Spring Framework

# Lý thuyết về Spring framework

## Khái niệm software framework

Software framework là một platform mà các chức năng tổng quát có thể được lựa chọn sử dụng hoặc ghi đè bởi người phát triển hoặc người sử dụng. Framework mang hình thức của thư viện nơi mà các API có thể tái sử dụng bất cứ đâu bên trong phần mềm.

Các tính năng do framework tạo ra thì khác với các hình thức thư viện khác, thể hiện ở 4 khía cạnh:

* Default Behavior: framework có các hành động mặc định của nó.
* Inversion of Control: không giống thư viện, framework điều khiển luồng xử lý bên trong nó.
* Extensibility: người dùng có thể mở rộng framework bằng cách thay mã lệnh của framework bằng mã lệnh của họ.
* Non-modifiable Framework Code: người dùng có thể mở rộng framework nhưng không thể chỉnh sửa mã lệnh của nó.

Mục đích của framework là làm đơn giản môi trường phát triển ứng dụng, cho phép người phát triển tập trung vào yêu cầu dự án thay vì các chức năng và thư viện lặp lại.

## Sơ lược về Spring framework

Bạn có thể tham khảo về Spring tại [đây](http://www.springsource.org/spring-framework). Các phiên bản của Spring: 1.1, 1.2, 2.0. 2.5, 3.0, 3.1, 3.2.

Spring framework cung cấp mô hình lập trình toàn diện và cấu hình cho các ứng dụng trên nền Java. Yếu tố key của Spring là hạ tầng hỗ trợ mức ứng dụng: Spring tập trung vào “plumbing” của các ứng dụng doanh nghiệp.

Spring bao gồm:

* Injection linh hoạt với kiểu cấu hình XML và annotations.
* Hỗ trợ nâng cao theo hướng aspect với các biến thể trên nền proxy và AspectJ.
* Hỗ trợ khai báo transactions, caching, validation và formatting.
* Trừu tượng hóa mạnh để làm việc với các đặc tả Java EE như: JDBC, JPA, JTA và JMS.
* Hỗ trợ open-source như: Hibernate và Quartz.
* Framework để xây dựng các ứng dụng RESTful MVC và các dịch vụ đầu cuối.

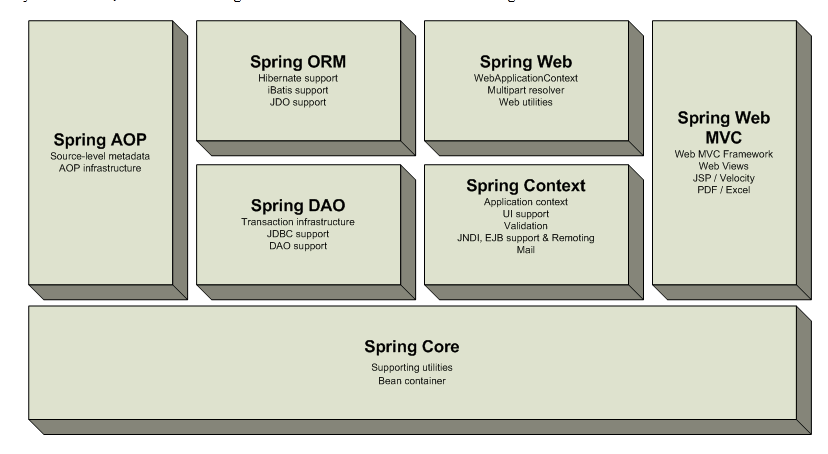
Spring thiết kế dạng mô-đun, cho phép gia tăng các thành phần riêng lẻ như core container hay hỗ trợ JDBC. Tất cả Spring services đều phù hợp với core container. Nhiều services có thể sử dụng bên ngoài container.

Hỗ trợ các platform để triển khai như Tomcat và Websphere.

Các project Spring gồm:

* Spring Security
* Spring Integration
* Spring Batch
* Spring Data
* Spring Web Flow
* Spring Web Services
* Spring Mobile
* Spring Social
* Spring Android

Spring có 7 chức năng được tổ chức thành 7 modules như hình:



Core package là thành phần cơ bản nhất của framework và nó cung cấp đặc tính Dependency Injection cho phép quản lý tính năng bean container. Khái niệm căn bản ở đây là BeanFactory.

Context package cung cấp cách thức truy cập beans theo kiểu của framework, giống với JNDI-registry. Context package kế thừa đặc tính từ beans package và thêm vào các hỗ trợ cho text messaging.

DAO package cung cấp tầng JDBC-abstraction mà loại bỏ mã JDBC và việc parsing các mã lỗi database-vendor. JDBC package cung cấp cách quản lý khai báo transaction cả cho các classes hiện thực các interfaces cụ thể và các POJOs (plain old java objects).

ORM package cung cấp các tầng tương tác cho các ORM APIs thông dụng, gồm JDO, Hibernate và iBatis. Dùng ORM package bạn có thể sử dụng các O/R-mappers kết hợp với các đặc tính khác của Spring cung cấp.

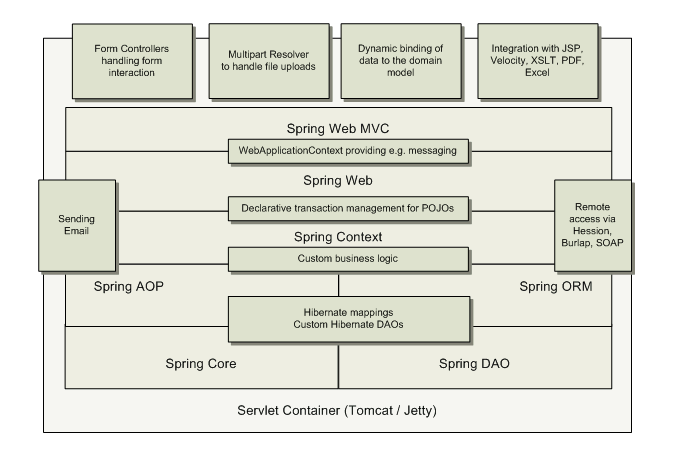
AOP package cung cấp một AOP Alliance dễ dàng lập trình aspect-oriented. Dùng chức năng metadata bạn có thể kết hợp tất cả loại thông tin vào mã lệnh.

Web package cung cấp các đặc trưng cơ bản tương tác với web như đa chức năng, khởi tạo ngữ cảnh dùng servlet listeners và ngữ cảnh ứng dụng web. Đây là package để Spring tương tác với WebWork hoặc Struts.

Web MVC package cung cấp một implementation của mô hình MVC cho ứng dụng web. Ngoài implementation nó còn cung cấp một sự phân tách rõ ràng giữa mã domain model với web forms và cho phép sử dụng bất cứ đặc trưng nào của Spring Framework như là validation.

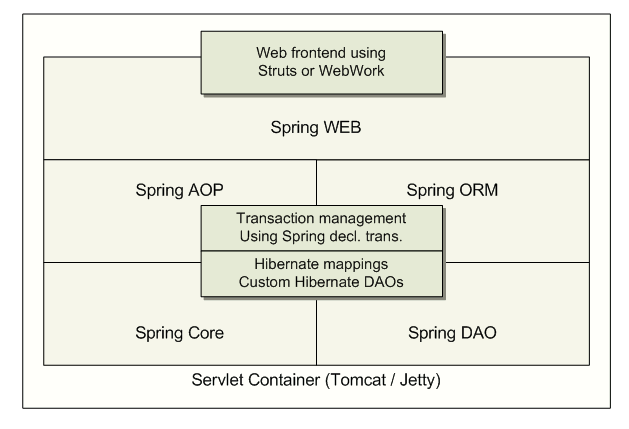
## Các kịch bản sử dụng Spring

### Ứng dụng web điển hình



Ứng dụng web điển hình sử dụng hầu hết đặc trưng của Spring. Tất cả business logic đều có thể được implemented bằng cách dùng các POJOs đơn giản và do Dependency Injection container quản lý. Các dịch vụ phát sinh thêm như gửi email và validation không phụ thuộc vào tầng web thì cho phép bạn lựa chọn nơi để đặt các quy tắc validation. ORM hỗ trợ tương tác với Hibernate, JDO và iBatis. Dùng HibernateDaoSupport bạn có thể tái sử dụng các Hibernate mappings đang tồn tại. Các Form controllers tương tác giữa tầng web và domain model, loại bỏ ActionForms hoặc các classes mà thực hiện việc truyền tham số HTTP cho domain model.

