalHost().getHostAddress()">

  <!ENTITY CodeServer.Port "9010" >

  <!ENTITY CodeServerURL "http://&CodeServer.IP;:&CodeServer.Port;/" >

  <!ENTITY Group SYSTEM

       "java://java.lang.System.getProperty(com.gs.jini_lus.groups)">

]>


<gs-deployment Name="Hello World Example">


  <!-- Declare global attributes for the Deployment Descriptor -->

   <Codebase>&CodeServerURL;</Codebase>


  <Groups>

      <Group>&Group;</Group>

  </Groups>


  <!-- Declare attributes for the Hello example -->

  <ServiceBean Name="Hello">

      <Interfaces>

          <Interface>example.Hello</Interface>

          <Resources>

              <!--

               <JAR>gs-dl.jar</JAR>

              -->
```

```
            <JAR>hello-dl.jar</JAR>
<JAR>JSpaces.jar</JAR>
        </Resources>
    </Interfaces>

    <ImplementationClass>example.HelloImpl
        <Resources>
            <JAR>hello.jar</JAR>
            <JAR>JSpaces.jar</JAR>
        </Resources>
    </ImplementationClass>

    <Configuration>
        <Component Name="service.load">
            <Parameter
                Name="serviceBeanFactory"
               Value="new com.gigaspaces.grid.bean.BeanFactory()"/>
        </Component>
    </Configuration>

    <!--
    <Associations>
        <Association Type="uses"
                    Name="GigaSpace"
                    Property="space"
                    OperationalString="GigaSpace Service Deployment"/>
    </Associations>
    -->
    <Maintain>1</Maintain>
```

```
  </ServiceBean>

</gs-deployment>
```

2.2.5.4. The build file

Here is the any bbuild.xml you should have to build the hello library:

```
<?xml version='1.0' encoding='ISO-8859-1' standalone='yes'?>

<!--

The build script will build the GigaSpaces Prime Number example

-->

<project name="Bean Example" default="all" >

  <property name="example.name" value="bean-example" />

  <property environment="env."/>

  <property name="examples.home" value="${basedir}/.."/>

  <!-- import the common elements -->

  <import file="../build_properties.xml"/>

  <property name="example.lib" value="${examples.home}/bean/lib"/>

  <property name="example.build" value="${examples.home}/bean/build"/>

  <property name="example.classes" value="${example.build}/classes" />

  <property name="example.src" value="${examples.home}/bean/src" />

  <property name="example.space-prop" value="${examples.home}/bean/space-prop" />


  <fileset dir="${example.src}">

      <patternset id="java.source">

          <include name="example/**/*.java"/>

      </patternset>

  </fileset>


  <fileset dir="${example.classes}" >
```

```
    <patternset id="project.classes" >
        <include name="example/*.class" />
    </patternset>
</fileset>

            <fileset dir="${example.classes}" >
    <patternset id="project.classes" >
        <include name="example/*.class" />
    </patternset>
</fileset>

            <fileset dir="${example.lib}">
    <patternset id="jar.files">
        <include name="hello.jar" />
   </patternset>
</fileset>

<target name="jars" depends="hello.jar, hello-dl.jar" />

<target name="hello.jar" depends="compile">
    <jar destfile="${example.lib}/hello.jar"
         basedir="${example.classes}"/>
</target>

<target name="hello-dl.jar" depends="compile">
    <jar destfile="${example.lib}/hello-dl.jar"
               basedir="${example.classes}"
               excludes="**/HelloImpl.class"/>
</target>
```

```
</project>
```

2.2.5.5. Deploying the Pojo

Start the \GigaSpacesEE5.0\ServiceGrid\bin\gsc.cmd – this will start Service Grid container

Start the \GigaSpacesEE5.0\ServiceGrid\bin\gsm.cmd – this will start Service Grid Manager

Start \GigaSpacesEE5.0\ServiceGrid\bin\gs.cmd – this will start the Service Grid command interactive line shell.

```
gs> deploy hello.xml

total 1

Deploying [Hello World Example], total services [1] ...

    [1] Hello provisioned to        10.0.0.13

Deployment notification time 1062 millis, Command completed
```

The Service Grid container should Display:

```
Jun 8, 2006 1:35:21 PM com.gigaspaces.grid.gsc.GSCImpl$InitialServicesLoadTask loadInitialServices

CONFIG: Loading [0] initialServices

**** Hello Service Started! ****
```

The Service Grid manager should display:

```
Jun 8, 2006 1:36:19 PM org.jini.rio.monitor.ServiceElementManager verify

FINE: ServiceElementManager.verify(): [Hello] actual [0], pending [0], maintain [1]

Jun 8, 2006 1:36:19 PM org.jini.rio.monitor.ServiceResourceSelector selectServiceResource

FINER: Grid Service Container at [10.0.0.13] has [0] instance(s), planned [1] of [Hello]

Jun 8, 2006 1:36:19 PM org.jini.rio.monitor.InstantiatorResource canProvision

FINER: Grid Service Container at [10.0.0.13] meets qualitative requirements for [Hello]

Jun 8, 2006 1:36:19 PM org.jini.rio.monitor.ServiceProvisioner$ProvisionTask doProvision

FINER: Allocating [Hello] ...

Jun 8, 2006 1:36:20 PM org.jini.rio.monitor.ServiceProvisioner$ProvisionTask doProvision
```

FINER: Allocated [Hello]

Jun 8, 2006 1:36:20 PM org.jini.rio.monitor.ServiceElementManager$JSBProvisionListener serviceProvisioned

FINE: [Hello] service provisioned, instance=Instance=[1] Proxy=[$Proxy15] ID=[863469f1-8974-4dbc-80d9-307148547b65] Host

Address=[10.0.0.13]

2.3. Integration Implementation Classes

The architecture of the GigaSpaces integration with Spring is very similar to Hibernate implementation on spring. The implementation is based on the Spring standards, including dependency injection, transaction attributes sources, configurable proxies/exportes for remote services, etc.

Basic support for accessing a Space is provided via a GigaSpacesFactoryBean which is configured in Spring's xml definition file. Configuration primarily includes the String array of space URLs. The Factory creates a singleton Space proxy or running an embedded space when using embedded space URL.

The factory extends AbstarctJavaSpaceFactoryBean which has createSpace() template method and add listeners implementation if specified.

2.3.1. org.springmodules.javaspaces.gigaspaces.GigaSpacesFactoryBean

An entry point for the GigaSpaces Spring support. This is a standard Spring factory bean.

The following properties are injected:

To ensure each client will get the relevant result object back from the worker the task and result injected with unique uid generated at the client side.

Table 6.

Property	Description	Type
urls	list of GigaSpaces space URL. When the URL represents remote URL these will be accessed one by one until a connection will be established. When using embedded space URL it will start a new space instance at the running application memory address	

listeners	notify templates allowing notifications when matching entries are written to the space	

The standard GigaSpacesFactoryBean.getObject() method creates or accesses IJSpace object according to the provided url's that used by the the GigaSpacesTemplate, GigaSpacesDaoSupport, or GigaSpacesLocalTransactionManagerFactoryBean.

2.3.2. org.springmodules.javaspaces.gigaspaces.GigaSpacesDaoSupport

The *GigaSpacesDaoSupport* extends the org.springframework.dao.support.DaoSupport.

This is a support class, intended for extension by the application developer for writing Data Access Objects which perform domain-level operations on the supplied space. In order to operate, the Dao object should be injected either with a pre instantiated GigaspacesTemplate, or with an IJSpace. Extending classes will typically use the getGigaSpaceTemplate() method for performing space operations, but direct access to the space via the IJSpace is also possible.

2.3.3. org.springmodules.javaspaces.JavaSpaceTemplate

Implementation of the Spring "template" concept for JavaSpaces. Translates exceptions into Spring exception hierarchy. Simplifies the performance of several operations in a single method.

The JavaSpaceTemplate provides the following methods:

Table 7.

Return value	*Method*
void	afterPropertiesSet()
java.lang.Object	execute(JavaSpaceCallback jsc) Perform multiple JavaSpaces tasks in the one transaction.
net.jini.space.JavaSpae	getSpace() Return the Javaspace this template operates on
boolean	isUseTransaction() return true if transaction used.

net.jini.core.event.EvntRegistration	notify(net.jini.core.entry.Entry template, net.jini.core.event.RemoteEventListener listener, long millis, java.rmi.MarshalledObject handback) When entries are written that match this template notify the given listener with a RemoteEvent that includes the handback object.
net.jini.core.entry.Enry	read(net.jini.core.entry.Entry template, long millis) Read using the current transaction any matching entry from the space, blocking until one exists.
net.jini.core.entry.Entry	readIfExists(net.jini.core.entry.Entry template, long millis) Read using the current transaction any matching entry from the space, returning null if there is currently is none
void	setSpace(net.jini.space.JavaSpace space)
void	setUseTransaction(Boolean useTransaction) set to true to use transactions with space operation.
net.jini.core.entry.Enry	snapshot(net.jini.core.entry.Entry entry) return formatted entry. The snapshot method gives the JavaSpaces service implementor a way to reduce the impact of repeated use of the same entry
net.jini.core.entry.Enry	take(net.jini.core.entry.Entry template, long millis) Take using the current transaction a matching entry from the space, waiting until one exists
net.jini.core.entry.Enry	takeIfExists(net.jini.core.entry.Entry template, long millis) Take using the current transaction a matching entry from the space, returning null if there is currently is none
net.jini.core.lease.Lease	write(net.jini.core.entry.Entry entry, long millis) Write using the current transaction a new entry into the space.

2.3.4. org.springmodules.javaspaces.gigaspaces.GigaSpacesTemplate

The GigaSpacesTemplate extends the JavaSpaceTemplate and provides GigaSpaces enhanced JavaSpaces operations.

Responsible for supplying application developers with a collection of helper methods for accessing the space, while wrapping specific checked exceptions thrown due to Space operations with Spring's generic runtime exceptions. The template also exposes one general-purpose method, which accepts a *JavaSpaceCallback* object from the client application. This callback is where application logic code may be implemented, directly working with the space. The callback mechanism allows exception conversion to take place even when writing low-level code.

The GigaSpacesTemplate method's accept not only objects implementing the Entry interface (as defined by JavaSpaces specification) but every type of object which has a void constructor and exposes its meaningful data members via accessor / mutator methods - in other words, a POJO.

The template object exposes a general purpose method, execute(), that accepts a JavaSpaceCallback object, where application logic is implemented. The method invokes the callback object, wrapping the applicative logic with exception conversion mechanism.

The GigaSpacesTemplate includes the following methods:

Table 8.

Return value	*method*
com.j_spaces.core.client.NotifyDelegator	addNotifyDelegatorListener(org.springframework.spaces.JavaSpac javaSpaceListener, boolean fifoEnabled, int notifyMask) When entries are written that match this template notify the given listener with a RemoteEvent that includes the handback object.
com.j_spaces.core.client.NotifyDelegator	addNotifyDelegatorListener(net.jini.core.event.RemoteEventListe listener, java.lang.Object templatePojo, java.rmi.MarshalledObject handback, boolean fifoEnabled, long lease, int notifyMask) When Pojo's are written that match this template notify the given listener with a RemoteEvent that includes the handback object.
void	afterPropertiesSet()

	Override the method in JavaSpaceTemplate not, throw exception if space is null
void	clean() Cleans this space.
void	clear(net.jini.core.entry.Entry entry) Removes the entries that match the specified template and the specified
void	clear(java.lang.Object pojo) Removes the entries that match the specified template and the specified transaction from this space.
int	count(net.jini.core.entry.Entry entry) Counts the number of entries that match the specified template and the specified transaction..
int	count(java.lang.Object pojo) Counts the number of entries that match the specified template and the specified transaction..
void	dropClass(java.lang.String className) Drops all Class's entries and all its templates from the space.
java.lang.Object	execute(org.springframework.spaces.JavaSpaceCallback jsc) Checks if the space is null before execute
java.lang.Object	getAdmin() Returns the admin object to the remote part of this space
java.lang.String	getName() Returns the name of this space.
int	getReadTakeModifiers() Gets the proxyReadTakeModifiers.

int	getUpdateModifiers() Gets the proxyUpdateModifiers.
boolean	isEmbedded() Checks whether proxy is connected to embedded or remote space.
boolean	isFifo() Returns true if this proxy FIFO enabled, otherwise false.
boolean	isNOWriteLeaseMode() Checks the write mode.
boolean	isOptimisticLockingEnabled() Returns status of Optimistic Lock protocol.
boolean	isSecured() Returns an indication : is this space secured.
net.jini.core.event.EventRegistration	notify(net.jini.core.entry.Entry template, net.jini.core.event.RemoteEventListener listener, long millis, java.rmi.MarshalledObject handback, net.jini.core.transaction.Transaction tx) When entries are written that match this template notify the given listener with a RemoteEvent that includes the handback object.
net.jini.core.event.EventRegistration	notify(java.lang.Object templatePojo, net.jini.core.event.RemoteEventListener listener, long millis, java.rmi.MarshalledObject handback, net.jini.core.transaction.Transaction tx) When Pojo's are written that match this template notify the given listener with a RemoteEvent that includes the handback object.
void	ping() Checks whether the space is alive and accessible.
java.lang.Object	read(java.lang.Object pojo, long lease)

	Read the pojo from the space
java.lang.Object	readIfExists(java.lang.Object pojo, long lease) Read the pojo from the space if exist
net.jini.core.entry.Entry[]	readMultiple(net.jini.core.entry.Entry entry, int maxEntries) Reads all the entries matching the specified template from this space.
Object[]	readMultiple(java.lang.Object pojo, int maxEntries) Reads all the entries matching the specified template from this space.
void	setFifo(boolean enable) Sets FIFO mode for proxy.
void	setNOWriteLeaseMode(boolean enable) Set noWriteLease mode enabled
void	setOptimisticLocking(boolean enable) Enable/Disable Optimistic Lock protocol.
int	setReadTakeModifiers(int newModifiers) Sets the read-take mode modifiers for proxy level.
int	setUpdateModifiers(int newModifiers) Sets the update mode modifiers for proxy level.
java.lang.Object	snapshot(java.lang.Object obj) Snapshot the pojo
java.lang.Object	take(java.lang.Object template, long millis) Take the pojo form the space
java.lang.Object	takeIfExists(java.lang.Object template, long millis) Take the pojo form the space if exists

java.lang.Object[]	takeMultiple(net.jini.core.entry.Entry entry, int maxEntries) Takes all the entries matching the specified template from this space.
java.lang.Object[]	takeMultiple(java.lang.Object pojo, int maxEntries) Takes all the entries matching the specified template from this space.
net.jini.core.entry.Entry	update(net.jini.core.entry.Entry newEntry, long lease, long timeout) Updates the first entry matching the specified template, if found and there is no transaction conflict.
net.jini.core.entry.Entry	update(net.jini.core.entry.Entry newEntry, long lease, long timeout, int updateModifiers) Updates the first entry matching the specified template, if found and there is no transaction conflict.
java.lang.Object	update(java.lang.Object newPojo, long lease, long timeout) Updates the first entry matching the specified template, if found and there is no transaction conflict.
java.lang.Object	update(java.lang.Object newPojo, long lease, long timeout, int updateModifiers) Updates the first entry matching the specified template, if found and there is no transaction conflict.
java.lang.Object[]	updateMultiple(net.jini.core.entry.Entry[] entries, long[] leases) Updates a group of entries.
java.lang.Object[]	updateMultiple(net.jini.core.entry.Entry[] entries, long[] leases, int updateModifiers) Updates a group of entries.
java.lang.Object[]	updateMultiple(java.lang.Object[] pojos, long[] leases) Updates a group of pojo's.
java.lang.Object[]	updateMultiple(java.lang.Object[] pojos, long[]

	leases, int updateModifiers) Updates a group of pojo's.
net.jini.core.lease.Lease	write(java.lang.Object pojo) Write the pojo to the space with lealse long.MAX_VALUE
net.jini.core.lease.Lease	write(java.lang.Object pojo, long lease) Write the pojo to the space
net.jini.core.lease.Lease[]	writeMultiple(net.jini.core.entry.Entry[] entries, long lease) Writes the specified entries to this space.
net.jini.core.lease.Lease[]	writeMultiple(java.lang.Object[] pojos, long lease) Writes the specified entries to this space.

2.3.5. org.springmodules.javaspaces.gigaspaces.GigaSpacesLocalTransactionManagerFactoryBean

Extends the org.springframework.transaction.jini.*AbstarctTransactionManagerFactoryBean* class defined in Spring, which integrates with Spring's existing transaction management mechanism.

The class implements the template method createTransactionManager() which create the local transaction manager using the GigaSpaces LocalTransactionManager.

The GigaSpaces Spring Transaction is responsible for creating, starting, suspending, resuming, committing and rolling back the transactions which encompass Space resource(s). The Transaction Manager is injected to Spring's generic Transaction Interceptor, which intercepts calls to services available on the application context using a proxy, and maintains transactional contexts for these calls, based on configuration details including propagation, isolation, etc. These configuration details may be defined as configuration data in the bean descriptor xml file, using Java 5 annotations in the code, or via any other valid implementation of Spring's TransactionAttributeSource interface.

The following transaction propagation behaviors are supported:

- · RequiredNew
- · Never
- · Required

- · Mandatory
- · Supports
- · NotSupported

2.4. Spring Configuration Files

2.4.1. Application Context xml

Includes the GigaSpacesFactoryBean:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

        <!-- Declaration of GigaSpace factory bean -->

        <bean id="gigaspace"

                class=" org.springmodules.javaspaces.gigaspaces.GigaSpacesFactoryBean">

                <property name="urls">

                        <list>

                                <value>jini://*/*/myCache</value>

                        </list>

                </property>

        </bean>

</beans>
```

2.4.2. The Dao xml

Defines client’s Pojo DAO. For each field will be indicator if its PK and whether it needs to be calculated.

```
<?xml version="1.0" encoding="UTF-8"?>

<!--

<!DOCTYPE gigaspaces-mapping SYSTEM "jar:file:.\..\..\..\..\lib\JSpaces.jar!\gigaspaces-spring.dtd">
```

```
-->

<gigaspaces-mapping>

        <class-descriptor name="com.gigaspaces.spring.examples.SimpleBean"

                 persistent="true" replicatable="false" fifo="true" timetolive="Long.MAX_VALUE">

                 <field-descriptor name="secondName" indexed="false"

                          primary-key="false" auto-generate-pk="false" />

                 <field-descriptor name="age" indexed="false" />

                 <reference-descriptor

                          class-ref="com.gigaspaces.spring.examples.BaseSimpleBean" />

        </class-descriptor>

        <class-descriptor

                 name="com.gigaspaces.spring.examples.BaseSimpleBean"

                 persistent="true" replicatable="false" fifo="true" timetolive="Long.MAX_VALUE">

                 <field-descriptor name="firstName" indexed="true"

                          primary-key="false" auto-generate-pk="false" />

        </class-descriptor>

</gigaspaces-mapping>
```

2.4.2.1.

A *class-descriptor* and the associated java class ClassDescriptor [http://db.apache.org/ojb/api/org/apache/ojb/broker/metadata/ClassDescriptor.html] encapsulate metadata information of concrete class.

Table 9.

Attribute	Description
name	contains the full qualified name of the specified class. As this attribute is of the XML type ID there can only be one class-descriptor per class.
persistent	indicates of the transient field in the ExternalEntry.

fifo	indicates if the Pojo will be save in a Fifo order in the space.
Timetolive	time (in milliseconds) left for this entry to live. This value is correct for the operation time

2.4.2.2.

A *field descriptor* contains mapping info for a primitive typed attribute of a persistent class.

Table 10.

Attribute	Description
Name	holds the name of the persistent classes attribute.
Index	indicates which fields are indexed in the space. Takes the first member indexed for hashing
Primarykey	specifies if the field is marked as a primary key, default value is false. It's possible to auto assign primary key fields (see more details below). Field must to have toString() method that can’t be changed in runtime
*auto-generate-p*k	specifies if the values for the persistent attribute should be automatically generated by the space. The filed must be from type java.lang.String

2.4.2.3.

A *reference-descriptor* contains mapping info for an attribute of a class that is not primitive but references another entity Object.

2.4.2.4.

The *class-ref* attribute contains the full qualified name of the specified class.

2.4.3. transaction.xml

This xml includes gigaspacesTransactionAttributeSource and the TransactionInterceptor settings:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">



<beans>

        <!-- Declaration of GigaSpace factory bean -->

        <bean id="gigaspaces"

                class="org.springmodules.javaspaces.gigaspaces.GigaSpacesFactoryBean">

                <property name="urls">

                        <list>

                                <value>rmi://localhost:10098/./myCache</value>

                        </list>

                </property>

        </bean>

        <!-- declaration of GigaSpaces local transaction -->

        <bean id="gigaspacesTransactionManager"

class="org.springmodules.javaspaces.gigaspaces.transaction.GigaSpacesLocalTransactionManagerFactoryBean">

                <property name="javaSpace" ref="gigaspaces" />

        </bean>

        <!-- declaration of jini transaction manager -->

        <bean id="transactionManager"

                class="org.springmodules.javaspaces.transaction.jini.JiniTransactionManager ">

                <property name="transactionManager" ref="gigaspacesTransactionManager" />

                <property name="transactionalContext" ref="gigaspaces" />

        </bean>

        <bean id="gigaspacesTransactionAttributeSource"

class="org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource">

                <property name="properties">

                        <props>
                                <prop key="writeMandatory*">
                                    PROPAGATION_MANDATORY
                                </prop>
                                <prop key="writeNever*">
                                    PROPAGATION_NEVER
                                </prop>
                                <prop key="writeRequired*">
                                    PROPAGATION_REQUIRED
                                </prop>
                                <prop key="writeRequiredNew*">
                                    PROPAGATION_REQUIRES_NEW
                                </prop>
                                <prop key="writeSupports*">
                                    PROPAGATION_SUPPORTS
                                </prop>
                                <prop key="writeNotSupported*">
                                    PROPAGATION_NOT_SUPPORTED,
                                    +java.lang.RuntimeException
                                </prop>
                                <prop key="writeRequiredNotSupportedWithPartialRollback">
                                    PROPAGATION_REQUIRES_NEW,
                                    +java.lang.RuntimeException
                                </prop>
                                <prop key="writeRequiredNotSupportedWithPartialCommit">
                                    PROPAGATION_REQUIRES_NEW,
                                    +java.lang.RuntimeException
                                </prop>
                        </props>
                </property>
```

```
</bean>



<!-- declaration of spring transaction interceptor for transaction declaration-->
<bean id="txInterceptor"
        class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <property name="transactionManager" ref="transactionManager" />
        <property name="transactionAttributeSource" ref="gigaspacesTransactionAttributeSource">
        </property>
</bean>


<!-- declaration of dao and the service -->
<bean id="txDaoTarget"
        class="com.gigaspaces.spring.examples.transaction.TransactedDao">
        <property name="space" ref="gigaspaces" />
</bean>
<!-- Declaration of GigaSpace template-->
<bean id="gigaspacesTemplate"
        class="com.gigaspaces.spring.GigaSpacesTemplate">
        <property name="space" ref="gigaspaces" />
</bean>
<bean id="txDao"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
                <value>
                        com.gigaspaces.spring.examples.transaction.ITransactedDao
                </value>
        </property>
        <property name="interceptorNames">
```

```
                        <list>
                                <value>txInterceptor</value>
                        </list>
                </property>
                <property name="target" ref="txDaoTarget" />
        </bean>
        <bean id="simpleBean"
                class="com.gigaspaces.spring.examples.transaction.SimpleBean"/>
</beans>
```

2.4.4. Pojo Primary Key setting

A Pojo can be declared with or without primary key. The primary key type can be java.lang.String or any other type, as long it implements the toString() where the toString() return value cannot be changes for the object life time period. The following table describes the operation support when using the Primary Key field.

Table 11.

Operation	Write	Take/Read	Update
Without primary key	Supported	Supported	Not Supported
With primary key - Auto generator	Supported	Supported when sending the pk field as not null or call toEntry() with parameter isIgnoreGenerateAutoPk = true.	Supported
With primary key -No auto generator	Supported	Supported	Supported

Note:

If there are more than one primary key with Auto generator, the converter will generate UID for each primary key. The UIDs will also be set for the Pojo primary key fields.

2.4.4.1. Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gigaspaces-mapping SYSTEM "src/main/resources/gigaspaces-spring.dtd">
<gigaspaces-mapping>
        <class-descriptor name="com.gigaspaces.spring.tests.app.SimpleBean"
                persistent="true" replicatable="false" fifo="true" timetolive="100">
                <field-descriptor name="name" indexed="false"
                        primary-key="false" auto-generate-pk="false" />
                <field-descriptor name="testLongObject" indexed="false" />
                <field-descriptor name="testDoubleObject" indexed="false" />
                <field-descriptor name="age" indexed="false" />
                <field-descriptor name="testFloatObject" indexed="false" />
                <field-descriptor name="testBooleanObject" indexed="false" />
                <field-descriptor name="testIntegerObject" indexed="false" />
                <reference-descriptor
                        class-ref="com.gigaspaces.spring.tests.app.BaseSimpleBean" />
        </class-descriptor>
        <class-descriptor
                name="com.gigaspaces.spring.tests.app.BaseSimpleBean"
                persistent="true" replicatable="false" fifo="true" timetolive="100">
                <field-descriptor name="teststring" indexed="true"
                        primary-key="false" auto-generate-pk="false" />
                <field-descriptor name="testboolean" indexed="true" />
                <field-descriptor name="testlong" indexed="true" />
                <field-descriptor name="testfloat" indexed="false" />
                <field-descriptor name="testint" indexed="false" />
                <field-descriptor name="testdouble" indexed="false" />
        </class-descriptor>
</gigaspaces-mapping>
```

2.5. 3rd party packages

The following libraries are used as part of the GigaSpaces Spring integration.

Apache Digester - commons-digester-gs-1.7.jar.

Apache Commons - commons-beanutils-gs.jar

Apache Velocity - velocity-1.4.jar

Remoting - cglib-nodep-2.1_3.jar

Transaction support - jta.jar

2.6. References

Spring Framework - http://www.springframework.org [http://www.springframework.org/] , http://www.interface21.com/

Beanutils - Using http://jakarta.apache.org/commons/beanutils for reflection investigating classes Meta data in order to build ExternalEntery from POJO.

Digester - Using http://jakarta.apache.org/commons/digester for parsing the gs.xml which describes the POJO to ExternalEntry.

Hibernate - http://hibernate.org [http://hibernate.org/]

Velocity - Using http://jakarta.apache.org/velocity

Chapter 9. jBPM 3.1.x

Note

The following documentation can be used as reference documentation for Spring Modules jBPM 3.0.x support as well.

9.1. Introduction

jBPM module offers integration between the Spring [http://www.springframework.org/] and jBPM [http://www.jboss.com/products/jbpm/] allowing for reuse of Spring's Hibernate [http://www.hibernate.org] support along with the IoC container. The module allows jBPM's underlying Hibernate sessionFactory to be configured through Spring and jBPM actions to access Spring's context.

9.2. Configuration

Users familiar with Spring will see that the jBPM module structure resembles with the orm package from the main Spring distribution. The module offers a central template class for working with jBPM, a callback to access the native JbpmContext and a local factory bean for configuration and creating a jBPM instance.

```
<beans>
  <!-- DataSource definition -->
  <bean id="dataSource" class="...">
    ...
  </bean>

  <!-- Hibernate SessionFactory definition -->
  <bean id="hibernateSessionFactory"
     class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
  </bean>

  <!-- helper for reading jBPM process definitions -->
  <bean id="simpleWorkflow"
      class="org.springmodules.workflow.jbpm31.definition.ProcessDefinitionFactoryBean">
    <property name="definitionLocation"
      value="classpath:org/springmodules/workflow/jbpm31/simpleWorkflow.xml"/>
  </bean>

  <!-- jBPM configuration -->
  <bean id="jbpmConfiguration"
      class="org.springmodules.workflow.jbpm31.LocalJbpmConfigurationFactoryBean">
    <property name="sessionFactory" ref="hibernateSessionFactory"/>
    <property name="configuration" value="classpath:jbpm.cfg.xml"/>
    <property name="processDefinitions">
     <list>
      <ref local="simpleWorkflow"/>
     </list>
    </property>
    <property name="createSchema" value="true"/>
    <property name="processDefinitionsResources">
     <list>
      <value>classpath:/org/springmodules/workflow/jbpm31/someOtherWorkflow.xml</value>
     </list>
    </property>
  </bean>

 <!-- jBPM template -->
 <bean id="jbpmTemplate" class="org.springmodules.workflow.jbpm31.JbpmTemplate">
  <constructor-arg index="0" ref="jbpmConfiguration"/>
  <constructor-arg index="1" ref="simpleWorkflow"/>
 </bean>
set
</beans>
```

The example above shows how (existing) Spring-managed Hibernate SessionFactories and transaction management can be reused with jBPM.

9.2.1. LocalJbpmConfigurationFactoryBean

The main element is *LocalJbpmConfigurationFactoryBean* which should be familiar to users acustomed to Spring. Based on the jbpm configuration file and the given SessionFactory, it will create a jBPM configuration which can be used for working with the given process definitions. It is possible to replace jBPM xml configuration with jBPM 3.1.x newly added ObjectFactory - note that if both are present the xml configuration is preffered. *LocalJbpmConfigurationFactoryBean* allows the creation of the underlying schema based on the process definitions loaded automatically at startup.

Note that the sessionFactory property is not mandatory - Hibernate SessionFactory can be reused with jBPM or jBPM can work by itself without any integration with the existing infrastructure. However, in most scenarios, using *LocalJbpmConfigurationFactoryBean* allows one to take advantage of Spring transaction management infrastructure [http://static.springframework.org/spring/docs/2.0.x/reference/transaction.html] so it's possible without any code change to use jBPM, Hibernate and jdbc-based code inside the same transactional context, be it managed locally or globally (JTA). Moreover, it is possible to use thread-bound [http://static.springframework.org/spring/docs/2.0.x/reference/orm.html#orm-hibernate] session or OpenSessionInView patterns with jBPM.

LocalJbpmConfigurationFactoryBean is also aware of the enclosing applicationContext lifecycle - jBPM will be initialized once the context is started (usually application startup) and will be closed properly when the context is destroyed (application is shutdown).

Note that *LocalJbpmConfigurationFactoryBean* can be configured programatically and can be used standalone only to build an jBPM context which can be used independently of Spring Modules jBPM support.

9.2.2. Inversion of Control: JbpmTemplate and JbpmCallback

Another important feature of Spring Modules jBPM support is *JbpmTemplate*. The template offers very convient ways of working directly with process definitions as well as jBPM API taking care of handling exceptions (be it jBPM or Hibernate based) in respect to the ongoing transaction (if it's present), the underlying Hibernate session (if pesistent services are used) and the jBPM context. jBPM exceptions (and the underlying Hibernate information) are translated into Spring's DAO exception hierarchy. Everything happens in a transparent and consistent manner.This is possible, as with every Spring-style template,even when direct access to the native JbpmContext is desired, through the *JbpmCallback*:

```
public ProcessInstance findProcessInstance(final Long processInstanceId) {
  return (ProcessInstance) execute(new JbpmCallback() {
     public Object doInJbpm(JbpmContext context) {
      // do something
      ...
      return context.getGraphSession().loadProcessInstance(processInstanceId.longValue());
     }
  });
}
```

As well, as *LocalJbpmConfigurationFactoryBean*, the *JbpmTemplate* can be configured programatically and can be used standalone on a pre-existing jbpmContext (configured through *LocalJbpmConfigurationFactoryBean* or not) and can be used independently of Spring Modules jBPM support.

9.2.3. ProcessDefinitionFactoryBean

ProcessDefinitionFactoryBean is a simple reader that loads jBPM process definition using Spring's ResourceLoader [http://static.springframework.org/spring/docs/2.0.x/reference/resources.html#d0e4913]s. Thus, the xml files can be load using the classpath, relative or absolute file path or even from the Servlet Context. See the official documentation [http://static.springframework.org/spring/docs/2.0.x/reference/resources.html] for more information.

Note

As reported [http://forum.springframework.org/showpost.php?p=64025&postcount=26] on the forums [http://forum.springframework.org/forumdisplay.php?f=37], using ProcessDefinitionFactoryBean jBPM 3.1.1will trigger a new process definition to be persisted(through `deployProcessDefinition`) at each startup. While this is useful in development when the database is created on application startup and destroyed on closing, for cases where the definition doesn't change, the process should not be declared inside Spring XML files.

Note

As reported here [http://opensource.atlassian.com/projects/spring/browse/MOD-193], due to the static nature of jBPM, process definitions which include sub processes are not loaded properly if a JbpmContext does not exist at the time of the loading (no exception is thrown whatsoever). As a workaround consider using the LocalJbpmConfigurationFactoryBean's processDefinitionsResources property.

9.2.4. Outside Spring container

It is important to note that while our example showed LocalJbpmConfigurationFactoryBean and JbpmTemplate template inside a Spring xml, these classes do not depend on each other or on Spring application context. They can be just as well configured programatically and can

9.3. Accessing Spring beans from jBPM actions

Another important feature of Spring Modules jBPM integration is allowing Spring configured beans to be reused inside jBPM actions. This allows one to leverage Spring container capabilities (bean lifecycles, scoping, injection, proxying just to name a few) in a transparent way with jBPM. Consider the following Spring application context:

```
<beans>
 <!-- Spring bean visible inside jBPM processed -->
 <bean id="jbpmAction" class="org.MyJbpmActionHandler" singleton="true">
   <property name="someProp" ref="anotherBean"/>
   ...
 </bean>
..
</beans>
```

and jBPM process definition:

```
<?xml version="1.0" encoding="UTF-8"?>

<process-definition name="simpleWorkflow">
 <start-state>
  <transition to="myState">
  </transition>
 </start-state>

 <state name="myState">
  <transition to="end">
```

```
   <action name="myAction" config-type="bean"
      class="org.springmodules.workflow.jbpm31.JbpmHandlerProxy">
    <targetBean>jbpmAction</targetBean>
    <factoryKey>jbpmConfiguration</factoryKey>
   </action>
  </transition>
 </state>

 <end-state name="end"/>
</process-definition>
```

JbpmHandlerProxy transparently locates Spring applicationContext and searches the bean identified by the *targetBean* parameter (in this case jbpmAction) and delegate all calls to the jBPM action. This way, one is not limited only to the injection offered by jBPM container and can integrate and communicate in a very easy manner with other Spring managed beans. Moreover, your action lifecycle can be sigleton (one shared instance) or prototype (every call gets a new instance) or in Spring 2.0 scoped to a certain application component (like one instance per http session).

The optional *factoryKey* parameter specified in this example should be used when one is dealing with more then one jBPM configuration inside the same classloader (not common in practice). The *factoryKey* should be the same as the bean name of the *LocalJbpmConfigurationFactoryBean* to be used (in our case jbpmConfiguration).

Chapter 10. Java Content Repository (JSR-170)

10.1. Introduction

JSR-170 defines "a standard, implementation independent, way to access content bi-directionally on a granular level within a content repository. A Content Repository is a high-level information management system that is a superset of traditional data repositories. A content repository implements "content services" such as: author based versioning, full textual searching, fine grained access control, content categorization and content event monitoring. It is these "content services" that differentiate a Content Repository from a Data Repository." (taken from the JSR-170 description page).

More information about Java Content Repository (from here on refered as JCR) can be found at here [http://www.jcp.org/en/jsr/detail?id=170].

The package has been designed to resemble as much as possible the ORM packages from the main Spring distribution. Users familiar with these can start using the JCR-support right away without much hassle; the documentation resembles the main documentation structure also. For those who haven't used them, please refer to the main Spring documentation, mainly chapter 12 (Data Access using O/R Mappers) [http://static.springframework.org/spring/docs/1.2.x/reference/orm.html] as the current documentation focuses on the JCR specific details, the Spring infrastructure being outside the scope of this document. As the ORM package, the main reason for the JCR support is to ease development using Spring unchecked DAOexception hierarchy, integrated transaction management, ease of testing.

Before going any further I would like to thank Guillaume Bort <guillaume.bort@zenexity.fr> and Brian Moseley <bcm@osafoundation.org> which worked on some implementation of their own and were kind enough to provide their code and ideas when I started working on this package.

10.2. JSR standard support

The standard support works only with the JSR-170 API (represented by `javax.jcr` package) without making any use of specific features of the implementations (which we will discuss later).

10.2.1. SessionFactory

JSR-170 doesn't provide a notion of `SessionFactory` but rather a repository which based on the credentials and workspace provided returnes a session. The `SessionFactory` interface describes a basic contract for retrieving session without any knowledge of credentials, it's implementation acting as a wrapper around the `javax.jcr.Repository`:

```
<bean id="sessionFactory" class="org.springmodules.jcr.JcrSessionFactory">
 <property name="repository" ref="repository"/>
</bean>
```

The only requirement for creating a sessionFactory is the repository (which will be discussed later). There are cases were credentials have to be submitted. One problem that new users have is that `javax.jcr.SimpleCredentials` requires a char array (char[]) as constructor parameter and not a String and the current Spring distribution (1.2.5) does not contains a `PropertyEditor` for char arrays. The following examples (taken from the sample) shows how we can use String statical methods to obtain a char array:

```
<bean id="sessionFactory" class="org.springmodules.jcr.JcrSessionFactory">
  <property name="repository" ref="repository"/>
  <property name="credentials">
   <bean class="javax.jcr.SimpleCredentials">
    <constructor-arg index="0" value="bogus"/>
    <!-- create the credentials using a bean factory -->
    <constructor-arg index="1">
     <bean factory-bean="password"
          factory-method="toCharArray"/>
    </constructor-arg>
   </bean>
  </property>
</bean>

<!-- create the password to return it as a char[] -->
<bean id="password" class="java.lang.String">
  <constructor-arg index="0" value="pass"/>
</bean>
```

Using the static `toCharArray` (from `java.lang.String`) we transformed the String supplied as password (with value 'pass') to `SimpleCredentials` for user 'bogus'. Note that `JcrSessionFactory` can also register namespaces, add listeners and has utility methods for determing the underlying repository properties - see the javadoc and the samples for more information.

10.2.1.1. Namespace registration

The `JcrSessionFactory` allows namespace registration based on the standard JSR-170 API. It is possible to override the existing namespaces (if any) and register namespaces just during the existence of the `JcrSessionFactory`. By default, the given namespaces are registered only if they occupy free prefixes and be kept in the repository even after the `SessionFactory` shuts down.

To register the namespaces, simply pass them as a property object, with the key representing the prefix and the value, representing the namespace:

```
<bean id="sessionFactory" class="org.springmodules.jcr.JcrSessionFactory">
  ...
 <property name="namespaces">
  <props>
    <prop key="foo">http://bar.com/jcr</prop>
    <prop key="hocus">http://pocus.com/jcr</prop>
   </props>
 </property>
</bean>
```

One can customize the behavior of the `JcrSessionFactory` using 3 flags:

- *forceNamespacesRegistration* - indicates if namespaces already registered under the given prefixes will be overridden or not(default). If `true`, the existing namespaces will be unregistered before registering the new ones. Note however that most (if not all) JCR implementations do not support namespace registration.
- *keepNewNamespaces* - indicates if the given namespaces are kept, after being registered (default) or unregistered on the `SessionFactory` destruction. If `true`, the namespaces unregistered during the registration process will be registered back on the repository. Again, as noted above, this requires the JCR implementation to support namespace un-registration.
- *skipExistingNamespaces* - indicates if the during the registration process, the existing namespaces are being skipped (default) or not. This flag is used as a workaround for repositories that don't support namespace un-registration (which render the *forceNamespacesRegistration* and *keepNewNamespaces* useless). If `true`, will allow registration of new namespaces only if they use a free prefix; if the prefix is taken, the namespace registration is skipped.

10.2.1.2. Event Listeners

JSR-170 repositories which support *Observation*, allow the developer to monitor various event types inside a workspace. However, any potential listener has to be register per-session basis which makes the session creation difficult. `JcrSessionFactory` eases the process by supporting global (across all sessions) listeners through its `EventListenerDefinition`, a simple wrapper class which associates a JCR `EventListener` with event types, node paths and uuids (which allows, if desired, the same `EventListener` instance to be reused across the sessions and event types).

Configuring the listener is straight forward:

```
<bean id="sessionFactory" class="org.springmodules.jcr.JcrSessionFactory">
  ...
  <property name="eventListeners">
   <list>
    <bean class="org.springmodules.jcr.EventListenerDefinition">
     <property name="listener">
      <bean class="org.springmodules.examples.jcr.DummyEventListener"/>
     </property>
     <property name="absPath" value="/rootNode/someFolder/someLeaf"/>
    </bean>
   </list>
  </property>
</property>
```

10.2.1.3. NodeTypeDefinition registration

JCR 1.0 specifications allows custom node types to be registered in a repository but it doesn't standardises the process, thus each JCR implementation comes with its own approach. For Jackrabbit, the JCR module provides a dedicated `SessionFactory`, the `JackrabbitSessionFactory` which allows node type definitions in the CND [http://jackrabbit.apache.org/doc/nodetype/cnd.html] format, to be added to the repository:

```
<bean id="jackrabbitSessionFactory" class="org.springmodules.jcr.jackrabbit.JackrabbitSessionFactory">
  ...
  <property name="nodeDefinitions">
    <list>
     <value>classpath:/nodeTypes/wikiTypes.cnd</value>
     <value>classpath:/nodeTypes/clientATypes.cnd</value>
    </list>
  </property>
</bean>
```

If there is no need to register any custom node types, it's recommended that the `JcrSessionFactory` is used since it works on all JCR repositories.

10.2.2. Inversion of Control: JcrTemplate and JcrCallback

Most of the work with the JCR will be made through the `JcrTemplate` itself or through a `JcrCallback`. The template requires a `SessionFactory` and can be configured to create sessions on demand or reuse them (thread-bound) - the default behavior.

```
<bean id="jcrTemplate" class="org.springmodules.jcr.JcrTemplate">
  <property name="sessionFactory" ref="sessionFactory"/>
  <property name="allowCreate" value="true"/>
</bean>
```

`JcrTemplate` contains many of the operations defined in `javax.jcr.Session` and `javax.jcr.query.Query` classes plus some convenient ones; however there are cases when they are not enought. With `JcrCallback`, one can work directly with the `Session`, the callback begin thread-safe, opens/closes sessions and deals with exceptions:

```
    public void saveSmth() {
        template.execute(new JcrCallback() {

            public Object doInJcr(Session session) throws RepositoryException {
                Node root = session.getRootNode();
                log.info("starting from root node " + root);
                Node sample = root.addNode("sample node");
                sample.setProperty("sample property", "bla bla");
                log.info("saved property " + sample);
                session.save();
                return null;
            }
        });
    }
```

10.2.2.1. Implementing Spring-based DAOs without callbacks

The developer can access the repository in a more 'traditional' way without using `JcrTemplate` (and `JcrCallback`) but still use Spring DAO exception hierarchy. SpringModules `JcrDaoSupport` offers base methods for retrieving `Session` from the `SessionFactory` (in a transaction-aware manner is transactions are supported) and for converting exceptions (which use `SessionFactoryUtils` static methods). Note that such code will usually pass "false" into `getSession`'s the "allowCreate" flag, to enforce running within a transaction (which avoids the need to close the returned `Session`, as it its lifecycle is managed by the transaction):

```
public class ProductDaoImpl extends JcrDaoSupport {

    public void saveSmth()
            throws DataAccessException, MyException {

        Session session = getSession();
        try {
                Node root = session.getRootNode();
                log.info("starting from root node " + root);
                Node sample = root.addNode("sample node");
                sample.setProperty("sample property", "bla bla");
                log.info("saved property " + sample);
                session.save();
                return null;
        }
        catch (RepositoryException ex) {
            throw convertJcrAccessException(ex);
        }
    }
}
```

The major advantage of such direct JCR access code is that it allows any checked application exception to be thrown within the data access code, while `JcrTemplate` is restricted to unchecked exceptions within the callback. Note that one can often defer the corresponding checks and the throwing of application exceptions to after the callback, which still allows working with `JcrTemplate`. In general, `JcrTemplate's` convenience methods are simpler and more convenient for many scenarios.

10.2.3. RepositoryFactoryBean

Repository configuration have not been discussed by JSR-170 and every implementation has a different approach. The JCR-support provides an abstract repository factory bean which defined the main functionality leaving subclasses to deal only with the configuration issues. The current version supports jackrabbit and jeceira as repository implementations but adding new ones is very easy. Note that through Spring, one can configure a repository without the mentioned `RepositoryFactoryBean`.

10.2.3.1. Jackrabbit

JackRabbit [http://incubator.apache.org/jackrabbit/] is the default implementation of the JSR-170 and it's part of the Apache Foundation. The project has graduated from the incubator and had an initial 1.0 release in early 2006. JackRabbit support both levels and all the optional features described in the specifications.

```
<!-- configuring the default repository -->
<bean id="repository" class="org.springmodules.jcr.jackrabbit.RepositoryFactoryBean">
  <!-- normal factory beans params -->
  <property name="configuration" value="classpath:jackrabbit-repo.xml"/>
  <property name="homeDir" value="/repo"/>
</bean>
```

-- or --

```
<!-- configuring a 'transient' repository (automatically starts when a session is opened
     and shutdowns when all sessions are closed) -->
<bean id="repository" class="org.springmodules.jcr.jackrabbit.TransientRepositoryFactoryBean">
  <!-- normal factory beans params -->
  <property name="configuration" value="classpath:jackrabbit-repo.xml"/>
  <property name="homeDir" value="/repo"/>
</bean>
```

Note that `RepositoryFactoryBean` makes use of Spring Resource to find the configuration file.

10.2.3.2. Jackrabbit RMI support

Jackrabbit's RMI server/client setup is provided through `org.springmodules.jcr.jackrabbit.RmiServerRepositoryFactoryBean` though Spring itself can handle most of the configuration without any special support:

```
<!-- normal repository -->
<bean id="repository" class="org.springmodules.jcr.jackrabbit.RepositoryFactoryBean">
   <!-- normal factory beans params -->
   <property name="configuration" value="/org/springmodules/jcr/jackrabbit/jackrabbit-repo.xml" />
   <!-- use the target folder which will be cleaned  -->
   <property name="homeDir" value="file:./target/repo" />
</bean>

<!-- rmi server -->

<!-- use Spring's RMI classes to retrieve the RMI registry -->
<bean id="rmiRegistry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean"/>

<bean id="rmiServer" class="org.springmodules.jcr.jackrabbit.RmiServerRepositoryFactoryBean">
  <property name="repository" ref="repository"/>
  <property name="remoteAdapterFactory">
    <bean class="org.apache.jackrabbit.rmi.server.ServerAdapterFactory"/>
  </property>

  <property name="registry" ref="rmiRegistry"/>
  <property name="rmiName" value="jackrabbit"/>
</bean>

<!-- rmi client -->
<bean id="rmiClientFactory" class="org.apache.jackrabbit.rmi.client.ClientRepositoryFactory"/>

<bean id="rmiClient" factory-bean="rmiClientFactory" factory-method="getRepository"
                   depends-on="rmiServer">
  <constructor-arg value="rmi://localhost:1099/jackrabbit"/>
</bean>
```

10.2.3.3. Jeceira

Jeceira [http://www.jeceira.com/] is another JSR-170 open-source implementation though as not as complete as Jackrabbit. Support for it can be found under `org.springmodules.jcr.jeceira` package:

```
<bean id="repository" class="org.springmodules.jcr.jeceira.RepositoryFactoryBean">
  <property name="repositoryName" value="myRepository"/>
</bean>
```

10.3. Extensions support

JSR-170 defines 2 levels of complaince and a number of optional features which can be provided by implementations, transactions being one of them.

10.3.1. Transaction Manager

One of the nicest features of the JCR support in Spring Modules is transaction management (find out more about Spring transaction management in Chapter 8 [http://static.springframework.org/spring/docs/1.2.x/reference/transaction.html] of the Spring official reference documentation). At the moment, only Jackrabbit is known to have dedicated transactional capabilities. One can use `LocalTransactionManager` for local transactions or Jackrabbit's JCA connector to enlist the repository in a XA transaction through a JTA transaction manager. As a side note the JCA scenario can be used within an application server along with a specific descriptor or using a portable JCA connector (like Jencks [http://www.jencks.org]) which can work outside or inside an application server.

10.3.1.1. LocalTransactionManager

For local transaction the `LocalTransactionManager` should be used:

```
<bean id="jcrTransactionManager"
 class="org.springmodules.jcr.jackrabbit.LocalTransactionManager">
   <property name="sessionFactory" ref="jcrSessionFactory"/>
</bean>

<!-- transaction proxy for Jcr services/facades -->
<bean id="txProxyTemplate" abstract="true"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
 <property name="proxyTargetClass" value="true"/>
 <property name="transactionManager" ref="jcrTransactionManager"/>
 <property name="transactionAttributes">
   <props>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED, readOnly</prop>
   </props>
 </property>
</bean>

<bean id="jcrService" parent="txProxyTemplate">
 <property name="target">
  <bean class="org.springmodules.examples.jcr.JcrService">
   <property name="template" ref="jcrTemplate"/>
  </bean>
 </property>
</bean>
```

for which only the `sessionFactory` is required.

Note that when using transactions in most cases you want to reuse the session (which means `allowCreate` property on `jcrTemplate` should be false (default)).

10.3.1.2. JTA transactions

For distributed transactions, using JCA is recommend in JackRabbit's case. An example is found inside the sample. You are free to use your application server JCA support; Jencks is used only for demonstrative purpose, the code inside the jackrabbit support having no dependency on it.

10.3.1.3. SessionHolderProviderManager and SessionHolderProvider

Because JSR-170 doesn't directly address transaction, details vary from repository to repository; JCR module contains (quite a lot of) classes to make this issue as painless as possible. Normally users should not be concern with these classes,however they are the main point for adding support for custom implementations.

In order to plug in extra capabilities one must supply a `SessionHolderProvider` implementation which can take advantage of the underlying JCR session feature. `SessionHolderProviderManager` acts as a registry of `SessionHolderProviders` for different repositories and has several implementations that return user defined provider or discover them automatically.

`By default, ServiceSessionHolderProviderManager` is used, which is suitable for most of cases. It uses JDK 1.3+ Service Provider specification [http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html#Service%20Provider] (also known as *META-INF/services*) for determining the holder provider. The class looks on the classpath under META-INF/services for the file named "`org.springmodules.jcr.SessionHolderProvider`" (which contains the fully qualified name of a `SessionHolderProvider` implementation). The providers found are instantiated and registered and later on used for the repository they support. The distribution for example, contains such a file to leverage Jackrabbit's transactional capabilities.

Besides `ServiceSessionHolderProviderManager`, one can use `ListSessionHolderProviderManager` to manually associate a `SessionHolder` with a certain repository:

```
<bean id="sessionFactory" class="org.springmodules.jcr.JcrSessionFactory">
  <property name="repository" ref="repository"/>
  <property name="credentials">
  ...
  </property>
  <property name="sessionHolderProviderManager">
    <bean class="org.springmodules.jcr.support.ListSessionHolderProviderManager">
     <list>
      <bean class="foo.bar.CustomSessionHolderProvider"/>
      <bean class="my.custom.AnotherSessionHolderProvider"/>
     </list>
    </bean>
  </property>
</bean>
```

10.4. Mapping support

Working with the JCR resembles to some degree to working with JDBC. Mapping support for the JCR seems to be the next logical step but the software market doesn't seem to offer any mature solution. The current package offers support for jcr-mapping [http://incubator.apache.org/graffito/project-structure.html] which is part of Graffito project [http://incubator.apache.org/graffito/] which belongs to the Apache foundation. However, Graffito itself is still in the incubator and the jcr-mapping is described as a prototype. The current support provides some base functionality taken from a snapshot which can be found inside the distribtution which most probably is old.

Moreover as jcr-mapping is clearly in alpha stage and a work in progress, users should not invest too much in this area but are encouraged to experiment and provide feedback. At the moment, the support contains a `JcrMappingCallback` and `Template` plus a `FactoryBean` for creating `MappingDescriptors` (which allows using more then one mapping file which is not possible at the moment in the jcr-mapping project).

10.5. Working with JSR-170 products

Even though the documentation uses as examples stand-alone JSR-170 implementations, the JCR module can work against any library/product which supports the JCR API. The only difference is that the JCR repository retrieval, which has to be changed based on the product used settings. Usually, reading the product documentation suffices however, to ease integration, this documentation includes hints for major JCR-compatible products (if you are working with a major product which is not mentioned below, please contribute the instructions through the project issue tracker [http://opensource.atlassian.com/projects/spring/browse/MOD]).

10.5.1. Alfresco

Alfresco [http://www.alfresco.com/] is an open-source enterprise content managment system which uses Spring framework [http://www.springframework.org] as its core. To get a hold of the JCR connector, one can:

bootstrap Alfresco application context and do a dependency lookup:

```
ApplicationContext context = new ClassPathXmlApplicationContext("classpath:alfresco/application-context.xml");
Repository repository = (Repository)context.getBean("JCR.Repository");
```

- or -

let the container inject the dependency:

```
<bean id="myBean" class="myBeanImplementation">
  <property name="JCRRepository"><ref bean="JCR.Repository"/></property>
  ...
</bean>
```

For more information, see Alfresco JCR documentation [http://wiki.alfresco.com/wiki/Introducing_the_Alfresco_Java_Content_Repository_API#Repository_Configuration].

10.5.2. Magnolia

Magnolia [http://www.magnolia.info/] aims to make Enterprise Content Management simple by being user-friendly, battle-tested, enterprise-ready and open-source. Mangnolia itself is not a repository and replies on a JCR implementation(Jackrabbit in particular) for its backend. Thus, connecting through JSR-170 to Magnolia is identical to connecting to a Jackrabbit repository. See Magnolia FAQ [http://www.magnolia.info/en/products/faq.html], for more information [http://www.magnolia.info/en/products/faq.html#DoyouprovideaJCRJSR170level2readw].

Chapter 11. JSR94

11.1. Introduction

As described in the scope section of the specification document, JSR94 defines a lightweight-programming interface. Its aim is to constitute a standard API for acquiring and using a rule engine.

"The scope of the specification specifically excludes defining a standard rule description language to describe the rules within a rule execution set. The specification targets both the J2SE and J2EE (managed) environments.

The following items are in the scope of the specification:

- The restrictions and limits imposed by a compliant implementation.
- The mechanisms to acquire interfaces to a compliant implementation.
- The mechanisms to acquire interfaces to a compliant implementation.
- The mechanisms to acquire interfaces to a compliant implementation.
- The interfaces through which rule execution sets are invoked by runtime clients of a complaint implementation.
- The interfaces through which rule execution sets are loaded from external resources and registered for use by runtime clients of a compliant implementation.

The following items are outside the scope of the specification:

- The binary representation of rules and rule execution sets.
- The syntax and file-formats of rules and rule execution sets.
- The semantics of interpreting rules and rule execution sets.
- The mechanism by which rules and rule execution sets are transformed for use by a rule engine.
- All minimal system requirements required to support a compliant implementation."

Spring Modules provides a support for this specification in order to simply the use of its APIs according to the philosophy of the Spring framework.

11.2. JSR94 support

This section describes the different abstractions to configure in order to administer and use rule engines with the JSR94 support.

11.2.1. Provider

The first step to use JSR94 in a local scenario is to configure the rule engine provider. You must specify its name with the *provider* property and its implementation class with the *providerClass* property.

These properties are specific to the used rule engine. For more informations about the configuration of different rule engines, see the following configuration section.

Here is a sample configuration of a rule provider:

```
<bean id="ruleServiceProvider"
      class="org.springmodules.jsr94.factory.DefaultRuleServiceProviderFactoryBean">
  <property name="provider">
    <value>org.jcp.jsr94.jess</value>
  </property>
  <property name="providerClass">
    <value>org.jcp.jsr94.jess.RuleServiceProviderImpl</value>
  </property>
</bean>
```

Important note: When you get the JSR94 *RuleAdministrator* and *RuleRuntime* from JNDI, you don't need to configure this bean in Spring.

11.2.2. Administration

There are two possibilities to configure the *RuleAdministrator* abstraction:

- Local configuration as a bean.
- Remote access from JNDI.

These two scenarios are supported. Firstly, the local configuration uses the *RuleAdministratorFactoryBean* which needs to have a reference to the JSR94 provider, configured in the previous section, with its *serviceProvider* property.

```
<bean id="ruleAdministrator" class="org.springmodules.jsr94.factory.RuleAdministratorFactoryBean">
  <property name="serviceProvider">
    <ref local="ruleServiceProvider"/>
  </property>
</bean>
```

The version 0.1 doesn't support the configuration of a *RuleAdministrator* from JNDI with a Spring's *FactoryBean*.

11.2.3. Execution

As for the RuleRuntime abstraction, there are two possibilities to configure the *RuleRuntime* abstraction (local and from JNDI).

Here is a sample of local configuration as bean:

```
<bean id="ruleRuntime" class="org.springmodules.jsr94.factory.RuleRuntimeFactoryBean">
  <property name="serviceProvider">
    <ref local="ruleServiceProvider"/>
  </property>
</bean>
```

The version 0.1 doesn't support the configuration of a *RuleRuntime* from JNDI with a Spring's *FactoryBean*.

11.2.4. Definition of a ruleset

To administer and execute rules, the JSR94 support introduces the *RuleSource* abstraction. It provides two different features:

- Automatic configuration of the rule or ruleset for a rule engine.
- Wrapper of JSR94 APIs for execution.

Important note: A *RuleSource* is a representation of an unique rule or ruleset.

These two features work respectively upon the JSR94 *RuleAdministrator* and *RuleRuntime* abstractions. That's why , to configure the *RuleSource*, you have two possibilities:

- Firstly, you can inject these two beans previously configured (see the two previous sections).
- Secondly, you can inject the JSR94 provider. So the rule source will create automatically these two beans .

You need to specify too some specific properties for the rule:

- Bind uri of the rule. The value of the *bindUri*property will be use when invoking the corresponding rule.
- Implementation of the rule. The JSR94 support is based on the Spring resource concept and the *source* property is managed in this way. So, by default, the ruleset source file is looked for in the classpath.

Here is a sample rule set configuration using the *DefaultRuleSource* class with a *RuleRuntime* and a *RuleAdministrator*:

```
<bean id="ruleSource" class="org.springmodules.jsr94.rulesource.DefaultRuleSource">
  <property name="ruleRuntime">
    <ref local="ruleRuntime"/>
  </property>
  <property name="ruleAdministrator">
    <ref local="ruleAdministrator"/>
  </property>
  <property name="source">
    <value>/testagent.drl</value>
  </property>
  <property name="bindUri">
    <value>testagent</value>
  </property>
</bean>
```

Here is an other sample of rule set configuration using the *DefaultRuleSource* with a *RuleServiceProvider*.

```
<bean id="ruleSource" class="org.springmodules.jsr94.rulesource.DefaultRuleSource">
  <property name="ruleServiceProvider">
    <ref local="ruleServiceProvider"/>
  </property>
  <property name="source">
    <value>/testagent.drl</value>
  </property>
  <property name="bindUri">
    <value>testagent</value>
  </property>
</bean>
```

Important note: If you don't specify the bindUri property, the JSR94 support will use the string returned by the *getName* method of the underlying *RuleExecutionSet* created for the *RuleSource*.

On the other hand, JSR94 provides some ways to specify additional configuration properties for specific rule engines.

Firstly some rule engines need to have custom properties to configure rules. These properties can be specify with the *ruleSetProperties* property of the type *Map*. This property is passed to the *createRuleExecutionSet* method (the last argument) of the JSR94 LocalRuleExecutionSetProvider interface. For example, JRules needs to specifiy the *rulesetProperties* property (see the configuration section).

Then some parameters need to be specified in order to get an implementation of the JSR94 *LocalRuleExecutionSetProvider* abstraction. These properties can be specify with the *providerProperties* property as a map.

Finally some parameters need to be specified in order to register a JSR94 *RuleExecutionSet* implementation.

Here is the the code of the registerRuleExecutionSets method of *DefaultRuleSource* class to show how the previous maps are used. Note that the *DefaultRuleSource* class is the default implementation of the *RuleSource* interface of the JSR94 support.

```
RuleExecutionSet ruleExecutionSet = ruleAdministrator.
    getLocalRuleExecutionSetProvider(providerProperties).createRuleExecutionSet(source.getInputStream(), ruleset
ruleAdministrator.registerRuleExecutionSet(bindUri, ruleExecutionSet, registrationProperties);
```

11.2.5. Configure the JSR94 template

In order to execute rules, you need to use the dedicated *JSR94Template* class. This class must be configured with a RuleSource instance.

There are two ways to configure this class.

Firstly, you can define the template directly in Spring as a bean. In this way, you can make your service extend the *Jsr94Support* abstract class. This class defines get/set methods for the *JSR94Template* and provides the associated template to the service thanks to the *getJSR94Template* method.

```
<bean id="ruleSource" class="org.springmodules.jsr94.rulesource.DefaultRuleSource">
  ...
</bean>

<bean id="jsr94Template" class="org.springmodules.jsr94.core.Jsr94Template">
  <property name="ruleSource"><ref local="ruleSource"/></property>
</bean>

<bean id="myService" class="MyService">
  <property name="template"><ref local="jsr94Template"/></property>
</bean>
```

Secondly, you can directly inject the configured *RuleSource* in your service. You can make too your service extend the *JSR94Support* abstract class. This class defines get/set methods for the *RuleSource*, creates automatically and provides the associated template to the service thanks to the *getJSR94Template* method.

```
<bean id="ruleSource" class="org.springmodules.jsr94.rulesource.DefaultRuleSource">
  ...
</bean>

<bean id="myService" class="MyService">
  <property name="ruleSource">
    <ref local="ruleSource"/>
  </property>
</bean>
```

Then the MyService class can directly use the template (injected or created with the *RuleSource*) with the help of the *getJSR94Template* method.

```
public class MyServiceImpl extends JSR94Support implements MyService {
  public void serviceMethod() {
    getJSR94Template.execute(...);
  }
}
```

Important note: Because Java doesn't support multiple inheritance, you can't always extend Jsr94Support class because your service classes can already have a super class. In this case, you need to define the get/set methods or instance the template by yourself.

11.2.6. Using the JSR94 template

In order to execute rules, you need to use the *JSR94Template* class configured in the previous section.

JSR94 defines two session modes to execute rules. A session is a runtime connection between the client and the rule engine.

- Stateless mode. "A stateless rule session provides a high-performance and simple API that executes a rule execution set with a List of input objects." (quotation of the JSR94 specification)
- Stateful mode. "A stateful rule session allows a client to have a prolonged interaction with a rule execution set. Input objects can be progressively added to the session and output objects can be queried repeatedly." (quotation of the JSR94 specification)

So this template defines two corresponding executing methods: *executeStateless* for stateless sessions and *executeStateful* for stateful ones.

To execute rules in a stateless mode, you need to use the following execute method of the template.

```
public Object executeStateless(final String uri, final Map properties,
                            final StatelessRuleSessionCallback callback) {
  //...
}
```

This method needs an implementation of the callback interface, *StatelessRuleSessionCallback*. This interface defines a method to which an instance of *StatelessRuleSession* is provided. The developer doesn't need to deal with the release of the resources and the management of technical exceptions.

Moreover, if you need to specify additional parameters to create the session, you can use the second parameter of the method (named properties and which is a map).

```
public interface StatelessRuleSessionCallback {
  Object execute(StatelessRuleSession session)
      throws InvalidRuleSessionException, RemoteException;
}
```

Here is a sample of use:

```
List inputObjects=...;
List outputObjects=getTemplate().executeStateless("ruleBindUri",null,
  new StatelessRuleSessionCallback() {
    public Object execute(StatelessRuleSession session)
                  throws InvalidRuleSessionException, RemoteException {
      return session.executeRules(inputObjects);
    }
});
```

The JSR94 support uses the same features to execute rules in a stateful mode. Here is the dedicated executing method.

```
public Object executeStateful(final String uri, final Map properties,
                          final StatefulRuleSessionCallback callback) {
  //...
}
```

This method needs an implementation of the callback interface, *StatefulRuleSessionCallback*. This interface defines a method to which an instance of *StatefulRuleSession* is provided. As for stateless sessions, the developer doesn't need to deal with the release of the resources and the management of technical exception.

Moreover, if you need to specify additional parameters to create the session, you can use the second parameter of the method (named properties and which is a map).

```
public interface StatefulRuleSessionCallback {
  Object execute(StatefulRuleSession session)
    throws InvalidRuleSessionException, InvalidHandleException, RemoteException;
}
```

Here is a sample of use:

```
List inputObjects=...;
List outputObjects=getTemplate().executeStateful("ruleBindUri",null,
  new StatefulRuleSessionCallback() {
    public Object execute(StatelessRuleSession session)
                  throws InvalidRuleSessionException, RemoteException {
      statefulRuleSession.addObjects(inputs);
      statefulRuleSession.executeRules();
      return statefulRuleSession.getObjects();
    }
});
```

11.3. Configuration with different engines

This section will describe the way to configure different rule engines in Spring using the JSR 94 support. This section describes the configuration of the following rule engines:

- Ilog JRules. See http://www.ilog.com/products/jrules/.
- Jess. See http://herzberg.ca.sandia.gov/jess/.
- Drools. See http://drools.org/.

Although all samples inject *RuleRuntime* and *RuleAdministrator* instances, you can inject the JSR94 provider used directly in a local scenario (according to previous sections of the documentation).

11.3.1. JRules

With JSR94, you can only access rules configured in an embedded rule engine. At the time of writing, JRules 5.0 doesn't provide an implementation of JSR94 to execute and administer rules deployed in an BRES (Business Rule Engine Server).

Important note: To use the BRES with Spring, you need to make your own integration code direclty based on the JRules APIs.

Firstly you need to configure the JSR94 provider specific to JRules. The name of the class for JRules is *ilog.rules.server.jsr94.IlrRuleServiceProvider*. There is no need to define specific parameters for the RuleRuntime and RuleAdministrator beans.

```
<bean id="ruleServiceProvider"
       class="org.springmodules.jsr94.factory.DefaultRuleServiceProviderFactoryBean">
  <property name="provider"><value>http://www.ilog.com</value></property>
  <property name="providerClass">
    <value>ilog.rules.server.jsr94.IlrRuleServiceProvider</value>
  </property>
</bean>

<bean id="ruleRuntime" class="org.springmodules.jsr94.factory.RuleRuntimeFactoryBean">
  <property name="serviceProvider"><ref local="ruleServiceProvider"/></property>
</bean>

<bean id="ruleAdministrator"
      class="org.springmodules.jsr94.factory.RuleAdministratorFactoryBean">
  <property name="serviceProvider"><ref local="ruleServiceProvider"/></property>
</bean>
```

Then you need to configure the different rulesets for the embedded rule engine. In order to do this, the *DefaultRuleSource* can be used. You need to inject the instances of *RuleRuntime* and *RuleAdministrator*, specifiy the source of the ruleset (an irl file in the case of JRule and the binding uri for this ruleset.

Note: The language to write JRules' ruleset is IRL (Ilog Rule Language). This language is similar to Java and introduces specific keyworks for rules.

Endly, you need to configure specific properties for JRules:

- *IlrName*: This key describes the internal name of the configured ruleset.
- *IlrRulesInILR*: This key specifies that the ruleset of the configured file is written in IRL.

```
<bean id="ruleSource" class="org.springmodules.jsr94.rulesource.DefaultRuleSource">
  <property name="ruleRuntime"><ref local="ruleRuntime"/></property>
  <property name="ruleAdministrator"><ref local="ruleAdministrator"/></property>
  <property name="source"><value>/cars_rules.irl</value></property>
  <property name="bindUri"><value>cars</value></property>
  <property name="rulesetProperties">
    <map>
      <entry key="IlrName"><value>cars_rules</value></entry>
      <entry key="IlrRulesInILR"><value>true</value></entry>
    </map>
  </property>
</bean>
```

11.3.2. Jess

The reference implementation of the JSR94 specification is a wrapper for the Jess rule engine. We have used the samples provided in the specification to describe the configuration of this rule engine.

Firstly you need to configure the *RuleServiceProvider*, *RuleAdministrator* and *RuleRuntime* abstractions as beans in Spring.

```
<bean id="ruleServiceProvider"
      class="org.springmodules.jsr94.factory.DefaultRuleServiceProviderFactoryBean">
  <property name="provider"><value>org.jcp.jsr94.jess</value></property>
  <property name="providerClass">
```

```
    <value>org.jcp.jsr94.jess.RuleServiceProviderImpl</value>
  </property>
</bean>

<bean id="ruleRuntime"
      class="org.springmodules.jsr94.factory.RuleRuntimeFactoryBean">
  <property name="serviceProvider"><ref local="ruleServiceProvider"/></property>
</bean>

<bean id="ruleAdministrator"
      class="org.springmodules.jsr94.factory.RuleAdministratorFactoryBean">
  <property name="serviceProvider"><ref local="ruleServiceProvider"/></property>
</bean>
```

Then you need to configure rulesets in Spring using the JSR94 support.

```
<bean id="ruleSource" class="org.springmodules.jsr94.rulesource.DefaultRuleSource">
  <property name="ruleRuntime"><ref local="ruleRuntime"/></property>
  <property name="ruleAdministrator"><ref local="ruleAdministrator"/></property>
  <property name="source"><value>/org/jcp/jsr94/tck/tck_res_1.xml</value></property>
  <property name="bindUri"><value>tck_res_1</value></property>
</bean>
```

Jess doesn't need specific additional configuration for the rule source.

11.3.3. Drools

An other interesting rule engine is Drools. It provides too an integration with JSR94. We have used the samples provided in the Drools distribution to describe its configuration.

Firstly you need to configure the *RuleServiceProvider*, *RuleAdministrator* and *RuleRuntime* abstractions as beans in Spring.

```
<bean id="ruleServiceProvider"
      class="org.springmodules.jsr94.factory.DefaultRuleServiceProviderFactoryBean">
  <property name="provider"><value>http://drools.org/</value></property>
  <property name="providerClass">
    <value>org.drools.jsr94.rules.RuleServiceProviderImpl</value>
  </property>
</bean>

<bean id="ruleRuntime"
      class="org.springmodules.jsr94.factory.RuleRuntimeFactoryBean">
  <property name="serviceProvider"><ref local="ruleServiceProvider"/></property>
</bean>

<bean id="ruleAdministrator"
      class="org.springmodules.jsr94.factory.RuleAdministratorFactoryBean">
  <property name="serviceProvider"><ref local="ruleServiceProvider"/></property>
</bean>
```

Then you need to configure rulesets in Spring using the JSR94 support.

```
<bean id="ruleSource" class="org.springmodules.jsr94.rulesource.DefaultRuleSource">
  <property name="ruleRuntime"><ref local="ruleRuntime"/></property>
  <property name="ruleAdministrator"><ref local="ruleAdministrator"/></property>
  <property name="source"><value>/testagent.drl</value></property>
  <property name="bindUri"><value>testagent</value></property>
</bean>
```

As Jess, Drools doesn't need specific additional configuration for the rule source.

Chapter 12. Lucene

12.1. Introduction

According to the home page project, "Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform".

The project is hosted by Apache. It allows to make scalable architecture based on distributed indexes and provides several kinds of indexes (in-memory, file-system based, database based).

Spring Modules offers a Lucene support in order to provide more flexibility in its APIs use. It adds some new abstractions to facilitate the management of *IndexReader*, *IndexWriter* and *Searcher*, the index locking and concurrent accesses, the query creation and the results extraction. It provides too facilities to index easily sets of files and database rows.

The support provides too a thin layer upon the Lucene API in order to hide the underlying resources and to make easier unit tests of classes using Lucene. As a matter of fact, Lucene does not use interfaces and is essentially based on concrete entities.

On the other hand, the support provides a generic document handling feature in order to offer dedicated entities to create documents . This feature manages too the document and handler association. The handler has the responsability to create a document from an object or an *InputStream*. The feature is particularly useful to manage the indexing of different file formats.

The Open Source community provides too an interesting tool which makes easier the use of Lucene, the Compass framework. The Lucene support of Spring Modules is different from this tool because the later hides all the interactions with the index by using a *CompassSession*, an entity similar to the Hibernate *Session*. The aim of the Lucene support is to leave access to the root resources of Lucene for more flexibilty.

On the other hand, if you are looking for a tool to manage paradigm conversions like object, resource or xml to index, the Compass framework is the right tool for you. This later provides too supports and integrations with different tools and frameworks like Hibernate with its GPS feature and allows the use of transactions upon Lucene, feature not supported natively by Lucene.

12.2. Indexing

In this section, we will describe how the Lucene support makes easier the manage of a Lucene index. We firtly deal with the root entities of the support and the way to configure them, then show how to interact with the index and finally describe how to manage concurrency.

12.2.1. Root entities

Lucene provides two main root entities in order to interact with an index and index documents, the classes *IndexReader* and *IndexWriter*. These classes are concrete and can make difficult the implementation of the tests unit. That's why the Lucene support introduces two interfaces, respectively the interfaces *LuceneIndexReader* and *LuceneIndexWriter*, in order to define the contracts of those classes. So these interfaces offers the same methods as the Lucene classes *IndexReader* and *IndexWriter*.