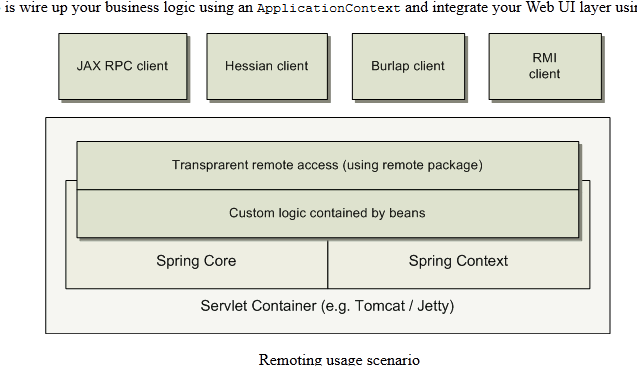
### Sử dụng framework của bên thứ ba



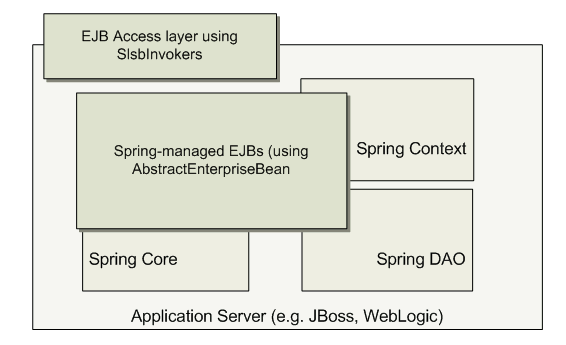
Spring không bắt buộc bạn phải sử dụng tất cả mọi thứ theo những gì nó đưa ra. Các frontends dùng WebWork, Struts, Tapestry, hoặc các UI frameworks khác có thể tương tác tốt với middle-tier dựa trên Spring, cho phép bạn dùng các đặc trưng transaction do Spring đưa ra. Điều bạn cần làm là dùng ApplicationContext để đưa vào các business logic và dùng WebApp-licationContext để tương tác với tầng Web UI.

### Kịch bản remoting

Khi cần truy cập mã lệnh đang tồn tại thông qua webservices, bạn có thể dùng các class Hessian-, Burlap-, Rmi- hoặc JaxRpcProxyFactory của Spring.



### EJBs - Wrapping các POJOs đang tồn tại



Spring cũng cung cấp tầng access và tầng abstraction cho Enterprise JavaBeans. Nó cho phép bạn tái sử dụng các POJOs đang tồn tại và wrap chúng trong Stateless Session Beans.

## Thông tin nền

Hai khái niệm là nền của Spring là: Inversion of Control hoặc Dependency Injection. Phần này theo quan điểm của Martin Fowler.

Các lightweight containers lắp ráp các components từ các projects khác nhau vào một application có liên kết chặt chẽ. Phía dưới các containers này là một pattern chỉ ra cách thực thi kết nối các components. Tên pattern này là "Inversion of Control", tên cụ thể là "Dependency Injection". Ví dụ về lightweight containers là PicoContainer và Spring.

**Thuật ngữ components và services**

Một component là một phần mềm mà các ứng dụng có sử dụng nó chỉ sử dụng mà không thay đổi mã lệnh của nó.

Một service giống một component nhưng nó được dùng cho các ứng dụng foreign. Điểm khác là component được dùng trong nội bộ (như: jar, assembly, dll, hoặc source import). Một service sẽ được dùng từ xa thông qua remote interface (như: web service, messaging system, RPC, hoặc socket).

### Inversion of Control

Ví dụ java truyền thống:

Khai báo lớp Movie chứa thông tin phim:

... class Movie {

...

}

Khai báo lớp MovieLister có phương thức lấy phim theo đạo diễn:

class MovieLister...

private MovieFinder finder;

...

public Movie[] moviesDirectedBy(String arg) {

List allMovies = finder.findAll();

for (Iterator it = allMovies.iterator(); it.hasNext();) {

Movie movie = (Movie) it.next();

if (!movie.getDirector().equals(arg)) it.remove();

}

return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);

}

Ta nói MovieLister dependency vào MovieFinder và Movie. Trong đó, MovieFinder là một interface:

public interface MovieFinder {

List findAll();

}

Khai báo lớp ColonDelimitedMovieFinder implements từ interface MovieFinder:

... class ColonDelimitedMovieFinder implements MovieFinder {

...

}

Lớp này có chức năng phân tách danh sách phim theo dấu phẩy. Khi đó, trong class MovieLister ta có thể thêm constructor:

public MovieLister() {

finder = new ColonDelimitedMovieFinder("movies1.txt");

}

Đối với một người coding thì các mã trên tương đối dễ hiểu. Tuy nhiên, sẽ có vấn đề xảy ra nếu tên phim không phân tách theo dấu phẩy hoặc không lưu trong movies1.txt. Khi đó, phải viết nhiều class để implements từ interface MovieFinder và phải có cách nào đó để lấy được chính xác class nào sẽ sử dụng.

Tình huống này liên tưởng đến Plugin: class implements từ MovieFinder không được liên kết vào lúc biên dịch, MovieLister có thể làm việc với bất kỳ class implement nào và nó được plugged vào sau này. Vấn đề là ta sẽ liên kết ra sao nếu dùng Plugin để lắp vào ứng dụng?

Inversion of Control (IoC) sẽ giải quyết vấn đề này.

Theo cách truyền thống, chương trình sẽ điều khiển. VD: bạn nhập dữ liệu và chương trình xuất ra kết quả xử lý. Inversion: bạn có thể điều khiển luồng xử lý của chương trình.

Vấn đề đặt ra cho inversion là chúng tìm các class implements như thế nào. Ví dụ trên khai báo implementation tường minh. Phương pháp của inversion là cho phép các module sẽ inject implementation của nó vào MovieLister.

IoC nên được đổi tên là Dependency Injection.

### Dependency Injection

Ý tưởng cơ bản của Dependency Injection là có một đối tượng phân biệt, assembler, sẽ populate một field trong MovieLister với một implementation phù hợp của MovieFinder.

Có 3 loại dependency injection: interface injection, setter injection và constructor injection. Với ví dụ trên, ta dùng setter injection với Spring.

Trong lớp MovieLister ta định nghĩa phương thức setting:

class MovieLister...

private MovieFinder finder;

public void setFinder(MovieFinder finder) {

this.finder = finder;

}

Phương thức này sẽ đặt implementation phù hợp của interface MovieFinder cho field finder của MovieLister. Sau đó, ta định nghĩa ColonMovieFinder là implementation của interface MovieFinder - có setting cho filename là tên tập tin chứa tên phim:

class ColonMovieFinder...

private String filename;

public void setFilename(String filename) {

this.filename = filename;

}

Tiếp đến, ta cấu hình cho các tập tin này. Spring hỗ trợ cả XML và coding. Ở đây, ta dùng XML:

<beans>

<bean id="MovieLister" class="spring.MovieLister">

<property name="finder">

<ref local="MovieFinder"/>

</property>

</bean>

<bean id="MovieFinder" class="spring.ColonMovieFinder">

<property name="filename">

<value>movies1.txt</value>

</property>

</bean>

</beans>

Với cấu hình trên, framework sẽ lấy ColonMovieFinder đặt cho field finder của MovieLister. Lấy movies1.txt đặt cho field filename của ColonMovieFinder. Khi cần thay đổi giá trị, ta chỉ cần đổi trong file XML này.

Ví dụ sử dụng mã lệnh:

public void testWithSpring() throws Exception {

ApplicationContext ctx = new FileSystemXmlApplicationContext("spring.xml");

MovieLister lister = (MovieLister) ctx.getBean("MovieLister");

Movie[] movies = lister.moviesDirectedBy("Sergio Leone");

assertEquals("Once Upon a Time in the West", movies[0].getTitle());

}

### Service Locator

Service Locator là một cách khác để thay đổi dependency. Ý tưởng của nó là có một đối tượng biết được làm sao để lấy ra tất cả services mà một ứng dụng cần dùng. Nó có phương thức để trả về một implementation của interface MovieFinder khi lớp đó cần dùng đến. Ta vẫn phải get locator vào trong MovieLister.

## Beans, BeanFactory và ApplicationContext

### Giới thiệu

Hai gói quan trọng của Spring là org.springframework.beans và org.springframework.\_ context - chúng có các đặc trưng của IoC. BeanFactory cung cấp tính năng configuration có khả năng quản lý các beans (objects). Nó cung cấp cơ chế configuration của framework và các tính năng cơ bản. ApplicationContext xây dựng trên nền BeanFactory và thêm vào các tính năng như: tương tác với Springs AOP, xử lý message resource, truyền event, có cơ chế khai báo để tạo ApplicationContext hay WebApplicationContext.

Thông thường với ứng dụng J2EE, ApplicationContext được dùng thay vì BeanFactory. BeanFactory chỉ nên dùng khi bạn không cần dùng các tính năng nâng cao của ApplicationContext.

### Cơ bản về BeanFactory và BeanDefinitions

#### BeanFactory

BeanFactory là container mà instantiates, configures và manages các beans. Các beans cộng tác và phụ thuộc (dependency) với nhau. BeanFactory dùng dữ liệu configuration để phản ánh sự phụ thuộc này (một số phụ thuộc không phản ánh trong configuration nhưng lại là tính năng của các beans lúc runtime).

Interface org.springframework.beans.factory.BeanFactory đại diện cho một BeanFactory - và có thể có nhiều implementations. Một implementation thường dùng là org.springframe\_ work.beans.factory.xml.XmlBeanFactory.

Instance một BeanFactory:

InputStream is = new FileInputStream("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(is);

hoặc

ClassPathResource res = new ClassPathResource("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(res);

hoặc

ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(

new String[] {"applicationContext.xml", "applicationContext-part2.xml"});

// of course, an ApplicationContext is just a BeanFactory

BeanFactory factory = (BeanFactory) appContext;

Spring Framework sẽ instantiate BeanFactory. Ví dụ: với ứng dụng web J2EE, tầng web hỗ trợ mã lệnh để load Spring ApplicationContext tự động khi khởi động ứng dụng.