The following code describes the methods offered by the *LuceneIndexReader* interface:

```
public interface LuceneIndexReader {
    void close() throws IOException;
    void deleteDocument(int docNum) throws IOException;
    int deleteDocuments(Term term) throws IOException;
    Directory directory();
    int docFreq(Term t) throws IOException;
    Document document(int n) throws IOException;
    Collection getFieldNames(IndexReader.FieldOption fldOption);
    TermFreqVector getTermFreqVector(int docNumber, String field) throws IOException;
    TermFreqVector[] getTermFreqVectors(int docNumber) throws IOException;
    long getVersion();
    boolean hasDeletions();
    boolean hasNorms(String field) throws IOException;
    boolean isCurrent() throws IOException;
    boolean isDeleted(int n);
    int maxDoc();
    byte[] norms(String field) throws IOException;
    void norms(String field, byte[] bytes, int offset) throws IOException;
    int numDocs();
    void setNorm(int doc, String field, byte value) throws IOException;
    void setNorm(int doc, String field, float value) throws IOException;
    TermDocs termDocs() throws IOException;
    TermDocs termDocs(Term term) throws IOException;
    TermPositions termPositions() throws IOException;
    TermPositions termPositions(Term term) throws IOException;
    TermEnum terms() throws IOException;
    TermEnum terms(Term t) throws IOException;
    void undeleteAll() throws IOException;
    LuceneSearcher createSearcher();
    Searcher createNativeSearcher();
}
```

The following code describes the methods offered by the *LuceneIndexWriter* interface:

```
public interface LuceneIndexWriter {
    void addDocument(Document doc) throws IOException;
    void addDocument(Document doc, Analyzer analyzer) throws IOException;
    void addIndexes(Directory[] dirs) throws IOException;
    void addIndexes(IndexReader[] readers) throws IOException;
    void close() throws IOException;
    int docCount();
    Analyzer getAnalyzer();
    long getCommitLockTimeout();
    Directory getDirectory();
    PrintStream getInfoStream();
    int getMaxBufferedDocs();
    int getMaxFieldLength();
    int getMaxMergeDocs();
    int getMergeFactor();
    Similarity getSimilarity();
    int getTermIndexInterval();
    boolean getUseCompoundFile();
    long getWriteLockTimeout();
    void optimize() throws IOException;
    void setCommitLockTimeout(long commitLockTimeout);
    void setInfoStream(PrintStream infoStream);
    void setMaxBufferedDocs(int maxBufferedDocs);
    void setMaxFieldLength(int maxFieldLength);
    void setMaxMergeDocs(int maxMergeDocs);
    void setMergeFactor(int mergeFactor);
    void setSimilarity(Similarity similarity);
    void setTermIndexInterval(int interval);
    void setUseCompoundFile(boolean value);
    void setWriteLockTimeout(long writeLockTimeout);
}
```

The main advantage of this mechanism is the possibility to dissociate logical and physiacal resources. The physical resources are the Lucene resources which directly interact with the index, i.e. the instances of *IndexReader* and *IndexWriter*.

The logical resources are high level resources which allow a more flexible management of resources in order to integrate concurrency and transaction managements. The logical resources are not provided by Lucene but interfaces of the Lucene support, the interfaces *LuceneIndexReader* and *LuceneIndexWriter*.

In order to create these resources, the Lucene support implements the factory pattern based on the *IndexFactory* interface. This interface allows and hide the creation of logical resources. So, with this mechanism, you only need to configure an implementation of this interface in order to specify the strategy of resource management.

The following code describes the methods offered by the *IndexFactory* interface:

```
public interface IndexFactory {
    LuceneIndexReader getIndexReader();
    LuceneIndexWriter getIndexWriter();
}
```

Because the factory handle only logical resources, it do not provide directly instances of *IndexReader* and *IndexWriter*. The latters are managed implementations of the *LuceneIndexReader* and *LuceneIndexWriter*.

The Lucene support introduces the following implementations of the *IndexFactory* interface:

Table 12.1. Different implementations of the IndexFactory interface

IndexFactory implementation	Logical resource implementations	Description
SimpleIndexFactory	SimpleLuceneIndexReader and SimpleLuceneIndexWriter	Simple wrapping of the physical ressources.

12.2.2. Configuration

Spring Modules provides at this time only one index factory based on a directory and an analyzer. It provides too support for configuring the main directory types.

12.2.2.1. Configuring directories

The root Lucene concept is the *Directory* which represents physically the index. Lucene supports different types of storage of the index. The Lucene support allows to configure an in-memory index (a RAM directory) and a persistent index (file system directory) with dedicated Spring *FactoryBeans*.

The first type of *Directory*, the RAM directory, can be configured using the *RAMDirectoryFactoryBean* class, as in the following code:

```
<bean id="ramDirectory" class="org.springmodules.lucene.index.support.RAMDirectoryFactoryBean"/>
```

You must be careful when you use this type of storage because all the informations of the index are in memory and are never persist on the disk.

The second type of *Directory*, the file system directory, can be configure using the *FSDirectoryFactoryBean* class. This class is much more advanced because it allows to manage the creation of the index. The only mandatory property is location which specifies the location of the index on the disk basing on the facilities of the *Resource* interface of Spring.

The following code describes how to configure a directory of this type:

```
<bean id="fsDirectory" class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/temp/lucene"/>
```

```
</bean>
```

The sandbox of the Lucene project defines other kinds of directory (database...) which are not supported by the Lucene support at this time.

12.2.2.2. Configuring a SimpleIndexFactory

The *SimpleIndexFactory* class is the default factory to manipulate index. This entity provides logical resources which simply wrap the physical resources on the index basing on the *SimpleLuceneIndexReader* and *SimpleLuceneIndexWriter* classes.

This factory must be configured with the Lucene directory to access and eventually a default analyzer. In order to configure this class in a Spring application context, the support provides the *SimpleIndexFactoryBean* class whose configuration is described below:

```
<bean id="fsDirectory" class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="C:/temp/lucene"/>
</bean>

<bean id="indexFactory" class="org.springmodules.lucene.index.support.SimpleIndexFactoryBean">
    <property name="directory" ref="fsDirectory"/>
    <property name="analyzer">
        <bean class="org.apache.lucene.analysis.SimpleAnalyzer"/>
    </property>
</bean>
```

The SimpleIndexFactory class allows too to management the creation and the locking of the index using respectively the *resolveLock* and *create* properties. The default values of these properties are *false*. The first one specifies that the index will be automaticaly unlock if it is lock during the first resource creation. The second specifies that the index structure will be created if not during the creation of the first *IndexWriter*.

This factory is based on the *IndexReaderFactoryUtils* and *IndexWriterFactoryUtils* classes to manage the *LuceneIndexReader* and *LuceneIndexWriter* creation and getting. However, no concurrency management is provided by this entity. You must be aware that opening an index in a write mode will lock it until the writer is closed. Moreover some operations are forbidden between the reader and the writer (for example, a document delete using the reader and document addition using the writing).

For more informations, see the following section about the *IndexFactory* management.

12.2.2.3. Dedicated namespace

The Lucene support provides a dedicated namespace *lucene* which make easier the configuration of an index and its associated *IndexFactory* based on the *index* tag. The configuration of the namespace is similar to those of the standard namespaces of Spring and is shown in the following code:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lucene="http://www.springmodules.org/schema/lucene"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
              http://www.springmodules.org/schema/lucene
                http://www.springmodules.org/schema/lucene/lucene-index.xsd">
    (...)
</beans>
```

An IndexFactory bean is configured with the *index* tag of this namespace. This tag supports both file system and ram directory thanks to the *location* attribute. This attribute allows to configure the location of an filesystem index. The following code describes the configuration of ram and filesystem *IndexFactory*:

```
<beans>
    <!-- Configuration of the ram directory -->
    <lucene:index/>

    <!-- Configuration of the filesystem directory -->
    <lucene:index location="/temp/lucene"/>
</beans>
```

In addition, an analyzer can be configured for the index tag. Two approaches are possible: the firts allows to reference a previously configured analyzer bean with the *analyzer-ref* attribute and the second to configure an analyzer as inner bean. The following code describes how to configure these approaches:

```
<beans>
    <!-- Configuration of an analyzer with the analyzer-ref attribute -->
    <bean id="analyzer" class="org.apache.lucene.analysis.SimpleAnalyzer"/>
    <lucene:index id="fsDirectory1" location="/temp/lucene" analyzer-ref="analyzer"/>

    <!-- Configuration of the filesystem directory -->
    <lucene:index id="fsDirectory2" location="/temp/lucene">
        <lucene:analyzer>
            <bean class="org.apache.lucene.analysis.SimpleAnalyzer"/>
        </lucene:analyzer>
    </lucene:index>
</beans>
```

12.2.3. Document type handling

The informations used to populate the index can come from different sources (types of files, objects...). An unified support is provided in order to

The central entity of this unified support is the interface *DocumentHandler* which defines the contract to create a document from an object. *DocumentHandler* is an high level interface while it does not tie to any source: it have no dependency with the Java IO API. The following code describes this interface:

```
public interface DocumentHandler {
    boolean supports(Class clazz);
    Document getDocument(Map description, Object object) throws Exception;
}
```

The interface provides two different methods. The first, the method *supports*, specifies which class the implementation of the interface will be able to handle. The second, the method *getDocument*, has the responsability to create a document from an object and different other informations.

Different implementations of the interface DocumentHandler are provided in the support to handle file formats, as shown in the following table:

Table 12.2. Different implementations of the DocumentHandler interface

Format	Tool	Implementation
Texte	-	TextDocumentHandler
PDF	PdfBox	PdfBoxDocumentHandler
Rtf	-	DefaultRtfDocumentHandler
Excel	JExcel	JExcelDocumentHandler
Excel	POI	POIExcelDocumentHandler

Format	Tool	Implementation
Word	POI	POIWordDocumentHandler

Other implementations are also provided in order to create Lucene documents from POJO. The informations to index are determined used metadatas configured from different ways, as shown in the following table:

Table 12.3. Different implementations of the DocumentHandler interface

Metadatas	Implementation
Properties file	PropertiesDocumentHandler
Object (with reflection)	ReflectiveDocumentHandler
Annotations	AnnotationDocumentHandler

The support offers too a generic entity which allows to handle the association between entities and document handlers. The aim is .

This mechanism is by the *DocumentHandlerManager* interface which allows to determine the document handler to use in order to create a Lucene document from a file or an object. If no document handler is available, a *DocumentHandlerException* is thrown. Other methods are provided too in order to register and unregister document handlers. The following code shows the code of the *DocumentHandlerManager* interface:

```
public interface DocumentHandlerManager {
    DocumentHandler getDocumentHandler(String name);
    void registerDefaultHandlers();
    void registerDocumentHandler(DocumentMatching matching, DocumentHandler handler);
    void unregisterDocumentHandler(DocumentMatching matching);
}
```

In order to determine with which document handler a entity type is associated, the *DocumentMatching* interface is introduced. The latter defines only one method which allows to check if a String corresponds to an internal criteria, as shown in the following code:

```
public interface DocumentMatching {
    boolean match(String name);
}
```

This mechanism allows to make correspond several entities to one document handle based on a criteria like an regexp for example. Different implementations of this interface are provided by the support, as shown in the following table:

Table 12.4. Different implementations of the DocumentHandler interface

Implementation	Description
IdentityDocumentMatching	PropertiesDocumentHandler
Object (with reflection)	ReflectiveDocumentHandler
Annotations	AnnotationDocumentHandler

According to the method signatures of the *DocumentHandlerManager* and *DocumentMatching* interfaces, the support offers the possibility to use different implementations of the *DocumentMatching* interface with the same *DocumentHandlerManager* entity.

Endly, a dedicated *FactoryBean* is provided by the support in order to configure programmatically *DocumentHandler* and when use them. This *FactoryBean* is generic and corresponds to a *DocumentHandlerManager*. In order to configure it, you must specify the implementations of *DocumentHandlerManager* and *DocumentMatching* interfaces to use basing respectively on the *documentHandlerManagerClass* and *documentMatchingClass* properties. By default, the implementations of these entities are *DefaultDocumentHandlerManager* and *IdentityDocumentMatching*.

The following code shows how to configure this *FactoryBean*:

```
<bean id="documentHandlerManager"
      class="org.springmodules.lucene.index.document.handler.DocumentHandlerManagerFactoryBean">
    <property name="documentHandlerManagerClass"
         value="org.springmodules.lucene.index.document.handler.DefaultDocumentHandlerManager"/>
    <property name="documentMatchingClass"
        value="org.springmodules.lucene.index.document.handler.file.ExtensionDocumentMatching"/>
    (...)
</bean>
```

The *FactoryBean* registers automatically the default handlers of the *DocumentHandlerManager* by calling its *registerDefaultHandlers* method. You can too register programmatically other *DocumentHandler*, as shown in the following code:

```
<bean id="documentHandlerManager"
      class="org.springmodules.lucene.index.document.handler.DocumentHandlerManagerFactoryBean">
    <property name="documentMatchingClass"
        value="org.springmodules.lucene.index.document.handler.file.ExtensionDocumentMatching"/>
    <property name="documentHandlers">
        <map>
            <entry key="pdf">
                org.springmodules.lucene.index.document.handler.file.PdfBoxDocumentHandler
            </entry>
            (...)
        </map>
    </property>
</bean>
```

We will see later that all these entities can be use directly or internally by templates. Moreover, by default, all the document handler implementations seen above are automatically registred.

The lucene dedicated namespace previously described provides too a support to easily configure a document handler manager thanks to a *document-handler* tag. Its supports the matching of documents with both identity and extension and can be configured as shown below:

```
<!-- Document handler manager using an extension matching -->
<lucene:document-handler id="documentHandlerManager1" type="extension"/>

<!-- Document handler manager using an identity matching -->
<lucene:document-handler id="documentHandlerManager2" type="identity"/>
```

12.2.4. Template approach

The Lucene support provides a template approach like Spring JDBC to make easier the use and the manipulation of an index.

LuceneIndexTemplate is the central interface of the Lucene support core package *org.springmodules.lucene.index.core*) for the indexing. It simplifies the use of the corresponding Lucene APIs since it handles the creation and release of resources and allow you to configure declaratively the resource management. This helps to avoid common errors like forgetting to always close the index reader/writer. It executes the common operations on an index leaving application code the way to create or delete a document and questionwork on the index (numDoc property, optimization of an index, deleted documents...).

The Lucene support provides a default implementation of this interface, the *DefaultLuceneIndexTemplate* class, which is created and used by default.

12.2.4.1. Template configuration and getting

In order to configure and get an instance of a LuceneIndexTemplate, the support provides the *LuceneIndexSupport* class. It allows to inject instances of the *IndexFactory* and *DocumentHandlerManager* interfaces and of the *Analyzer* class. These entities are used to create an instance of a template which can be reached by using the *getTemplate* method.

The following code shows a class based on the *LuceneIndexSupport* class:

```
public class TestIndexImpl extends LuceneIndexSupport implements TestIndex {
    public void getElement() {
        LuceneIndexTemplate template = getLuceneIndexTemplate();
        (...)
    }
}
```

The following code shows the configuration of the *TestIndexImpl* class in a Spring application context:

```
<bean id="indexFactory" class="org.springmodules.lucene.index.support.SimpleIndexFactoryBean">
    (...)
</bean>

<bean id="testIndex" class="org.springmodules.lucene.samples.index.TestIndexImpl">
    <property name="indexFactory" ref="indexFactory"/>
</bean>
```

12.2.4.2. Basic operations

The *LuceneIndexTemplate* class provides the basic operations in order to manipulate an index: create, update and delete documents. Different groups of methods can be distinguished, as shown in the following code:

```
public interface LuceneIndexTemplate {
    (...)
    /* Document(s) addition(s) */
    void addDocument(Document document);
    void addDocument(Document document, Analyzer analyzer);
    void addDocument(DocumentCreator creator);
    void addDocument(DocumentCreator documentCreator, Analyzer analyzer);
    void addDocuments(List documents);
    void addDocuments(List documents, Analyzer analyzer);
    void addDocuments(DocumentsCreator creator);
    void addDocuments(DocumentsCreator creator, Analyzer analyzer);

    /* Document(s) update(s) */
    void updateDocument(Term identifierTerm, DocumentModifier documentModifier);
    void updateDocument(Term identifierTerm, DocumentModifier documentUpdater, Analyzer analyzer);
    void updateDocuments(Term identifierTerm, DocumentsModifier documentsModifier);
    void updateDocuments(Term identifierTerm, DocumentsModifier documentsModifier, Analyzer analyzer);

    /* Document(s) deletion(s) */
    void deleteDocument(int internalDocumentId);
    void deleteDocuments(Term term);
    void undeleteDocuments();
    boolean isDeleted(int internalDocumentId);
    boolean hasDeletions();
    (...)
}
```

The first group of methods allows to create and add documents to an index. Lucene document can be used as parameters of the *addDocument* methods. The Lucene support provides too the interface *DocumentCreator* which defines the way to create the document, exceptions thrown during the creation of a document are now managed by the template. The following code describes this interface:

```
public interface DocumentCreator {
    Document createDocument() throws Exception;
}
```

The same mechanism is available to create and add several documents and the interface used is DocumentsCreator which returns a list of documents. The following code describes this interface:

```
public interface DocumentsCreator {
    List createDocuments() throws Exception;
}
```

The following code shows an example of creation of a document based on an *addDocument* method of the template:

```
getLuceneIndexTemplate().addDocument(new DocumentCreator() {
    public Document createDocument() throws Exception {
        Document newDocument = new Document();
        (...)
        return newDocument;
    }
});
```

Lucene do not provide support in order to modify a document in the index. An addition and a deletion must be made successively and you need to use an *IndexReader* instance and then an *IndexWriter* instance. The Lucene support provide an interface, the interface *DocumentModifier*, in order to specify how to update a document, as shown in the following code:

```
public interface DocumentModifier {
    Document updateDocument(Document document) throws Exception;
}
```

You can use then the *updateDocument* method to really update the document basing on this interface. The first parameter of these method, a parameter of type *Term*, is used to identify the document to update. It must identify only one document.

The template provides too two other methods in order to update several documents at the same time, the *updateDocuments* methods. These later use the same mechanism as the *updateDocument* method and are based on the *DocumentsModifier* interface which takes a list of documents to update, as shown in the following code:

```
public interface DocumentsModifier {
    List updateDocuments(LuceneHits hits) throws IOException;
}
```

You can use then the *updateDocument*s method to really update a set of documents basing on this interface. The first parameter of these method, a parameter of type *Term*, is used to identify the set of documents to update.

The following code shows a example of use of the *updateDocument* method:

```
getLuceneIndexTemplate.updateDocument(new Term("id", "anId"), new DocumentModifier() {
    public Document updateDocument(Document document) throws Exception {
        Document newDocument = new Document();
        (...)
        return newDocument;
    }
});
```

The last group of methods can be used to delete documents in the index. The *deleteDocument* method deletes only one document based on its internal identifier whereas the *deleteDocuments* deletes several documents based on a *Term*. The following code shows an example of use of this method:

```
getLuceneIndexTemplate.deleteDocument(new Term("attribute", "a value"));
```

12.2.4.3. Usage of InputStreams with templates

The template offers the possibility to create a document basing on an *InputStream* with two different *addDocument* methods, as shown in the following code:

```
public interface LuceneIndexTemplate {
    (...)
    void addDocument(InputStreamDocumentCreator creator);
    void addDocument(InputStreamDocumentCreator documentCreator, Analyzer analyzer);
    (...)
}
```

These later methods are the responsability to manage the *InputStream*, i.e. to get an instance of it, to manage *IOExceptions* and to close the *InputStream*.

These *addDocument* methods are based on the InputStreamDocumentCreator which specifies how to initialize the *InputStream* and use it in order to create a document. The following code shows the detail of this interface and its two methods, *createInputStream* and *createDocumentFromInputStream*:

```
public interface InputStreamDocumentCreator {
    InputStream createInputStream() throws IOException;
    Document createDocumentFromInputStream(InputStream inputStream) throws Exception;
}
```

The following code shows a sample use of this interface within the *addDocument* of the template:

```
final String fileName = "textFile.txt";

getTemplate().addDocument(new InputStreamDocumentCreator() {
    public InputStream createInputStream() throws IOException {
        return new FileInputStream(fileName);
    }

    public Document createDocumentFromInputStream(InputStream inputStream) throws Exception {
        Document document = new Document();
        String contents = IOUtils.getContents(inputStream);
        document.add(new Field("contents", contents, Field.Store.YES, Field.Index.TOKENIZED));
        document.add(new Field("fileName", fileName, Field.Store.YES, Field.Index.UN_TOKENIZED));
        return document;
    }
});
```

12.2.4.4. Usage of the DocumentHandler support with templates

The template offers the possibility to use the *DocumentHandler* support in order to create a document basing on an *InputStream*. This feature is based on the mechanism described in the previous section. An dedicated implementation of the *InputStreamDocumentCreator*, the *InputStreamDocumentCreatorWithManager* class, is provided. This class takes an instance of the *DocumentHandlerManager* interface and selects the right *DocumentHandler* to use in order to create the document from an *InputStream*.

The *InputStreamDocumentCreatorWithManager* class defines two abstract methods in order to select the name and the description of the resource associated with the *InputStream*, as following in the following code:

```
public abstract class InputStreamDocumentCreatorWithManager implements InputStreamDocumentCreator {
    (...)
    public InputStreamDocumentCreatorWithManager(DocumentHandlerManager documentHandlerManager) {
        this.documentHandlerManager = documentHandlerManager;
    }

    protected abstract String getResourceName();
    protected abstract Map getResourceDescription();
    (...)
}
```

Note that the InputStreamDocumentCreatorWithManager class must be initialized with an instance of the *DocumentHandlerManager* interface.

The following code shows an example of use of this class with an *addDocument* method of the template:

```
DocumentHandlerManager manager = (...)

final String fileName = "textFile.txt";
getLuceneIndexTemplate.addDocument(new InputStreamDocumentCreatorWithManager(manager) {
    public InputStream createInputStream() throws IOException {
        return new FileInputStream(fileName);
    }

    protected String getResourceName() {
        return fileName;
    }

    protected Map getResourceDescription() {
        return null;
    }
});
```

12.2.4.5. Work with root entities

Some other methods of the template allow you to work directly on logical resources of the index basing on callback interfaces and methods. The template uses these callbacks in order to provide instances of *LuceneIndexReader* and *LuceneIndexWriter* to the application. The following code describes the *read* and *write* methods of the template which are based on these callbak interfaces:

```
public interface LuceneIndexTemplate {
    (...)
    Object read(ReaderCallback callback);
    Object write(WriterCallback callback);
    (...)
}
```

These two methods are based on the *ReaderCallback* and *WriterCallback* which allow the template to give the resources to the application. The following code describes these two interfaces:

```
public interface ReaderCallback {
    Object doWithReader(LuceneIndexReader reader) throws Exception;
}

public interface WriterCallback {
    Object doWithWriter(LuceneIndexWriter writer) throws Exception;
}
```

The following code shows an sample of use of the *WriterCallback* interface with the template in order to index documents:

```
LuceneIndexTemplate template = (...)
template.write(new WriterCallback() {
    public Object doWithWriter(LuceneIndexWriter writer) throws IOException {
        Document document = new Document();
        (...)
        writer.addDocument(document);
        return null;
    }
});
```

12.2.4.6. Template and used resources

The *LuceneIndexTemplate* hides the resource used in order to execute an operation and its managment. The developer has now no need to know the Lucene API. The following table shows the underlying resources used by the template's methods:

Table 12.5. Resource used by the template methods

LuceneIndexTemplate method group	Corresponding resource used
deletion methods	IndexReader
addition methods	IndexWriter
get methods	IndexReader
optimize methods	IndexWriter

In the context of the *LuceneIndexTemplate*, the calls of different methods have sense only if the underlying resources stay opened during several calls. For exemple, the call of the *hasDeletion* method always returns false if resources are used only for the call of a method.

12.2.5. Mass indexing approach

The support offers facilities to index an important number of documents or datas from a directory (or a set of directory) or a database. It is divided into two parts:

- Indexing a directory and its sub directories recursively. This approach allows you to register custom handlers to index several file types.
- Indexing a database. This approach allows you to specify the SQL requests in order to get the datas to index. A callback is then provided to create a Lucene document from a *ResultSet*. this feature is based on the Spring JDBC framework.

Every classes of this approach are located in the *org.springmodules.lucene.index.object* package and its sub packages.

12.2.5.1. Indexing directories

Indexing directories is implemented by the *DirectoryIndexer* class. To use it, you simply call its *index* method which needs the base directory. This class will browse this directory and all its sub directories, and tries to index every files which have a dedicated handler.

```
public class DirectoryIndexer extends AbstractIndexer {
    (...)
    public void index(String dirToParse) { ... }
    public void index(String dirToParse,boolean optimizeIndex) { ... }
    (...)
}
```

Important note: If you set the *optimizeIndex* parameter as true, the index will be optimized after the indexing.

This class is based on a mechanism to handle different file types. It uses the *DocumentHandlerManager* interface seen in the previous section. It allows the indexer to be extended and supports other file formats.

You can add too listeners to be aware of directories and files processing. In this case, you only need to implement the *DocumentIndexingListener* on which different methods will be called during the indexing. So the implementation will receive the following informations:

- The indexer begins to handle all the files of a directory.
- The indexer has ended to handle all the files of a directory
- The indexing of a file begins.
- The indexing of a file is succesful.
- The indexing of a file has failed. The exception is provided to the callback.
- The indexer haven't the specific handler for the file type.

```
public interface DocumentIndexingListener {
    void beforeIndexingDirectory(File file);
    void afterIndexingDirectory(File file);

    void beforeIndexingFile(File file);
    void afterIndexingFile(File file);
    void onErrorIndexingFile(File file,Exception ex);
    void onNotAvailableHandler(File file);
}
```

To associate a listener with the indexer, you can simply use its *addListener* method and to remove one, the *removeListener* method. The following code describes these two methods:

```
public class DirectoryIndexer extends AbstractIndexer {
    (...)
    public void addListener(DocumentIndexingListener listener) { ... }
    public void removeListener(DocumentIndexingListener listener) { ... }
    (...)
}
```

The following code shows of use of all these entities:

```
public class SimpleDirectoryIndexingImpl
                implements DirectoryIndexing,InitializingBean {
    private IndexFactory indexFactory;
    private DocumentHandlerManager documentHandlerManager;
    private DirectoryIndexer indexer;

    public SimpleDirectoryIndexingImpl() {}

    public void afterPropertiesSet() throws Exception {
        if( indexFactory!=null ) {
            throw new IllegalArgumentException("indexFactory is required");
        }
        this.indexer = new DirectoryIndexer(indexFactory,documentHandlerManager);
    }

    public void indexDirectory(String directory) { indexer.index(directory,true); }

    public void prepareListeners() {
        DocumentIndexingListener listener = new DocumentIndexingListener() {
            public void beforeIndexingDirectory(File file) {
                System.out.println("Indexing the directory : "+file.getPath()+" ...");
            }
            public void afterIndexingDirectory(File file) {
                System.out.println(" -> Directory indexed.");
            }
            public void beforeIndexingFile(File file) {
                System.out.println("Indexing the file : "+file.getPath()+" ...");
            }
            public void afterIndexingFile(File file) {
                System.out.println(" -> File indexed ("+duration+").");
            }
            public void onErrorIndexingFile(File file, Exception ex) {
                System.out.println(" -> Error during the indexing : "+ex.getMessage());
            }
            public void onNotAvailableHandler(File file) {
                System.out.println("No handler registred for the file : "+file.getPath()+" ...");
            }
        };
        indexer.addListener(listener);
    }

    public IndexFactory getIndexFactory() { return indexFactory; }
    public void setIndexFactory(IndexFactory factory) { indexFactory = factory; }
    public DocumentHandlerManager getDocumentHandlerManager() {
        return documentHandlerManager;
    }
    public void setDocumentHandlerManager(DocumentHandlerManager manager) {
        documentHandlerManager = manager;
    }
}
```

The following code describes the configuration of the later class in a Spring application context:

```
<bean id="fsDirectory"
      class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/temp/lucene"/>
</bean>

<bean id="indexFactory"
      class="org.springmodules.lucene.index.support.SimpleIndexFactoryBean">
    <property name="directory" ref="fsDirectory"/>
    <property name="analyzer">
        <bean class="org.apache.lucene.analysis.SimpleAnalyzer"/>
    </property>
</bean>

<bean id="documentHandlerManager" class=
 "org.springmodules.lucene.index.object.file.ExtensionDocumentHandlerManagerFactoryBean">
    (...)
</bean>

<bean id="indexingDirectory"
  class="org.springmodules.samples.lucene.index.console.SimpleDirectoryIndexingImpl">
    <property name="indexFactory" ref="indexFactory"/>
    <property name="documentHandlerManager" ref="documentHandlerManager"/>
</bean>
```

12.2.5.2. Indexing databases

The support for the database indexing looks like the previous. It is implemented by the *DatabaseIndexer* class. To use it, you simply use its index method which needs the JDBC *DataSource* to use. This class will execute every sql requests registred, and tries to index every corresponding resultsets with the dedicated request handlers.

```
public class DatabaseIndexer extends AbstractIndexer {
    (...)
    void index(DataSource dataSource) { ... }
    void index(DataSource dataSource,boolean optimizeIndex) { ... }
    (...)
}
```

Important note: If you set the *optimizeIndex* parameter as true, the index will be optimized after the indexing.

This class is based on a mechanism to handle different queries. It allows the indexer to execute every specified requests. To make a new handler, we only need to implement the *SqlDocumentHandler* interface which specifies the way to construct a Lucene document from a result set.

```
public interface SqlDocumentHandler {
    Document getDocument(SqlRequest request,ResultSet rs) throws SQLException;
}
```

As you can see in the method signatures, we need to use the *SqlRequest* class to specify the SQL request to execute and its parameters. It defines two constructors according to the request (with or without parameters):

```
public class SqlRequest {
    (...)
    public SqlRequest(String sql) { ... }
    public SqlRequest(String sql,Object[] params,int[] types) { ... }
    (...)
}
```

To add and remove requests, you can respectively use the *registerDocumentHandler* and *unregisterDocumentHandler* methods. The following code describes these two methods:

```
public class DatabaseIndexer extends AbstractIndexer {
    (...)
    public void registerDocumentHandler(SqlRequest sqlRequest,
                                      SqlDocumentHandler handler) { ... }
    public void unregisterDocumentHandler(SqlRequest sqlRequest) { ... }
    (...)
}
```

You can add too listeners to be aware of requests processing. In this case, you only need to implement the *DatabaseIndexingListener* on which different methods will be called during the indexing. So the implementation will receive the following informations:

- The indexing of a request begins.
- The indexing of a request is succesful.
- The indexing of a request has failed. The exception is provided to the callback.

```
public interface DatabaseIndexingListener {
    void beforeIndexingRequest(SqlRequest request);
    void afterIndexingRequest(SqlRequest request);
    void onErrorIndexingRequest(SqlRequest request,Exception ex);
}
```

To associate a listener with the indexer, you can simply use its *addListener* method.

```
public class DatabaseIndexer extends AbstractIndexer {
    (...)
    public void addListener(DatabaseIndexingListener listener) { ... }
    public void removeListener(DatabaseIndexingListener listener) { ... }
    (...)
}
```

The following code shows of use of all these entities:

```
public class SimpleDatabaseIndexingImpl
                  implements DatabaseIndexing, InitializingBean {

    private DataSource dataSource;
    private IndexFactory indexFactory;
    private DatabaseIndexer indexer;

    public SimpleDatabaseIndexingImpl() {}

    public void afterPropertiesSet() throws Exception {
        if( indexFactory!=null ) {
            throw new IllegalArgumentException("indexFactory is required");
        }
        this.indexer=new DatabaseIndexer(indexFactory);
    }

    public void prepareDatabaseHandlers() {
        //Register the request handler for book_page table without parameters
        this.indexer.registerDocumentHandler(
                new SqlRequest("select book_page_text from book_page"),
                new SqlDocumentHandler() {
            public Document getDocument(SqlRequest request,
                                        ResultSet rs) throws SQLException {
                Document document=new Document();
                document.add(Field.Text("contents", rs.getString("book_page_text")));
                document.add(Field.Keyword("request", request.getSql()));
                return document;
            }
        });
    }

    public void indexDatabase() {
        indexer.index(dataSource,true);
    }

    public void prepareListeners() {
        DatabaseIndexingListener listener=new DatabaseIndexingListener() {
            public void beforeIndexingRequest(SqlRequest request) {
                System.out.println("Indexing the request : "+request.getSql()+" ...");
            }
            public void afterIndexingRequest(SqlRequest request) {
                System.out.println(" -> request indexed.");
            }
            public void onErrorIndexingRequest(SqlRequest request, Exception ex) {
                System.out.println(" -> Error during the indexing : "+ex.getMessage());
            }
        };
        indexer.addListener(listener);
    }

    public IndexFactory getIndexFactory() { return indexFactory; }
    public void setIndexFactory(IndexFactory factory) { indexFactory = factory; }
    public DataSource getDataSource() { return dataSource; }
    public void setDataSource(DataSource source) { dataSource = source; }
}
```

The following code describes the configuration of the later class in a Spring application context:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
</bean>

<bean id="fsDirectory"
      class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/temp/lucene"/>
```

```
</bean>

<bean id="indexFactory"
      class="org.springmodules.lucene.index.support.SimpleIndexFactoryBean">
    <property name="directory" ref="fsDirectory"/>
    <property name="analyzer">
        <bean class="org.apache.lucene.analysis.SimpleAnalyzer"/>
    </property>
</bean>

<bean id="indexingDatabase"
   class="org.springmodules.samples.lucene.index.console.SimpleDatabaseIndexingImpl">
    <property name="indexFactory" ref="indexFactory"/>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

12.3. Search

In this section, we will describe how the Lucene support makes easier the search on a Lucene index. We firtly deal with the root entities of the support and the way to configure them, then show how to make a search on the index.

12.3.1. Root entities

Lucene provides two entities in order to make search on index, the classes *Searcher* and *Hits*. These classes are concrete and can make difficult the implementation of the tests unit. That's why the Lucene support introduces two interfaces, respectively the interfaces *LuceneSearcher* and *LuceneHits*, in order to define the contracts of those classes. So these interfaces offers the same methods as the Lucene classes *Searcher* and Hits.

The following code describes the methods offered by the *LuceneSearcher* interface:

```
public interface LuceneSearcher {
    void close() throws IOException;
    Document doc(int i) throws IOException;
    int docFreq(Term term) throws IOException;
    int[] docFreqs(Term[] terms) throws IOException;
    Explanation explain(Query query, int doc) throws IOException;
    Explanation explain(Weight weight, int doc) throws IOException;
    Similarity getSimilarity();
    int maxDoc() throws IOException;
    Query rewrite(Query query) throws IOException;
    LuceneHits search(Query query) throws IOException;
    LuceneHits search(Query query, Filter filter) throws IOException;
    void search(Query query, Filter filter, HitCollector results) throws IOException;
    TopDocs search(Query query, Filter filter, int n) throws IOException;
    TopFieldDocs search(Query query, Filter filter, int n, Sort sort) throws IOException;
    LuceneHits search(Query query, Filter filter, Sort sort) throws IOException;
    void search(Query query, HitCollector results) throws IOException;
    LuceneHits search(Query query, Sort sort) throws IOException;
    void search(Weight weight, Filter filter, HitCollector results) throws IOException;
    TopDocs search(Weight weight, Filter filter, int n) throws IOException;
    TopFieldDocs search(Weight weight, Filter filter, int n, Sort sort) throws IOException;
    void setSimilarity(Similarity similarity);
    IndexReader getIndexReader();
}
```

The following code describes the methods offered by the *LuceneHits* interface:

```
public interface LuceneHits {
    int length();
    Document doc(int n) throws IOException;
    float score(int n) throws IOException;
    int id(int n) throws IOException;
    Iterator iterator();
}
```

The main advantage of this mechanism is the possibility to dissociate logical and physiacal resources. The physical resources are the Lucene resources which directly make search on the index, i.e. the instances of *Searcher*.

In order to create these resources, the Lucene support implements the factory pattern based on the *SearcerFactory* interface. This interface allows and hide the creation of logical resources. So, with this mechanism, you only need to configure an implementation of this interface in order to specify the strategy of resource management.

The following code describes the methods offered by the *SearcherFactory* interface:

```
public interface SearcherFactory {
    LuceneSearcher getSearcher() throws IOException;
}
```

Because the factory handle only logical resources, it do not provide directly instances of *Searcher*. The latters are managed implementations of the *LuceneSearcher*.

12.3.2. Configuration

Spring provides several factories to make searchs on a single index, on several indexes in a simple or parallel maner and on one or several remote indexes.

12.3.2.1. Configuring a SimpleSearcherFactory

The *SimpleSearcherFactory* class is the simplest factory in order to get instances of the *LuceneSearcher* interface. This factory is only based on a single *Directory*. Its configuration is described in the following code:

```
<bean id="fsDirectory"
      class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/lucene/index1/"/>
</bean>

<bean id="searcherFactory"
      class="org.springmodules.lucene.search.factory.SimpleSearcherFactory">
    <property name="directory" ref="fsDirectory"/>
</bean>
```

12.3.2.2. Configuring a MultipleSearcherFactory

The *MultipleSearcherFactory* class allows to make searchs across several indexes. It is based on the Lucene *MultiSearcher* class and can be configured with several *Directory*, as shown in the following example:

```
<bean id="fsDirectory1"
      class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/lucene/index1/"/>
</bean>

<bean id="fsDirectory2"
      class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/lucene/index2/"/>
</bean>

<bean id="searcherFactory"
      class="org.springmodules.lucene.search.factory.MultipleSearcherFactory">
    <property name="directories">
        <list>
            <ref local="fsDirectory1"/>
            <ref local="fsDirectory2"/>
        </list>
    </property>
```

```
</bean>
```

12.3.2.3. Configuring a ParallelMultipleSearcherFactory

The *MultipleSearcherFactory* class allows to make searchs across several indexes in a parallel manner. It is based on the Lucene *ParallelMultiSearcher* class and can be configured with several *Directory*, as shown in the following example:

```
<bean id="fsDirectory1"
      class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/lucene/index1/"/>
</bean>

<bean id="fsDirectory2"
      class="org.springmodules.lucene.index.support.FSDirectoryFactoryBean">
    <property name="location" value="/lucene/index2/"/>
</bean>

<bean id="searcherFactory"
   class="org.springmodules.lucene.search.factory.ParallelMultipleSearcherFactory">
    <property name="directories">
        <list>
            <ref local="fsDirectory1"/>
            <ref local="fsDirectory2"/>
        </list>
    </property>
</bean>
```

12.3.3. Template approach

The Lucene support provides a template approach like Spring for JDBC, JMS... to make searchs. The developer has not to know how to interact with the Lucene API in order to make searchs.

LuceneSearchTemplate is the central class of the Lucene support core package (*org.springmodules.lucene.search.core*) for the search. It simplifies the use of the corresponding Lucene APIs since it handles the creation and release of resources. This helps to avoid common errors like forgetting to always close the searcher. It executes the search leaving application code the way to create a search query and extract datas from results.

The template uses the *QueryCreator* abstraction to create a query basing on its *createQuery* method which must be contain the way to create the query. The following code describes the definition of this interface:

```
public interface QueryCreator {
    Query createQuery(Analyzer analyzer) throws ParseException;
}
```

If you don't inject an *Analyzer* instance in the template, this *analyzer* parameter of the method *createQuery* will be null. As a matter of fact, an analyzer isn't always mandatory to create a query.

The support provides a *ParsedQueryCreator* implementation to help to construct a query based on a *QueryParser* or a *MultiFieldQueryParser*. It uses an inner class *QueryParams* to hold the document fields to use and the query string. This class is used at the query creation and must be created by the *configureQuery* method. If you need to configure the created query (for example with a call of the *setBoost* method), you must overwrite the *setQueryProperties* method which gives it as method parameter.

```
public abstract class ParsedQueryCreator implements QueryCreator {
    public abstract QueryParams configureQuery();
    protected void setQueryProperties(Query query) { }

    public final Query createQuery(Analyzer analyzer) throws ParseException { (...) }
}
```

In order to construct a collection of objects from the result of a search, the Lucene support provides the *HitExtractor* interface, as described in the following code:

```
public interface HitExtractor {
    Object mapHit(int id, Document document, float score);
}
```

The *LuceneSearcherTemplate* class provides several *search* methods in order to make a search on the index. These methods use as parameters different entities of Lucene (Query, Fitler, Sort and HitCollector) and the Lucene support (QueryCreator, HitExtractor and SearcherCallback). The following code describes the *LuceneSearchTemplate* interface:

```
public interface LuceneSearchTemplate {
    List search(QueryCreator queryCreator, HitExtractor extractor);
    List search(Query query, HitExtractor extractor);
    List search(QueryCreator queryCreator, HitExtractor extractor, Filter filter);
    List search(Query query, HitExtractor extractor, Filter filter);
    List search(QueryCreator queryCreator, HitExtractor extractor, Sort sort);
    List search(Query query, HitExtractor extractor, Sort sort);
    List search(QueryCreator queryCreator, HitExtractor extractor, Filter filter, Sort sort);
    List search(Query query, HitExtractor extractor, Filter filter, Sort sort);
    void search(QueryCreator queryCreator, HitCollector results);
    Object search(SearcherCallback callback);
}
```

The following example constructs a query (basing on the *QueryParser* class) to search a text in the "contents" property of indexed documents. Then it constructs *SearchResult* objects with the search results. These objects will be added in a list by the support.

The following code describes an example of use of a *search* method of the template:

```
final String textToSearch = (...)
List results = getTemplate().search(new ParsedQueryCreator() {
    public QueryParams configureQuery() {
        return new QueryParams("contents", textToSearch);
    }
}, new HitExtractor() {
    public Object mapHit(int id, Document document, float score) {
        return new SearchResult(document.get("filename"), score);
    }
});
```

Finally the search template provides a callback order to work directly on a *LuceneSearcher* instance, the logical resource to use to make searchs.The callback is based on the *SearcherCallback* interface, as shown in the following code:

```
public interface SearcherCallback {
    Object doWithSearcher(LuceneSearcher searcher) throws Exception;
}
```

The callback interface is used by a dedicated *search* method of the LuceneSearchTemplate interface, as show in the following code:

```
public class LuceneSearchTemplate {
    (...)
    Object search(SearcherCallback callback);
    (...)
}
```

The following code describes an example of use of this *search* method of the template:

```
final String textToSearch = (...)
List results = getTemplate().search(new SearcherCallback() {
    public Object doWithSearcher(LuceneSearcher searcher) throws Exception {
        Query query = new TermQuery(new Term("attribute", textToSearch));
        Hits hits = searcher.search(query);
        (...)
    }
});
```

12.3.4. Object approach

The Lucene support allows the creation of search queries. Every classes of this approach are internally based on the *LuceneSearchTemplate* class and its mechanisms.

The base class is *LuceneSearchQuery* of the queries. The internal LuceneSearchTemplate instance is configured by injecting the *SearcherFactory* and *Analyzer* instances to use. As this class is abstract, you must implement the *search* method in order to specify the way to make your search and how handle the results.

```
public abstract class LuceneSearchQuery {
    private LuceneSearchTemplate template = new LuceneSearchTemplate();

    public LuceneSearchTemplate getTemplate() { (...) }
    public void setAnalyzer(Analyzer analyzer) { (...) }
    public void setSearcherFactory(SearcherFactory factory) { (...) }

    public abstract List search(String textToSearch);
}
```

As this class is very generic, Spring Modules providers a simple sub class to help you to implement your search queries. The abstract *SimpleLuceneSearchQuery* class implements the search methods leaving you to construct the query and specify the way to extract the results.

```
public abstract class SimpleLuceneSearchQuery extends LuceneSearchQuery {
    protected abstract Query constructSearchQuery(
                             String textToSearch) throws ParseException;
    protected abstract Object extractResultHit(int id,
                                        Document document, float score);

    public final List search(String textToSearch) { ... }
}
```

The following code describes an exampel of use based on the *SimpleLuceneSearchQuery* class:

```
String textToSearch = (...)
LuceneSearchQuery query = new SimpleLuceneSearchQuery() {
    protected abstract Query constructSearchQuery(
                             String textToSearch) throws ParseException;
    QueryParser parser = new QueryParser("contents",getAnalyzer());
    return parser.parse(textToSearch);
  }

  protected abstract Object extractResultHit(int id,
                               Document document, float score) {
    return document.get("filename");
  }
};
List results = query.search(textToSearch);
```

Chapter 13. Apache OJB

Note

Starting with release 0.6, Spring Modules hosts the Apache OJB project found in the main Spring distribution previous to 2.0 RC4.

Apache OJB (http://db.apache.org/ojb) offers multiple API levels, such as ODMG and JDO. Aside from supporting OJB through JDO, Spring also supports OJB's lower-level `PersistenceBroker` API as data access strategy. The corresponding integration classes reside in the `org.springmodules.orm.ojb` package.

13.1. OJB setup in a Spring environment

In contrast to Hibernate or JDO, OJB does not follow a factory object pattern for its resources. Instead, an OJB `PersistenceBroker` has to be obtained from the static `PersistenceBrokerFactory` class. That factory initializes itself from an OJB.properties file, residing in the root of the class path.

In addition to supporting OJB's default initialization style, Spring also provides a `Local`OjbConfigurer class that allows for using Spring-managed `DataSource` instances as OJB connection providers. The `DataSource` instances are referenced in the OJB repository descriptor (the mapping file), through the "jcd-alias" defined there: each such alias is matched against the Spring-managed bean of the same name.

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="ojbConfigurer" class="org.springmodules.orm.ojb.support.LocalOjbConfigurer"/>

  ...
</beans>
```

```
<descriptor-repository version="1.0">

  <jdbc-connection-descriptor jcd-alias="dataSource" default-connection="true" ...>
        ...
  </jdbc-connection-descriptor>

  ...
</descriptor-repository>
```

A `PersistenceBroker` can then be opened through standard OJB API, specifying a corresponding "PBKey", usually through the corresponding "jcd-alias" (or relying on the default connection).

13.2. `PersistenceBrokerTemplate` and `PersistenceBrokerDaoSupport`

Each OJB-based DAO will be configured with a "PBKey" through bean-style configuration, i.e. through a bean property setter. Such a DAO could be coded against plain OJB API, working with OJB's static `PersistenceBrokerFactory`, but will usually rather be used with Spring's `PersistenceBrokerTemplate`:

```
<beans>
  ...
  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="jcdAlias" value="dataSource"/> <!-- can be omitted (default) -->
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private String jcdAlias;

    public void setJcdAlias(String jcdAlias) {
        this.jcdAlias = jcdAlias;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        PersistenceBrokerTemplate pbTemplate =
                new PersistenceBrokerTemplate(new PBKey(this.jcdAlias);
        return (Collection) pbTemplate.execute(new PersistenceBrokerCallback() {
            public Object doInPersistenceBroker(PersistenceBroker pb)
                    throws PersistenceBrokerException {

                Criteria criteria = new Criteria();
                criteria.addLike("category", category + "%");
                Query query = new QueryByCriteria(Product.class, criteria);

                List result = pb.getCollectionByQuery(query);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

A callback implementation can effectively be used for any OJB data access. `PersistenceBrokerTemplate` will ensure that `PersistenceBrokers` are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single-step actions such as a single `getObjectById`, `getObjectByQuery`, `store`, or `delete` call, `PersistenceBrokerTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `PersistenceBrokerDaoSupport` base class that provides a `setJcdAlias` method for receiving an OJB JCD alias, and `getPersistenceBrokerTemplate` for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends PersistenceBrokerDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        Criteria criteria = new Criteria();
        criteria.addLike("category", category + "%");
        Query query = new QueryByCriteria(Product.class, criteria);

        return getPersistenceBrokerTemplate().getCollectionByQuery(query);
    }
}
```

As alternative to working with Spring's `PersistenceBrokerTemplate`, you can also code your OJB data access against plain OJB API, explicitly opening and closing a `PersistenceBroker`. As elaborated in the corresponding Hibernate section, the main advantage of this approach is that your data access code is able to throw checked exceptions. `PersistenceBrokerDaoSupport` offers a variety of support methods for this scenario, for fetching and releasing a transactional `PersistenceBroker` as well as for converting exceptions.

13.3. Transaction management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For 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:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
  ...

  <bean id="myTxManager" class="org.springmodules.orm.ojb.PersistenceBrokerTransactionManager">
    <property name="jcdAlias" value="dataSource"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

Note that OJB's `PersistenceBroker` level does not track changes of loaded objects. Therefore, a `PersistenceBroker` transaction is essentially simply a database transaction at the `PersistenceBroker` level, just with an additional first-level cache for persistent objects. Lazy loading will work both with and without the `PersistenceBroker` being open, in contrast to Hibernate and JDO (where the original `Session` or `PersistenceManager`, respectively, needs to remain open).

`PersistenceBrokerTransactionManager` is capable of exposing an OJB transaction to JDBC access code that accesses the same JDBC `DataSource`. The `DataSource` to expose the transactions for needs to be specified explicitly; it won't be autodetected.

Chapter 14. O/R Broker

14.1. Introduction

O/R Broker [http://orbroker.sf.net] is a convenience framework for applications that use JDBC. It allows you to externalize your SQL statements into individual files, for readability and easy manipulation, and it allows declarative mapping from tables to Java objects. Not just JavaBeans.

Spring Modules Integration for O/R Broker aims at simplifying the use of O/R Broker from within Spring applications. This module supports the same template style programming provided for JDBC, Hibernate, iBATIS, JPA...

Transaction management can be handled through Spring's standard facilities. As with iBATIS, there are no special transaction strategies for O/R Broker, as there is no special transactional resource involved other than a JDBC Connection. Hence, Spring's standard JDBC DataSourceTransactionManager or JtaTransactionManager are perfectly sufficient.

14.2. Setting up the Broker

To use O/R Broker you need to create the Java classes and configure the mappings. Spring Modules Integration for O/R Broker provides a factory called BrokerFactoryBean that loads the resources and creates the Broker.

```
public class Account {

    private Integer id;
    private String name;
    private String email;

    public void setId(Integer id) {
      this.id = id;
    }

    public Integer getId() {
      return id;
    }

    public String getName() {
        return this.name;
    }

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

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

To map this class, we need to create the following account-broker.xml. The Sql statement "getAccountById" is used to retrieve the accounts through their ids. "insertAccount" is used to create new accounts.

```
<broker name="Petclinic" version="2.0">

  <result-object id="Account" class="Account" key-columns="id">
    <property name="id"> <column name="id"/> </property>
    <property name="name"> <column name="name"/> </property>
    <property name="email"> <column name="email"/> </property>
  </result-object>

  <sql-statement id="getAccountById" result-object="Account"><![CDATA[
    SELECT id, name, email
      FROM acounts
     WHERE id = :id
  ]]></sql-statement>

  <sql-statement id="insertAccount"><![CDATA[
    INSERT INTO accounts (id, name, email)
    VALUES (:account.id, :account.name, :account.email)
  ]]></sql-statement>

</broker>
```

Using Spring, we can now configure a Broker through the BrokerFactoryBean:

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <!-- BrokerFactoryBean -->
  <bean id="broker" class="org.springmodules.orm.orbroker.BrokerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="configLocation" value="classpath:META-INF/account-broker.xml"/>
  </bean>

  ...
</beans>
```

As you can see from the previous config, account-broker.xml is saved under the META-INF folder and loaded using a classpath resource.

14.3. BrokerTemplate and BrokerDaoSupport

The BrokerDaoSupport class offers a supporting class similar to the HibernateDaoSupport and the JdoDaoSupport classes. Let's implement a DAO:

```
public class BrokerAccountDao extends BrokerDaoSupport implements AccountDao {

    public Account getAccount(Integer id) throws DataAccessException {
        return (Account) getBrokerTemplate().selectOne("getAccountById", "id", id);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getBrokerTemplate().execute("insertAccount", "account", account);
    }
}
```

In the DAO, we use the pre-configured BrokerTemplate to execute the queries, after setting up the BrokerAccountDao in the application context and wiring it with our Broker instance:

```
<beans>
  ...

  <bean id="accountDao" class="examples.BrokerAccountDao">
    <property name="broker" ref="broker"/>
  </bean>
```

```
</beans>
```

The BrokerTample offers a generic execute method, taking a custom BrokerCallback imlpementation as argument. This can be used as follows:

```
public class BrokerAccountDao extends BrokerDaoSupport implements AccountDao {
    ...

    public void insertAccount(final Account account) throws DataAccessException {
        getBrokerTemplate().execute(new BrokerCallback() {
            public Object doInBroker(Executable executable) throws BrokerException {
                executable.execute("insertAccount", "account", account);
            }
        });
    }
}
```

Any BrokerException thrown will automatically get converted to Spring's generic DataAccessException hierarchy.

14.4. Implementing DAOs based on plain O/R Broker API

DAOs can also be written against plain O/R Broker API, without any Spring dependencies, directly using an injected Broker. A corresponding DAO implementation looks like as follows:

```
public class BrokerAccountDao implements AccountDao {

    private Broker broker;

    public void setBroker(Broker broker) {
        this.broker = broker;
    }

    public Account getAccount(Integer id) {
        Query qry = this.broker.startQuery();
        qry.setParameter("id", id);
        try {
            return (Account) qry.queryForOne("getAccountById");
        }
        catch (Throwable ex) {
            throw new MyDaoException(ex);
        }
        finally () {
          qry.close();
        }
    }

    ...
}
```

Configuring such a DAO can be done as follows:

```
<beans>
  ...

  <bean id="accountDao" class="example.BrokerAccountDao">
    <property name="broker" ref="broker"/>
  </bean>

</beans>
```

14.5. Unit Testing

You can leverage the excellent Spring Framework testing support [http://static.springframework.org/spring/docs/2.0.x/reference/testing.html] to test your O/R Broker based DAOs. More informations on integration testing DAOs (using a DataSource, TransactionManager...) can be found at **8.3.3. Transaction management** [http://static.springframework.org/spring/docs/2.0.x/reference/testing.html#testing-tx].

O/R Broker module also allows for quick creation of O/R Broker intances without providing an external configuration file. Statements could then be created dynamically.

Let's suppose we would like to test the following method:

```
public class BrokerUserDao extends BrokerDaoSupport implements UserDao {

    public int countUsers() throws DataAccessException {
        Long count = (Long) getBrokerTemplate().selectOne("userCount");
        return count.intValue();
   }

}
```

The application context file would be:

```
<beans>

    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="jdbc.properties"/>
    </bean>

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="broker" class="org.springmodules.orm.orbroker.BrokerFactoryBean">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

and the test case:

```
public class UserDaoTests extends AbstractTransactionalDataSourceSpringContextTests {

    protected Broker broker;

    public void testCountUsers() throws Exception {
        // setup the dao
        broker.addStatement("userCount", "select count(*) from users");
        BrokerUserDao dao = new BrokerUserDao();
        dao.setBroker(broker);

        // empty users table
        deleteFromTables(new String[]{"users"});

        // run the target method and check the returned value
        assertEquals(0, dao.countUsers());
    }

    public void setBroker(Broker broker) {
        this.broker = broker;
    }

    protected String getConfigPath() {
        return "applicationContext.xml";
    }

}
```

Chapter 15. OSWorkflow

15.1. Introduction

OSWorkflow module offers Spring-style [http://www.springframework.org] support for OSWorkflow [http://www.opensymphony.com/osworkflow/] allowing easy configuration and interaction with its API. For OSWorkflow version 2.8 and upwards, beans maintained by Spring contained can be accessed by OSWorkflow definition as conditions, functions, etc...

15.2. Configuration

OSWorkflow module offers ConfigurationBean for configuring OSWorkflow resources:

```
<bean id="configuration" class="org.springmodules.workflow.osworkflow.configuration.ConfigurationBean">
  <property name="workflowLocations">
   <props>
    <prop key="documentApproval">
      classpath:/org/springmodules/examples/workflow/osworkflow/service/documentApproval.xml</prop>
   </props>
  </property>
</bean>
```

`ConfigurationBean` is not a `FactoryBean` as OSWorkflow already manages the creation of workflow instances. The bean extends OSWorkflow's `DefaultConfiguration` and allows workflows to be loaded using Spring's ResourceLoader [http://static.springframework.org/spring/docs/2.0.x/reference/resources.html#d0e4913]s and the underlying persistence store to be either injected or configured and managed by OSWorkflow. Note that by default, `ConfigurationBean` uses a memory based storage (`MemoryWorkflowStore`).

15.3. Inversion of Control: OsWorkflowTemplate and OsWorkflowCallback

One of the core classes of OSWorkflow module is `OsWorkflowTemplate` which greatly simplifies interaction with the OSWorkflow API by hiding the management of context parameters such as the caller and workflow ID besides offering the usual advantages of Spring's template pattern such as exception translation (from OSWorkflow checked exception into unchecked ones). The template mirrors most of the OSWorkflow API methods; however for lengthy interactions or cases where the native Workflow is required, `OsWorkflowCallback` should be used.

It is important to note, that OsWorkflowTemplate manages all instances of a single workflow within an application - that is there should be one template per workflow definition. This results in simple method calls as the workflow name or id are not required - they will be automatically passed in by the template. Consider the following example:

```
<bean id="workflowTemplate" class="org.springmodules.workflow.osworkflow.OsWorkflowTemplate">
  <property name="configuration" ref="configuration"/>
  <property name="workflowName" value="documentApproval"/>
</bean>

<bean id="someWorkflowClass" class="some.example.SomeWorkflowFacade">
  <property name="workflowTemplate"/>
</bean>
```

```
public class SomeWorkflowFacade
{
  private OsWorkflowTemplate template;

  public void setTemplate(OsWorkflowTemplate template) {
    this.template = template;
  }
  ...

  public void executeSomeAction(int actionNumber) {
    template.doAction(actionNumber);
  }

  public void addSomeInput(Object input) {
    template.doAction(INPUT_ACTION, "some_input", input);
  }

  public void accessNativeWorkflowObject() {
     template.execute(new OsWorkflowCallback()
     {
        public Object doWithWorkflow(Workflow workflow) throws WorkflowException {
            // call the OSWorkflow API directly
            workflow.changeEntryState(someInstanceId, someState);
        }
     });
  }
}
```

In this case, the facade uses the injected template to execute several actions on the workflow - note that workflow id or caller are never specified as the template determines them internally. The template is thread safe; the same template instance can be used with different instances of the same workflow.

15.4. Working with workflow instances

As mentioned previously, the template transparently handles the workflow instance ID and caller on which the methods are executed. Both ID and caller values can be retrieved and set using `OsWorkflowContextHolder` and `WorkflowContext`. OSWorkflow module offers two convenient classes when working with Spring MVC:

- `AbstractWorkflowContextHandlerInterceptor` - abstract base class which can set the workflow id from HTTP parameters and store it on the `HttpSession`
- `DefaultWorkflowContextHandlerInterceptor` - default implementation which retrieves the workflow caller from `HttpRequest`
- `AcegiWorkflowContextHandlerInterceptor` - Acegi [http://www.acegisecurity.org/] specific implementation - the workflow caller will be retrieved from Acegi [http://www.acegisecurity.org/].

Spring Modules CVS contains an osworkflow sample which shows the Spring MVC Handler in action along with the rest of the OSWorkflow module.

15.5. Acegi integration

Besides the already mentioned Acegi web handler , OSWorkflow module also offers out of the box an Acegi aware OSWorkflow condition that can be used inside workflow definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC
 "-//OpenSymphony Group//DTD OSWorkflow 2.7//EN"
 "http://www.opensymphony.com/osworkflow/workflow_2_7.dtd">
<workflow>
```

```
 ...
 <steps>
  <step id="1" name="Awaiting Document">
   <actions>
    <action id="1" name="Create Document" view="upload">
     <restrict-to>
      <conditions>
       <condition type="class">
        <arg name="class.name">org.springmodules.workflow.osworkflow.support.AcegiRoleCondition</arg>
        <arg name="role">ROLE_CREATOR</arg>
       </condition>
      </conditions>
     </restrict-to>
     <results>
      <unconditional-result old-status="Finished" status="Document Uploaded" step="2"/>
     </results>
     ...
    </action>
   </actions>
  </step>
  ...
  </steps>
</workflow>
```

`AcegiRoleCondition` will check the current Acegi authorities against the 'role' parameter specified in the workflow definition and return `true` if a match is found or `false` otherwise.

Note

Spring Modules CVS contains a comprehensive OSWorkflow module sample which uses the classes discussed.

15.6. OSWorkflow 2.8+ support

OSWorkflow 2.8 release added two important componets:

- TypeResolver [http://www.opensymphony.com/osworkflow/api/com/opensymphony/workflow/TypeResolver.html] - which allows business components to be resolved and used inside workflow definitions. For Spring users, the most important subclass is SpringTypeResolver [http://www.opensymphony.com/osworkflow/api/com/opensymphony/workflow/util/SpringTypeResolver.html] which creates a bridge between Spring application context and OSWorkflow so it's possible to simply reuse beans just by using their name.

- VariableResolver [http://www.opensymphony.com/osworkflow/api/com/opensymphony/workflow/util/VariableResolver.html] which adds translation capabilities for variables.

However, as OsWorkflow 2.7 is widely deploy, OSWorkflow module adds support for the new features under a special package : org.springmodules.workflow.osworkflow.v28. A typical configuration under OSWorkflow 2.8 might look like this:

-- Spring application context --

```
<bean id="workflowTemplate" class="org.springmodules.workflow.osworkflow.v28.OsWorkflowTemplate">
  <property name="configuration" ref="configuration"/>
  <property name="workflowName" value="documentApproval"/>
  <property name="typeResolver">
    <bean class="com.opensymphony.workflow.util.SpringTypeResolver"/>
  </property>
</bean>
```

```

<bean id="whiteHorseFunction" class="mypackage.WhiteHorse" singleton="false"/>
```

-- OSWorkflow workflow definition --

```
<workflow>
...
<function type="spring">
  <arg name="bean.name">whiteHorseFunction</arg>
</function>
...
</workflow>
```

In this case, the whiteHorseFunction is retrieved from Spring application context and used inside the workflow instance; this is a power concept as the business components can be configured inside Spring and take advantage of the advanced IoC functionality as transaction demarcation or custom scoping.

Chapter 16. Spring MVC extra

16.1. About

The Spring MVC extra module contains classes for improving and extending the Spring MVC Framework.

16.2. Usage guide

16.2.1. Using the ReflectivePropertyEditor

The *org.springmodules.web.propertyeditors.ReflectivePropertyEditor* is a *property editor* [http://java.sun.com/j2se/1.4.2/docs/api/java/beans/PropertyEditor.html] implementation capable of converting any object type to and from text, using Java reflection. It converts objects to text and vice-versa thanks to four configurable parameters:

- **dataAccessObject**, the object used for converting from text to the actual desired object: it could be a Factory, or a DAO.
- **dataAccessMethod**, the method of the dataAccessObject object to call for converting from text to object.
- **propertyName**, the name of the property which will represent the object text value.
- **stringConvertor**, for converting the string value to be passed to the dataAccessMethod.
- **stringConvertorOutput**, the *Class* of the object converted by the string convertor; if not specified, the property editor will try to guess it by directly inspecting the converted object.

16.2.2. Using the ReflectiveCollectionEditor

The *org.springmodules.web.propertyeditors.ReflectiveCollectionEditor* is a *property editor* [http://java.sun.com/j2se/1.4.2/docs/api/java/beans/PropertyEditor.html] implementation for converting a collection of strings to a collection of objects and vice-versa. For converting you have to define the following:

- **dataAccessObject**, the object used for converting from text to the actual desired object: it could be a Factory, or a DAO.
- **dataAccessMethod**, the method of the dataAccessObject object to call for converting from text to object.
- **propertyName**, the name of the property which will represent the object text value.
- **stringConvertor**, for converting the string value to be passed to the dataAccessMethod.
- **stringConvertorOutput**, the *Class* of the object converted by the string convertor; if not specified, the property editor will try to guess it by directly inspecting the converted object.

This class is to be used for binding collections in Spring MVC: for example, if you want to bind a collection of customers starting from a collection of customer ids, obtained from some kind of selection list, you can use this class for automatically converting the collection of customer ids (strings) to a collection of actual customer objects.

16.2.3. Using EnhancedSimpleFormController and EnhancedAbstractWizardFormController

The *org.springmodules.web.servlet.mvc.EnhancedSimpleFormController* and *org.springmodules.web.servlet.mvc.EnhancedAbstractWizardFormController* are Spring MVC controllers which provide facilities for setting custom *property editors* [http://java.sun.com/j2se/1.4.2/docs/api/java/beans/PropertyEditor.html] in a declarative way. Using the *setCustomEditor(Map)* method you can set a map of custom property editors containing, as **key**, the class of the property in the form *class:CLASS_NAME* if you want to edit all properties of the given type, or its path in the form *property:PROPERTY_PATH* if you want to edit only the given property, and as **value** the name of a bean in the application context. Please note that the bean have to be a PropertyEditor and must be declared as *prototype*. If the *class* and *property* prefixes above are missed, the key is treated as a class name. So, if you extend the EnhancedSimpleFormController for your controllers, you can use the method above and avoid to overwrite and manually code the *initBinder* [http://www.springframework.org/docs/api/org/springframework/web/servlet/mvc/BaseCommandController.html#initBinde method.

Here is an XML snippet of EnhancedSimpleFormController custom editors configuration into the Spring application context, using the default prefix:

```
<bean id="exampleController" class="org.acme.ExampleController">
  <property name="customEditors">
    <map>
     <entry>
       <key><value>org.acme.Office</value></key>
       <value>officeEditor</value>
     </entry>
    </map>
  </property>
</bean>
```

This one uses the *class* prefix:

```
<bean id="exampleController" class="org.acme.ExampleController">
   <property name="customEditors">
     <map>
       <entry>
         <key><value>class:org.acme.Office</value></key>
         <value>officeEditor</value>
       </entry>
     </map>
   </property>
</bean>
```

Finally, this one uses the *property* prefix:

```
<bean id="exampleController" class="org.acme.ExampleController">
   <property name="customEditors">
     <map>
      <entry>
        <key><value>property:office</value></key>
        <value>officeEditor</value>
      </entry>
     </map>
   </property>
</bean>
```

The same applies to the EnhancedAbstractWizardFormController.

16.2.4. Using the FullPathUrlFilenameViewController

The *org.springmodules.web.servlet.mvc.FullPathUrlFilenameViewController* is an *AbstractUrlViewController* [http://www.springframework.org/docs/api/org/springframework/web/servlet/mvc/AbstractUrlViewController.html] which like the *UrlFilenameViewController* [http://www.springframework.org/docs/api/org/springframework/web/servlet/mvc/UrlFilenameViewController.html] transforms the page name at the end of a URL into a view name, but preserves the full path in the web URL. For example, the URL "/foo/index.html" will correspond to the the "foo/index" view name.

16.2.5. Using the AbstractRssView

The *org.springmodules.web.servlet.view.AbstractRssView* is an abstract superclass for creating RSS views, with the capability of supporting many syndication formats through the use of the *Rome* [http://wiki.java.net/bin/view/Javawsxml/Rome] library.

AbstractRssView uses ATOM 1.0 as its default syndication format: you can change it by setting the feed type through the following method:

```
public void setDefaultFeedType(String)
```

Moreover, you can select the syndication format on the fly, using the HTTP request parameter *type*; i.e., the *http://www.example.org/example.xml?type=rss_1.0* make a request for an RSS 1.0 file.

The feed type naming format is explained *here* [https://rome.dev.java.net/apidocs/0_8/com/sun/syndication/feed/WireFeed.html].

Then, for constructing your feed, you need to override the following method:

```
abstract protected void buildFeed(Map,HttpServletRequest,HttpServletResponse,SyndFeed)
```

Here, you can construct your feed filling the *SyndFeed* [https://rome.dev.java.net/apidocs/0_8/com/sun/syndication/feed/synd/SyndFeed.html] object.

Chapter 17. Validation

17.1. Commons Validator

The Commons Validator is a library that allows you to perform validation based on rules specified in XML configuration files.

TODO: Describe the concepts of Commons Validator in more details.

17.1.1. Configure an Validator Factory

Firstly you need to configure the Validator Factory which is the factory to get Validator instances. To do so, the support provides the class DefaultValidatorFactory in the package org.springmodules.validation.commons

You need to specify with the property validationConfigLocations the file containing the Commons Validator rules and the file containing the validation rules specific to the application.

The following code shows how to configure this factory.

```
<bean id="validatorFactory"
      class="org.springmodules.validation.commons.DefaultValidatorFactory">
  <property name="validationConfigLocations">
    <list>
      <value>/WEB-INF/validator-rules.xml</value>
      <value>/WEB-INF/validation.xml</value>
    </list>
  </property>
</bean>
```

17.1.2. Use a dedicated validation-rules.xml

The file *validation-rules.xml* must contain Commons Validator elements based on classes provided by the support of this framework in Spring Modules.

For example, the configuration of the entities "required" and "requiredif" must be now in the *validation-rules.xml* file.

```
<validator name="required"
          classname="org.springmodules.validation.commons.FieldChecks"
          method="validateRequired"
          methodParams="java.lang.Object,
                        org.apache.commons.validator.ValidatorAction,
                        org.apache.commons.validator.Field,
                        org.springframework.validation.Errors"
          msg="errors.required">

                <javascript><![CDATA[
                (...)
                ]]></javascript>
                </validator>

                <validator name="requiredif"
                classname="org.springmodules.validation.commons.FieldChecks"
                method="validateRequiredIf"
                methodParams="java.lang.Object,
                org.apache.commons.validator.ValidatorAction,
                org.apache.commons.validator.Field,
                org.springframework.validation.Errors,
                org.apache.commons.validator.Validator"
                msg="errors.required">
                </validator>
```

The validation sample of the distribution provides a complete *validation-rules.xml* based on the classes of the support.

You must note that the support of *validwhen* is not provided at the moment in the support. However, some codes are provides in JIRA. For more informations, see the issues MOD-38 [http://opensource2.atlassian.com/projects/spring/browse/MOD-38] and MOD-49 [http://opensource2.atlassian.com/projects/spring/browse/MOD-49] .

17.1.3. Configure a Commons Validator

Then you need to configure the Validator itself basing the previous Validator Factory. It corresponds to an adapter in order to hide Commons Validator behind a Spring Validator.

The following code shows how to configure this validator.

```
<bean id="beanValidator"
                class="org.springmodules.validation.commons.DefaultBeanValidator">
                <property name="validatorFactory" ref="validatorFactory"/>
                </bean>
```

17.1.4. Server side validation

Spring MVC provides the implementation *SimpleFormController* of the interface *Controller* in order to process HTML forms. It allows a validation of informations processing by the controller by using the property v *alidator* of the controller. In the case of Commons Validator, this property must be set with the bean *beanValidator* previously configured.

The following code shows how to configure a controller which validates a form on the server side using the support of Commons Validator.

```
<bean id="myFormController" class="org.springmodules.sample.MyFormController">
                (...)
                <property name="validator" ref="beanValidator"/>
                <property name="commandName" value="myForm"/>
                <property name="commandClass" value="org.springmodules.sample.MyForm"/>
                (...)
                </bean>
```

The *beanValidator* bean uses the value of the property *commandClass* of the controller to select the name of the form tag in the *validation.xml* file. The configuration is not based on the *commandName* property. For example, with the class name *org.springmodules.sample.MyForm* , Commons Validator must contain a form tag with *myForm* as value of the name property. The following code shows the contents of this file.

Important

In version 0.6 the logic to resolve the form names has changed. In the previous versions *org.springframework.util.StringUtils.uncapitalize(...)* was used to transform the command class name to the form name. From version 0.6 *java.beans.Introspector.decapitalize(...)* is used instead. The main difference between the two approaches is that the second one better complies to the javabean naming conventions, so for example, *URLCommand* would be translated to URLCommand and not *uRLCommand* .

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
                <!DOCTYPE form-validation PUBLIC
                "-//Apache Software Foundation//DTD Commons Validator Rules Configuration 1.1//EN"
                "http://jakarta.apache.org/commons/dtds/validator_1_1.dtd">

                <form-validation>
                <formset>
                <form name="myForm">
                <field property="field1" depends="required">
                <arg0 key="error.field1" />
                </field>
                <field property="field2" depends="email">
                <arg0 key="error.field2" />
                </field>
                </form>
                </formset>
                </form-validation>
```

17.1.5. Partial Bean Validation Support

Partial validation support enables partial validation of beans where not all properties are validated but only selected ones.

Commons validator enables partial validation by specifying the *page* attribute for each field in the form configuration:

```
<form name="personDataWizard">
                <field property="firstName" depends="required" page="0">
                <arg0 key="person.first.name" />
                </field>
                <field property="lastName" depends="required" page="0">
                <arg0 key="person.last.name" />
                </field>
                <field property="email" depends="required,email" page="0">
                <arg0 key="person.email" />
                </field>
                <field property="password" depends="required" page="1">
                <arg0 key="person.password" />
                </field>
                <field property="verifyPassword" depends="validwhen" page="1">
                <arg0 key="person.password.not.matching" />
                <var>
                <var-name>test</var-name>
                <var-value>(*this* == password)</var-value>
                </var>
                </field>
                </form>
```

The *org.springmodules.validation.commons.ConfigurablePageBeanValidator* and *org.springmodules.validation.commons.DefaultPageBeanValidator* classes support partial validation by setting their *page* property. The value of this property will be matched with the page attribute in the form configuration, and only the fields with the appropriate page configured will be validated.

The following is an example of a partial validation support usage within a wizard controller:

```
<bean id="personWizardController" class="PersonWizardController">
                <property name="pages">
                <list>
                <value>personPage0</value>
                <value>personPage1</value>
                </list>
                </property>
                <property name="allowDirtyForward" value="false"/>
                <property name="validators">
                <list>
                <ref bean="pageValidator0"/>
                <ref bean="pageValidator1"/>
                </list>
                </property>
                <property name="commandName" value="person"/>
                <property name="commandClass" value="PersonData"/>
                </bean>

                <bean id="pageValidator0" class="ConfigurablePageBeanValidator" parent="pageValidator">
                <property name="page" value="0"/>
                </bean>

                <bean id="pageValidator1" class="ConfigurablePageBeanValidator" parent="pageValidator">
                <property name="page" value="1"/>
                </bean>

                <bean id="pageValidator" abstract="true">
                <property name="formName" value="personDataWizard"/>
                <property name="validatorFactory" ref="validatorFactory"/>
                </bean>

                ...
```

The controller will look like this:

```
public class PersonWizardController extends AbstractWizardFormController {

                ...

                protected void validatePage(Object command, Errors errors, int page) {
                Validator[] validators = getValidators();
                for (int i=0; i<validators.length; i++) {
                Validator validator = validators[i];
                if (validator instanceof PageAware) {
                if (((PageAware)validator).getPage() == page) {
                validator.validate(command, errors);
                }
                }
                }
                }
                }
```

17.1.6. Client side validation

The support of Commons Validator in Spring Modules provides too the possibility to use a client side validation. It provides a dedicated taglib to generate the validation javascript code. To use this taglib, we firstly need to declare it at the beginnig of JSP files as following.