Dữ liệu configuration của BeanFactory sẽ định nghĩa các beans mà BeanFactory phải quản lý. Trong XmlBeanFactory, từng bean được cấu hình bằng thẻ <bean> bên trong thẻ <beans> như VD sau:

<?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="..." class="...">

...

</bean>

<bean id="..." class="...">

...

</bean>

...

</beans>

#### BeanDefinition

Bean định nghĩa trong DefaultListableBeanFactory (giống XmlBeanFactory) được hiểu là BeanDefinition. Nó chứa các điểm sau:

* Class name: là lớp implementation của bean. Tuy nhiên, nếu bean được constructed qua phương thức static factory thì class name là tên lớp của factory class.
* Các thành phần cấu hình bean behavioral: diễn tả bean sẽ behave ra sao trong container (VD: prototype hoặc singleton, kiểu autowiring, kiểu kiểm tra dependency, phương thức initialization và destruction).
* Giá trị của các tham số và thuộc tính khởi tạo được thiết lập khi tạo bean mới. VD: số kết nối của connection pool, giới hạn kích thước pool.
* Các beans khác mà một bean cần đến chúng để làm việc. Chúng được gọi là collaborators hay dependencies.

Một số nhóm element khác:

* class
* id và name
* singleton hoặc prototype
* constructor arguments
* bean properties
* autowiring mode
* dependency checking mode
* initialization method
* destruction method

Interface org.springframework.beans.factory.config.BeanDefinition đại điện cho BeanDefinition.

BeanFactory có thể đăng ký các bean instances đang tồn tại. DefaultListableBeanFactory hỗ trợ việc này thông qua phương thức registerSingleton.

#### Class bean

Phần này đề cập đến thuộc tính class của bean.

Thông thường, khi BeanFactory tạo bean thông qua gọi constructor thì class chỉ đến tên lớp của bean được constructed. Khi BeanFactory gọi phương thức static factory để tạo bean thì class chỉ đến tên lớp chứa phương thức đó.

**Tạo bean qua constructor**

BeanFactory không giới hạn ở việc chỉ quản lý các JavaBeans, nó có thể quản lý bất cứ lớp nào bạn muốn. Có thể có các lớp non-bean-style trong BeanFactory.

Dùng XmlBeanFactory để chỉ định bean như sau:

<bean id="exampleBean"

class="examples.ExampleBean"/>

<bean name="anotherExample"

class="examples.ExampleBeanTwo"/>

**Tạo bean qua phương thức static factory**

Theo cách này, bạn chỉ định cả thuộc tính class và factory-method. Spring sẽ gọi phương thức này để trả về bean. Dữ liệu definition không định nghĩa kiểu trả về của phương thức.

<bean id="exampleBean"

class="examples.ExampleBean2"

factory-method="createInstance"/>

**Tạo bean qua phương thức instance factory**

Tương tự như tạo bean qua static factory chỉ khác về kiểu static/non-static của phương thức được gọi. Dùng cách này sẽ không chỉ định thuộc tính class. Đồng thời thuộc tính factory-bean phải chỉ đến một bean khác.

<!-- The factory bean, which contains a method called createInstance -->

<bean id="myFactoryBean"

class="...">

...

</bean>

<!-- The bean to be created via the factory bean -->

<bean id="exampleBean"

factory-bean="myFactoryBean"

factory-method="createInstance"/>

Factory bean có thể được managed và configured thông qua Dependency Injection bởi container.

#### Các định danh bean identifiers (id và name)

Mọi bean có một/nhiều ids (cũng gọi là identifiers hay names). Các ids này phải duy nhất trong BeanFactory/ApplicationContext chứa bean. Một bean chỉ có một id, nếu có nhiều hơn thì là aliases.

Trong XmlBeanFactory (bao gồm ApplicationContext variants), bạn dùng id/name để xác định bean id(s), và ít nhất một id phải được chỉ định. Thuộc tính id cho phép bạn xác định một id và nó giống như thuộc tính ID của XML. Dùng thuộc tính name để chỉ định một hoặc nhiều ids (aliases) phân biệt bằng dấu phẩy hoặc chấm phẩy.

#### To singleton or not to singleton

Beans được định nghĩa để deployed theo một trong hai modes: singleton hoặc non-singleton (cũng gọi là prototype). Khi bean là một singleton (mặc định), chỉ một shared instance của bean được quản lý và tất cả mọi requests tới bean qua id/ids khi phù hợp cũng chỉ trả về một bean instance cụ thể. Với non-singleton (prototype), khi một request tới bean được thực hiện sẽ tạo ra một bean instance mới để trả về.

Chỉ nên dùng mode non-singleton (prototype) khi thực cần thiết. Ví dụ:

<bean id="exampleBean"

class="examples.ExampleBean" singleton="false"/>

<bean name="yetAnotherExample"

class="examples.ExampleBeanTwo" singleton="true"/>

Theo định nghĩa, Spring không thể quản lý việc hoàn thành vòng đời của một non-singleton/prototype bean từ sau khi nó được tạo ra. Nó được chuyển cho client và container không giữ track của nó nữa.

### Properties, collaborators, autowiring và dependency checking

#### Thiết lập bean properties và collaborators

Beans định nghĩa các dependencies thông qua constructor arguments. Các arguments cho phương thức factory hoặc properties được đặt cho instance của đối tượng sau khi nó được constructed hoặc được trả về từ phương thức factory. Container sẽ inject các dependencies này vào khi tạo bean. Việc này là cơ bản của inverse (IoC) của bean instantianceting, hoặc một số thứ giống với mẫu Service Locator.

Inversion of Control/Dependency Injection (IoC/DI) tồn tại trong 2 variants chính:

* **setter-based**: DI được thực hiện bằng cách gọi setters trên các beans sau khi gọi một phương thức no-argument constructor hoặc no-argument static factory để instantiate beans. Các beans đã định nghĩa trong BeanFactory và được dùng ở đây là true JavaBeans (có no-argument constructor, getters, setters).
* **constructor-based**: DI được thực hiện bằng cách gọi một constructor với một số arguments, từng đối số đại diện cho một collaborator hay property. Thêm vào là gọi một phương thức static factory với các arguments cụ thể để khởi tạo bean. Spring ủng hộ việc sử dụng setter-based.

BeanFactory hỗ trợ cả hai variants này để inject dependencies vào trong beans do nó quản lý. Dữ liệu configuration cho các dependencies đặt dưới hình thức của BeanDefinition, được dùng với JavaBeans PropertyEditors để biết cách chuyển định dạng của properties.

Resolution của bean dependency như sau:

* BeanFactory được tạo mới và khởi tạo với configuration cho tất cả beans. Thường dùng một BeanFactory hoặc ApplicationContext variant có hỗ trợ XML.
* Từng bean có dependencies dưới hình thức properties, constructor arguments hoặc arguments của phương thức static-factory khi nó được dùng thay cho constructor. Các dependencies này được cung cấp cho bean khi nó thực sự được tạo ra.
* Từng property hoặc constructor arguments hoặc là có định nghĩa giá trị hoặc là một tham chiếu đến một bean khác trong BeanFactory. Nếu là ApplicationContext, tham chiếu có thể chỉ đến một bean trong một ApplicationContext cha.
* Từng property hoặc constructor argument với giá trị của nó phải được chuyển kiểu phù hợp.
* Spring validates dữ liệu configuration của từng bean trong BeanFactory khi BeanFactory được tạo ra, bao gồm cả các thuộc tính mà bean tham chiếu. Tuy nhiên, bean properties không được thiết lập cho tới khi bean thực sự được tao ra.
* Spring có thể đưa ra các vấn đề của configuration như tham chiếu đến một non-existent beans và phụ thuộc lòng vòng tại load-time của BeanFactory. BeanFactory khi đã loaded thành công vẫn có thể phát sinh lỗi về sau nếu có vấn đề về tạo beans hay do các dependencies.

Một số ví dụ.

Đầu tiên, dùng BeanFactory theo setter-based DI, bao gồm một phần XmlBeanFactory configuration và mã lệnh hiển thị khai báo setters.

<bean id="exampleBean" class="examples.ExampleBean">

<property name="beanOne"><ref bean="anotherExampleBean"/></property>

<property name="beanTwo"><ref bean="yetAnotherBean"/></property>

<property name="integerProperty"><value>1</value></property>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

Mã lệnh

public class ExampleBean {

private AnotherBean beanOne;

private YetAnotherBean beanTwo;

private int i;

public void setBeanOne(AnotherBean beanOne) {

this.beanOne = beanOne;

}

public void setBeanTwo(YetAnotherBean beanTwo) {

this.beanTwo = beanTwo;

}

public void setIntegerProperty(int i) {

this.i = i;

}

}

Ví dụ dùng BeanFactory cho IoC loại 3, bao gồm XML configuration chỉ định constructor arguments và mã lệnh hiển thị constructor:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>

<constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>

<constructor-arg type="int"><value>1</value></constructor-arg>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

Mã lệnh

public class ExampleBean {

private AnotherBean beanOne;

private YetAnotherBean beanTwo;

private int i;

public ExampleBean(AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

this.beanOne = anotherBean;

this.beanTwo = yetAnotherBean;

this.i = i;

}

}

Arguments được dùng để chuyển đối số vào cho constructor.