```
<%@ tglib uri="http://www.springmodules.org/tags/commons-validator" prefix="validator" %>
```

You need then to include the generated javascript code in the JSP file as following by using the *javascript* tag.

```
<validator:javascript formName="account"
                staticJavascript="false" xhtml="true" cdata="false"/>
```

At last, you need to set the *onSubmit* attribute on the *form* tag in order to trigger the validation on the submission of the form.

```
<form method="post" action="(...)" onsubmit="return
                validateMyForm(this)">
```

17.2. Valang

Valang (*Va* -lidation *Lang* -uage), provides a simple and intuitive way for creating spring validators. It was initially create with three goals in mind:

- Enables writing validation rules quickly, without the need of writing classes or even any java code.
- Ease the use of Spring validation tools.
- Make validation rules compact, readable and easily maintainable.

Valang is built upon two major constructs - The valang expression language and valang validators. The former is a generic boolean expression language that enables expressing boolean rule in a "natural language"-like fashion. The later is a concrete implementation of the Spring *Validator* interface that is built around the expression language.

Before going into details, lets first have a look at a small example, just to have an idea of what valang is and how it can be used. For this example, we'll assume a *Person* class with two properties - firstName and lastName. In addition, there are two main validation rules that need to be applied:

- The first name of the person must be shorter than 30 characters.
- The last name of the person must be shorter than 50 characters.

One way of applying these validation rules (and currently the most common one) is to implement the *Validator* interface specifically for the *Person* class:

```
public class PersonValidator implements Validator {

            public boolean supports(Class aClass) {
            return Person.class.equals(aClass);
            }

            public void validate(Object person, Errors errors) {
            String firstName = ((Person)person).getFirstNam();
            String lastName = ((Person)person).getLastName();
            if (firstName == null || firstName.length() >= 30) {
            errors.reject("first_name_length", new Object[] { new Integer(30) },
            "First name must be shorter than 30");
            }
            if (lastName == null || lastName.length() >= 50) {
            errors.reject("last_name_length", new Object[] { new Integer(50) },
            "Last name must be shorter than 50");
            }
            }
            }
```

While this is a perfectly valid approach, it has its downsides. First, it is quite verbose and time consuming - quite a lot of code to write just for two very simple validation rules. Second, it required an additional class which clutters the code (in case it is an inner-class) or the design - just imagine having a validator class for each of the domain model objects in the application.

The following code snippet shows how to create a valang validator to apply the same rules as above:

```
<bean id="personValidator" class="org.springmodules.validation.valang.ValangValidator">
            <property name="valang">
            <value>
            <![CDATA[
            { firstName : length(?) < 30 : 'First name too long' : 'first_name_length' : 30}
            { lastName : length(?) < 50 : 'Last name too long' : 'last_name_length' : 50 }
            ]]>
            </value>
            </property>
            </bean>
```

There are a few things to notice here. First, no new class is created - with valang, one can reuse a predefined validator class (as shown here). Second, This validator is not part of the java code, but put in the application context instead - In the above case, the *ValangValidator* is instantiated and can be injected to other objects in the system. Last but not least, The validation rules are defined using the valang expression language which is very simple and quick to define.

The following two sections will elaborate on the expression language and the use of the Valang validator in greater details.

17.2.1. Valang Syntax

The valang syntax is based on the valang expression language and the valang validation rule configuration. As mentioned above, the former is a boolean expression language by which the validation rules predicates (conditions) are expressed. The later binds the rule predicates to a key (usually a bean property), error message, and optionally error code and arguments.

17.2.1.1. Rule Configuration

Here is the basic structure of the valang rule configuration:

```
{ <key> : <predicate_expression> : <message> [: <error_code> [:
                    <args> ]] }
```

- <key> - The key to which the validation error will be bound to. *(mandatory)*
- <predicate_expression> - A valang expression that defines the predicate (condition) of the validation rule. *(mandatory)*
- <message> - The error message of the validation rule. The message is mandatory but can be an empty string if not used. This message is also used as the default message in case the error code could not be resolved. *(mandatory)*
- <error_code> - An error code that represents the validation error. Used to support *i18n* . *(optional)*
- <args> - A comma separated list of arguments to associate with the error code. When error codes are resolved, this arguments may be used in the resolved message. *(optional)*

17.2.1.2. Expression Language

As mentioned, the valang expression language is used to define the predicate to be associated with the validation rule. The expression is always evaluated against a context bean. The expression can be defined as follows:

```
<expression> ::= <expression> ( ( "AND" | "OR" ) <expression> )+ |
                    <predicate>
```

The <predicate> in an evaluation that is composed of operators, literals, bean properties, functions, and mathematical expressions.

Operators

The following are the supported operators:

- Binary Operators:
  - String, boolean, date and number operators:
    - = | == | IS | EQUALS
    - != | <> | >< | IS NOT | NOT EQUALS
  - Number and date operators:
    - > | GREATER THAN | IS GREATER THAN
    - < | LESS THAN | IS LESS THAN
    - >= | => | GREATER THAN OR EQUALS | IS GREATER THAN OR EQUALS
    - <= | =< | LESS THAN OR EQUALS | IS LESS THAN OR EQUALS
- Unary Operators:
  - Object operators:
    - NULL | IS NULL
    - NOT NULL | IS NOT NULL
  - String operators:
    - HAS TEXT
    - HAS NO TEXT
    - HAS LENGTH
    - HAS NO LENGTH
    - IS BLANK
    - IS NOT BLANK
    - IS UPPERCASE | IS UPPER CASE | IS UPPER
    - IS NOT UPPERCASE | IS NOT UPPER CASE | IS NOT UPPER
    - IS LOWERCASE | IS LOWER CASE | IS LOWER
    - IS NOT LOWERCASE | IS NOT LOWER CASE | IS NOT LOWER
    - IS WORD
    - IS NOT WORD
- Special Operators:
  - BETWEEN
  - NOT BETWEEN
  - IN
  - NOT IN
  - NOT

These operators are case insensitive. Binary operators have a left and a right side. Unary operators only have a left side.

Value types on both sides of the binary operators must always match. The following expressions will throw an exception:

```
name > 0
                        age == 'some string'
```

BETWEEN / NOT BETWEEN Operators

The BETWEEN and NOT BETWEEN operators have the following special syntax:

```
<between_operator> ::= <left_side> BETWEEN <value> AND <value>
                        <not_between_operator> ::= <left_side> NOT BETWEEN <value> AND
                        <value>
```

Both the left side and the values can be any valid combination of literals, bean properties, functions and mathematical operations.

Examples:

```
width between 10 and 90
                        length(name) between minLength and maxLength
```

IN / NOT IN Operators

The IN and NOT IN operators have the following special syntax:

```
<in_operator> ::= <left_side> IN <value> ( "," <value> )*
                        <not_in_operator> ::= <left_side> NOT IN <value> ( "," <value>
                        )*
```

Both the left side and the values can be any valid combination of literals, bean properties, functions and mathematical operations.

There's another special syntax where a *java.util.Collection* , *java.util.Enumeration* , *java.util.Iterator* or object array instance can be retrieved from a bean property. These values are then used as right side of the operator. This feature enables to create dynamic sets of values based on other properties of the bean.

```
<special_in_operator> ::= <left_side> IN "@"<bean_property>
                        <special_not_in_operator> ::= <left_side> NOT IN
                        "@"<bean_property>
```

Examples:

```
size in 'S', 'M', 'L', 'XL'
                        size in @sizes
```

NOT Operator

The not operator has the following special syntax:

```
<not_operator> ::= "NOT" <expression>
```

This operator inverses the result of one or a set of predicates.

Literals

Four type of literals are supported by valang: *string* , *number* , *date* , and *boolean* .

Strings are quoted with single quotes:

```
'Bill', 'George', 'Junior'
```

Number literals are unquoted and are parsed by *java.math.BigDecimal* :

```
0.70, 1, 2000, -3.14
```

Date literals are delimited with square brackets and are parsed upon each evaluation by a special date parser. [TODO: write documentation for date parser]

```
[T<d], [2005-05-28]
```

Boolean literals are not quoted and have the following form:

```
<boolean> ::= ( "TRUE" | "YES" | "FALSE" | "NO" )
```

Bean Properties

As mentioned above, the valang always evaluates the expressions against a context bean. Once can access this bean's properties directly within the expression. To better understand how this works lets assume a Person class with the following properties:

- name (String)
- address (Address)
- specialFriends (Map<String, Object>)
- friends (Person[])
- enemies (List<Person>)

The Address class has the following properties:

- street (String)
- city (String)
- Country (String)

The context bean properties can be accessed directly by using their names:

```
name, address, attributes
```

Accessing nested properties is also supported by using a dot-separated expression. For example, accessing the street of the person can be done as follows:

```
address.street
```

List and/or array elements can be access by their index number as follows:

```
friends[1].name
                            enemies[0].address.city
```

Map entries can also be accessed by their keys:

```
specialFriends[bestFriend].name
```

Functions

Valang expressions can contain functions. A function is basically an operation which accepts arguments and returns a result. Functions can accept one or more arguments where each may be either a literal, bean property, or a function as described in the following definition:

```
function ::= <function_name> "(" <arg> [ "," <arg> ]* ")"
                            <arg> ::= <literal> | <bean_property> | <function>
```

Valang ships with the following predefined functions:

Table 17.1. Functions

Name	Description
length	Returns the size of the passed in collection or array. If the passed in argument is neither, the length of the string returned from the *toString()* call on the passed in argument.
len	See *length* above
size	See *length* above
count	See *length* above
match	Matches the given regular expression (first argument) to the string returned from the *toString()* call on the passed in value (second argument).
matches	See *match* above.
email	Returns *true* if the string returned from the *toString()* call on the passed in argument represents a valid email
upper	Converts the string returned from the *toString()* call on the argument to upper case.
lower	Converts the string returned from the *toString()* call on the argument to lower case.
!	Not operation on a boolean value.
resolve	Wrap string in *org.springframework.context.support.DefaultMessageSourceResolvable* .
inRole	Accepts a role name as an argument and returns *true* if the current user has this role. This function uses *Acegi* to fetch the current user.

Examples:

```
length(?)
                            size(upper('test'))
                            upper(address.city)
```

One of the more powerful features in Valang expression language is that it is extensible with custom functions. To add a custom function one first needs to implement the *org.springmodules.validation.valang.functions.Function* interface or extend the *org.springmodules.validation.valang.functions.AbstractFunction* . Then, when using the *ValangValidatorFactoryBean* or *ValangValidator* , register the new function with the *customFunctions* property using the function name as the key. [TODO: show an example of a custom function]

Mathematical Expressions

The following mathematical operators are supported:

- +
- -
- *
- / | div
- % | mod

Parentheses are supported and expression are parsed left to right so that

```
2 - 3 + 5 = 4
```

Values in the mathematical expression can be literals, bean properties, and functions.

Examples:

```
(2 * (15 - 3) + ( 20 / 5 ) ) * -1
                            (22 / 7) - (22 div 7)
                            10 % 3
                            length(?) mod 4
```

17.2.2. Valang Validator Support

As we saw in the previous chapter, Valang offers quite a reach and powerful expression language to represent the validation rules. Language that for most cases relieves the user from creating custom Validator classes.

The only missing piece of the puzzle now is to see how this expression language and the validation rule configuration integrate with Spring validation support.

The 2 most important constructs of Spring validation are the *org.springframework.validation.Validator* and *org.springframework.validation.Errors* classes. The *Errors* class serves as a registry for validation errors that are associated with an object (a.k.a the target object). The *Validator* interface provides a mechanism to validate objects and register the various validation error within the passed in *Errors* .

Valang ships with some support classes that leverage the power of the Valang expression language and validation rule configuration, and integrates nicely with Spring validation. The most important of them all is the *org.springmodules.validation.valang.ValangValidator* class.

17.2.2.1. ValangValidator

The *org.springmodules.validation.valang.ValangValidator* class is a concrete implementation of Spring's *Validator* interface. The most important property of this validator is the *valang* property.

The *valang* property is of type *java.lang.String* and holds a textual representation of the validation rules that are applied by the validator. We saw in the previous section that a single validation rule is represented in valang using the following format:

```
{ <key> : <predicate_expression> : <message> [: <error_code>
                        [: <args> ]] }
```

Since, a validator may apply more then just one rule, the *valang* property accepts a set of such rule definitions.

Example:

```
{ firstName : length(?) < 30 : 'First name too long' : 'first_name_length' : 30}
                        { lastName : length(?) < 50 : 'Last name too long' : 'last_name_length' : 50
                        }
```

There are two ways to use the valang validator. It can be explicitly instantiated and initialized with the rule definitions by calling the *setValang(String)* method on it. But the recommended way is actually to let the Spring IoC container do this job for you. The valang validator was design as a POJO specifically for that reason - to easily define it within Spring application context and inject it to all other dependent objects in the application.

Here is an example of how to define a simple valang validator within the application context:

```
<bean id="personValidator"
                        class="org.springmodules.validation.valang.ValangValidator">
                        <property name="valang">
                        <value>
                        <![CDATA[
                        { firstName : length(?) < 30 : 'First name too long' : 'first_name_length' : 30}
                        { lastName : length(?) < 50 : 'Last name too long' : 'last_name_length' : 50 }
                        ]]>
                        </value>
                        </property>
                        </bean>
```

This validator defines two validation rules - one for the maximum size of the first name of the person and the other for the maximum size of the last name of the person.

Also notice that the above validator is unaware of the object type it validates. The valag validator is not restricted to a specific class to be validated. It will always apply the defined validation rules as long as the validated object has the validated properties (firstName and lastName in this case).

This configuration should be enough for most cases. But there are some cases in which you need to apply extra configuration. With *ValangValidator* it is possible to register custom function (thus, extend the valang expression language). This can be done by registering the functions within the *customFunctions* property, where the function name serves as the registration key.

Here is an example of a valang validator configuration with a custom function:

```
<bean id="personValidator"
                        class="org.springmodules.validation.valang.ValangValidator">
                        <property name="customFunctions">
                        <map>
                        <entry key="doIt">
                        <value>org.springmodules.validation.valang.functions.DoItFunction</value>
                        </entry>
                        </map>
                        </property>
                        <property name="valang">
                        <value>
                        <![CDATA[
                        { firstName : doIt(?) and length(?) < 30 : 'First name too long' : 'first_name_length' :
                        { lastName : length(?) < 50 : 'Last name too long' : 'last_name_length' : 50 }
                        ]]>
                        </value>
                        </property>
                        </bean>
```

It is also possible to register extra property editors and custom date parsers for valang to use. For more details about valang validator configuration options, please refer to the class javadoc.

17.2.3. Client Side Validation

The Valang to JavaScript conversion service is a simple extension to the Valang validator package that allows you to use the same Valang validation rules for client-side JavaScript validation that you are already using for your server-side controller validation.

The default JavaScript validator will be activated when the user tries to submit the form being validated and any errors detected by the validator will be presented in an alert box, but if you wish, you may customize any aspect of the validator to suit you needs.

17.2.3.1. Getting Started

Note

This Getting Started guide assumes that you are using Valang as the Validator implementation for your Spring MVC controllers and using JSP for your views.

There are 3 simple steps that are needed to enable the Valang to JavaScript translation:

Step 1 - Add the Valang rules exporter to your DispatcherServlet configuration

Because the translation of your validation rules happens in a custom tag it is necessary for any Valang validation rules used in your controller to be exported into the JSP page context. A convenient Spring MVC interceptor is provided that will automatically export the Valang validation rules for any controllers that make use of them.

If you are using the default handler mapping provided by the DispatcherServlet you will need to add the following to your dispatcher config file:

```
<bean id="handlerMapping"
                    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
                    <property name="interceptors">
                    <list>
                    <bean class="org.springmodules.validation.valang.javascript.taglib.ValangRulesExportIntercep
                    />
                    </list>
                    </property>
                    </bean>
```

or, if you have already configured an alternative handler mapping, all you need to do is include the additional ValangRulesExportInterceptor in the list of interceptors used by your custom handler mapping:

```
<bean id="myHandlerMapping" class="org.springframework.web.servlet.handler.?">
                    ...
                    <property name="interceptors">
                    <list>
                    ...
                    <bean class="org.springmodules.validation.valang.javascript.taglib.ValangRulesExportIntercep
                    />
                    </list>
                    </property>
                    </bean>
```

Step 2 - Import the Valang custom tag and JavaScript codebase into your JSP view

In the JSP file that is used to render the view the form you wish to validate you will need to import the Valang custom tag library by including the following line at the top of the file:

```
<%@taglib uri="/WEB-INF/tlds/valang.tld" prefix="valang" %>
```

then somewhere in the HTML *<head>* section of your JSP template include the JavaScript codebase that you previously saved to your web application.

```
<html>
                    <head>
                    ...
                    <valang:codebase includeScriptTags="true" fieldErrorsIdSuffix="_error"
                    globalErrorsId="global_error"/>
                    ...
                    </head>
```

- *includeScriptTags* - indicates whether the generated code should also generate the wrapping <script> tags. Should be set to *false* if this this line is put within an already existing <script> tag.
- *fieldErrorsIdSuffix* - indicates suffix of the id of the html element (usually a <div> or a <span>) that would render the field error. The generated javascript code will render a filed error in the html element with the id *<field_name><suffix>* . For example, in the case of a suffix "_error" and field name "firstName" the error will be rendered within the element with id "firstName_error". If no element with that id is found, the error will be rendered in a javascript alert dialog.
- *globalErrorsId* - indicates the id of the html element where the global errors will be rendered.

Step 3 - Use the <valang:validate> tag to generate the JavaScript validator

Inside the HTML for the form element you wish to validate use the <vl:validate> tag to generate the JavaScript validator

```
<form method="post">
                    <valang:validate commandName="command" />
                    ...
                    </form>
```

or if you have some validation rules you would like to have applied in addition to the rules that were exported in Step 1 you may include these in the body of the <vl:validate> tag:

```
<form method="post">
                    <vl:validate commandName="command">
                    { firstName : length(?) < 30 : 'This rule only gets evaluated in JavaScript' }
                    </vl:validate>
                    ...
                    </form>
```

Note

The <vl:validate> tag must be inside the <form></form> tags for the form you wish to have the validation apply. If you do not do this the JavaScript validator will be unable to locate the form it is expected to validate and a JavaScript exception will be thrown

17.2.3.2. Customization

Custom Validation Functions

When the Valang to JavaScript translator encounters a custom validation function it generates a call to a JavaScript function with the same name as the Java class of the custom function. e.g. if your validation rule uses a custom validation function whose class is *my.custom.validators.IsEmailFunction* then the translated JavaScript validation call will be to a function *ValangValidator.Rules.prototype.IsEmailFunction()* .

It is therefore necessary that you provide a JavaScript implementation for the function call generated by the JavaScript translator which is as simple as adding a new function to the *ValangValidator.Rules.prototype* object. So for example - the *IsEmailFunction* described above would need the following JavaScript function to be defined after the <valang:codebase...> tag:

```
ValangValidator.Rule.prototype.IsEmailFunction = function(value) {
```

```
                    // very simple email validation
                    return value && value.indexOf(".") > 2 && value.indexOf("@") > 0
                    }
```

Customizing the Validation feedback

If you wish to customize the validation feedback that is provided by the JavaScript Valang validator you will need to override the function *ValangValidator.prototype.showValidationFeedback* . This function, which is only called when a validation error has been detected, takes one argument which is an array containing all of the failed validation rules.

It is recommended that you look at the existing implementation as a guide to how the validator handles feedback.

17.2.3.3. Localization

The current implementation supports localized error messages as long as they do not use any arguments. Messages that take arguments will currently be ignored and only the default error message will be used.

17.2.3.4. Troubleshooting

If the validator is not working the first thing you should do is have a look at the JavaScript console; there is likely to be an error message showing there.

For more detail of what the JavaScript Valang validator doing you can enable the extensive logging functionality. To enable the log output all you need to do is include the following HTML (it is recommended that this be placed at the end of the page):

```
<div id="valangLogDiv"></div>
```

Once you reload the page the JavaScript Valang validator will detect and output logging information into this div.

17.3. Bean Validation Framework

17.3.1. Introduction

The Bean Validation Framework was developed to fill in gaps in the other available validation approaches, namely commons validator and valang. The main goals of this framework are:

- Enable clear separation of concerns when it comes to validation
- Extract the best features from the other approaches and unite them under one framework
- Make use of latest technologies and methodologies
- Make it as intuitive and simple as possible to define the validation rules in the application

Separation of Concerns

Essentially, validation is there to help enforce the integrity of the data within an application. The integrity is defined by a set of well defined constraints which the application relies on in order to work properly. There are several types of such constraints:

- *Domain model constraints* - These constraints are defined on top of the domain model data that is used within the program. These are dictated by the domain in which the application operates in.
- *Data constraints* - These are not part of the domain but more of technical limitations that define additional constraints. An example of such constraint is a database constraint where a column can only hold a string value with a maximum number of characters.
- *Business (Logic) Rules* - Some constraints are defined by the business rules that drive the application. These rules define the valid interactions between the domain model and the services provided by the application. The business rules can be seen as domain oriented constraints applied on services rather than on data. An example of such a rule can be "Username must be unique between all users".
- *Application Constraints* - Constraints on data/services that are not part of the domain per se, but part of the application. These constraints come on top of the domain oriented constraints and define additional contract between the application and its clients. For example, in the user registration form, the user needs to fill in its password twice for confirmation. This is clearly an application constraint. As far as the domain is concerned, there is only one password, but the application need to accept two passwords as an input.

The validation framework focuses on the first two constraint types above. It enables defining the domain model and data constraints in an explicit manner. Application constraints are also partially supported as the same techniques that are used to define the domain model constraints can also be use to define the application data constraints.

Uniting Best Practices

Perhaps the most known and used validation framework is commons-validation. Originally this framework was developed as the validation mechanism for the Struts web framework, and later it was extracted and became an independent project. There are good and bad sides to this framework. The good is that in a way it introduced declarative validation and showed that validation can be configured and does not necessarily needs to be part of the code. The bad part is that, as mentioned above, it originated in the Strut web framework making it very web oriented. Commons validation does the job, but there is quite a lot of room for improvement - mainly in the extensive, complex, and non-intuitive xml configurations. The web orientation, for example, is clearly apparent in the configuration as it deals with terms like forms and javascript.

In early [TODO: date] Steven Devijver created Valang and committed it to spring modules. Valang (stands for *Va* -lidation *Lang* -uage) introduced a new approach towards validation. Instead of using predefined validator classes, valang provides a powerful and extensible expression language to represent validation rules. One of the most appealing features in Valang is the ability to express complex boolean expressions using a quite natural language which is both intuitive to write and very easy to read. The downside of Valang is that it mixes these boolean expressions and the validation configuration in one language. Experience shows that complex validations become quite verbose due to this tight coupling.

The validation framework takes the good parts from both of these approaches and aims to fix and improve the bad ones. One major improvement is the abstraction of the validation configuration and the clear separation between it and the actual alidation rules constructs. This abstraction enables the usage of different types of configuration mechanism implementations. The framework comes with out of the box XML files and Java 5 annotation support, but other mechanisms can easily be created as well. With the lessons learned from Valang, the framework also makes use of a boolean expression language where possible. The expression language itself can change according to the developer preferences.

Makes Use of Latest Technologies

As mentioned above, the validation framework tried to make use of the latest technologies. The use of Java 5 annotations and advanced expression languages serve evident for that.

Simple & Intuitive

One of the main goals of this framework is to make it as intuitive and simple as possible for the developer to use. If XML configuration is used, the element names are very descriptive and as opposed to commons-validator, do not use any web related terms, but more of generic terms (e.g. beans, properties, etc...). The use of annotations also makes using this framework very intuitive to work with. While the two different configuration mechanisms are totally independent, they both use the same terms and names of common validation rules.

17.3.2. Using the Framework

In this section you'll learn how to make use of the validation framework. The core components in this framework will be introduced and we'll show how they interact with each other. Along the way you'll also learn how to configure the validation rules both in XML and annotations and how Spring 2.0 namespaces support makes integration in a typical spring application even easier.

17.3.2.1. The Validation Rule

Spring defines the *org.springframework.validation.Validator* interface to abstract the task of object validation. While this is a good abstraction, in practice, all the implementation of this interface does is applying one or more validation rules on the validated object and populating the passed in *org.springframework.validation.Errors* argument with all encountered validation errors.

Realizing that, a finer grained abstraction is introduced in this framework - The *org.springmodules.validation.bean.rule.ValidationRule* . Here is the complete interface:

```
public interface ValidationRule {

    boolean isApplicable(Object obj);

    Condition getCondition();

    String getErrorCode();

    Object[] getErrorArguments(Object obj);

    String getDefaultErrorMessage();

}
```

A validation rule can be seen as a <condition, error> pair. The condition represents the predicate to be evaluated up on the object and determine whether this rule sees this object as valid or not. The error defines the validation error in case the rule sees this validated object as invalid. The interface above supports this definition as follows:

- The *getCondition()* method returns the condition/predicate of this rule.
- The *getErrorCode()* method returns the error code of the rule. This enables associating codes to different validation errors which can be resolved to a descriptive message using a resource bundle.
- The *getErrorArguments(Object)* method returns any arguments that might be useful for the error indication.

For example, when a validation rule checks that a string value is at least 4 chars long, two arguments that might be useful in case of validation failure are the actual and required lengths of the string. Notice that this method accepts the validated object as an argument. This enables returning arguments based on the validated object (e.g. This is how the actual string length would be returned as an error argument in the example above).

- The *getDefaultErrorMessage()* method returns a string message describing the error that can be used as the default error message when the error code is not used.

Last but not least, the *isApplicable(Object)* method is there to indicate whether or not the rule can be applied on the validated object in the first place. This is an important feature for it enables support for conditional validation rules. Conditional validation rules are applied on an object only if a certain condition is met. For example, lets assume we have an account in a web shop where a typical account can hold two addresses: shipping address and billing address. Just to make the point, we'll also assume that an account holds another flag indicating whether the shipping address should also be used as the billing address. In this scenario, validation rules defined for the billing address should only be applied if that flag is set to false. This is where the conditional rule will play part.

AbstractValidationRule and Predefined Rules

The framework comes with AbstractValidationRule class which is a parent class with default behavior to make it easier for developer to write custom validation rules. This class relies on the following two strategies:

- *Applicability Condition* - This condition helps in the decision whether the validation rule can be applied to a given validated object.
- *ErrorArgumentResolver* - A strategy to extract the error arguments based on the validated object.

The following are all the pre-defined validation rules the framework comes with:

- *NotNullValidationRule* - Checks that the validated object is not null.
- *ExpressionValidationRule* - Checks the validated object against a condition expression [TODO see...]
- *DateInThePastValidationRule* - Checks that the validated Date/Calendar occurred in the past (relative to the validation time)
- *DateInTheFutureValidationRule* - Checks that the validated Date/Calendar occurred in the future (relative to the validation time)
- *InstantInThePastValidationRule* - Checks that the validated joda-time instant occurred in the past (relative to the validation time)
- *InstantInTheFutureValidationRule* - Checks that the validated joda-time instant occurred in the future (relative to the validation time)
- *LengthValidationRule* - Checks that the length of the validated string is within specific bounds
- *MaxLengthValidationRule* - Checks that the length of the validated string is less than or equals a specific bound.
- *MinLengthValidationRule* - Checks that the length of the validated string is greater than or equals a specific bound.

- *NotBlankValidationRule* - Checks that the validated String is not blank, that is, has length greater than zero.
- *RegExpValidationRule* - Checks that the validated String matches a specific regular expression.
- *EmailValidationRule* - Checks that the validated string is a valid email address
- *SizeValidationRule* - Checks that the size of the validated collection/array is within specific bounds
- *MaxSizeValidationRule* - Checks that the size of the validated collection/array is less than or equals a specific bound.
- *MinSizeValidationRule* - Checks that the length of the validated collection/array is greater than or equals a specific bound.
- *NotEmptyValidationRule* - Checks that the validated collection/array is not empty.
- *RangeValidationRule* - Checks that the validated Comparable is within a specific range.
- *MaxValidationRule* - Checks that the validated Comparable is less than or equals a specific bound.
- *MinValidationRule* - Checks that the validated Comparable is greater than or equals a specific bound.
- *PropertyValidationRule* - A validation rule that applies another validation rule on the value of a specific property of the validated object.

17.3.2.2. Validation Configuration & Configuration Loader

As mentioned above, one of the key features in this framework is the clear abstraction of the validation configuration. This abstraction is represented by the *org.springmodules.validation.bean.conf.BeanValidationConfiguration* interface. Here is the complete interface:

```
public interface BeanValidationConfiguration {

    ValidationRule[] getGlobalRules();

    ValidationRule[] getPropertyRules(String propertyName);

    String[] getValidatedProperties();

    Validator getCustomValidator();

    CascadeValidation[] getCascadeValidations();

}
```

As its name suggests, the bean validation framework focuses on validating beans. One of the main goals of this framework was to enable defining validation rules for bean classes. The ValidationConfiguration interface provides all information needed about those validation rules and it's largely based on the same terminology used in Spring's Errors interface:

- The *getGlobalRules()* method returns all validation rules that are associated globally with the validated object. In practice, this association means the encountered validation errors will be registered globally for this object.
- The *getPropertyRules(String)* method returns all validation rules that are associated with the given property name. In practice, this association means the encountered validation errors will be registered at the field/property level. This method is closely related to the *getValidatedProperties()* method (described below) in the sense that only those property names returned by that method should be passed in as an

argument to this method.

- The *getValidatedProperties()* method returns the names of all the validated properties, that is, all properties that have validation rules associated with them.
- The *getCustomValidator()* method returns a Spring Validator implementation that should be used to validate the validated object. This enables associating custom validator implementation for the object and still configure it using the same configuration mechanism used to configure the validation rules. It enables easy pluggability of validation that cannot be easily captured by neither the global nor the property validation rules.
- The *getCascadeValidations()* method returns all the cascade validations associated with this configuration (see below).

Cascade Validation

Cascade validation is probably best explained by an example: Consider a Person class that is associated with an Address class. The two classes are indeed associated but still are independent and so are their validation rules. Nonetheless, it is sensible to say that a person instance is valid only if its associated address is valid. Thus, when validating the person you'd like to notify the validation mechanism that it should also validate the address that associated with it. This is where the cascade validation comes in the picture. The org.springmodules.validation.bean.conf.CascadeValidation class holds a property name indicating the property/association that should be validated as part of the validation of its parent. This class also holds an applicability condition to determine when this cascading behavior should be applied.

Providing an abstraction over the configuration information is nice, but it's not complete. To complete the abstraction we also need to abstract the manner in which this configuration is being loaded. This is reason behind the org.springmodules.validation.bean.conf.loader.BeanValidationConfigurationLoader interface. Here is the complete interface:

```
public interface BeanValidationConfigurationLoader {

    boolean supports(Class clazz);

    BeanValidationConfiguration loadConfiguration(Class clazz);

}
```

This is a simple interface that represents the strategy in which the validation configuration is loaded for a bean class.

- The *supports(Class)* method indicates whether this loader can load (or can try to load) the validation configuration for the given class.
- The *loadConfiguration(Class)* method does the actual loading of the validation configuration for the given class.

The validation framework supports two configuration loading mechanisms out of the box: XML configuration files and Java 5 annotations.

17.3.2.3. XML Configuration

The org.springmodules.validation.bean.conf.loader.xml.DefaultXmlBeanValidationConfigurationLoader class is the configuration loader implementation that loads the configuration from XML files.

If you already worked with XML mapping files such as Hibernate's ones, you'd find the structure of the validation XML files quite familiar. Here is a sample configuration file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<validation xmlns="http://www.springmodules.org/validation/bean"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springmodules.org/validation/bean
    http://www.springmodules.org/validation/bean/validation.xsd">

    <class name="Person">

        <validator class="PersonValidator" />

        <global>
           <!-- global validation rules go here -->
        </global>

        <property name="firstName">
            <!-- property validation rules go here -->
        </property>

        <property name="address" cascade="true"/>

    </class>

</validation>
```

- The *validation* element is the root element of the xml configuration. This element can hold zero or more *class* elements. This enables configuring validation rules for multiple classes in one file (Although the recommend approach is to have a separate mapping file per class).
- The *class* element defines the validation configuration for a specific class. The class is indicated by the *name* attribute. This can be a fully qualified class name or the class simple name. In the later case, the package name can be defined globally in the *package* attribute of the *validation* element. (Note: even if the package name is defined in the *validation* level, it is still possible to define fully qualified names in the *name* attribute of the *class* element. In that case, the *package* attribute will be ignored.
- The *validator* element enables configuring custom validators for the class. The validator class is indicated by the *class* required attribute. This attribute must hold the fully qualified name of a class that implements Spring's Validator interface. Currently it is required that this class will expose a default constructor which will be used to instantiate the validator. There can be zero or more *validator* elements under the *class* element.
- The *global* element contains all the global validation rules for the class. For more information on supported global validation rules see [TODO]. Zero or one *global* elements can be defined under the *class* element.
- The *property* element defines the configuration rules for a specific property identified by the *name* attribute. It is also possible to define the cascading validation for the associated property using the *cascade* property (which accepts *true|false*) and the *cascade-condition* property (which accepts a boolean expression that defines the condition under which the validation cascading should be applied). This element contains all the property validation rule definitions for the associated property. There can be zero or more *property* element under the *class* element.

Global Validation Rule Definitions

There two global validation rule definition that come out-of-the-box: The *expression* definition and the *validation-ref*.

<validation-ref>

This rule looks up a spring Validator in the application context and applies it on the validated bean. Obviously, for this rule to work, the appropriate validation configuration loader needs to be defined in an application context (or at least be associated with it). The attributes of this element are described below:

Table 17.2.

Name	Required	Type	Description
name	yes	string	Holds a the name of a Validator bean as defined in the application context.

<expression>

This element can be used to define a validation rule using an expression language. The attributes of this element are described below:

Table 17.3.

Name	Required	Type	Description
condition	yes	string	Holds a condition expression to indicate the global validation rule constraint that should be checked when validating an object. For more information on condition expressions see [TODO]
code	no	string	Specifies the error code associated with this validation rule
message	no	string	Specifies the default error message that should be used in case the error code cannot be used
args	no	string	Specifies a comma-separated list of arguments that should be passed along with the error code. The strings in the list can either be static strings or function expressions. For more information on function expressions see [TODO]
apply-if	no	string	Holds a condition expression to represent the applicability condition of this validation rule. For more information on condition expressions see [TODO]
contexts	no	string	A comma-separated list of validation context tokens to associate with the rule. The rule will only be applied if any of the configured context tokens is supported by the current validation context.

Property Validation Rule Definitions

All property validation rule definitions that come with the framework share common attributes. These attributes are listed in the following table:

Table 17.4.

Name	Required	Type	Description
code	no	string	Specifies the error code associated with this validation rule
message	no	string	Specifies the default error message that should be used in case the error code cannot be used

Name	Required	Type	Description
args	no	string	Specifies a comma-separated list of arguments that should be passed along with the error code. The strings in the list can either be static strings or object expressions. For more information on object expressions see [TODO]
apply-if	no	string	Holds a condition expression to represent the applicability condition of this validation rule. For more information on condition expressions see [TODO]
contexts	no	string	A comma-separated list of validation context tokens. The rule will only be applied if one of the configured context tokens is supported by the current validation context. see [TODO]

<not-null>

Represents a org.springmodules.validation.bean.rule.NotNullValidationRule

<in-future>

Represents either org.springmodules.validation.bean.rule.DateInTheFutureValidationRule or org.springmodules.validation.bean.rule.InstantInTheFutureValidationRule (depending on the data type). This validation rule checks that the validated date/calendar/instant occurred in the future (relative to the time of validation).

<in-past>

Represents either org.springmodules.validation.bean.rule.DateInThePastValidationRule or org.springmodules.validation.bean.rule.InstantInThePastValidationRule (depending on the data type). This validation rule checks that the validated date/calendar/instant occurred in the past (relative to the time of validation).

<email>

Represents a org.springmodules.validation.bean.rule.EmailValidationRule. This rule validates that a string value holds a valid email address.

<length>

Represents a org.springmodules.validation.bean.rule.LengthValidationRule. This rule checks that the range of a string's length is within specific bounds.

Table 17.5. Extra Attributes

Name	Required	Type	Description
min	only if max is not defined	non-negative integer	Specifies the minimum length the string value can have
max	only if min is not defined	non-negative integer	Specifies the maximum length the string value can have

Of course, both *min* and *max* attribute can be defined, in which case the length of the sptring value will have to be within the specified range.

<not-blank>

Represents a org.springmodules.validation.bean.rule.NotBlankValidationRule. This rule checks that a string value is not blank (that is, it holds some characters).

<not-empty>

Represents a org.springmodules.validation.bean.rule.NotEmptyValidationRule. This rule checks that a collection/array is not empty (that is, it holds at least one element).

<size>

Represents a org.springmodules.validation.bean.rule.SizeValidationRule. This rule checks that the size of a collection/array is within specific bounds.

Table 17.6. Extra Attributes

Name	Required	Type	Description
min	only if max is not defined	non-negative integer	Specifies the minimum size of the collection or array
max	only if min is not defined	non-negative integer	Specifies the maximum size of the collection or array

Of course, both *min* and *max* attribute can be defined, in which case the size of the collection or array will have to be within the specified range.

<range>

Represents a org.springmodules.validation.bean.rule.RangeValidationRule. This rule checks that a *java.lang.Comparable* value is within a specific range.

Table 17.7. Extra Attributes

Name	Required	Type	Description
min	only if max is not defined	string | number	Specifies the lower bound of the range
max	only if min is not defined	string | number	Specifies the upper bound of the range

<regexp>

Represents a org.springmodules.validation.bean.rule.RegExpValidationRule. This rule checks that a string value matches a specific regular expression.

Table 17.8. Extra Attributes

Name	Required	Type	Description
expression	yes	string	Specifies the regular expression

<expression>

Represents a org.springmodules.validation.bean.rule.ExpressionValidationRule. This rule checks the validated value against a condition expression (see [TODO CEL]).

Table 17.9. Extra Attributes

Name	Required	Type	Description
expression	yes	string	Specifies the condition expression
scope	no	enum [global|property]	Defines the evaluation scope of the condition expression. *global* will define the validated object as the scope, while *property* will define the property value as the scope. Using the property scope enables the definition of conditions that apply directly on the property value (e.g. "length > 5"). The global scope enables the definition of conditions that apply on other properties of the validated object (e.g. "equals some_other_property")

<condition-ref>

Represents a org.springmodules.validation.bean.rule.ConditionReferenceValidationRule. This rule uses a condition bean that is defined in the application context to perform the validation. NOTE: in order for this rule to work, the appropriate validation configuration loader needs to be defined within an application context (or at least be associated with it).

Table 17.10. Extra Attributes

Name	Required	Type	Description
name	yes	string	Specifies name of the condition bean as defined in the application context.

Custom Validation Rule Definitions

The pre-defined set of validation rules is pretty extensive and with the *<expression>* definition the majority of validation rules are supported out-of-the-box (well, at least the common ones). However, there will always be occasions where you would want to write your own custom validation rule and you would want it to be configured within the same XML configuration file where all other rules are configured.

The xml configuration loader org.springmodules.validation.bean.conf.loader.xml.handler.ClassValidationElementHandler org.springmodules.validation.bean.conf.loader.xml.handler.PropertyValidationElementHandler to parse the validation rules in the configuration. These handlers are registered within a org.springmodules.validation.bean.conf.loader.xml.ValidationRuleElementHandlerRegistry which the loader uses to find the proper handlers.

Here is the complete ClassValidationElementHandler interface:

```
public interface ClassValidationElementHandler {

    boolean supports(Element element, Class clazz);

    void handle(Element element, MutableBeanValidationConfiguration configuration);

}
```

- The *supports(Element, Class)* method determines whether the handler knows how to handle the given validation rule definition element for the given class. Based on this method, the loader knows whether the *handle(...)* method should be called on this handler.
- The *handle(Element, MutableBeanValidationConfiguration)* method handles the given validation rule definition element by manipulating the given configuration accordingly.

Here is the complete PropertyValidationElementHandler interface:

```
public interface PropertyValidationElementHandler {

    boolean supports(Element element, Class clazz, PropertyDescriptor descriptor);

    void handle(Element element,
                String propertyName,
                MutableBeanValidationConfiguration configuration);

}
```

- The *supports(Element, Class, PropertyDescriptor)* method determines whether this handler knows how to handle the given property validation rule definition element for the given class and property descriptor.
- The *handle(Element, String, MutableBeanValidationConfiguration)* method handles the given validation rule definition element by manipulating the given configuration accordingly.

There are two abstract implementations of these interfaces above to make it easy for developers to create custom element handlers. These classes rely on the fact that creating a custom handler usually goes hand in hand with the creation of custom validation rules. In fact, this approach (of creating both) is the one we recommend on. When using these parent classes, all that is needed is basically to create the appropriate validation rule based on the element structure (e.g. the "min" attribute in the range element represents the lower bound of the range) and define the element name. All pre-defined element handlers are implemented in this way, so it might be good to use them as a reference.

17.3.2.4. Java 5 Annotation Configuration

The org.springmodules.validation.bean.conf.loader.annotation.AnnotationBeanValidationConfigurationLoader class is the configuration loader implementation that loads the configuration from Java 5 annotations.

Here is the same configuration as described above in XML defined using annotations:

```
@Validator(PersonValidator.class)
    @Expression("address is not null")
    public class Person {

    @Length(min = 2, max = 10)
    private String firstName;

    @CascadeValidation
    private Address address;

    ...
}
```

- The *@Validation* annotation defines a custom validator class to be used for extra validation. This is the annotation counterpart of the *<validator>* xml definition.
- All global validation rules can be defined as annotations on the class level. In this example the *@Expression* annotation defines a condition expression as a global validation rule.

- The property validation rules can be defined either on the property setter method or on the associated class field. We believe that putting the annotations on the class fields is cleaner and more intuitive, but it all boils down to your personal preference. In this example the *@Length* annotation defines a LengthValidationRule on the *firstName* property. We can also see that the validation cascading behavior can be defined using the *@CascadeValidation* (as in the address property example)

Global Validation Rule Annotations

There four global validation rule annotation that comes with the framework out-of-the-box:

@Expression

This annotation represents an expression based validation rule.

Table 17.11. Attributes:

Name	**Type**	**Required**	**Description**
value	string	yes	Holds a condition expression to indicate the global validation rule constraint that should be checked when validating an object. For more information on condition expressions see [TODO]
errorCode	string	no	Specifies the error code associated with this validation rule
message	no	string	Specifies the default error message that should be used in case the error code cannot be used
args	no	string	Specifies a comma-separated list of arguments that should be passed along with the error code. The strings in the list can either be static strings or function expressions. For more information on function expressions see [TODO]
applyIf	no	string	Holds a condition expression to represent the applicability condition of this validation rule. For more information on condition expressions see [TODO]
scope	no	ExpressionScope	This attribute is ignored when put in the class level.
contexts	no	string	A comma-separated list of validation context tokens to associate with the rule. The rule will only be applied if any of the configured context tokens is supported by the current validation context.

@Expressions

This annotation enables defining multiple *@Expression* annotations on the class level.

@Validator

This annotation defines a custom validator class to be used in the validation process. It has only one required attribute *class* that defines the class of the validator.

@Validators

This annotation enabled defining multiple *@Validator* annotation on the class level.

@ValidatorRef

This annotation defines a Validator bean that is deployed in a spring application context to be used in the validation process. It has only one required attribute *value* (the default attribute) that defines the name of the bean as defined in the application context.

Property Validation Rule Annotations

All property validation rule annotation that come with the framework share common attributes. These attributes are listed in the following table:

Table 17.12.

Name	Type	Required	Description
errorCode	string	no	Specifies the error code associated with this validation rule
message	no	string	Specifies the default error message that should be used in case the error code cannot be used
args	no	string	Specifies a comma-separated list of arguments that should be passed along with the error code. The strings in the list can either be static strings or function expressions. For more information on function expressions see [TODO]
applyIf	no	string	Holds a condition expression to represent the applicability condition of this validation rule. For more information on condition expressions see [TODO]
contexts	no	string	A comma-separated list of validation context tokens to associate with the rule. The rule will only be applied if any of the configured context tokens is supported by the current validation context.

@NotNull
Represents a org.springmodules.validation.bean.rule.NotNullValidationRule. This rule validates that the validated value is not null.

@InTheFuture
Represents either org.springmodules.validation.bean.rule.DateInTheFutureValidationRule or org.springmodules.validation.bean.rule.InstantInTheFutureValidationRule (depending on the data type). This validation rule checks that the validated date/calendar/instant occurs in the future (relative to the time of validation).

@InThePast
Represents either org.springmodules.validation.bean.rule.DateInThePastValidationRule or org.springmodules.validation.bean.rule.InstantInThePastValidationRule (depending on the data type). This validation rule checks that the validated date/calendar/instant occurred in the past (relative to the time of validation).

@Email
Represents a org.springmodules.validation.bean.rule.EmailValidationRule. This rule validates that a string value holds a valid email address.

@Length
Represents a org.springmodules.validation.bean.rule.LengthValidationRule. This rule checks that the range of a string's length is within specific bounds.

Table 17.13. Extra Attributes

Name	Required	Type	Description
min	yes	non-negative integer	Specifies the minimum length the string value can have
max	yes	non-negative integer	Specifies the maximum length the string value can have

@MinLength

Represents a org.springmodules.validation.bean.rule.LengthValidationRule. This rule checks that a string's length is greater than or equals a specific lower bound.

Table 17.14. Extra Attributes

Name	Required	Type	Description
value	yes	non-negative integer	Specifies the minimum length the string value can have

@MaxLength

Represents a org.springmodules.validation.bean.rule.LengthValidationRule. This rule checks that the range of a string's length is less than or equals a specific upper bound.

Table 17.15. Extra Attributes

Name	Required	Type	Description
max	yes	non-negative integer	Specifies the maximum length the string value can have

@NotBlank

Represents a org.springmodules.validation.bean.rule.NotBlankValidationRule. This rule checks that a string value is not blank (that is, it holds some characters).

@NotEmpty

Represents a org.springmodules.validation.bean.rule.NotEmptyValidationRule. This rule checks that a collection/array is not empty (that is, it holds at least one element).

@Size

Represents a org.springmodules.validation.bean.rule.SizeValidationRule. This rule checks that the size of a collection/array is within specific bounds.

Table 17.16. Extra Attributes

Name	Required	Type	Description
min	yes	non-negative integer	Specifies the minimum size of the collection or array
max	yes	non-negative integer	Specifies the maximum size of the collection or array

@MinSize

Represents a org.springmodules.validation.bean.rule.SizeValidationRule. This rule checks that the size of a collection/array is greater than or equals a specific lower bound.

Table 17.17. Extra Attributes

Name	Required	Type	Description
value	yes	non-negative integer	Specifies the minimum size of the collection or array

@MaxSize

Represents a org.springmodules.validation.bean.rule.SizeValidationRule. This rule checks that the size of a collection/array is less than or equals a specific bound.

Table 17.18. Extra Attributes

Name	Required	Type	Description
value	yes	non-negative integer	Specifies the maximum size of the collection or array

@Range

Represents a org.springmodules.validation.bean.rule.RangeValidationRule. This rule checks that a *java.lang.Comparable* value is within a specific range.

Table 17.19. Extra Attributes

Name	Required	Type	Description
min	only if max is not defined	double	Specifies the lower bound of the range
max	only if min is not defined	double	Specifies the upper bound of the range

@Min

Represents a org.springmodules.validation.bean.rule.RangeValidationRule. This rule checks that a *java.lang.Comparable* value is greater than or equals a specific lower bound.

Table 17.20. Extra Attributes

Name	Required	Type	Description
value	only if max is not defined	double	Specifies the lower bound of the range

@Max

Represents a org.springmodules.validation.bean.rule.RangeValidationRule. This rule checks that a *java.lang.Comparable* value is less than or equals a specific upper bound.

Table 17.21. Extra Attributes

Name	Required	Type	Description
value	only if min is not defined	double	Specifies the upper bound of the range

@RegExp

Represents a org.springmodules.validation.bean.rule.RegExpValidationRule. This rule checks that a string value matches a specific regular expression.

Table 17.22. Extra Attributes

Name	Required	Type	Description
value	yes	string	Specifies the regular expression

@Expression

Represents a org.springmodules.validation.bean.rule.ExpressionValidationRule. This rule checks the validated value against a condition expression (see [TODO CEL]).

Table 17.23. Extra Attributes

Name	Required	Type	Description
value	yes	string	Specifies the condition expression
scope	no	ExpressionScope	Defines the evaluation scope of the condition expression. *global* will define the validated object as the scope, while *property* will define the property value as the scope. Using the property scope enables the definition of conditions that apply directly on the property value (e.g. "length > 5"). The global scope enables the definition of conditions that apply on other properties of the validated object (e.g. "equals some_other_property")

@Expressions

This annotation enables defining multiple *@Expression* annotations on a property.

@ConditionRef

Represents a org.springmodules.validation.bean.rule.ConditionReferenceValidationRule. This rule uses a condition that is defined in the application context to perform the validation check.

Table 17.24. Extra Attributes

Name	Required	Type	Description
value	yes	string	Specifies the name of the condition bean as defined in the application context.

17.3.2.4.1. Hibernate Validator & JPA Annotations

Hibernate Validator Support
The framework will pick up any hibernate validator annotation that is used and register the validation rules appropariately.

JPA Support
Four JPA annotations are currently supported:

- *@Column* - If the *nullable* attribute is *false*, the framework will treat it as if a *@NotNull* annotation was defined. The framework will also extract the value from the *length* attribute and treat it as the *@MaxLength*.
- *@Basic* - If the *optional* attribute is *false*, the framework will treat it as if *@NotNull* annotation was defined.
- *@OneToOne* - If the *optional* attribute is *false*, the framework will treat it as if *@NotNull* annotation was defined.
- *@ManyToOne* - If the *optional* attribute is *false*, the framework will treat it as if *@NotNull* annotation was defined.

17.3.2.5. Condition Expression Language (CEL) & Function Expression Language (FEL)

If there's one good lesson learned from Valang, is the significant role an expression language can play when it comes to validation. By no means, the validation framework core constructs depends on any expression language, but as mentioned above, the default implementations do use them heavily. Using expression language enables defining relatively complex and less common validation rules without the need to write custom java code for it. Of course it also depends on the richness of the language used.

The validation framework recognizes two types of expression languages: Condition Expression Language (CEL) and Function Expression Language (FEL).

Condition Expression Language (CEL)

As the name suggests, CEL represents an expression language that defines conditions. A Condition can be seen as a predicate, that given an object it evaluates to either *true* or *false* .

As you probably know by now, CEL fits easily within the validation framework, and practically can be used whenever a condition is used. The various validation rule handlers (annotation or xml element) are good examples where you'd rather use CEL in the configuration than writing java code to represent an applicability condition for example.

When designing the framework, we didn't want to tie the user to a specific expression language and an abstraction was added to support such languages. The relevant constructs for this abstraction are located in the *org.springmodules.validation.util.cel* package and are listed below:

ConditionExpressionParser

This interface represents a parser that can parse a CEL expression and construct a *org.springmodules.validation.util.condition.Condition* from it. Here is the complete interface:

```
public interface ConditionExpressionParser {

                    Condition parse(String expression) throws CelParseException;

                    }
```

Note

All exceptions thrown by the CEL constructs are unchecked exceptions. The CelParseException is declared above just for clarity.

There are two concrete implementations of this interface supplied by the framework out of the box: The first is *org.springmodules.validation.util.cel.valang.ValangConditionExpressionParser* which is based on the Valang language, and the second is *org.springmodules.validation.util.cel.ognl.OgnlConditionExpressionParser* which is based on the OGNL [TODO add link] expression language. The former is the preferred and the default one used by the framework (although changing it is just a matter of configuration).

ConditionExpressionBased

This interface should be implemented by any class that depends on *ConditionExpressionParser* . It defines only one method which is a setter to set the condition expression parser. *org.springmodules.validation.bean.conf.loader.xml.handler.AbstractPropertyValidationElementHandler* and *org.springmodules.validation.bean.conf.loader.annotation.handler.AbstractPropertyValidationAnnotationHandler* are two examples of classes that implement this interface. They both depend on a *ConditionExpressionParser* to be used when the applicability conditions of the validation rules are parsed.

Function Expression Language (FEL)

FEL represents an expression language that defines functions. A function is a construct that accepts an object as an argument, evaluates it, and returns the evaluation result. The following is the Function interface:

```
public interface Function {

                    Object evaluate(Object argument) throws FelEvaluationException;

                    }
```

Note

All exceptions thrown by the different FEL constructs are unchecked exceptions.

As with CEL, FEL also provides an abstraction over the expression language. All of FEL constructs are located in the *org.springmodules.validation.util.fel* package and are listed bellow:

FunctionExpressionParser

This interface represents a parser that can parse a FEL expression and create a Function from it. Here is the complete interface:

```
public interface FunctionExpressionParser {

                    Function parse(String expression) throws FelParseException;

                    }
```

There are three concrete implementation of this parser that come with the framework:

- *ValangFunctionExpressionParser* - This parser can parse Valang function expressions.
- *OgnlFunctionExpressionParser* - This parser can parse Ognl expression that can be evaluated upon an object.
- *PropertyPathFunctionExpressionParser* - This parser is based on Spring's BeanWrapper and creates functions that can resolve property paths on the functions argument.

FunctionExpressionBased
This interface can be implemented any object that depends on FEL - specifically on a FunctionExpressionParser. Here is the complete interface:

```
public interface FunctionExpressionBased {

                    void setFunctionExpressionParser(FunctionExpressionParser functionExpressionParser);

                    }
```

The core constructs of the framework don't depend on FEL. Some default implementations however do make use of FEL. The *AbstractPropertyValidationAnnotationHandler* for example, uses FEL to represent the error arguments. That is, one can define error arguments within a validation rule annotation using function expression language. The handler will then parse these expressions, create functions, and evaluate them passing the property value as an argument.

17.3.2.6. The BeanValidator

Now that we know how to define the validation rules and how to load the validation configuration, it is time to put everything together and see how it can all be used to perform the actual validation.

Surprisingly or not, there's only one class that does this job - The *org.springmodules.validation.bean.BeanValidator* . This class is an implementation of Spring's *org.springframework.validation.Validator* interface. It is configured with the configuration loader we discussed above which is used to load all the validation rules for the validated objects. Here are a few facts to know about the bean validator:

- The BeanValidator support any class that its configuration loader supports.
- When the *validate(Object, Errors)* method is called on the BeanValiator The following occurs:
  - The validation configuration is loaded for the validated object based on its class.
  - The global validation rules are extracted from the configuration and applied on the validated object.
  - The property validation rules are extracted from the configuration and applied on the validated object.
  - The custom validators are extracted from the configuration and are applied on the object.
  - All the cascade validations are extracted from the configuration. This validation process is executed recursively for each cascade validation where the validated object is the cascaded property value. This is done while keeping the original validated object as the validation context to perform proper error registration. For example, consider a *Person* with an *address* property. If the *street* property of the *address* is invalid, the error code will be registered in the Errors object under the *person.address.street* property. This is because the original validation context was the person and not the address itself.

- The Bean Validator uses the *org.springmodules.validation.bean.converter.ErrorCodeConverter* abstraction to convert the default validation error codes to different ones. [TODO see...]
- The BeanValidator is just a spring validator, thus it can be used just like you'd use nay other validator.
- It is possible to use BeanValidator to validate multiple object of different types - for each object, different validation rules will be applied depending on its type and the validation configuration loader.
- The BeanValidator supports short-circuit validation process of properties. Short-circuiting means that if multiple validation rules are defined for a single property, the first rule to fail will stop the vlaidation process of that property, that is, all pending validation rules will not be executed. This support is enabled by default, but it is possible to disable it by calling *BeanValidator#setShortCircuitFieldValidation(false)*.

17.3.2.7. Application Context Configuration

In this section we'll walk through the steps needed to configure the validation framework within the application context.

There are two options when it comes to configuring the framework. You can use the traditional spring bean definitions in the application context or if spring 2.0 is used you can use the validation namespace to configure the framework in even more intuitive and easy manner.

Traditional Spring Bean Definition

All the constructs in the framework were designed as java beans, thus they can easily be configured in the application context as beans. Here is an example of a BeanValidator configuration using plain old spring bean tags:

```
<beans>

    <bean id="validator" class="BeanValidator">
        <property name="configurationLoader" ref="configurationLoader"/>
    </bean>

    <bean id="configurationLoader" class="DefaultXmlBeanValidationConfigurationLoader">
        <property name="resource" value="classpath:validation.xml"/>
    </bean>

</beans>
```

Note

The packages of the classes where dropped in the configuration for brevity.

The above example defines the bean validator "validator" which is configured with a configuration loader that is also defined in the application context. In this case, the loader is an xml configuration loader that loads the configuration rules from the *validation.xml* file in the classpath. (of course, the loader could have also be defined as an anonymous bean within the bean validator definition).

This is the simplest form of configuration. Of course, we might want to customize the defaults in the configuration. For example, lets say we would like to use valang as the CEL implementation and further more extend it with a custom function. Here is how we would configure it:

```
<beans>

    <bean id="validator" class="BeanValidator">
        <property name="configurationLoader" ref="configurationLoader"/>
    </bean>

    <bean id="configurationLoader" class="DefaultXmlBeanValidationConfigurationLoader">
        <property name="resource" value="classpath:validation.xml"/>
        <property name="conditionExpressionParser" ref="conditionExpressionParser"/>
        <property name="elementHandlerRegistry">
            <bean class="DefaultValidationRuleElementHandlerRegistry">
              <property name="conditionExpressionParser" ref="conditionExpressionParser"/>
            </bean>
        </property>
    </bean>

    <bean id="conditionExpressionParser" class="ValangConditionExpressionParser">
        <property name="customFunctions">
            <map>
                <entry key="tupper" value="UpperCaseFunction"/>
            </map>
        </property>
    </bean>

</beans>
```

As you can see, all classes are just normal java beans that are configured just like any other beans. Nonetheless, looking at the configuration above, it is clearly too verbose for such a small customization. First, you need to redefine the validation rule element handler registry just to introduce the new CEL parser. Then you need to set the new CEL parser twice (on the loader itself and the element handler). Luckily, a lot of beans in the framework are optimized to work within an application context. This optimization enables you to configure the above configuration as follows:

```
<beans>

    <bean id="validator" class="BeanValidator">
        <property name="configurationLoader" ref="configurationLoader"/>
    </bean>

    <bean id="configurationLoader" class="DefaultXmlBeanValidationConfigurationLoader">
        <property name="resource" value="classpath:validation.xml"/>
    </bean>

    <bean id="conditionExpressionParser" class="ValangConditionExpressionParser">
        <property name="customFunctions">
            <map>
                <entry key="tupper" value="UpperCaseFunction"/>
            </map>
        </property>
    </bean>

</beans>
```

In this configuration, the CEL parser implementation is picked up from the application context by all beans that depend on it. This form of configuration hides the complex wiring from the developer and enables focusing on the specific customization that is needed.

Note: FEL parser implementation can also be customized and configured in the same way.

Spring 2.0 Namespaces

The traditional bean configuration is definitely a viable option but if you're working with Spring 2.0 an even better one exists. Spring 2.0 introduced support for richer xml configuration via the use of XML namespaces. Using this support enables transforming the generic cross-domain spring configuration to a domain specific configuration (some sort of DSL) by using xml constructs (elements and attributes) that are specific per domain.

The validation framework comes with its own namespace making the configuration of its constructs even simpler, more readable, and more intuitive to work with. Here is the example above configured using the validation namespace:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:vld="http://www.springmodules.org/validation/bean/validator"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://.../beans http://.../spring-beans.xsd
    http://.../bean/validator http://.../bean/validator.xsd">

    <vld:valang-condition-parser>
        <vld:function name="tupper" class="UpperCaseFunction"/>
    </vld:valang-condition-parser>

    <vld:xml-based-validator id="validator">
        <vld:resource location="classpath:validation.xml"/>
    </vld:xml-based-validator>

</beans>
```

The *xml-based-validator* practically defines a *BeanValidator* that is configured with *DefaultXmlBeanValidationConfigurationLoader*. The *resource* element defines the xml configuration file where the validation rules are defined.

The *valang-condition-parser* practically defines a *ValangConditionExpressionParser*. The *function* sub element defines a custom valang function to be registered with the parser.

As you can see, this xml is much clearer than the generic spring one. It is very easy to read and understand it as a configuration. It is also much easier to write, especially when using schema supported editor/IDE's. The use of xml schema practically leads the developer in how the validator should and can be configured. The amount of code (text) is also reduced.

17.3.2.8. Contextual Validation

Sometimes a domain object may have different validation rules depending on the context in which it is being used. For example, it may be that certain validation rules of an object should be applied when the application is running in a *normal* mode, but only some of them should be applied when in *admin* mode. This type of validation is refered as contextual validation.

Having applicability conditions associated with the different validation rules certiainly enables support of contextual validation. It is also quite easy to extend the condition expression language that is used with functions to help check the current execution context. That said, from 0.8 release the framework supports contextual validation in its core.

ValidationContext & Validation Context Tokens

The *ValidationContext* interface stands in the center of this support:

```
public interface ValidationContext {

    boolean supportsTokens(String[] tokens);

}
```

This validation context is based on the context tokens notion. Validation context tokens are string tokens that are used to relate other constructs to specific contexts. The main (and only) method in this interface is the *supportTokens(String[])* method. This method can be called to determine whether or not certain tokens are supported by the context.

ValidationContextHolder

Attempting to adhere to spring's common practices and patterns, the validation context is accessible by calling one of the public static methods of the *ValidationContextHolder*:

```
public class ValidationContextHolder {

    ...

    public static ValidationContext getValidationContext() {
        ...
    }

    public static void setValidationContext(ValidationContext context) {
        ...
    }

    public static void clearContext() {
        ...
    }

    ...

}
```

This static method can be called from anywhere in the code in order to get,set, and clear the validation context.

Validation Context Demarcation

Like any other context (transactional, locale, request, etc...) the validation context need to be demarcated, that is, it should be defined where exactly the execution enters to and exits from a specific context. Normally, the demarcation is done using interceptors and the framework comes with three interceptors out of the box:

- *ValidationContextFilter* - This is a servlet filter that can be configured with validation context mappings. The mapping in paractice map validation context tokens to URL pattern.
- *ValidationContextHanlderInterceptor* - This is the spring mvc counter part of the ValidationContextFilter (It is also configured the same).
- *SimpleValidationContextInterceptor* - This is aop allians method interceptor that can be configured with a list of context tokens. (It extends AbstractValidationContextInterceptor which can easily be extended in order to further customize this support). In addition to the context setting, this interceptor also "remembers" the already existing validation context before the interception and sets it back when the execution is over. It is also possible to order the interceptor to extend the alreay existing validation context with more token as opposed to the default behavior where the context is completely reset for the duration of the execution.

Here are some examples of how to demarcate the validation context:

```
<filter>
    <filter-name>validationContextFilter</filter-name>
    <filter-class>..ValidationContextFilter</filter-class>
    <init-param>
        <param-name>validationContextUrlMappings</param-name>
        <param-value>
            /user/*=ctx1,ctx2
            /admin/*=ctx3
        </param-value>
    </init-param>
</filter>
```

```
<bean id="validationContextInterceptor" class="..ValidationContextHandlerInterceptor">
    <property name="validationContextUrlMappings">
        <value>
            /user/*=ctx1,ctx2
            /admin/*=ctx3
        </value>
    </property>
</bean>
```

```
<bean id="userValidationContextInterceptor" class="SimpleValidationContextInterceptor">
    <property name="contextMappings">
        <value>
            do.*=ctx1,ctx2
        </value>
    </property>
</bean>

<aop:config>
    <aop:pointcut id="serviceLayer" expression="execution(* .*Service.*(..))"/>
    <aop:advisor advice-ref="userValidationContextInterceptor" pointcut-ref="userValidationContext"/>
</aop:config>
```

Validation Rule Context Association

In order to associate the different validation rules with specific contexts, the rule's applicability condition was used. Further more, All rules that extend *AbstractValidationRule* now can be configured with specific validation context tokens that are associated with the rule. These tokens are then checked against the current validation context when the *isApplicable(Object)* method is called on the rule.

As far as the xml and annotation configuration is concerned, new attributes were added that accept a comma-separated list of context tokens to associate with the configured rule. For example:

```
@NotNull(contexts = "ctx1")
private String bla;
```

```
<property name="bla">
    <not-null contexts="ctx1"/>
</property>
```

17.3.3. Future Directions

The validation framework is quite new and its architecture and design is a subject to changes. The more experience we'll gain with this framework, the better we'll understand the demands of such a framework, and we hope to continuously improve it with time. Here some of the future planned features and improvements:

- *Valang* - alang was proven to be very powerful expression language when it comes to validation. Nonetheless, it was originally developed independently and somewhat in an experimental manner. Nowadays, we have a much better grasp of the requirements of this language and there are plans to improve it even more.
- *Client side support* - Both valang and common-validator support javascript client side validation. Adding such support to this framework is essential and is planned for one of the releases following 0.6.

We also keep a close eye on the recently submitted JSR-303. The following is taken from the original proposal:

> "This JSR will define a meta-data model and API for JavaBean validation. The default meta-data source will be annotations, with the abilty to override and extend the meta-data through the use of XML validation descriptors. It is expected that the common cases will be easily accomplished using the annotations, while more complex validations or context-aware validation configuration will be available in the XML validation descriptors.
>
> The validation API developed by this JSR will not be specific to any one tier or programming model. It will specifically not be tied to either the web tier or the persistence tier, and will be available for both server-side application programming, as well as rich client Swing application developers. This API is seen as a general extension to the JavaBeans object model, and as such

is expected to be used as a core component in other specifications, such as JSF, JPA, and Bean Binding."

—Jason Carreira

Chapter 18. Templates

18.1. Templates & Template Engines

Important

The template module was completely rewritten for 0.9 release. The old version of this module was actually experimental and should have never been released. As a consequence, that version will not be mantained or supported in any way from 0.9 on.

The Templates module provides an abstraction over common template engines. This abstraction mainly targets frameworks and products developers who don't wish to tie their implementation to a specific template engine.

18.1.1. Core Constructs & Interfaces

This abstraction is based on the following three core interfaces:

- Template - This interface represents the notion of a template that is capable of generating an output based on a given model.

```
public interface Template {

    void generate(OutputStream out, Map model) throws TemplateGenerationException;

    void generate(Writer writer, Map model) throws TemplateGenerationException;

    String generate(Map model) throws TemplateGenerationException;

}
```

  All Template's method essentially do the same thing - generate an output (whether by writing it to an output stream, writer, or just generate a string) based on a given model which is abstracted as a Map. Although declared, the *TemplateGenerationException* is a runtime exception (Actually, all exceptions thrown by all constructs are runtime exceptions). This exception is thrown when for some reason the template could not generate the desired output.

- TemplateEngine - This interface represents the a template engine that is capable of creating the appropriate template based on a template resource. As there are many different template engines out there and each one defines its own template language, a template generated by one engine is naturally different than the one generated by another engine.

```
public interface TemplateEngine {

    Template createTemplate(Resource resource) throws TemplateCreationException;

    Template createTemplate(Resource resource, String encoding) throws TemplateCreationException;

}
```

  As seen above, the template engine methods accept a Spring resource as an input (the template definition) and create the appropriate template from it. A runtime *TemplateCreationException* is thrown when for some reason the engine could not create a template from the given resource (perhaps bad template syntax).

- TemplateResolver - This is essentially a strategy for resolving templates based on a template names. For those who are familiar with Spring MVC, this resembles the ViewResolver.

```
public interface TemplateResolver {

    Template resolve(String name);

    Template resolve(String name, Locale locael);

    Template resolve(String name, String encoding);

    Template resolve(String name, String encoding, Locale locale);



}
```

As we'll see later on, this interface enables defining different strategies by which templates can be loaded/resolved which makes the development of a template base class simple and clean.

18.1.2. Supported Template Engines

18.1.2.1. Apache Velocity

The *org.springmodules.template.engine.velocity.VelocityTemplate* class is the Apache Velocity [http://velocity.apache.org/] implementation of the *TemplateEngine* interface. Under the hood it uses the VelocityEngine that can be configured by setting the appropriate Velocity settings using the *setConfiguration(Properties)* method. Here is an example how the engine can be configured in Spring application context:

```
<bean id="engine" class="org.springmodules.template.engine.velocity.VelocityTemplateEngine">
    <property name="defaultEncoding" value="UTF-8"/>
    <property name="configuration">
        <props>
            <!-- Velocity specific properties -->
            <prop key="velocimacro.context.localscope">false</prop>
        </props>
    </property>
</bean>
```

A common feature among template engines is support for inter-resource referencing. This feature enables the template author to modularize the structure of the templates and practice code reuse. Velocity support this feature using the *#include* and *#parse* directives. The *VelocityTemplateEngine* integrates tightly with Velocity resource management and leverages Spring's resource & resource loader support to resolve the referenced resources. This enables the template author to define the references in the Spring common resource paths. For example:

```
<html>
    <body>
        <div>#parse("classpath:header.vm")</div>
        <div>
            The content body
        </div>
        <div>#parse("classpath:footer.vm")</div>
    </body>
</html>
```

The *VelocityTemplateEngine* implements Spring's *ResourceLoaderAware* interface so the application context serves as the resource loader by default.

18.1.2.2. Freemarker

Freemarker [http://freemarker.sourceforge.net/] is a mature and feature rich template engine that is well designed and performs well. The *org.springmodules.template.engine.freemarker.FreemarkerTemplateEngine* class is capable of reading FreeMarker template sources and generate the appropriate FreeMarker templates. The following is an example of application context configuration:

```
<bean id="freemarker" class="org.springmodules.template.engine.freemarker.FreemarkerTemplateEngine">
    <property name="defaultEncoding" value="UTF-8"/>
    <property name="settings">
        <props>
            <prop key="locale">en_US</prop>
        </props>
    </property>
    <property name="sharedVariables">
        <map>
            <entry key="name" value="${name}"/>
        </map>
    </property>
</bean>
```

As shown above, it is also possible to define variables that will be shared among all templates. In this case the shared variable is configured using a property placeholder.

Like with Velocity, *FreemarkerTemplateEngine* also leverages Spring's resource support to support inter-resource referencing. This enables writing a template like:

```
<html>
    <body>
        <div><#include "classpath:header.html" parse=false></div>
        <div>
            The content body
        </div>
        <div><#include "classpath:footer.ftl"></div>
    </body>
</html>
```

18.1.2.3. Groovy Template

Groovy Template [http://groovy.codehaus.org/Groovy+Templates] is a library that ships with groovy which enables writing templates in the groovy language (GString's for those familiar with groovy). While current implementation performance is somewhat lagging behind Velocity and Freemarker, groovy as a template language brings quite a lot of power to the hands of template authors. The *org.springmodules.template.engine.groovy.GroovyTemplateEngine* is the template engine that create Groovy templates. Here is how this engine would be configured in an application context:

```
<bean id="groovy" class="org.springmodules.template.engine.groovy.GroovyTemplateEngine">
    <property name="defaultEncoding" value="UTF-8"/>
</bean>
```

As seen above, there isn't much to configure when defining a groovy template engine. This engine is much more simplistic and less featured than both Velocity and Freemarker (for example, it has no support inter-template referencing).

18.1.3. Template Resolvers

We've seen how we can create/define the template engines that enables the creation of template of different types. However, in order to utilize them to create those actual templates, one needs to first load the appropriate template resource and hand it over to the engine. Although it is perfectly valid approach, template resolvers come to simplify this process even more.

The following are the different types of template resolvers that are provided out-of-the-box:

18.1.3.1. BasicTemplateResolver

As the name suggests, this is a very basic implementation of the *TemplateResolver* interface. It is configured with a template engine and a resource loader. The resource loader is used to load the template resources based on the template name, while the template engine is used to create the templates from the loaded template resources. In the simplest case, the resolver's *resolve(String)* calls the resource loader's *getResource(String)* with the provided template name. This resolver can also (recommended) be configured with an extension. The extension is used to help when resolving using a locale. If no extension is set, the resolver will figure out the extension from the given template name (basically, taking treating the susbstring from the last '.' in the name as the extension). Furthermore, it is also possible to tell the resolver to resolve the locale from the current *org.springframework.context.i18n.LocaleContext* when the locale is not supplied by the client (or when the client supplies *null* as the locale). Here is how this resolver can be defined within an application context:

```
<bean id="resolver" class="org.springmodules.template.resolver.BasicTemplateResolver">
    <property name="engine" ref="velocity"/>
    <property name="extension" value=".vm"/>
    <property name="resolveLocalFromContextWhenAbsent" value="true"/>
</bean>
```

The *BasicTemplateResolver* also implements the *ResourceLoaderAware* interface, so when defined in the application context, by default the application context is used as the resource loader implementation.

Most of the time you wouldn't choose to use this implementation in your application. As we'll see soon, this class mainly serves as a base class for the other template resolver implementations.

18.1.3.2. CachingTemplateResolver

This is a simple extension on top of the *BasicTemplateResolver* which caches the resolved templates by their names. Calling *resolve(String)* on this resolver for the first time, will go through the same process as with the *BasicTemplateResolver* but with one difference, just before the resolved template is returned, it is cached internally by its name. As a consequence, the next call for the same template (using the same template name) will just return the cached template.

Although for most cases using a cached template resolver is the preferred choice, there are some things to consider. First, the cache effectiveness high depends on the concrete implementation of the template that is used. For example, it could be that a template is implemented in such a way that every call to *generate(Writer, Map)* the template resource is being re-read. Another thing to consider is that some engines has a built-in support for automatic reload (that is, when the template source changes, the template is reloaded). For some implementations, caching the template will prevent from the reload to actually work. Hopefully, in a future release we'll have our own support for such reload functionality.

18.1.3.3. SimpleTemplateResolver

Again, as its name suggests, this is a simple template resolver that extends the caching resolver. It is very likely that you will use this resolver implementation most of the time. In addition to all functionality this resolver inherits from the caching and basic resolver, it one additional features. With this resolver it is possible to define a prefix that when combined with the template name, is used to create the template resource path. Here's how this resolver can be configured within an application context:

```
<bean id="simpleResolver" class="org.springmodules.template.resolver.SimpleTemplateResolver">
    <property name="engine" ref="freemarker"/>
    <property name="prefix" value="classpath:/templates/"/>
    <property name="extension" value=".ftl"/>
</bean>
```

In the example above, when trying to resolve a template named "email", the resolver will try to load the template resource from the "classpath:/templates/email.ftl" location, or when the locale is provided the resolver will try one of the following first: classpath:/templates/email_en_US.ftl" and "classpath:/templates/email_en.ftl". This is quite powerful mechanism as it lets you define all template resources in a well defined and proprietary location, while still the user only needs to deal with the logical name of the template (i.e. "email").

18.2. Email Framework

Spring comes with an out-of-the-box support for emailing services. The main goal of this support is to provide a higher abstraction of this type of service and hide all the complexities when dealing with the *javax.mail* and *java.activation* API.

While this abstraction succeeds in what it was set up to do, it only provides a new (simpler) set of API's for the developer to work with. When it comes to integrating the email service in an enterprise application, a quite common approach is to write as set of templates (using Velocity or FreeMarker) to represent the email body, so instead of having the content of the email hardcoded in java, one can customize this content on demand without changing the codebase. While this approach is better than the pure programmatic one, it's only a partial solution to the problem as the developer still needs to hard code the other properties of the email such as subject, sender, recipients, and more.

This email framework tries to answer all these issues by introducing an XML based email descriptor which can serve as a template for creating email definitions. This email definitions are then handled appropriately by leveraging Spring's mail services.

18.2.1. Core Constructs & Interfaces

- *Email* - A simple POJO representing an email message. As opposed to all Spring's *MailMessage* implementation, this POJO tries to capture all possible data of a mail message as bean properties including normal and embedded (inline) attachements.
- *EmailDispatcher* - An interface used to send Email messages. Can be seen as Spring's *MailSender* counterpart which works with *Email* objects.
- *EmailParser* - Represents a parser that can read email descriptors, parse them, and create Email objects from them.

18.2.2. EML files - Email Descriptor

The email descriptor is a simple XML source describing an email. It holds most (if not all) data associated with an email, making it possible to keep all email definitions outside of the code base. Below you can find a sample email descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<email xmlns="http://www.springmodules.org/schema/template/email-1.0">

    <from name="John Doe">john@doe.org</from> ❶

    <reply-to name="No Reply">noreply@doe.org</reply-to> ❷

    <to> ❸
        <address name="Client">client@foo.com</address> ❹
    </to>
    <cc> ❺
        <address name="Managing Director">md@doe.org</address>
        <address name="VP Marketing">vpm@doe.org</address>
    </cc>

    <bcc> ❻
        <address>anonymous@doe.org</address>
    </bcc>

    <priority>HIGH</priority> ❼

    <subject>The subject of the mail</subject> ❽

    <text-body> ❾
        This is the text based body of the email
    </text-body>

    <html-body> ❿
        <![CDATA[
            <html><body>
                <div><img src="cid:logo"/></div>
                <div>This is the HTML based body of the email</div>
            </body></html>
        ]]>
    </html-body>

    <attachments> 11
        <attachment name="Marketing.doc">classpath:/documents/marketing.doc</attachment> 12
    </attachments>

    <inline-attachments> 13
        <attachment name="logo">classpath:/images/logo.gif</attachment> 14
    </inline-attachments>

</email>
```

❶ Specifying the sender's email address. The "name" attribute is optional and can be used to define the personal name of the sender.
❷ An optional element specifying the address to which the recipients should reply
❸ Holds a list of primary recipient addresses to which this email is sent
❹ Defines an address. The optional "name" attribute can be used to define a personal name for the address
❺ Holds a list of addresses to which this email is will be cc'ed
❻ Holds a list of addresses to which this email is will be bcc'ed
❼ Optional element to specify the priority of the email message. The priority can be on of 5 options: LOWEST, LOW, NORMAL, HIGH, and HIGHEST. By default the priority is set to NORMAL.
❽ Specifies the subject of the email
❾ Specifies the plain text body of the email
❿ Specifies the html body of the email.
Holds a list of resources to be attached to the email as normal attachments.
Defines an attachment where the name serves as the file name and the value specifies the location of the resource. As we'll later see, the location path is determined by the *ResourceLoader* implementation that is used by the *EmailParser*. Most commonly the *DefaultResourceLoader*.
Holds a list of resources to be attached to the email as inline (embedded) attachments.
Defines an inline attachment where the "name" serves as the id of the attachment by which it can be referenced in the body of the content (for example, assuming the name is set to "001" , the html body can contain the following image element <img src="cid:theId""/>). Here too the value of the element specifies the location of the resource. As we'll later see, the location path is determined by the *ResourceLoader* implementation that is used by the *EmailParser*. Most commonly the *DefaultResourceLoader*.