Giờ là ví dụ về variant Spring gọi để trả về một instance của đối tượng:

<bean id="exampleBean" class="examples.ExampleBean"

factory-method="createInstance">

<constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>

<constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>

<constructor-arg><value>1</value></constructor-arg>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

Mã lệnh

public class ExampleBean {

...

// a private constructor

private ExampleBean(...) {

...

}

// a static factory method

// the arguments to this method can be considered the dependencies of the bean that

// is returned, regardless of how those arguments are actually used.

public static ExampleBean createInstance(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

ExampleBean eb = new ExampleBean(...);

// some other operations

...

return eb;

}

}

#### Constructor Argument Resolution

Việc so khớp đối số của Constructor xảy ra dựa vào loại của đối số. Khi dùng kiểu đơn giản, như <value>true<value>, Spring không thể xác định loại của giá trị.

Ví dụ bean:

package examples;

public class ExampleBean {

private int years; //No. of years to the calculate the Ultimate Answer

private String ultimateAnswer; //The Answer to Life, the Universe, and Everything

public ExampleBean(int years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

**So khớp kiểu của Constructor Argument**

Sử dụng kiểu đơn giản kết hợp chỉ rõ kiểu của constructor argument:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int"><value>7500000</value></constructor-arg>

<constructor-arg type="java.lang.String"><value>42</value></constructor-arg>

</bean>

**Chỉ số của Constructor Argument**

Dùng thuộc tính chỉ số (bắt đầu từ 0) để chỉ định rõ vị trí đối số:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0"><value>7500000</value></constructor-arg>

<constructor-arg index="1"><value>42</value></constructor-arg>

</bean>

IoC thích sử dụng kiểu chỉ số hơn để tránh nhập nhằng trường hợp có nhiều đối số cùng kiểu dữ liệu.

#### Chi tiết bean properties và constructor arguments

Hai mục này được định nghĩa hoặc tham chiếu đến các beans khác hoặc giá trị. XmlBean\_ Factory hỗ trợ qua một số loại sub-element trong property của nó và constructor argument elements.

Element value chỉ ra giá trị của property hay constructor argument dưới định dạng chuỗi. JavaBeans PropertyEditors chuyển các chuỗi này sang loại thực sự của property hay argument.

Ví dụ:

<beans>

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<!-- results in a setDriverClassName(String) call -->

<property name="driverClassName">

<value>com.mysql.jdbc.Driver</value>

</property>

<property name="url">

<value>jdbc:mysql://localhost:3306/mydb</value>

</property>

</bean>

</beans>

Spring xem empty arguments của property là chuỗi rỗng.

<bean class="ExampleBean">

<property name="email"><value></value></property>

</bean>

Giá trị trên tương đương lệnh java: exampleBean.setEmail("").

Khi muốn truyền giá trị null bạn phải dùng element null.

<bean class="ExampleBean">

<property name="email"><null/></property>

</bean>

Giá trị trên tương đương lệnh java: exampleBean.setEmail(null).

Các element list, set, map, props cho phép các properties và arguments có kiểu java là List, Set, Map, Properties. VD:

<beans>

...

<bean id="moreComplexObject" class="example.ComplexObject">

<!-- results in a setPeople(java.util.Properties) call -->

<property name="people">

<props>

<prop key="HarryPotter">The magic property</prop>

<prop key="JerrySeinfeld">The funny property</prop>

</props>

</property>

<!-- results in a setSomeList(java.util.List) call -->

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource"/>

</list>

</property>

<!-- results in a setSomeMap(java.util.Map) call -->

<property name="someMap">

<map>

<entry key="yup an entry">

<value>just some string</value>

</entry>

<entry key="yup a ref">

<ref bean="myDataSource"/>

</entry>

</map>

</property>

<!-- results in a setSomeSet(java.util.Set) call -->

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource"/>

</set>

</property>

</bean>

</beans>

Element bean nằm trong element property được dùng để định nghĩa giá trị cho bean inline, không tham chiếu đến bean khác. Inline bean không cần định nghĩa id.

<bean id="outer" class="...">

<!-- Instead of using a reference to target, just use an inner bean -->

<property name="target">

<bean class="com.mycompany.PersonImpl">

<property name="name"><value>Tony</value></property>

<property name="age"><value>51</value></property>

</bean>

</property>

</bean>

Element idref dùng để đặt một property tham chiếu đến id/name của một bean khác. VD:

<bean id="theTargetBean" class="...">

</bean>

<bean id="theClientBean" class="...">

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

Lúc runtime, nó tương đương với:

<bean id="theTargetBean" class="...">

</bean>

<bean id="theClientBean" class="...">

<property name="targetName">

<value>theTargetBean</value>

</property>

</bean>

Element ref dùng đặt thuộc tính tham chiếu tới một bean khác. Giá trị của thuộc tính bean hoặc giống id hoặc giống một trong số các name của target bean. VD:

<ref bean="someBean"/>

Dùng thuộc tính local của element ref để XML parser sẽ validate XML id trong cùng file. Giá trị của thuộc tính local phải giống id của target bean.

<ref local="someBean"/>

Dùng thuộc tính parent của element ref để tham chiếu đến bean trong parent BeanFactory/\_ ApplicationContext của BeanFactory/ApplicationContext hiện hành. Giá trị của thuộc tính này có thể là id hoặc một trong số các name.

<ref parent="someBean"/>

#### Method Injection

Thông thường, khi hoặc hai bean singleton hoặc hai bean non-singleton cần cộng tác với nhau thì sẽ định nghĩa một bean là property của bean kia. Tuy nhiên, sẽ có vấn đề vì vòng đời của các beans khác nhau.

Method Injection, một đặc điểm tiện lợi của BeanFactory, cho phép thực hiện điều này.

**Lookup method Injection**

Lookup method injection là khả năng container ghi đè các phương thức abstract hoặc concrete trên các beans trong container, để trả về kết quả tìm kiếm từ một bean khác trong container. Thông thường, lookup sẽ là một non-singleton bean. Spring thực hiện thông qua việc phát sinh động một subclass ghi đè phương thức đó, dùng thư viện CGLIB để phát sinh bytecode.

Class chứa method được injected phải định nghĩa phương thức (abstract/concrete) theo dạng:

protected abstract SingleShotHelper createSingleShotHelper();

Nếu phương thức không abstract thì Spring sẽ override implementation đang tồn tại. VD:

<!-- a stateful bean deployed as a prototype (non-singleton) -->

<bean id="singleShotHelper" class="..." singleton="false">

</bean>

<!-- myBean uses singleShotHelper -->

<bean id="myBean" class="...">

<lookup-method name="createSingleShotHelper"

bean="singleShotHelper"/>

<property>

...

</property>

</bean>

Bean myBean sẽ gọi phương thức createSingleShotHelper khi nó cần một instance mới của singleShotHelper bean. Bean singleShotHelper phải được deploy theo mode non-singleton. Nếu nó deploy theo mode singleton thì cùng một instance của singleShotHelper được trả về.

Lookup method injection có thể kết hợp với Constructor Injection và Setter Injection.

**Thay thế phương thức tùy ý**

Cách ít thông dụng hơn Lookup Method Injection là thay thế các phương thức tùy ý trong bean với một method implementation khác.

Trong XmlBeanFactory dùng element replaced-method để thực hiện việc này. VD:

public class MyValueCalculator {

public String computeValue(String input) {

... some real code

}

... some other methods

}

Class implementing từ interface org.springframework.beans.factory.support.MethodReplacer cần cung cấp định nghĩa phương thức mới:

/\*\* meant to be used to override the existing computeValue

implementation in MyValueCalculator \*/

public class ReplacementComputeValue implements MethodReplacer {

public Object reimplement(Object o, Method m, Object[] args) throws Throwable {

// get the input value, work with it, and return a computed result

String input = (String) args[0];

...

return ...;

}

BeanFactory deploy definition và chỉ định phương thức override như sau:

<bean id="myValueCalculator class="x.y.z.MyValueCalculator">

<!-- arbitrary method replacement -->

<replaced-method name="computeValue" replacer="replacementComputeValue">

<arg-type>String</arg-type>

</replaced-method>

</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplaceMentComputeValue">

</bean>

Element replaced-method có thể dùng một/nhiều element arg-type.

#### Sử dụng depends-on

Một bean là dependency của một bean khác được diễn giải bằng cách đặt bean là property của bean kia. Thực hiện việc này bằng cách dùng element ref trong XmlBeanFactory.

Element depends-on có thể dùng để bắt buộc một/nhiều bean phải được khởi tạo trước khi bean sử dụng element này khởi tạo. VD:

<bean id="beanOne" class="ExampleBean" depends-on="manager">

<property name="manager"><ref local="manager"/></property>

</bean>

<bean id="manager" class="ManagerBean"/>

#### Autowiring collaborators

BeanFactory có thể autowire mối quan hệ giữa các collaborating beans. Nghĩa là để Spring giải quyết các collaborators bằng cách kiểm tra nội dung của BeanFactory. Autowiring có 5 mode:

* no: (default) không có autowiring. Các tham chiếu bean phải định nghĩa qua element ref.
* byName: autowiring theo tên property. Chương trình sẽ duyệt BeanFactory và tìm bean có tên chính xác theo tên property cần được autowired.
* byType: cho phép một property được autowired nếu có một bean của property type này trong BeanFactory. Nếu có nhiều hơn một thì sẽ bị lỗi.
* constructor: tương tự byType nhưng áp dụng cho constructor arguments. Nếu không có chính xác một bean của constructor argument type trong bean factory thì sẽ có lỗi.
* autodetect: tự bean class sẽ chọn constructor hoặc byType. Nếu tìm thấy một default constructor thì sẽ chọn byType.