This XML based email descriptor is the only descriptor that the framework is delivered with. Nonetheless, it is possible to define other descriptors as well. The *EmailParser* interface enables loading an email descriptor resource into an Email object. The default implementation is *SaxEmailParser* with parses the XML based descriptor.

18.2.3. Email Dispatcher

The email dispatcher is the heart of this framework. This interface enables sending emails. Here is the *EmailDispatcher* interface:

```
public interface EmailDispatcher {

    void send(Email email); ❶

    void send(String emailName); ❷

    void send(String emailName, Locale locale); ❸

    void send(String emailName, Map model); ❹

    void send(String emailName, Map model, Locale locale); ❺

    void send(String emailName, Map model, EmailPreparator preparator); ❻

    void send(String emailName, Map model, Locale locale, EmailPreparator preparator); ❼

}
```

❶ Sends the given email
❷ Resolves and sends the email identified by the given name.
❸ Resolves and sends the email identified by the given name and locale.
❹ Resolves the email identified by the given name, populating it using the given model and sends it.
❺ Resolves and sends the email identified by the given name and locale. The resolved email is populated by the given given model.
❻ Resolves the email identified by the given name, populating it using the given model, preparing it using the given email preparator and sends it.
❼ Resolves and sends the email identified by the given name and locale. The resolved email is populated by the given model and is prepared by the given preparator prior to dispatch.

As can be seen, there are essentially two ways of sending an email. A direct way which mean the client of this interface provides the email to be sent. The other way is more implicit way where the client requests the dispatcher to send an email identified by its name and possibly by a specific locale.

AbstractEmailDispatcher is a base class for all concrete implementation of this interface. The abstract class takes care of resolving and preparing the emails. It does this by first loading the appropriate template for the given email, generating the output using the given model, The output is expected to be an email descriptor and so an *EmailParser* is used to parse it to a concrete *Email* object. And last but not least, if necessary, preparing the result email using an *EmailPreparator*. In this implementation, the email name essentially serves as an email template name that will eventually be generating the actual emails descriptors.

There are four concrete *EmailDispatcher* implementations:

- *ConfigurableEmailDispatcher* - This dispatcher is configured with an *EmailSender* which takes care of the email dispatching task. There are two implmentations of *EmailSender*: *JavaMailEmailSender* and *SimpleEmailSender*. Most of the time, you won't be using this dispatcher but instead use one of the following two.

- *JavaMailEmailDispatcher* - This dispatcher should be used when configured with a *JavaMailSender* as the underlying *MailSender*. It is also possible to configured the *JavaMailSender* properties (e.g. "host", "port", "protocol", etc..) directly on the dispatcher and let it create a new instance of *JavaMailSenderImpl* internally.

- *SimpleEmailDispatcher* - This dispatcher should be used when configured with other *MailSender* implementations such as *CosMailSenderImpl*. As *MailSender* works with *SimpleMailMessage*, this dispatcher makes sure that only the supported information is extracted from the provided *Email* objects. For example, the attachments in the provided emails will be ignored since *SimpleMailMessage* does not support attachments of any kind.
- *AysncEmailDispatcher* - Sometimes you would like to send the emails in an asynchronous fashion. This dispacher enables exactly that by leveraging Spring's *TaskExecutor* abstraction. The asynchronous behavior is determined by the configured *TaskExecutor*. It is also possible to configure a *DispatchingErrorHandler* to properly handle any error that occur during an email dispatch, and *DispatchingCallback* that will be called after the email was dispatched (whether successfully or not). By default a *LoggingDispatchingErrorHandler* and an *EmptyDispatchingCallback* are set. The logging handler will log any error using Apache Commons Logging logger while the empty callback will do nothing.

Note

It is possible to still define a synchronous behaviour using the *AsyncEmailDispatcher* by configuring a *SyncTaskExecutor* as the task executor, and for handling the dispatching errors differently a *RethrowingDispatchingErrorHandler* can be set (which will rethrow all occured errors which might make sense when using the *SyncTaskExecutor*).

Here's an example of how the *JavaMailEmailDispatcher* can be configured in an application context:

```
<bean id="emailDispatcher" class="org.springmodules.email.dispatcher.JavaMailEmailDispatcher">
    <property name="templateResolver" ref="templateResolver"/>
    <property name="host" value="${mail.host}"/>
    <property name="port" value="${mai.port}"/>

</bean>

<bean id="templateResolver" class="org.springmodules.template.resolver.SimpleTemplateResolver">
    <property name="engine" ref="freemarker"/>
    <property name="prefix" value="classpath:/emails/"/>
    <property name="extension" value=".eml"/>
</bean>

<bean id="freemarker" class="org.springmodules.template.engine.freemarker.FreemarkerTemplateEngine"/>
```

18.2.3.1. Preparing emails for dispatch

As mentioned above, one can prepare the email before it is being dispatched by the dispatcher. This can be done by passing an *EmailPreparator* to the *send* method. Here is the *EmailPreparator* interface:

```
public interface EmailPreparator {

    Email prepare(Email email);

}
```

The *prepare* method accepts the email to prepare (this method will be called by the dispatcher just before sending the email) and returns the prepared email. Alghough, most of the time implementation of this method will return the same email instance that is passed in, it is perfectly possible to return a different email instance (We'll see an example shortly).

There are some preparators that are already implemented which can be used out-of-the-box:

- *AttachmentEmailPreparator* - This preparator enables adding an attachment to an existing email. It can be useful for example when there's a need to attach runtime generated reports to an email (for example PDF reports generated by JasperReports).

- *InlineAttachmentEmailPreparator* - This preparator enables embedding an attachment in the body of the (html) message. This can be useful for example when there's a need to display an image of a runtime generated graph in the body of the email.
- *IdentifiableEmailPreparator* - This is an example of a preparator that actually replaces the passed in email with another one. This preparator wraps the passed in email with an IdentifiableEmail and by doing that, associates an id with the email. A scenario where you'd like to do that is when the system keeps all emails in some storage, and an asynchronous dispather is used to send them. When an email is sent it should be removed from the storage. In this case, one can wrap the dispached emails with an IdentifiableEmail where the id represents the identity of the email in the storage. A special *DispatchingCallback* can then be registered with the async. dispatcher to remove the sent email if the dispatch was successful.
- *ChainEmailPreparator* - This is a compound email preparator which holds a chain of other email preparators. When asked to prepare an email, this preparator iterates over the preparator chain so that each preparator in the chain prepares the email. This preparator can bue used to easily combine multiple preparators into one.

Chapter 19. XT Framework

19.1. About XT Framework

The XT Framework is a *Spring* [http://www.springframework.org] module for developing applications with "richer domain models and richer user interfaces", following the *Domain Driven Design* [http://domaindrivendesign.org] practices.

Domain Driven Design puts the *domain model* [http://martinfowler.com/eaaCatalog/domainModel.html] in the heart of the development process. The domain model becomes the core of your application and the main focus of your development. It concentrates all business logic, rules and constraints, being **totally independent** from other application parts. In particular, considering a standard layered architecture, domain models are designed to be independent from the so called *presentation layer* and *infrastructure/data access layer*. Unfortunately, practically speaking, this causes a sort of *impedence mismatch* among layers, causing code duplication, domain model corruption, or the infamous *anemic domain model* [http://www.martinfowler.com/bliki/AnemicDomainModel.html] phenomenon.

The XT Framework aims at solving this mismatch providing the so called **XT Modeling Framework** for developing *rich domain models*, which gracefully adapt to other layers.

And if you have a rich domain model, why not having a *rich user interface*? The XT Framework provides also the **XT Ajax Framework**, which integrates Spring and its MVC framework with *AJAX* [http://adaptivepath.com/publications/essays/archives/000385.php] technologies.

19.1.1. XT Modeling Framework

XT Modeling Framework provides components for helping in developing rich domain models and making them collaborate with other application layers without violating DDD principles. At the moment it provides:

- **Bean Introductor** : Built on top of Spring AOP, it introduces JavaBeans style interfaces with getter/setter methods into domain objects, for adapting/decorating them.
- **Implementor Introductor** : Built on top of Spring AOP, it introduces additional interfaces to already existent object, and automatically delegates the implementation of those interfaces to an external, independent, object.
- **Introductor Collections** : Standard Java collections decorators for introducing interfaces into objects stored in collections, by using the introductor implementations above.
- **Dynamic Generator** : Generic interface for dynamically generating Java objects.
- **Dynamic Factory Generator** : Dynamic Generator that let you generate factory objects on the fly, providing only the factory interface.
- **Notifications** : *Notification* [http://www.martinfowler.com/eaaDev/Notification.html] objects, for carrying messages between different application layers.
- **Specifications** : Generic interface for implementing *Specifications* [http://www.martinfowler.com/apsupp/spec.pdf] capable of collecting messages of different types into notification objects

- **Composite Specifications** : Specifications implementation capable of being combined through logical operators, using a *fluent interface* [http://www.martinfowler.com/bliki/FluentInterface.html].
- **Events** : Additional facilities for implementing event-based architectures built on the standard Spring events system: events filtering, events collection and Event Driven POJOs.

If you want to learn more, take a look at XT Modeling Framework base concepts.

19.1.2. XT Ajax Framework

XT Ajax Framework is an event based Ajax framework fully integrated with Spring MVC. Here are some of its major features:

- No javascript code required, nor special template tags, just plain old HTML: we believe in Separation of Concerns and web pages with no extra logic.
- Based on web page level events fired as *javascript events* [http://www.w3schools.com/js/js_events.asp].
- Pure server side handling of page events.
- Ready to use actions and HTML components for page updating.
- Dramatically simplified handling of complex HTML forms logic.
- Only minor changes required for adapting your existing Spring MVC controllers and web pages.
- Secure.

If you want to learn more, take a look at XT Ajax Framework base concepts.

19.1.3. JVM Requirements

The XT Frameworks is based on Java 1.5.

However, since version 0.9, it is distributed with two jar archives backported for working in Java 1.4 environments.

19.2. XT Modeling Framework

19.2.1. Introduction

XT Modeling Framework provides several facilities for constructing rich domain models which gracefully collaborate and interact with other layers and components in your application. It comprises different parts we'll illustrate in the section below.

19.2.2. Base Concepts

19.2.2.1. Introductor

The XT *Introductor* permits you to make existent objects implementing additional interfaces defined at runtime, that is, it *introduces* new interfaces into already existent Plain Old Java Objects. This is achieved through the use of *Spring AOP* [http://www.springframework.org/documentation] *Introduction*, for creating proxy objects that behave like the original one but also implement new interfaces.

19.2.2.1.1. DynamicIntroductor

The core interface is the **DynamicIntroductor** (see *org.springmodules.xt.model.introductor.DynamicIntroductor* javadoc):

```
public interface DynamicIntroductor {

    public Object introduceInterfaces(Object target, Class[] introducedInterfaces);

    public Object introduceInterfaces(Object target, Class[] introducedInterfaces,
                                      Class[] targetInterfaces);

    public Object getTarget(Object proxy);
}
```

- *public Object introduceInterfaces(Object target, Class[] introducedInterfaces)* : Introduce to the given target object a set of interfaces defined at runtime. The resulting proxy object **extends** the target object class.
- *public Object introduceInterfaces(Object target, Class[] introducedInterfaces, Class[] targetInterfaces)* : Introduce to the given target object a set of interfaces defined at runtime. The resulting proxy object **implements** a set of target interfaces defined at runtime (without extending the target object).
- *public Object getTarget(Object proxy)* : Get the original target object from the proxied (introduced) one.

19.2.2.1.2. DynamicBeanIntroductor

The **DynamicBeanIntroductor** (see *org.springmodules.xt.model.introductor.bean.DynamicBeanIntroductor* javadoc) is a concrete implementation for introducing *JavaBeans-style* interfaces with getter/setter methods into already existent **target** objects. You can introduce any interface **without** manually implementing it, because its getter/setter methods will be automatically implemented; the only **constraints** to follow when using the DynamicBeanIntroductor are to not introduce interfaces with methods different than getter or setters, and to not use primitive types as getter return types, using instead their object wrappers.

For example, take this Employee object:

```
public class Employee implements IEmployee {

    private String matriculationCode;
    private String firstname;
    private String surname;

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }
    public String getMatriculationCode() {
        return matriculationCode;
    }

    public void setMatriculationCode(String matriculationCode) {
        this.matriculationCode = matriculationCode;
    }
}
```

You can automatically introduce the following interface:

```
public interface EmployeeView {

    void setOffice(Office office);

    Office getOffice();

    void setSelected(Boolean selected);

    Boolean isSelected();
}
```

By simply doing:

```
IEmployee employee = new Employee();

DynamicBeanIntroductor introductor =  new DynamicBeanIntroductor();

Object newEmployee = introductor.introduceInterfaces(employee,
                                                     new Class[]{EmployeeView.class});
```

You can now use your brand new object:

```
IEmployee employee = (IEmployee) newEmployee;
EmployeeView view = (EmployeeView) newEmployee;

employee.setMatriculationCode(...);
view.setSelected(...);
```

When calling methods on an object created through the DynamicBeanIntroductor, the following happens:

- Every call to a getter/setter method declared into the introduced interface is delegated to the target object if this declares and implements a public method with equal signature, otherwise it is automatically implemented by the proxy.
- Every call to a non getter/setter method declared into the introduced interface throws an exception.
- Every call to a method not declared into the introduced interface is directly delegated to the target object.

Moreover, the DynamicBeanIntroductor behaviour can be customized: see DynamicIntroductor annotations.

19.2.2.1.3. DynamicImplementorIntroductor

The **DynamicImplementorIntroductor** (see *org.springmodules.xt.model.introductor.implementor.DynamicImplementorIntroductor* javadoc) is a concrete implementation for introducing additional interfaces into already existent **target** objects and delegate the implementation of those interfaces to an external Java object called **implementor**.

For example, take this Manager object:

```
public class Manager implements IManager {

    private List<IEmployee> managedEmployees;
    private String matriculationCode;

    public void setManagedEmployees(List<IEmployee> employees) {
        this.managedEmployees = employees;
    }

    public List<IEmployee> getManagedEmployees() {
        return this.managedEmployees;
    }
}
```

You can introduce the following interface:

```
public interface IEmployee {

    String getMatriculationCode();

    String getFirstname();

    String getSurname();

    void setMatriculationCode(String matriculationCode);

    void setFirstname(String firstname);

    void setSurname(String surname);
}
```

With the following implementation:

```
public class Employee implements IEmployee {

    private String matriculationCode;
    private String firstname;
    private String surname;

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public String getMatriculationCode() {
        return matriculationCode;
    }

    public void setMatriculationCode(String matriculationCode) {
        this.matriculationCode = matriculationCode;
    }
}
```

By simply doing:

```
IManager manager = new Manager();
IEmployee employee = new Employee();
DynamicImplementorIntroductor introductor  = new DynamicImplementorIntroductor(employee); // (1)

Object newManager = introductor.introduceInterfaces(manager,
                                                    new Class[]{IEmployee.class}); // (2)
```

You have to simply construct the *DynamicImplementorIntroductor* object (1) by passing it the *implementor* object, that is, the object that will implement the interfaces to introduce. Then, you have to actually introduce the interfaces into the *target* object (2).

You can now use your brand new object:

```
IManager manager = (IManager) newManager;
IEmployee employee = (IEmployee) newManager;

manager.setManagedEmployees(...);
employee.setMatriculationCode(...);
```

When calling methods on an object created through the DynamicImplementorIntroductor, the following happens:

- Every call to a method declared into an introduced interface will be delegated to the **implementor** object if it implements the introduced interface; otherwise, an exception occurs.
- Every call to a method not declared into the introduced interface is as always executed by the **target** object.

19.2.2.1.4. Introductor Collections

Introductor Collections (see *org.springmodules.xt.model.introductor.collections* javadocs) are *Java Collections* [http://java.sun.com/docs/books/tutorial/collections/index.html] decorators that apply Dynamic Introductors to contained objects. You can decorate a Java *Collection* [http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html], *Set* [http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html], or *List* [http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html]. Here is how you construct an Introductor Collection:

```
public IntroductorCollection(Collection target, Class[] introducedInterfaces,
                             Class[] targetObjectsInterfaces, DynamicIntroductor introductor);

public IntroductorCollection(Collection target, Class[] interfaces,
                             DynamicIntroductor introductor);

public IntroductorSet(Collection target, Class[] introducedInterfaces,
                      Class[] targetObjectsInterfaces, DynamicIntroductor introductor);

public IntroductorSet(Collection target, Class[] interfaces,
                      DynamicIntroductor introductor);

public IntroductorList(Collection target, Class[] introducedInterfaces,
                       Class[] targetObjectsInterfaces, DynamicIntroductor introductor);

public IntroductorList(Collection target, Class[] interfaces,
                       DynamicIntroductor introductor);
```

You can specify the following constructor parameters:

- *Collection target* : The target collection to decorate
- *Class[] introducedInterfaces* : The interfaces to introduce into every contained object.
- *Class[] targetObjectsInterfaces* : The interfaces of the target object that the proxy object must implement.
- *DynamicIntroductor introductor* : The Dynamic Introductor to use.

You can safely add and remove objects to and from Introductor Collections like they were normal collections: all changes are backed by the original non-decorated collection. Every object you get from an Introductor Collection implements a given set of interfaces using a particular, concrete, Dynamic Introductor, but **note** that objects stored in the original collection **are not modified**.

19.2.2.2. Generator

The XT *Generator* let you generate Java objects starting from simple Java interfaces.

19.2.2.2.1. DynamicGenerator

The core interface is the **DynamicGenerator** (see *org.springmodules.xt.model.generator.DynamicGenerator* javadoc):

```
public interface DynamicGenerator<T> {

    public T generate();
}
```

- *public T generate()* : Generate an object of generic type *T*.

Given its simple and generic interface, the DynamicGenerator can be used to generate all kind of objects.

19.2.2.2.2. DynamicFactoryGenerator

The **DynamicFactoryGenerator** (see *org.springmodules.xt.model.generator.factory.DynamicFactoryGenerator* javadoc) is a concrete implementation for generating factory objects by simply providing a plain Java interface with a number of setter methods, representing constructor arguments and properties to use for constructing and assembling the *product object*, and a factory method returning the class (or superclass) of that object.

Going into details, the factory interface methods must obey the following rules:

- **Setter** methods must be annotated either with:
    - The ConstructorArg (see *org.springmodules.xt.model.generator.annotation.ConstructorArg*) annotation if they are constructor arguments, plus the optional ConstructorArgType (see *org.springmodules.xt.model.generator.annotation.ConstructorArgType*) annotation.
    - The Property (see *org.springmodules.xt.model.generator.annotation.Property*) annotation if they are properties to be set.
    - The Value (see *org.springmodules.xt.model.generator.annotation.Value*) annotation if they are just simple values to use later for other purposes.

- The **factory** method must be annotated with the **FactoryMethod** (see *org.springmodules.xt.model.generator.annotation.FactoryMethod* javadoc) annotation.
- Moreover, there can be **getter** methods corresponding to the setter methods above.

Any other method, if called, will throw an exception.

So here is a simple annotated interface:

```
public interface EmployeeFactory {

    // Setter methods must be annotated because they represent constructor arguments or
    // just object properties:

    @ConstructorArg(position = 0)
    void setNickname(String nickname);

    @ConstructorArg(position = 1)
    void setMatriculationCode(String matriculationCode);

    @Property(access = Property.AccessType.FIELD)
    void setFirstname(String firstname);

    @Property()
    void setSurname(String surname);

    // Getter methods must not be annotated:

    String getNickname();

    String getMatriculationCode();

    String getFirstname();

    String getSurname();

    // The factory method must be annotated in order to make it clear its purpose:

    @FactoryMethod()
    IEmployee make();
}
```

And here is how to use it with the DynamicFactoryGenerator:

```
// Generate an EmployeeFactory for creating Employees:
DynamicFactoryGenerator<EmployeeFactory, Employee> generator =
    new DynamicFactoryGenerator(EmployeeFactory.class, Employee.class);

// Actually generates the factory:
EmployeeFactory factory = generator.generate();

// Set values for constructing and assembling the employee:
factory.setNickname("SB");
factory.setMatriculationCode("111");
factory.setFirstname("Sergio");
factory.setSurname("Bossa");

// Actually creates the employee:
IEmployee emp = factory.make();
```

Note

The DynamicFactoryGenerator generates a new factory object for every call to its *generate* method, and the factory generates a new product object for every call to its factory method.

19.2.2.3. Notifications

Notifications [http://www.martinfowler.com/eaaDev/Notification.html] are lightweight objects capable of collecting and carrying messages between different application layers, even if most of the time they will carry messages toward the presentation layer.

XT Notifications implementation is based on a *Message* interface and a *Notification* interface.

19.2.2.3.1. The Message Interface

The Message interface looks like this:

```
public interface Message {

    public enum Type { ERROR, WARNING, INFO };

    public String getCode();

    public Message.Type getType();

    public String getPropertyName();

    public String getDefaultMessage();
}
```

A message object can hold the following information:

- A message type: *ERROR*, *WARNING* or *INFO*.
- A message code.
- A default message text.
- The name of an object property this message refers to, used for example when messages refer to validation errors.

19.2.2.3.2. The Notification Interface

The Notification interface looks like this:

```
public interface Notification {

    public void addMessage(Message message);

    public boolean removeMessage(Message message);

    public Message[] getMessages(Message.Type type);

    public boolean hasMessages(Message.Type type);

    public Message[] getAllMessages();

    public boolean hasMessages();

    public void addAllMessages(Notification notification);
}
```

It is indeed straightforward: you can add and remove Message objects, and ask for Messages of a given type.

Notifications are best used together with Specifications.

19.2.2.4. Specifications

Specifications [http://www.martinfowler.com/apsupp/spec.pdf] are predicate-like objects for powerfully expressing all kind of business/validation/selection rules and matching business objects against them.

Every Specification implementation must have a boolean method for verifying if a business object satisfies the Specification rules.

XT Specification implementation is based on the following interface:

```
public interface Specification<O> {

    public boolean evaluate(O object);

    public boolean evaluate(O object, Notification notification);

    public void addMessage(Message message, boolean whenSatisfied);

    public boolean removeMessage(Message message, boolean whenSatisfied);
}
```

The Specification has one generic type, *O*, representing the object type you want to evaluate.

As you may note, you can specify what Messages will be thrown when the specification is satisfied or unsatisfied, and collect them into a Notification object at evaluation time.

19.2.2.4.1. Composite Specifications

Composite Specifications let you compose your own specification objects through logical operators and define what Messages will be issued by what specifications when satisfied or unsatisfied, all using a nice *fluent interface* [http://www.martinfowler.com/bliki/FluentInterface.html].

Other than this, the main advantage of using the Composite Specification is that you can combine your own specifications whatever their class is, without modifying one line of your code.

Here is the CompositeSpecification interface:

```
public interface CompositeSpecification<S, O> extends Specification<O> {

    public CompositeSpecification and(CompositeSpecification<S, O> specification);

    public CompositeSpecification and(S specification);

    public CompositeSpecification andNot(CompositeSpecification<S, O> specification);

    public CompositeSpecification andNot(S specification);

    public CompositeSpecification compose(S specification);

    public CompositeSpecification withMessage(Message message, boolean whenSatisfied);

    public boolean evaluate(O object);

    public CompositeSpecification or(CompositeSpecification<S, O> specification);

    public CompositeSpecification or(S specification);

    public CompositeSpecification orNot(CompositeSpecification<S, O> specification);

    public CompositeSpecification orNot(S specification);
}
```

The CompositeSpecification has two generic types, *S* and *O*: the former represents the type of the specification you want to compose, the latter the type of the object you want to evaluate.

Some code is worth a lot of words, so for taking a glimpse on how powerful a CompositeSpecification can be, say you have implemented these two specifications in your own application (not depending on the XT library):

```
public class OfficeIdSpecification implements BaseSpecification<IOffice> {

    public boolean isSatisfiedBy(IOffice o) {
        return o.getOfficeId().matches("\\d+") || o.getOfficeId().matches("o\\d+");
    }

}
```

```
public class FullOfficeSpecification implements BaseSpecification<IOffice> {

    private int limit = 3;

    public boolean isSatisfiedBy(IOffice o) {
        return o.getEmployees().size() > this.limit;
    }

}
```

You can powerfully compose them in a composite specification by simply writing the following:

```
// Create your business-specific specifications:
OfficeIdSpecification idSpecification = new OfficeIdSpecification();
FullOfficeSpecification fullOfficeSpecification = new FullOfficeSpecification();

// Create a CompositeSpecification for composing BaseSpecification specifications (your base class):
CompositeSpecification<BaseSpecification, IOffice> compositeSpecification =
    new CompositeSpecificationImpl(BaseSpecification.class, "isSatisfiedBy");

// Create an error message for the wrong id:
Message wrongIdMessage = new MessageImpl(OfficeErrorCodes.WRONG_ID,
    Message.Type.ERROR,
    "officeId",
    "Wrong office id");

// Create an error message for the full office:
Message fullOfficeMessage = new MessageImpl(OfficeErrorCodes.FULL,
Message.Type.ERROR,
    "employees",
    "Too many employees");

// Compose all:
compositeSpecification.compose(idSpecification).withMessage(wrongIdMessage, false)
    .andNot(fullOfficeSpecification).withMessage(fullOfficeMessage, true);
```

This will compose a specification matching an office that has a correct id and is not full.

19.2.2.5. Events

Introducing events into your application architecture is proven to be an effective technique for:

- Decoupling otherwise intricated application modules, letting them communicate through proper events.
- Enabling, in a typical layered application architecture, lower layers communication towards higher layers; i.e., the communication from the domain layer to the presentation one.
- Multicasting system changes to an unknown number of interested listeners.
- Recording system changes in order to track/undo/correct them.

The XT Modeling Framework provides several facilities for implementing an event-based architecture in your application. It is built on the standard Spring events system [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#context-functionality-events], which provides the following base components:

- *org.springframework.context.event.ApplicationEventMulticaster*.
- *org.springframework.context.ApplicationListener*.
- *org.springframework.context.ApplicationEvent*.

ApplicationListeners configured into the Spring context will be automatically notified of ApplicationEvents published through an ApplicationEventMulticaster.

The XT Modeling Framework extends those components by providing the following additional features:

- **Events filtering**.
- **Events collection**.
- **Event Driven POJOs**.

19.2.2.5.1. Filtering Events

Filtering events is important when your application generates a great number of different events and you want your listeners to be notified only of those events they are interested to.

The XT Modeling Framework provides two key classes for enabling events filtering capabilities:

- The **FilteringApplicationEventMulticaster** (see *org.springmodules.xt.model.event.filtering.FilteringApplicationEventMulticaster* javadoc).
- The **FilteringApplicationListener** (see *org.springmodules.xt.model.event.filtering.FilteringApplicationListener* javadoc).

The FilteringApplicationEventMulticaster is an ApplicationEventMulticaster [http://www.springframework.org/docs/api/org/springframework/context/event/ApplicationEventMulticaster.html] implementation that used in conjunction with FilteringApplicationListener implementations permits to efficiently filter published events, in order to avoid notifying *every* listener of *every* published event.

The FilteringApplicationListener interface defines supported events, providing a clear separation between event processing and event filtering logic:

```
public interface FilteringApplicationListener extends ApplicationListener {

    public Class[] getSupportedEventClasses();

    public boolean accepts(ApplicationEvent event);
}
```

- *public Class[] getSupportedEventClasses()* : defines supported event classes.
- *public boolean accepts(ApplicationEvent event)* : defines the actual filtering logic, often based on event state.

Concrete FilteringApplicationListener implementations must then be registered to a FilteringApplicationEventMulticaster.

Once events gets published through the FilteringApplicationEventMulticaster, it will notify each registered FilteringApplicationListener implementation only of those events whose class is supported by the given listener, and that are accepted by the listener itself.

Note

The FilteringApplicationEventMulticaster filters events only for registered FilteringApplicationListener implementations; however, it is possible to register other ApplicationListener [http://www.springframework.org/docs/api/org/springframework/context/ApplicationListener.html] implementations, losing filtering capabilities: those ApplicationListeners will be notified (as always) of every published event.

19.2.2.5.2. Collecting Events

While standard ApplicationListener [http://www.springframework.org/docs/api/org/springframework/context/ApplicationListener.html] implementations directly process events once they get notified, *event collectors* are special ApplicationListeners that collect and store published events, in order to provide FIFO (First In First Out) access to external, interested, objects.

Event collectors all share the following interface:

```
public interface ApplicationCollector extends ApplicationListener {

    public ApplicationEvent pollEvent();

    public List<ApplicationEvent> getEvents();

    public void clear();
}
```

- *public ApplicationEvent pollEvent()* : poll the first collected event, removing it from the queue of collected events.
- *public List<ApplicationEvent> getEvents()* : get all collected events, without removing them from the queue of collected events.
- *public void clear()* : clear all collected events.

Once registered into an ApplicationEventMulticaster as with any other ApplicationListener, ApplicationCollector implementations will start to collect notified events.

The XT Modeling Framework provides several, ready to use, ApplicationCollector implementations:

- **SimpleCollector** (see *org.springmodules.xt.model.event.collector.SimpleCollector* javadoc) : simple, non thread safe, ApplicationCollector implementation that simply collect events in a queue.
- **ThreadLocalCollector** (see *org.springmodules.xt.model.event.collector.ThreadLocalCollector* javadoc) : thread safe ApplicationCollector implementation storing collected events with per-thread scope.
- **BlockingCollector** (see *org.springmodules.xt.model.event.collector.BlockingCollector* javadoc) : thread

safe, synchronous, ApplicationCollector implementation that blocks when polling for events if no event is actually available.

- **ConcurrentCollector** (see *org.springmodules.xt.model.event.collector.ConcurrentCollector* javadoc) : synchronized ApplicationCollector implementation that wraps another non thread safe collector, in order to make it possible to safely share it among threads.

19.2.2.5.3. Event Driven POJOs

Event Driven POJOs (**EDPs**) are a powerful XT Modeling Framework feature that makes whatever application-specific object capable of processing events published through a Spring ApplicationEventMulticaster in a completely **transparent non-invasive way** , without having to implement any Spring specific interface.

So, an EDP is a plain old (framework agnostic) Java object with methods invoked on event publishing.

For enabling EDPs in your application, you have to configure an **EDPInvoker** (see *org.springmodules.xt.model.event.edp.EDPInvoker* javadoc) for every bean you want to make an EDP.

The EDPInvoker defines:

- The invoked EDP bean.
- The name of the method to invoke on published events.

More specifically, the XT Modeling Framework provides the **DefaultEDPInvoker** (see *org.springmodules.xt.model.event.edp.DefaultEDPInvoker* javadoc), which, once configured the EDP bean to use and the name of the method to invoke, will actually invoke the EDP if the bean has a method with name equal to the configured one, and that accepts a single parameter whose type is the same as, or is a super class/interface of, the published event.

Note

Please note that if the EDP bean provides several overloaded implementations of the invokable method, the DefaultEDPInvoker will invoke **all** methods supporting a given published event, that is, all methods accepting a parameter whose type is the same as, or is a super class/interface of the published event type.

So, given the following POJO:

```
public class SimpleBean {

    // This is the method that will be invoked
    public void onEvent(CustomEvent event) {
        // ...
    }
}
```

Here is how to configure a DefaultEDPInvoker for invoking the SimpleBean on publishing of CustomEvent events:

```
<!-- The EDP to invoke -->
<bean id="simpleBean" class="org.example.SimpleBean">

<!-- The EDP invoker -->
<bean id="edpInvoker" class="org.springmodules.xt.model.event.edp.DefaultEDPInvoker">
    <!-- The bean configuration -->
    <property name="invokedBean" ref="simpleBean"/>
    <!-- The method name configuration -->
    <property name="invokedMethodName" value="onEvent"/>
</bean>
```

Tip

If you want an EDPInvoker with filtering capabilities, to use with a FilteringApplicationEventMulticaster, you can pick the FilteringEDPInvoker: see the *org.springmodules.xt.model.event.edp.FilteringEDPInvoker* javadoc for more information.

19.2.3. Advanced Concepts

19.2.3.1. Other annotations

19.2.3.1.1. DynamicIntroductor Annotations

DynamicIntroductor concrete implementations let you modify the standard introductor behaviour by applying the following annotations on introduced interfaces:

- **org.springmodules.xt.model.introductor.annotation.MapToTargetField** : Determines if a setter or getter method directly maps to the corresponding field in the target class. I.E., if this annotation is applied to a *getName()* method, the introductor will have to access the *name* attribute in the target object.
- **org.springmodules.xt.model.introductor.annotation.OverrideTarget** : Determines if a method of an introduced interface must always override the corresponding method in the target class. The two methods must have equal signature.

Please read the javadoc of DynamicIntroductor implementations to verify supported annotations.

19.2.3.2. Apache Commons Predicates integration

The XT Modeling Framework provides ways for adapting generic Specification interfaces, as well as the XT Specification interfaces, to the well known *Apache Commons Predicate* [http://jakarta.apache.org/commons/collections/api-release/org/apache/commons/collections/Predicate.html] interface through the use of two Adapter objects: PredicateGenericAdapter and PredicateCompositeAdapter.

19.2.3.2.1. Using the PredicateGenericAdapter

The *org.springmodules.xt.model.specifications.adapter.PredicateGenericAdapter* adapts generic specification objects to the Apache Commons Predicate interface.

You can construct the adapter using one of the two constructors:

```
public PredicateGenericAdapter(Object specification, Method specificationMethod)
```

```
public PredicateGenericAdapter(Object specification, String specificationMethod)
```

The constructor requires the specification object to adapt and the specification method to call for evaluating the Predicate.

Once constructed, the Predicate adapter can be evaluated through the standard evaluate(Object) [http://jakarta.apache.org/commons/collections/api-release/org/apache/commons/collections/Predicate.html#evaluate(java.la method.

19.2.3.2.2. Using the PredicateCompositeAdapter

The *org.springmodules.xt.model.specifications.adapter.PredicateCompositeAdapter* adapts the XT Composite Specification to the Apache Commons Predicate interface.

You can construct the adapter using the following constructor:

```
public PredicateCompositeAdapter(CompositeSpecification specification)
```

The constructor requires the CompositeSpecification object to adapt.

Once constructed, the Predicate adapter can be evaluated through the standard evaluate(Object) [http://jakarta.apache.org/commons/collections/api-release/org/apache/commons/collections/Predicate.html#evaluate(java.la method.

19.3. XT Ajax Framework

19.3.1. Introduction

XT Ajax Framework is based on the following fundamental concepts:

- Events
- Handlers
- Actions
- Components
- Responses

They work together in a simple processing flow:

1. Web pages fire **events** using standard *JavaScript events* [http://www.w3schools.com/js/js_events.asp].
2. Each event causes an Ajax request that is processed by simple Java objects acting as server side **handlers**.
3. The handler processes events creating **actions**.
4. Each action may render one or more HTML **components** in the page.
5. At the end, actions are canalised into an Ajax **response** sent back to the web page.

We'll take a more detailed look at each of the concepts above in the next section.

19.3.2. Base Concepts

19.3.2.1. Ajax Events

In XT Ajax Framework you can raise events from your web pages as standard *JavaScript events* [http://www.w3schools.com/js/js_events.asp], and handle them by the server side, **without having to write any javascript code**. There are actually two types of Ajax events:

- **Action events**, a generic event type directly handled by configured handlers **without** passing control to any Spring MVC controller: this is most used for updating the web page with data from the server.
- **Submit events**, a special type of event handled by configured handlers **after** having executed the on submit method of the Spring MVC controller associated with the page: this event causes the submit of the page and so the handling of the submit by the controller, and can be used for further processing the submit in order to properly updating the view.

Ajax **action events** are basically fired in web pages as follows:

```
<input type="button" value="Go"
    onclick="XT.doAjaxAction(event-id, source-element, server-parameters, client-parameters);">
```

You have to simply call the **XT.doAjaxAction** JavaScript function, passing:

- The **name** (also called **id**) of the event you want to fire (*event-id*, required).
- The HTML element which fired the event, most of the time through the **this** objects (*source-element*, optional).
- A *JSON* [http://www.json.org/] object containing extra parameters to pass to the server (*server-parameters*, optional).
- A *JSON* [http://www.json.org/] object containing extra parameters for configuring the client side Ajax request (*client-parameters*, optional).

Warning

The old way of executing an Ajax action, through the **doAjaxAction** JavaScript function, is since version 0.8 **no more supported**.

Once you receive the event in your server side handler, you can access the following information (see XT javadocs for more information):

- The event id (corresponding to the event name above).
- The Java *HttpServletRequest* [http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/ServletRequest.html] object.
- The name and id of the HTML element firing the event.
- A map of string key/value pairs, containing the extra parameters defined above as *server-parameters*.
- The command object created by the Spring MVC controller.

Here are the Java interfaces:

```
public interface AjaxEvent {

    public String getEventId();

    public HttpServletRequest getHttpRequest();

    public String getElementName();

    public void setElementName(String elementName);

    public String getElementId();

    public void setElementId(String elementName);

    public Map<String, String> getParameters();

    public void setParameters(Map<String, String> parameters);
}
```

```
public interface AjaxActionEvent extends AjaxEvent {

    public Object getCommandObject();

    public void setCommandObject(Object command);
}
```

For firing Ajax **submit events**, you have to basically write:

```
<input type="button" value="Press"
    onclick="XT.doAjaxSubmit(event-id, source-element, server-parameters, client-parameters);">
```

You have to simply call the **XT.doAjaxSubmit** JavaScript function, passing it the same information defined above.

Warning

The old way of executing an Ajax submit, through the **doAjaxSubmit** JavaScript function, is since version 0.8 **no more supported**.

Once you receive the event in your server side handler, you can access the following information in addition to those defined above (see XT javadocs for more information):

- The *validation errors* [http://static.springframework.org/spring/docs/1.2.x/api/org/springframework/validation/Errors.html], for managing form submission errors (if any).
- The model map returned by the controller submit method (if any).

Here are the Java interfaces:

```
public interface AjaxSubmitEvent extends AjaxEvent {

    public Errors getValidationErrors();

    public void setValidationErrors(Errors errors);

    public Object getCommandObject();

    public void setCommandObject(Object command);

    public Map getModel();
    public void setModel(Map model);
}
```

Note

Given that submit events are handled after calling the configured Spring MVC controller, the command object, model map, and validation errors contained in the submit event are **the same** objects you manage in the controller.

Important

How to access the command object and validation errors outside of Spring MVC environments?

The XT Ajax Framework is by default configured for working with Spring MVC, so in order to access the command object and validation errors the MVC controller must adhere to the requirements provided by the *org.springmodules.xt.ajax.MVCFormDataAccessor* class (see related javadocs for details). If you want to use our framework in other environments, like Spring Web Flow, you have to implement a proper *org.springmodules.xt.ajax.FormDataAccessor* and set it into the *org.springmodules.xt.ajax.AjaxInterceptor* (again, see related javadocs for details).

Important

Why are the command object and model map contained into the AjaxSubmitEvent null?

This is probably because your Spring MVC Controller submit method missed to return a proper ModelAndView object containing the BindException model; for helping that, you are strongly encouraged to construct and return an **AjaxModelAndView** object (see *org.springmodules.xt.ajax.web.servlet.AjaxModelAndView* javadoc).

Under the hood, Ajax action events are nothing more than asynchronous GET requests, while Ajax submit events are asynchronous PUT requests.

If there are some web forms in the page, the Ajax request sends by default all values belonging to the first form in the page. If you have just one form in your page, all form values will be automatically sent with the Ajax event, so you have nothing special to do. If you have more than one form, and you want to send values belonging to a specifically named form, you have to specify the form name between client parameters:

```
XT.doAjaxAction(event-id, source-element, server-parameters, {'formName' : 'form_name'});
XT.doAjaxSubmit(event-id, source-element, server-parameters, {'formName' : 'form_name'});
```

Moreover, in order to be XHTML strict compatible, you can use the form id instead:

```
XT.doAjaxAction(event-id, source-element, server-parameters, {'formId' : 'form_id'});
XT.doAjaxSubmit(event-id, source-element, server-parameters, {'formId' : 'form_id'});
```

Now you have enough information about how to fire events by the client side, and how they are represented by the server side. Let's talk about how to **handle** them.

19.3.2.2. Ajax Handlers

The Ajax handler is the server side component to implement for processing Ajax events. Here is its interface, followed by a description of its methods:

```

```

```
public interface AjaxHandler {

    public boolean supports(AjaxEvent event);

    public AjaxResponse handle(AjaxEvent event);
}
```

- *public boolean supports(AjaxEvent event)* : Check if *this* handler supports (can handle) the given **AjaxEvent**
- *public AjaxResponse handle(AjaxEvent event)* : Handle the given **AjaxEvent** (if supported) and return an appropriate **AjaxResponse** (see Ajax Response).

Out of the box, XT Ajax Framework provides an AjaxHandler abstract implementation which provides a straightforward way of implementing Ajax handlers: the **org.springmodules.xt.ajax.AbstractAjaxHandler**.

AbstractAjaxHandler is an event dispatcher based on Java reflection and Ajax events name (*id*): it implements the two methods above by dispatching events to methods named after the event id. So, for handling different events, you have to simply implement a method named after the event id you want to handle, with the following signature:

```
public AjaxResponse eventId(AjaxEvent )
```

Given an Ajax event with id *countrySelection*, the handling method will be:

```
public AjaxResponse countrySelection(AjaxEvent )
```

Doing so, you handle generic *AjaxEvent*s. If you want to handle *AjaxActionEvent*s or *AjaxSubmitEvent*s, simply write methods with the following signatures:

```
public AjaxResponse eventId(AjaxActionEvent )
```

```
public AjaxResponse eventId(AjaxSubmitEvent )
```

However, you are free to implement a different handling strategy, directly implementing the interface above.

Note

When returning from an handler, if you don't want to send any response you can safely return a null AjaxResponse. What happens after that depends on the kind of event you are handling; see the section: More about the Ajax request processing flow

19.3.2.3. Associating events with handlers: the AjaxInterceptor

Once defined your events and implemented your handlers, you need to **associate** them.

For **security reasons**, web page events and Ajax handlers are not automatically associated, nor can be globally associated. If it were so, malicious users could hack your JavaScript and execute unwanted events.

So, events and handlers are associated **per page path**, that is, you have to configure every handler to manage events fired by a pre-determined, restricted, set of pages. Specifically, you can configure an handler to manage events fired by a specific, single, web page, or by a set of web pages defined with *Ant* [http://ant.apache.org] style pattern matching.

This is done in two simple steps.

First, you have to define in your Spring application context an **AjaxInterceptor**: it is the bean which associates web pages and Ajax handlers, making under the cover the actual event dispatching. Here is an example of AjaxInterceptor configuration:

```
<bean id="ajaxInterceptor" class="org.springmodules.xt.ajax.AjaxInterceptor">

    <property name="handlerMappings">
        <props>
            <prop key="/ajax/ex*">genericAjaxHandler</prop>
            <prop key="/ajax/ex1.page">ajaxHandler1</prop>
            <prop key="/ajax/ex2.page">ajaxHandler2, ajaxHandler3</prop>
        </props>
    </property>

</bean>
```

The *handlerMappings* property is a list of key/value entries associating **a web path** (the key) **with a comma separated list of handlers** (the value), which must be declared as beans in the Spring application context.As you can notice, the first entry associates an handler with an Ant style pattern, the second associates a single page with a single handler, the third associates a single page with two handlers. So, you can also apply more handlers to the same group of pages (or page). When more handlers apply to the same event, the one configured for matching the longest web path wins.

Finally, the second step requires you to associate the AjaxInterceptor with the interceptors chain of your Spring MVC *URL handler mapping* [http://static.springframework.org/spring/docs/1.2.x/api/org/springframework/web/servlet/HandlerMapping.html]. Here is how:

```
<bean id="urlHandlerMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

    <property name="interceptors">
        <list>
            <ref bean="ajaxInterceptor"/>
        </list>
    </property>

    <property name="mappings">
        <props>
        <!-- ... -->
        </props>
    </property>

</bean>
```

19.3.2.4. Ajax Actions

Ajax actions are created by handlers and act on client web pages.

XT Ajax Framework provides many useful actions ready to be used: see the *org.springmodules.xt.ajax.action* package for details; however, creating custom actions is just a matter of implementing the following simple interface:

```
public interface AjaxAction {
    public String render();
}
```

The *render()* method is the one you call for producing the action result.

However, all actions that render data into the web page implement the following interface:

```
public interface AjaxRenderingAction extends AjaxAction {

    public ElementMatcher getElementMatcher();
}
```

So, there are actions that render data into the web page, modifying page parts, but ... **how can actions identify page parts to modify**?

19.3.2.5. Identifying page parts: ElementMatchers.

Ajax actions that render data into the web page are all implementations of the *org.springmodules.xt.ajax.AjaxRenderingAction* interface.

By default, Ajax actions that need to identify page parts to modify use the HTML **id** attribute. So, assigning an identifier to your HTML elements is a way to make Ajax actions identify them:

```
<select id="toFill">
</select>
```

However, Ajax actions are not limited to HTML identifiers.

More specifically, **all actions that modify page contents must use either**:

- The element identifier (as previously stated).
- An *org.springmodules.xt.ajax.ElementMatcher* implementation.

ElementMatchers are a flexible way of identifying page parts by matching elements with different strategies. There are several ElementMatchers:

org.springmodules.xt.ajax.action.matcher.DefaultMatcher
: The DefaultMatcher matches a single element by its identifier. It is the default matcher used by actions that directly use element identifiers (as stated above).

org.springmodules.xt.ajax.action.matcher.ListMatcher
: The ListMatcher matches multiple elements against a list of identifiers.

org.springmodules.xt.ajax.action.matcher.WildcardMatcher
: The WildcardMatcher matches multiple elements whose identifier starts with a substring of a given source identifier and ends with the "_" **(underscore)** wildcard; for instance, a WildcardMatcher with the "foo.bar" id to match, will match an element with a "foo.bar" id, and an element with the "foo_" id too.

org.springmodules.xt.ajax.action.matcher.SelectorMatcher
: The SelectorMatcher matches multiple elements identified by a list of CSS2 [http://www.w3.org/TR/REC-CSS2/selector.html]/CSS3 [http://www.w3.org/TR/css3-selectors/] Selectors

(see the org.springmodules.xt.ajax.action.matcher.SelectorMatcher javadoc for supported selectors).

See javadocs of the classes above for additional details.

19.3.2.6. Components

Generally speaking, **components** represent web page parts; more specifically, most of the time they represent HTML elements, like input fields, tables and so.

XT Ajax Framework provides many useful components ready to be used:

- **Static components**, representing simple, static, HTML elements (see the *org.springmodules.xt.ajax.component* package for details).
- **Dynamic components**, capable of dynamically rendering page parts (see the *org.springmodules.xt.ajax.component.dynamic* package for details).

However, creating custom components is just a matter of implementing the following simple interface:

```
public interface Component {

    public String render();
}
```

Each component renders itself through the **render()** method, returning a string representation of its rendering.

But how do actions render components?

Components are added to actions in a way that depends on the concrete action implementation, and then rendered through the action *execute()* method.

Now that you have your Ajax actions and components, the only thing left to do is filling your Ajax response.

19.3.2.7. Ajax Response

AjaxResponse objects are filled with actions whose result is carried to clients, that is, to web pages. This is the interface of the AjaxResponse, followed by a description of its methods:

```
public interface AjaxResponse {

    public void addAction(AjaxAction action);

    public String render();

    public String getEncoding();

    public boolean isEmpty();
}
```

- *public void addAction(AjaxAction action)* : This method is used for adding actions that will be executed and whose result will be sent to the client.
- *public String render()* : This method renders the added actions and returns the result.
- *public String getEncoding()* : This method returns the charset name of the encoding to use for the response.

- *public boolean isEmpty()* : This method tells if the response has no actions to return.

Note

The default encoding of the Ajax response is ISO-8859-1. Just use a proper *org.springmodules.xt.ajax.AjaxResponseImpl* constructor for changing it.

Note

In the context of an Ajax response, actions are rendered (and executed client side) **following their order of adding**.

19.3.2.8. More about the Ajax request processing flow

As seen above, Ajax handlers process events, dispatched by the AjaxInterceptor, and return an appropriate AjaxResponse.

Here is a more schematic, detailed look at the Ajax request processing flow.

On Ajax Action Events :

1. The web page fires an Ajax action event.
2. The event is directly dispatched by the AjaxInterceptor toward a configured AjaxHandler.
3. If no handler supporting the event is found, an exception is thrown, else if the event is supported by a proper handler, it is handled.
4. After handling, if the handler returns a not null and not empty AjaxResponse, it is actually rendered. Anyway, **since version 0.9**, the AjaxInterceptor **stops** the execution chain after processing any Ajax request.

Important

Since 0.9 version, the AjaxInterceptor stops the execution chain **even after** an empty or null AjaxResponse.

On Ajax Submit Events :

1. The web page fires an Ajax submit event, causing a form submit.
2. The (eventually) configured Spring MVC Controller is called.
3. The Spring MVC Controller handles the form submit, eventually calling Validators and making other actions. At the end, it must return a ModelAndView with the model contained into the BindException object; otherwise, the model and command objects into the Ajax event object will be null. You can use the **AjaxModelAndView** object here (see *org.springmodules.xt.ajax.web.servlet.AjaxModelAndView* javadoc).
4. After the Controller handling, the event is dispatched by the AjaxInterceptor toward a configured AjaxHandler.
5. If no handler supporting the event is found, an exception is thrown, else if the event is supported by a proper handler, it is handled.

6. After handling, if the handler returns a not null and not empty AjaxResponse, it is rendered; otherwise the AjaxInterceptor automatically tries to redirect to the view configured in your ModelAndView object. The view must contain the **ajax-redirect** prefix (i.e. : *ajax-redirect:/start.page*) or the standard **redirect** prefix (i.e. : *redirect:/start.page*); if the view doesn't contain such prefixes, an exception is thrown.

19.3.2.9. Handling exceptions

Spring MVC deals with exceptions occurred during request processing by using HandlerExceptionResolver [http://static.springframework.org/spring/docs/2.0.x/api/org/springframework/web/servlet/HandlerExceptionResolver.html]

The XT Ajax Framework deals with exceptions occurred during Ajax requests in the same way, by using a special HandlerExceptionResolver: the **org.springmodules.xt.ajax.AjaxExceptionHandlerResolver.**

It works by mapping exception classes to instances of **AjaxExceptionHandler** beans (see *org.springmodules.xt.ajax.AjaxExceptionHandler* javadoc) configured in the Spring context.

Here is a sample AjaxExceptionHandlerResolver configuration snippet:

```
<bean id="ajaxExceptionResolver" class="org.springmodules.xt.ajax.AjaxExceptionHandlerResolver">
    <property name="exceptionMappings">
        <map>
            <entry key="java.lang.Exception" value-ref="exceptionHandler"/>
        </map>
    </property>
</bean>
```

The AjaxExceptionHandler bean then takes care of resolving the exception to an AjaxResponse to send to clients. Here is its interface:

```
public interface AjaxExceptionHandler {

    public AjaxResponse handle(HttpServletRequest request, Exception ex);
}
```

Out of the box, XT Ajax Framework provides the **RedirectExceptionHandler** (see *org.springmodules.xt.ajax.support.RedirectExceptionHandler* javadoc), an AjaxExceptionHandler implementation for redirecting to a given error page. Here is a sample configuration snippet:

```
<bean id="exceptionHandler" class="org.springmodules.xt.ajax.support.RedirectExceptionHandler">
    <property name="redirectUrl" value="/error.page"/>
    <property name="exceptionMessageAttribute" value="exceptionMessage"/>
</bean>
```

Warning

Since 0.9 version, the old *org.springmodules.xt.ajax.AjaxExceptionHandler* is now called *org.springmodules.xt.ajax.AjaxExceptionHandlerResolver*, while the old *org.springmodules.xt.ajax.AjaxExceptionResolver* and *org.springmodules.xt.ajax.support.RedirectExceptionResolver* are respectively called *org.springmodules.xt.ajax.AjaxExceptionHandler* and *org.springmodules.xt.ajax.support.RedirectExceptionHandler*.

19.3.2.10. Showing feedback to users : loading sign management

Ajax requests are asynchronous, and while they're processed users are free to continue interacting with the web page.

It's often important to give users a feedback about the Ajax request loading time: XT Ajax let you easily configure a *loading sign*, that is, **an image that will be automatically shown during Ajax request processing**.

The loading sign is configured by simply choosing the identifier of the element where the loading sign will be shown, and the URL of the image to show. You have two different configuration option:

Per Page configuration

By using the per page configuration, every Ajax request will show the same image in the same element into the page.

The identifier of the element where to show the loading sign must be assigned to the following Javascript variable:

```
XT.defaultLoadingElementId
```

While the image URL must be assigned to the following Javascript variable:

```
XT.defaultLoadingImage
```

So, just load a Javascript file with the following assignments:

```
XT.defaultLoadingElementId = "..."
XT.defaultLoadingImage = "..."
```

Per Request Configuration

By using the per request configuration, every Ajax request will be able to show a different image in a different element into the page.

Every Ajax request must specify its own loading element and image by putting the following information between *client parameters*:

```
XT.doAjaxAction(event-id, source-element, server-parameters, {'loadingElementId' : '...', 'loadingImage' : '...'
XT.doAjaxSubmit(event-id, source-element, server-parameters, {'loadingElementId' : '...', 'loadingImage' : '...'
```

As you can see, you have to specify the following parameters:

- *loadingElementId* : The id of the element where the image will be shown during the processing of the Ajax request related to *this* event.
- *loadingImage* : The URL of the image to show during the processing of the Ajax request related to *this* event.

You can use both per page and per request configuration options: the latter will prevail on the former.

Tip

For awesome effects, we strongly suggest to use gif images; you can find great loading gif images *here*. [http://www.ajaxload.info/]

19.3.2.11. Handling Javascript errors

By default, Javascript errors occurring during client-side Ajax request processing don't get handled.

However, you can handle them by configuring a proper **error handler function.**

As with the loading sign configuration described in the previous section, you have two different configuration option:

Per Page configuration

By using the per page configuration, every Ajax request will use the same error handler function.

You have to simply configure the following property of the *XT* javascript object:

```
XT.defaultErrorHandler = your_function_name
```

Per Request Configuration

By using the per request configuration, every Ajax request will use a different error handler function.

Every Ajax request must specify its own function by putting the following information between *client parameters*:

```
XT.doAjaxAction(event-id, source-element, server-parameters, {'errorHandler' : your_function_name});
XT.doAjaxSubmit(event-id, source-element, server-parameters, {'errorHandler' : your_function_name});
```

You can use both per page and per request configuration options: the latter will prevail on the former.

Finally, the error handler function should look like the following:

```
function yourFunction(ajaxRequest, exception) {
    // ...
}
```

As you can see, your function will have to accept two parameters: the ajax request object and the exception object.

19.3.2.12. Uploading files

XT Ajax transparently supports file uploading in Ajax enabled forms.

Uploading a file through Ajax is just a matter of sending an Ajax submit request configured as follows:

```
XT.doAjaxSubmit(event-id, source-element, server-parameters, {'enableUpload' : true})
```

You have to set the *enableUpload* Ajax client parameter to *true* (as a boolean value): that's all. XT Ajax will transparently do the rest.

19.3.3. Advanced Concepts

19.3.3.1. Core Javascript libraries

The client side API of the XT Ajax Framework is based on the *Taconite* [http://taconite.sourceforge.net] and *JSON* [http://www.json.org/js.html] JavaScript libraries, plus some other JavaScript stuff.

These are the core XT Ajax Javascript libraries: you can find them under the *js* directory.

They are put, at compile time, in the **springxt.js** file under the *js/lib/core* directory. This is the only **mandatory** file you need to include in your web page:

```
<script type="text/javascript" src='springxt.js'></script>
```

Tip

Under the *js/lib/core* you'll find also a **springxt-min.js** file: it is a minified version of *springxt.js* you can include in your pages for reducing the download size.

19.3.3.2. Optional Javascript libraries

XT Ajax Framework integrates with a number of other Javascript libraries, in order to provide additional Ajax functionalities:

- *Prototype* [http://prototype.conio.net/] (version 1.5.0 rc1 or higher) : see javadocs for the *org.springmodules.xt.ajax.action.prototype* classes.
- *Script.aculo.us* [http://script.aculo.us/] (version 1.6.4 or higher) : see javadocs for the *org.springmodules.xt.ajax.action.prototype.scriptaculous* classes.
- *Behaviour* [http://www.bennolan.com/behaviour/] (version 1.1 or higher) : see javadocs for the *org.springmodules.xt.ajax.action.behaviour* classes.

19.3.3.3. Tweaking your Ajax request

In previous sections you've seen how to enable several XT Ajax features by setting the proper client parameter when sending the Ajax request. Here is a list of additional client parameters to set for adding more tweaks to your request.

clearQueryString
: Set this parameter to *true* (as a boolean value) to remove all current GET parameters from the request query string.

19.3.4. Tutorials

In this section we'll show you, by practical, step-by-step tutorials, how to work with XT Ajax Framework. All tutorials are based on XT Ajax samples: take a look at the samples provided with the main distribution or *check out them* [https://springmodules.dev.java.net/source/browse/springmodules/samples/xt/] if you want to take a look at the full source code.

19.3.4.1. Working with Ajax action events

Ajax action events are used for updating pages without submitting data to (eventually configured) Spring MVC controllers: so, **the execution of an Ajax action event doesn't call controllers**.

In this tutorial we'll implement a simple Ajax sample that let you **fill a selection box with a list of Office names after clicking a button**, showing you how to:

- Write the web page.
- Write the Ajax handler.
- Map the Ajax handler to the web page URL.

19.3.4.1.1. Step 1 : Writing the web page

Writing a web page that fires an Ajax action event is not different than writing a normal JSP based web page as you'd usually do.

First, import the core XT Ajax javascript library:

```
<script type="text/javascript" src='springxt.js'></script>
```

Our web page must fill the selection list after clicking a button. So, write a button input field that fires an Ajax action event with *loadOffices* as event id:

```
<input type="button" value="Press" onclick="XT.doAjaxAction('loadOffices', this);">
```

Then, write the *select* HTML element to update and give it an id:

```
<select id="offices">
    <option>--- ---</option>
</select>
```

Recall that the *id* attribute is used for identifying the page part to update, that is, the element to fill with new content.

That's all ... let's write our Ajax handler!

19.3.4.1.2. Step 2 : Writing the Ajax handler

Our Ajax handler will extend the *org.springmodules.xt.ajax.AbstractAjaxHandler*, so it will have a method called after the Ajax event to handle, that will accept an *org.springmodules.xt.ajax.AjaxActionEvent*:

```
public AjaxResponse loadOffices(AjaxActionEvent event)
```

Now, let us analyze how to handle the event, by implementing the *loadOffices* method above.

First, we have to retrieve a list of offices from some kind of data access object:

```
Collection<IOffice> offices = store.getOffices();
```

Then, we have to create the components to render: a list of *org.springmodules.xt.ajax.component.Option* components, representing the *option* HTML elements and containing the office id as value and the office name as content.

```
// Create the options list:
List options = new LinkedList();
// The first option is just a dummy one:
```

```
Option first = new Option("-1", "Select one ...");
options.add(first);
// Create options representing offices:
for(IOffice office : offices) {
    Option option = new Option(office, "officeId", "name");
    options.add(option);
}
```

Now, we have to replace the HTML content of the *select* element showed above, so we have to create an *org.springmodules.xt.ajax.action.ReplaceContentAction*, adding it the components to render (the list of options):

```

ReplaceContentAction action = new ReplaceContentAction("offices", options);
```

Note that the *ReplaceContentAction* updates the HTML element with *offices* as id.

Finally, we create an *org.springmodules.xt.ajax.AjaxResponse*, add the action and return it!

```

AjaxResponse response = new AjaxResponseImpl();
response.addAction(action);
return response;
```

That's the simple implementation of the *loadOffices* method!

19.3.4.1.3. Step 3 : Mapping the Ajax handler to the web page URL

Say the web page URL is: *www.example.org/xt/ajax/tutorial1.page*. Mapping the Ajax handler is simply a matter of configuring the Ajax handler bean (*LoadOfficesHandler* in the snippet below) in the Spring application context and mapping it in the *AjaxInterceptor*:

```

<bean id="ajaxLoadOfficesHandler" class="org.springmodules.xt.examples.ajax.LoadOfficesHandler">
    <property name="store" ref="store"/>
</bean>
```

```

<bean id="ajaxInterceptor" class="org.springmodules.xt.ajax.AjaxInterceptor">
    <property name="handlerMappings">
        <props>
            <prop key="/ajax/tutorial1.page">ajaxLoadOfficesHandler</prop>
        </props>
    </property>
</bean>
```

19.3.4.2. Working with Ajax submit events

Ajax submit events are used for updating pages **after** submitting data to your Spring MVC controllers.

In this tutorial we'll implement a simple Ajax sample that let you **choose an office and list its employees in a table after submitting the form by clicking a button**. We'll see how to:

- Write the web page.
- Write the Spring MVC controller.
- Write the Ajax handler.

- Map the Ajax handler to the web page URL.

19.3.4.2.1. Step 1 : Writing the web page

Writing a web page that fires a submit event is not different than writing a normal JSP based web page as you'd usually do.

First, import the core XT Ajax javascript library:

```
<script type="text/javascript" src='springxt.js'></script>
```

Employees are listed in an HTML table after clicking a button. So, you have to write a button input field that fires an Ajax submit event with *listEmployees* as event id:

```
<input type="button" value="List" onclick="XT.doAjaxSubmit('listEmployees', this);">
```

Then, write the HTML *table* element to use for listing the employees:

```
<table border="1">

    <thead>
       <tr>
           <th>Firstname</th>
           <th>Surname</th>
           <th>Matriculation Code</th>
       </tr>
    </thead>

   <tbody id="employees">
   </tbody>

</table>
```

Please note the table body, with an *employees* id attribute: this is the page part that will be updated with the employees list.

That's all ... let's take a look at our Spring MVC controller!

19.3.4.2.2. Step 2 : Writing the Spring MVC controller

XT Ajax Framework requires only little changes to the way you write Spring MVC controllers.

For the purposes of our example, the most interesting part of our Spring MVC controller is the *onSubmit* method:

```
protected ModelAndView onSubmit(Object command, BindException errors)
throws Exception {
    // Take the command object and the office contained in it:
    EmployeesListForm form = (EmployeesListForm) command;
    Office office = form.getOffice();

    // Take a list of employees by office:
    Collection<IEmployee> employees = store.getEmployeesByOffice(office);

    // Construct and return the ModelAndView:
    Map model = new HashMap(1);
    model.put("employees", employees);
    return new AjaxModelAndView(this.getSuccessView(), errors, model);
    // The model map contains the employee list that will be rendered using ajax!
}
```

The only difference is the use of the **AjaxModelAndView** (see *org.springmodules.xt.ajax.web.servlet.AjaxModelAndView* javadoc), carrying the BindException errors object required by the Ajax framework.

Note

The AjaxModelAndView object behaves exactly the same as a standard ModelAndView object.

Let's go with our Ajax handler!

19.3.4.2.3. Step 3 : Writing the Ajax handler

Our Ajax handler will extend the *org.springmodules.xt.ajax.AbstractAjaxHandler*, so it will have a method called after the Ajax event to handle, that will accept an *org.springmodules.xt.ajax.AjaxSubmitEvent*:

```
public AjaxResponse listEmployees(AjaxSubmitEvent event)
```

Let's talk about the *listEmployees* method implementation.

We want to show the employees belonging to the selcted office, so we have to retrieve the model map from the event object, and the employee list contained in it:

```
Map model = event.getModel();
Collection<IEmployee> employees = (Collection) model.get("employees");
```

Then, we have to create the components to render: a list of *org.springmodules.xt.ajax.component.TableRow* components, containing the employees:

```
// Create the rows list:
List rows = new LinkedList();
for(IEmployee emp : employees) {
    // Every row is an employee:
    TableRow row = new TableRow(emp, new String[]{"firstname", "surname", "matriculationCode"}, null);
    rows.add(row);
}
```

Now we have to replace all the rows in the HTML table, so we have to create an *org.springmodules.xt.ajax.action.ReplaceContentAction*, adding it the components to render:

```
ReplaceContentAction action = new ReplaceContentAction("employees", rows);
```

Note that the *ReplaceContentAction* updates the HTML element with *employees* as id.

Finally, we have to create an *org.springmodules.xt.ajax.AjaxResponse* and return it!

```
AjaxResponse response = new AjaxResponseImpl();
response.addAction(action);
return response;
```

That's the *listEmployees* method implementation!

19.3.4.2.4. Step 4 : Mapping the Ajax handler to the web page URL

Say the web page URL is: *www.example.org/xt/ajax/tutorial2.page*. Mapping the Ajax handler is simply a matter of configuring the Ajax handler bean (*ajaxListEmployeesHandler* in the snippet below) in the Spring application context and mapping it in the *AjaxInterceptor*:

```
<bean id="ajaxListEmployeesHandler" class="org.springmodules.xt.examples.ajax.ListEmployeesHandler">
    <property name="store" ref="store"/>
</bean>
```

```
<bean id="ajaxInterceptor" class="org.springmodules.xt.ajax.AjaxInterceptor">
    <property name="handlerMappings">
        <props>
            <prop key="/ajax/tutorial2.page">ajaxListEmployeesHandler</prop>
        </props>
    </property>
</bean>
```

19.3.4.3. Working with Ajax validation

Ajax validation is a common use case, so the XT Ajax Framework provides the **org.springmodules.xt.ajax.validation.DefaultValidationHandler** for doing Ajax based validation in a very simple way.

In this tutorial we'll implement a simple Ajax validation use case: **validating an employee matriculation code**. We'll see how to:

- Configure the DefaultValidationHandler.
- Write the Spring MVC validator.
- Write the web page.

19.3.4.3.1. Step 1 : Configuring the DefaultValidationHandler

If you want to use the DefaultValidationHandler without any customization, you must simply configure and map it into the Spring application context as you'd usually do with any other handler:

```
<bean id="ajaxValidationHandler" class="org.springmodules.xt.ajax.validation.DefaultValidationHandler">
```

```
<bean id="ajaxInterceptor" class="org.springmodules.xt.ajax.AjaxInterceptor">
    <property name="handlerMappings">
        <props>
            <prop key="/ajax/tutorial3.page">ajaxValidationHandler</prop>
        </props>
    </property>
</bean>
```

By default, the DefaultValidationHandler displays and highlights error messages in the submitted web page, and redirects to the success page on successfull. However, you can **customize how error messages are rendered**, by providing a custom implementation of the *org.springmodules.xt.ajax.validation.ErrorRenderingCallback* class, and **how successfull validation is handled**, by providing a custom implementation of the *org.springmodules.xt.ajax.validation.SuccessRenderingCallback* class.

19.3.4.3.2. Step 2 : Writing the Spring MVC validator

XT Ajax Framework doesn't require to change the validator code: it is completely independent.

So here is the validator:

```
public class EmployeeValidator implements Validator {

    public boolean supports(Class aClass) {
        return IEmployee.class.isAssignableFrom(aClass);
    }

    public void validate(Object object, Errors errors) {
        if (this.supports(object.getClass())) {
            IEmployee emp = (IEmployee) object;
            if (emp.getMatriculationCode() == null || emp.getMatriculationCode().equals("")) {
                errors.rejectValue("matriculationCode",
                "employee.null.code", "No Matriculation Code!");
            }
        }
    }
}
```

Please note the *employee.null.code* error code: as we are going to see in the next section, it is the message that will be rendered in the web page by the DefaultValidationHandler.

19.3.4.3.3. Step 3 : Writing the web page

First, import the core XT Ajax javascript library, plus the Prototype and Script.aculo.us libraries:

```
<script type="text/javascript" src='springxt.js'></script>
<script type="text/javascript" src='prototype.js'></script>
<script type="text/javascript" src='scriptaculous.js?load=effects'></script>
```

Then, you have to mark the HTML elements where to show the errors sent by the validator: this requires only to **write HTML elements whose id is the same as the error codes you want to show**.

In our sample, we have an *employee.null.code* error code, and we want to have a *div* element containing just the *employee.null.code* error; here is what we have to write:

```
<div id="employee.null.code"/>
```

The DefaultValidationHandler will fill the element above with the proper error message.

Note

Error messages filled by the DefaultValidationHandler are **internationalized**.

Finally, you have to simply call the DefaultValidationHandler by firing an Ajax submit event in the following way:

```
<input type="button" value="Fire" onclick="XT.doAjaxSubmit('validate', this);">
```

Note

validate is the mandatory event name associated with the DefaultValidationHandler.

19.3.4.3.4. Step 4 : Organizing error messages

The DefaultValidationHandler puts your error messages in HTML elements whose identifier is equal to the validator error code.

In the previous section, the handler used the following element:

```
<div id="employee.null.code"/>
```

to render the *employee.null.code* error.

What if you want to group error messages, i.e. all errors with the error code starting with the "employee" substring?

The DefaultValidationHandler automatically uses wildcard matching; by doing so, it let you organize your error messages in HTML elements whose identifier either:

- Is exactly equal to the error code whose message has to be displayed.
- Starts with a substring of the error code whose message has to be displayed, and ends with the "_" wildcard.

Error messages will be displayed into all elements with matching identifier.

An example will help clarifying.

Say you have two error codes (with related messages): *employee.null.code* and *employee.null.username*. If you want to display a different HTML element for each error message, plus an HTML element containing all error messages whose code starts with "employee", just put the following elements in your web page:

```
<div id="employee_"/>
......
<div id="employee.null.code"/>
<div id="employee.null.username"/>
```

The first div element will display all error messages starting with the "employee" substring, while the other two will display each one the proper message.