Các dependencies tường minh trong property và constructor argument luôn ghi đè autowiring. Autowire có thể được kết hợp với dependency checking.

Một số điểm tiện lợi của autowiring:

* Nó có thể giảm volume của configuration required.
* Nó có thể làm cho configuration được up to date.

Một số điểm bất lợi của autowiring:

* Nó mơ hồ hơn explicit wiring.
* Thông tin wiring có thể không có sẵn cho các công cụ phát sinh tài liệu từ Spring application context.
* Autowiring byType chỉ làm việc khi có single bean definition của loại đó được chỉ định bởi phương thức setter hoặc constructor argument.

#### Checking for dependencies

Spring có khả năng kiểm tra sự tồn tại của các unresolved dependencies của một bean được deployed vào BeanFactory. Đây là các JavaBeans properties của bean.

Đặc điểm này hữu dụng khi bạn muốn chắc chắn tất cả properties được thiết lập trên một bean. Dependency checking có thể enabled/disabled đối với bean. Mặc định là không kiểm tra dependencies. Dependency checking có nhiều modes. Trong XmlBeanFactory chỉ định kiểm tra thông qua thuộc tính dependency-check với một số mode sau:

* none: không kiểm tra dependency.
* simple: kiểm tra dependency cho các primitive types và collections.
* object: kiểm tra cho collaborators.
* all: kiểm tra cho cả collaborators và primitive.

### Customizing the nature of a bean

#### Lifecycle interfaces

Spring cung cấp marker interfaces InitializingBean và DisposableBean để cho phép thay đổi đặc tính bean trong BeanFactory. Spring dùng BeanPostProcessors để xử lý marker interfaces mà nó tìm thấy và gọi các phương thức phù hợp. Bạn có thể implement BeanPostProcessors để custom các features hoặc lifecycle của Spring.

##### InitializingBean: init-method

Implementing interface org.springframework.beans.factory.InitializingBean cho phép một bean thực thi công việc khởi tạo sau khi tất cả properties cần thiết của bean được BeanFactory thiết lập. Interface này chỉ có một phương thức:

/\*

\* Invoked by a BeanFactory after it has set all bean properties supplied

\* (and satisfied BeanFactoryAware and ApplicationContextAware).

\* <p>This method allows the bean instance to perform initialization only

\* possible when all bean properties have been set and to throw an

\* exception in the event of misconfiguration.

\* @throws Exception in the event of misconfiguration (such

\* as failure to set an essential property) or if initialization fails.

\*/

void afterPropertiesSet() throws Exception;

##### DisposableBean : destroy-method

Implementing interface org.springframework.beans.factory.DisposableBean cho phép bean lấy về một callback khi BeanFactory chứa bean bị hủy. Interface này chỉ có một method:

/\*\*

\* Invoked by a BeanFactory on destruction of a singleton.

\* @throws Exception in case of shutdown errors.

\* Exceptions will get logged but not re-thrown to allow

\* other beans to release their resources too.

\*/

void destroy() throws Exception;

#### Knowing who you are

##### BeanFactoryAware

Class implements interface org.springframework.beans.factory.BeanFactoryAware được cung cấp một tham chiếu đến BeanFactory tạo ra nó khi nó được tạo.

public interface BeanFactoryAware {

/\*\*

\* Callback that supplies the owning factory to a bean instance.

\* <p>Invoked after population of normal bean properties but before an init

\* callback like InitializingBean's afterPropertiesSet or a custom init-method.

\* @param beanFactory owning BeanFactory (may not be null).

\* The bean can immediately call methods on the factory.

\* @throws BeansException in case of initialization errors

\* @see BeanInitializationException

\*/

void setBeanFactory(BeanFactory beanFactory) throws BeansException;

}

##### BeanNameAware

Nếu một bean implements từ interface org.springframework.beans.factory.BeanNameAware và deployed vào một BeanFactory thì BeanFactory sẽ gọi bean đó thông quan interface.

#### FactoryBean

Interface org.springframework.beans.factory.FactoryBean cung cấp 3 methods:

Object getObject()

Method trả về một instance của đối tượng mà factory tạo ra.

boolean isSingleton()

Method trả về true nếu FactoryBean trả về singletons, ngược lại là false.

Class getObjectType()

Trả về kiểu của đối tượng của hàm getObject() hoặc null.

### Abstract and child bean definitions

Một bean definition chứa những thông tin cụ thể, constructor arguments và property values. Một child bean definition cũng là một bean definition mà kế thừa dữ liệu configuration từ parent definition. Nó có thể override một số giá trị. Đây là một hình thức templating.

Với BeanFactory các child bean definitions được diễn tả qua lớp ChildBeanDefinition. VD về child bean trong XmlBeanFactory bean definition:

<bean id="inheritedTestBean" abstract="true" class="org.springframework.beans.TestBean">

<property name="name"><value>parent</value></property>

<property name="age"><value>1</value></property>

</bean>

<bean id="inheritsWithDifferentClass" class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBean" init-method="initialize">

<property name="name"><value>override</value></property>

<!-- age should inherit value of 1 from parent -->

</bean>

Child bean definition sẽ dùng bean class từ parent definition nếu không chỉ định và override nếu chỉ định. Child bean definition sẽ kế thừa các giá trị constructor argument, property và method từ parent và có option để thêm giá trị mới. Nếu có chỉ định init/destroy method, static factory method thì sẽ override các method phù hợp của parent.

Các settings sau sẽ luôn lấy từ child definition: depends on, autowire mode, dependency check, singleton, lazy init.

VD về không chỉ định class cho bean:

<bean id="inheritedTestBeanWithoutClass">

<property name="name"><value>parent</value></property>

<property name="age"><value>1</value></property>

</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBeanWithoutClass" init-method="initialize">

<property name="name"><value>override</value></property>

<!-- age should inherit value of 1 from parent -->

</bean>

### Tương tác với BeanFactory

BeanFactory có khả năng giữ lại một registry của các beans khác nhau và các dependencies của chúng. BeanFactory cho phép bạn đọc bean definitions và truy cập tới chúng.

VD:

InputStream is = new FileInputStream("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(is);

Dùng getBean(String) để lấy instances của các beans. Interface BeanFactory chỉ có 5 methods để client gọi:

boolean containsBean(String)

Method trả về true nếu BeanFactory chứa một bean definition hoặc bean instance mà khớp với tên đưa vào.

Object getBean(String)

Method trả về một instance của bean được đăng ký theo tên đưa vào. Có lỗi xảy ra nếu không tìm thấy bean hoặc việc instantiating và preparing bean có vấn đề.

Object getBean(String, Class)

Trả về bean theo tên đưa vào. Bean sẽ được cast qua Class. Nếu bean không thể cast thì sẽ văng lỗi.

boolean isSingleton(String)

Xác định bean đăng ký theo mode singleton hay prototype. Nếu không tìm thấy bean cũng sẽ văng lỗi.

String[] getAliases(String)

Trả về aliases của bean theo tên đưa vào.

Đôi khi có thể cần lấy instance của chính BeanFactory chứ không phải bean do nó tạo ra. Làm việc này bằng cách gắn & trước bean id khi gọi method getBean của BeanFactory (bao gồm cả ApplicationContext). VD: bean có id là myBean, gọi getBean("&myBean") sẽ trả về chính FactoryBean instance.

### Customizing beans with BeanPostprocessors

Một bean post-processor là một lớp java implements từ interface org.springframework.beans. factory.config.BeanPostProcessor, có 2 callback methods. Một bean post-processor sẽ kiểm tra marker interfaces, hoặc wrap một bean với một proxy.

Một ApplicationContext sẽ tự phát hiện các beans được deployed mà implement interface BeanPostProcessor, đăng ký chúng là post-processors, để gọi lúc thích hợp. Với BeanFactories, bean post-processors phải được đăng ký manually một cách tường minh. VD:

ConfigurableBeanFactory bf = new .....; // create BeanFactory

... // now register some beans

// now register any needed BeanPostProcessors

MyBeanPostProcessor pp = new MyBeanPostProcessor();

bf.addBeanPostProcessor(pp);

// now start using the factory

...

Khuyến khích dùng ApplictionContexts cho bean post-processors khi cần dùng.

### Customizing bean factories with BeanFactoryPost\_ processors

Là một java class implements từ interface org.springframework.beans.factory.config.Bean\_ FactoryPostProcessor. Nó được thực thi manually (trường hợp BeanFactory) hoặc automatically (trường hợp ApplicationContext) để áp dụng các thay đổi cho toàn bộ BeanFactory. Spring có một số bean factory post-processors như PropertyResourceConfigurer, PropertyPlaceHolder-Configurer và BeanNameAutoProxyCreator.

VD trường hợp BeanFactory:

XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));

// create placeholderconfigurer to bring in some property

// values from a Properties file

PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();

cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement

cfg.postProcessBeanFactory(factory);

ApplicationContext sẽ phát hiện beans implement từ interface BeanFactoryPostProcessor được deployed và tự động dùng chúng như bean factory post-processors lúc thích hợp.

Khuyến khích dùng ApplicationContext cho bean factory post-processors khi cần.

#### PropertyPlaceholderConfigurer

Được dùng để externalize một số property values từ BeanFactory definition vào một file khác theo định dạng Java Properties.

VD dưới định nghĩa datasource có cấu hình một vài thuộc tính từ file Properties. Lúc runtime, ta dùng PropertyPlaceholderConfigurer để lấy giá trị:

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName"><value>${jdbc.driverClassName}</value></property>

<property name="url"><value>${jdbc.url}</value></property>

<property name="username"><value>${jdbc.username}</value></property>

<property name="password"><value>${jdbc.password}</value></property>

</bean>

Giá trị được thiết lập trong file Properties (jdbc.properties):

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

Bean factory post-processor thực thi manually như sau:

XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));

PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();

cfg.setLocation(new FileSystemResource("jdbc.properties"));

cfg.postProcessBeanFactory(factory);

ApplicationContexts tự nhận ra và áp dụng các beans được deployed mà implement BeanFactoryPostProcessor. Khuyến khích dùng ApplicationContext thay cho BeanFactory.

PropertyPlaceHolderConfigurer cũng có thể dùng giá trị trong Java System properties. Dùng thuộc tính systemPropertiesMode để thực hiện việc này.

#### PropertyOverrideConfigurer

Nó tương tự PropertyPlaceholderConfigurer. Các original definitions có thể có/không thiết lập giá trị mặc định cho bean properties. Nếu file Properties không có entry cho một bean property thì definition mặc định được sử dụng.

Nếu có nhiều PropertyOverrideConfigurers cùng định nghĩa giá trị cho cùng bean property thì sẽ lấy định nghĩa cuối cùng. Định dạng cấu hình thuộc tính như sau:

beanName.property=value

Ví dụ:

dataSource.driverClassName=com.mysql.jdbc.Driver

dataSource.url=jdbc:mysql:mydb

### Registering additional custom PropertyEditors

Khi thiết lập giá trị dạng chuỗi cho bean properties, BeanFactory dùng JavaBeans PropertyEditors để convert chuỗi sang type của property.

### Giới thiệu ApplicationContext

Bean package cung cấp chức năng cơ bản để quản lý và thao tác beans. Context package dùng ApplicationContext. Cơ bản của context package là interface ApplicationContext, đặt trong org.springframework.context. Nó cung cấp tất cả chức năng của BeanFactory. Để làm việc hướng framework, dùng layering và hierarchical contexts, context package cung cấp:

* MessageSource cung cấp cách truy cập messages theo kiểu i18n.
* Truy cập resources, như URLs và files.
* Event propagation để beans implementing từ ApplicationListener interface.
* Nạp nhiều (hierarchical) contexts, cho phép tập trung vào một layer riêng biệt.

Khuyến khích dùng ApplicationContext thay cho BeanFactory trừ phi thật cần thiết.

### Added functionality of the ApplicationContext

ApplicationContext có một số chức năng phân biệt khỏi BeanFactory.

#### Dùng MessageSource

Interface ApplicationContext extends interface MessageSource và cung cấp chức năng messaging i18n. NestingMessageSource có khả năng giải quyết hierarchical messages. Đây là các basic interfaces do Spring cung cấp để làm message. Các methods gồm:

String getMessage (String code, Object[] args, String default, Locale loc)

Dùng lấy một message từ MessageSource. Khi không tìm thấy message sẽ trả về default message. Chuỗi default phải dùng chức năng MessageFormat để định dạng.

String getMessage (String code, Object[] args, Locale loc)

Cũng dùng để lấy về message nhưng nếu không tìm thấy sẽ văng lỗi NoSuchMessageException.

String getMessage(MessageSourceResolvable resolvable, Locale locale)

Cũng dùng để lấy message.

Khi ApplicationContext loaded, nó tự tìm MessageSource bean - tên messageSource - định nghĩa trong context. Nếu không tìm thấy thì nó sẽ tìm trong parent containing. Nếu cuối cùng vẫn không tìm thấy thì một empty StaticMessageSource sẽ được instantiated.

Spring cung cấp hai MessageSource implementations là ResourceBundleMessageSource và StaticMessageSource. Cả 2 đều implement NestingMessageSource. StaticMessageSource cung cấp cách thêm messages vào source. VD về ResourceBundleMessageSource:

<beans>

<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>format</value>

<value>exceptions</value>

<value>windows</value>

</list>

</property>

</bean>

</beans>

#### Propagating events

Class ApplicationEvent và interface ApplicationListener dùng để handle các events trong ApplicationContext. Nếu một bean implements từ interface ApplicationListener được deployed vào context thì mọi lần ApplicationEvent published tới ApplicationContext thì bean đó sẽ được notified. Spring cung cấp 3 events chuẩn:

* ContextRefreshedEvent: event được published khi ApplicationContext được initialized hay refreshed.
* ContextClosedEvent: event được published khi ApplicationContext được closed.
* RequestHandledEvent: một web-specific event thông báo cho tất cả beans rằng một HTTP request được serviced (nó được published sau khi request được finished). Chỉ dùng cho ứng dụng web dùng Spring's DispatcherServlet.

Dùng method publishEvent() để implementing các custom events.

VD ApplicationContext:

<bean id="emailer" class="example.EmailBean">

<property name="blackList">

<list>

<value>black@list.org</value>

<value>white@list.org</value>

<value>john@doe.org</value>

</list>

</property>

</bean>

<bean id="blackListListener" class="example.BlackListNotifier">

<property name="notificationAddress">

<value>spam@list.org</value>

</property>

</bean>

Các beans thực tế như sau:

public class EmailBean implements ApplicationContextAware {

/\*\* the blacklist \*/

private List blackList;

public void setBlackList(List blackList) {

this.blackList = blackList;

}

public void setApplicationContext(ApplicationContext ctx) {

this.ctx = ctx;

}

public void sendEmail(String address, String text) {

if (blackList.contains(address)) {

BlackListEvent evt = new BlackListEvent(address, text);

ctx.publishEvent(evt);

return;

}

// send email

}

}

public class BlackListNotifier implement ApplicationListener {

/\*\* notification address \*/

private String notificationAddress;

public void setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

public void onApplicationEvent(ApplicationEvent evt) {

if (evt instanceof BlackListEvent) {

// notify appropriate person

}

}

}

#### Dùng resources bên trong Spring

Nhiều ứng dụng cần truy cập resources. Interface ApplicationContext dùng method getResource(String) để làm việc này. Class Resource định nghĩa một số methods có chia sẻ cho tất cả Resource implementations:

|  |  |
| --- | --- |
| **Method** | **Giải thích** |
| getInputStream() | Mở và trả về InputStream cho resource. |
| exists() | Kiểm tra resource tồn tại hay không. |
| isOpen() |  |
| getDescription() | Trả về mô tả của resource. |

Dùng ResourceEditor để cấu hình resource.

### Customized behavior in the ApplicationContext

BeanFactory điều khiển được vòng đời của các beans được deployed trong nó. Configuration của beans tương đương thuộc tính init-method và destroy-method trong một XmlBeanFactory. Trong ApplicationContext, những cấu hình này vẫn hoạt động nhưng có thể customize được behavior của beans và container.

#### ApplicationContextAware marker interface

Tất cả marker interfaces mà BeanFactories có sẵn vẫn làm việc được trong ApplicationContext. ApplicationContext thêm một marker interface - org.springframework.context.Application-ContextAware - mà beans có thể implement. Một bean implements từ interface này và được deployed vào trong context thì sẽ có tham chiếu đến context để có thể tương tác với context.

#### BeanPostProcessor

Bean post-processors là Java class implement từ interface org.springframework.beans.factory.-config.BeanPostProcessor.

#### BeanFactoryPostProcessor

Bean factory post-processors là Java class implement từ interface org.springframework.beans.-factory.config.BeanFactoryPostProcessor.

#### PropertyPlaceholderConfigurer

Nó cũng được đề cập phía trên.

### Registering additional custom PropertyEditors

Đã đề cập phía trên, PropertyEditors dùng convert các property values. Bean factory post-processor CustomEditorConfigurer dùng register PropertyEditors tới một ApplicationContext.

VD hai class ExoticType và DependsOnExoticType:

public class ExoticType {

private String name;

public ExoticType(String name) {

this.name = name;

}

}

public class DependsOnExoticType {

private ExoticType type;

public void setType(ExoticType type) {

this.type = type;

}

}

ExoticType là một property của DependsOnExoticType. Khi gán giá trị:

<bean id="sample" class="example.DependsOnExoticType">

<property name="type"><value>aNameForExoticType</value></property>

</bean>

Ta phải dùng PropertyEditor để convert giá trị sang type của property:

// converts string representation to ExoticType object

public class ExoticTypeEditor extends PropertyEditorSupport {

private String format;

public void setFormat(String format) {

this.format = format;

}

public void setAsText(String text) {

if (format != null && format.equals("upperCase")) {

text = text.toUpperCase();

}

ExoticType type = new ExoticType(text);

setValue(type);

}

}

Cuối cùng, ta dùng bean factory CustomEditorConfigurer để register class PropertyEditor với ApplicationContext:

<bean id="customEditorConfigurer"

class="org.springframework.beans.factory.config.CustomEditorConfigurer">

<property name="customEditors">

<map>

<entry key="example.ExoticType">

<bean class="example.ExoticTypeEditor">

<property name="format">

<value>upperCase</value>

</property>

</bean>

</entry>

</map>

</property>

</bean>

### Setting a bean property or constructor argument from a property expression

PropertyPathFactoryBean là một FactoryBean mà định giá một property path theo một đối tượng target đưa ra. Đối tượng target có thể được chỉ định trực tiếp hay thông qua một bean name. Giá trị có được do định giá có thể được dùng trong một bean definition khác như một property value hoặc constructor argument.

VD:

// target bean to be referenced by name

<bean id="person" class="org.springframework.beans.TestBean" singleton="false">

<property name="age"><value>10</value></property>

<property name="spouse">

<bean class="org.springframework.beans.TestBean">

<property name="age"><value>11</value></property>

</bean>

</property>

</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'

<bean id="theAge" class="org.springframework.beans.factory.config.PropertyPathFactoryBean">

<property name="targetBeanName"><value>person</value></property>

<property name="propertyPath"><value>spouse.age</value></property>

</bean>

VD:

// will result in 12, which is the value of property 'age' of the inner bean

<bean id="theAge" class="org.springframework.beans.factory.config.PropertyPathFactoryBean">

<property name="targetObject">

<bean class="org.springframework.beans.TestBean">

<property name="age"><value>12</value></property>

</bean>

</property>

<property name="propertyPath"><value>age</value></property>

</bean>

VD:

// will result in 10, which is the value of property 'age' of bean 'person'

<bean id="person.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

VD:

<bean id="..." class="...">

<proprty name="age">

<bean id="person.age"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

</property>

</bean>

### Setting a bean property or constructor argument from a field value

FileRetrievingFactoryBean là một FactoryBean mà lấy về một static/non-static field value. Thông thường, nó được dùng để lấy các hằng số public static final, giá trị mà có thể được dùng để thiết lập một property value hay constructor argument cho một bean khác.

VD:

<bean id="myField"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">

<property name="staticField">

<value>java.sql.Connection.TRANSACTION\_SERIALIZABLE</value></property>

</bean>

VD:

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>

VD:

<bean id="..." class="...">

<proprty name="isolation">

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>

</property>

</bean>

Cũng có thể truy cập một non-static field của một bean khác.

### Invoking another method and optionally using the return value

Đôi khi cũng cần gọi một static/non-static method trong một class. Đôi lúc cũng cần thiết lập property của một bean là kết quả gọi method của một bean khác hay gọi một static method trên bất cứ class nào. Class MethodInvokingFactoryBean có thể được dùng cho cả hai mục đích trên. Nó là một FactoryBean trả về một giá trị là kết quả của việc gọi một static/instance method.

VD:

<bean id="force-init" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="staticMethod"><value>com.example.MyClass.initialize</value></property>

</bean>

<bean id="bean1" class="..." depends-on="force-init">

...

</bean>

Definition của bean1 dùng thuộc tính depends-on để tham chiếu đến bean force-init. Khi lần đầu tiên bean1 initialized thì bean force-init cũng đã phải initialized và do đó sẽ gọi static initializer method.

VD gọi một static factory method:

<bean id="myClass" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="staticMethod">

<value>com.whatever.MyClassFactory.getInstance</value></property>

</bean>

VD gọi một static method:

<bean id="sysProps" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="targetClass"><value>java.lang.System</value></property>

<property name="targetMethod"><value>getProperties</value></property>

</bean>

<bean id="javaVersion" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="targetObject"><ref local="sysProps"/></property>

<property name="targetMethod"><value>getProperty</value></property>

<property name="arguments">

<list>

<value>java.version</value>

</list>

</property>

</bean>

MethodInvokingFactoryBean thao tác mặc định theo singleton.

### Importing Bean Definitions from One File Into Another

Container definitions thường được đưa vào nhiều file XML. Dùng constructor của application context mà sẽ lấy nhiều Resource locations để load các file XML này vào context. Với một bean factory, reader có thể được dùng nhiều lần để đọc các definitions này trong các file XML.

Cách khác là trong một file XML bean definition có dùng một/nhiều elemnt import để load definitions từ các file khác. VD:

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

. . .

Lưu ý đến đường dẫn của file được import vào. Nội dung của file được imported sẽ phải valid với file nhận imported.

### Creating an ApplicationContext from a web application

ContextLoader có thể dùng để tạo ApplicationContexts. Nó có hai implementations là Context-LoaderListener và ContextLoaderServlet. Cả hai giống nhau về chức năng. Ở đây ta dùng ContextLoaderListener.

Dùng ContextLoaderListener để register một ApplicationContext như sau:

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

Hoặc dùng ContextLoaderServlet:

<servlet>

<servlet-name>context</servlet-name>

<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

Listener kiểm tra tham số contextConfigLocation. Nếu tham số không tồn tại thì listener sẽ dùng giá trị mặc định là /WEB-INF/applicationContext.xml. Khi tham số tồn tại thì listener sẽ dùng giá trị của tham số phân tách bằng dấu phẩy, chấm phẩy hoặc khoảng trắng để lấy tất cả file theo location đưa ra.

### Glue code and the evil singleton

Cách viết mã trong ứng dụng tốt nhất là theo style Dependency Injection (hay IoC). Tuy nhiên, một số layers nhỏ có thể cần viết theo style singleton để truy cập BeanFactory hay ApplicationContext.

## PropertyEditors, data binding, validation and the BeanWrapper

### Giới thiệu

Spring đưa ra một thiết kế cho validation (và data binding). Spring cung cấp Validator interface mà có thể dùng cho mọi layer của ứng dụng. Spring cung cấp DataBinder để data binding.

BeanWrapper là một khái niệm cơ bản của Spring và được sử dụng nhiều.

### Binding data using the DataBinder

DataBinder xây dựng trên nền BeanWrapper.

### Bean manipulation and the BeanWrapper

Sun cung cấp package org.springframework.beans. Một JavaBean là một class có no-argument constructor, các phương thức setter và getter cho thuộc tính.

Interface BeanWrapper và class BeanWrapperImpl được đặt trong cùng package beans. BeanWrapper cung cấp chức năng để set/get giá trị của property, lấy property descriptors, và query các properties để xác định chúng readable hay writable. Nó cũng hỗ trợ cho các nested properties, cho phép setting các properties và sub-properties. Nó hỗ trợ để thêm vào các JavaBeans PropertyChangeListeners và VetoableChangeListeners, hỗ trợ setting cho các indexed properties. BeanWrapper thường không được sử dụng trực tiếp bởi application code mà bởi DataBinder và BeanFactory.

#### Setting and getting basic and nested properties

Setting/getting cho properties sẽ dùng methods setPropertyValue(s) và getPropertyValue(s). VD về một số biểu thức:

|  |  |
| --- | --- |
| **Biểu thức** | **Giải thích** |
| name | Thuộc tính name, có: getName() và setName() |
| account.name | Sub-property. Có: getAccount().setName() hoặc getAccount().getName() |
| account[2] | Phần tử thứ ba của indexed property account. |
| account[COMPANYNAME] | Map entry indexed bởi khóa COMPANYNAME của Map property account. |

VD có 2 class:

public class Company {

private String name;

private Employee managingDirector;

public String getName() {

return this.name;

}

public void setName(String name) {

this.name = name;

}

public Employee getManagingDirector() {

return this.managingDirector;

}

public void setManagingDirector(Employee managingDirector) {

this.managingDirector = managingDirector;

}

}

public class Employee {

private float salary;

public float getSalary() {

return salary;

}

public void setSalary(float salary) {

this.salary = salary;

}

}

Class Company dependence class Employee. VD retrieve/manipulate các properties của chúng:

Company c = new Company();

BeanWrapper bwComp = BeanWrapperImpl(c);

// setting the company name...

bwComp.setPropertyValue("name", "Some Company Inc.");

// ... can also be done like this:

PropertyValue v = new PropertyValue("name", "Some Company Inc.");

bwComp.setPropertyValue(v);

// ok, let's create the director and tie it to the company:

Employee jim = new Employee();

BeanWrapper bwJim = BeanWrapperImpl(jim);

bwJim.setPropertyValue("name", "Jim Stravinsky");

bwComp.setPropertyValue("managingDirector", jim);

// retrieving the salary of the managingDirector through the company

Float salary = (Float)bwComp.getPropertyValue("managingDirector.salary");

#### Built-in PropertyEditors, converting types

Một số VD hiệu chỉnh property dùng trong Spring:

* setting properties của beans dùng PropertyEditors.
* parsing HTTP request parameters trong MVC framework của Spring.

Spring có một số built-in PropertyEditors trong org.springframework.beans.propertyeditors. Mặc định, hầu hết chúng đã được registered bởi BeanWrapperImpl.

|  |  |
| --- | --- |
| **Class** | **Giải thích** |
| ByteArrayPropertyEditor | Editor cho mảng byte. |
| ClassEditor | Parse String đại diện cho Class thành Class thực sự. Khi không tìm thấy Class sẽ có lỗi. |
| CustomBooleanEditor | Customizable property editor cho các properties kiểu Boolean. |
| CustomDateEditor | Customizable property editor cho các properties kiểu Date. Không được registered mặc định. |
| CustomNumberEditor | Customizable property editor cho các properties là kiểu con của Number như Long, Float, Double, Integer. |
| FileEditor | Khả năng chuyển String sang File-objects. |
| InputStreamEditor | Khả năng lấy một String và tạo ra một InputStream. |
| LocaleEditor | Khả năng chuyển String sang Locale-objects. |
| PropertiesEditor | Khả năng chuyển String (theo định dạng của java.lang.Properties) sang Properties-objects. |
| StringArrayPropertyEditor | Khả năng chuyển một danh sách comma-delimited kiểu String sang một String-array. |
| StringTrimmerEditor | Property editor mà dùng để trim cho một Strings. Không được registered mặc định. |
| URLEditor | Khả năng chuyển một String đại diện cho một URL sang URL-object thực sự. |

Spring dùng java.beans.PropertyEditorManager để thiết lập path cần thiết cho property editors. Search path cũng include sun.bean.editors mà dùng include PropertyEditors cho Font, Color và tất cả các primitive types.

#### Other features

Một số đặc trưng khác:

* Xác định readability và writability: dùng các methods isReadable() và isWritable().
* Lấy PropertyDescriptors: dùng các method getPropertyDescriptor(String) và getPropertyDescriptors() bạn có thể lấy về các đối tượng của type java.beans.PropertyDescriptor.

## Spring AOP

Mục này đề cập: **A**spect **O**riented **P**rogramming với Spring.

### Concepts

AOP bổ sung cho OOP bằng cách cung cấp một cách nghĩ khác về cấu trúc chương trình. OOP phân tán ứng dụng vào một hierarchy của objects, AOP phân tán chương trình vào trong các aspects hoặc concerns.

AOP framework là một key components của Spring. Các Spring IoC containers (BeanFactory và ApplicationContext) không phụ thuộc vào AOP.

AOP được dùng trong Spring:

* Để cung cấp declarative enterprise services, thay cho EJB declarative services.
* Để cho phép users implement các custom aspects, dùng OOP với AOP để bổ sung cho các aspects đó.

Nếu chỉ thích generic declarative services hoặc các pre-packaged declarative middleware services khác như pooling thì bạn không cần làm việc với Spring AOP.

#### AOP concepts

Một số khái niệm trung tâm của AOP:

* Aspect: một modularization của một concern mà implementation có thể cut across qua nhiều đối tượng. Transaction management là một VD của crosscutting concern trong J2EE.
* Joinpoint: point trong suốt quá trình thực thi chương trình, như method invocation hoặc một particular exception đang throwing. Trong Spring AOP, một joinpoint luôn là một method invocation.
* Advice: hành động do AOP framework thực hiện tại một particular joinpoint. Các loại khác nhau của advice gồm: around, before và throws.
* Pointcut: tập các joinpoints khi một advice bật lên.
* Introduction: thêm methods hay fields vào một advised class. Spring cho phép bạn introduce các interfaces mới tới bất kỳ advised object nào.
* Target object: object chứa joinpoint. Nó cũng được tham chiếu như advised hoặc proxied object.
* AOP proxy: object được AOP framework tạo ra, bao gồm cả advice. In Spring, nó là một JDK dynamic proxy hoặc một CGLIB proxy.
* Weaving: assembling aspects để tạo ra một advised object.

Các advice types khác:

* Around advice: advice bao quanh một joinpoint như method invocation. Đây là loại advice mạnh nhất. Nó thực hiện custom behavior lúc before và after của method invocation.
* Before advice: advice thực thi before một joinpoint. Nó không có khả năng ngăn luồng thực thi của joinpoint (trừ phi throw exception).
* Throws advice: advice được thực thi nếu một method throw một exception. Bạn có thể viết mã để catches exception trong nó.
* After returning advice: advice được thực thi sau khi joinpoint hoàn thành một cách bình thường (không có exception).

Hầu hết interception-based AOP frameworks, như Nanning Aspects, chỉ cung cấp around advice. Khuyến khích dùng least powerful advice nếu nó đáp ứng được yêu cầu của bạn.

Khái niệm pointcut là key của AOP. Pointcuts cho phép advice được targeted một cách độc lập với OO hierarchy.

#### Spring AOP capabilities and goals

Spring AOP không cần điều khiển class loader hierarchy, nó dùng được trong một J2EE web container hoặc application server. Spring hỗ trợ interception của các method invocations. Field interception không được implemented.

Spring cung cấp các classes đại diện cho các pointcuts và các advice types khác nhau. Spring dùng thuật ngữ advisor cho một đối tượng đại diện cho một aspect, gồm cả advice và pointcut.

Spring AOP khác các AOP frameworks khác. Nó cung cấp một tương tác chặt chẽ giữa AOP implementation và Spring IoC để giúp giải quyết các vấn đề trong enterprise applications. Các chức năng của Spring AOP thường sử dụng với Spring IoC container. AOP advice được chỉ định để sử dụng các cú pháp bean definition thông thường. Các advices và pointcuts sẽ do Spring IoC quản lý.

#### AOP Proxies in Spring

Mặc định, Spring dùng J2SE dynamic proxies cho AOP proxies. Spring cũng có thể dùng CGLIB proxies.

### Pointcuts in Spring

#### Các khái niệm

Pointcut model của Spring cho phép pointcut tái sử dụng độc lập với advice types. Interface org.springframework.aop.Pointcut dùng để target các advices tới các particular classes và methods.

public interface Pointcut {

ClassFilter getClassFilter();

MethodMatcher getMethodMatcher();

}

Interface ClassFilter dùng restrict một pointcut tới một tập các target classes. Nếu matches() method luôn trả về true thì tất cả target classes sẽ được matched:

public interface ClassFilter {

boolean matches(Class clazz);

}

Nội dung của MethodMatcher interface:

public interface MethodMatcher {

boolean matches(Method m, Class targetClass);

boolean isRuntime();

boolean matches(Method m, Class targetClass, Object[] args);

}

Method matches(Method, Class) dùng kiểm tra một pointcut có match với một method của một target class hay không.

#### Operations on pointcuts

Spring hỗ trợ các operations cho pointcuts: union và intersection.

Pointcuts có thể được sử dụng kết hợp các static methods trong class org.springframework.-aop.support.Pointcuts hoặc dùng class ComposablePointcut trong cùng package.

#### Convenience pointcut implementations

##### Static pointcuts

Static pointcuts dựa vào method và target class. Spring chỉ có thể định giá một static pointcut một lần khi một method được gọi lần đầu tiên. Sau đó, không cần định giá lại static pointcut.

**Regular expression pointcuts**

Dùng regular expressions để chỉ định static pointcuts. Class org.springframework.aop.support.-RegexpMethodPointcut là một regular expression pointcut, dùng cú pháp của Perl 5.

Dùng class này, bạn có thể cung cấp một danh sách các pattern Strings. Nếu một mục phù hợp thì pointcut sẽ định giá trị là true. VD:

<bean id="settersAndAbsquatulatePointcut"

class="org.springframework.aop.support.RegexpMethodPointcut">

<property name="patterns">

<list>

<value>.\*get.\*</value>

<value>.\*absquatulate</value>

</list>

</property>

</bean>

RegexpMethodPointcutAdvisor cho phép tham chiếu đến một Advice. VD:

<bean id="settersAndAbsquatulateAdvisor"

class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice">

<ref local="beanNameOfAopAllianceInterceptor"/>

</property>

<property name="patterns">

<list>

<value>.\*get.\*</value>

<value>.\*absquatulate</value>

</list>

</property>

</bean>

RegexpMethodPointcutAdvisor có thể dùng với bất cứ Advice type nào. RegexpMethodPointcut class yêu cầu package Jakarta ORO regular expression.

**Attribute-driven pointcuts**

Một loại quan trọng của static pointcut là metadata-driven pointcut. Nó dùng các giá trị của các thuộc tính metadata: thông thường là source-level metadata.

##### Dynamic pointcuts

Dynamic pointcuts lấy các đối số của method cũng như thông tin static. Chúng luôn phải được định giá cho mỗi method invocation. Kết quả của định giá không thể được cached. Tiêu biểu là control flow pointcut.

**Control flow pointcuts**

Một control flow pointcut match với lệnh gọi current. Nó có thể được fire lên khi một method gọi một joinpoint. Định giá Control flow pointcuts tốn tài nguyên hơn các loại dynamic pointcuts khác.

#### Pointcut superclasses

Spring cung cấp các pointcut superclasses cho phép bạn implement các custom pointcuts. Static pointcuts hữu dụng nhất. Bạn extends lớp StaticMethodMatcherPointcut như sau:

class TestStaticPointcut extends StaticMethodMatcherPointcut {

public boolean matches(Method m, Class targetClass) {

// return true if custom criteria match

}

}

#### Custom pointcuts

Pointcuts trong Spring là các Java classes nên có thể khai báo các custom pointcuts theo dạng static/dynamic.

### Advice types in Spring

#### Advice lifecycles

Spring advices có thể được shared ngang qua tất cả các advised objects, hoặc unique cho từng advised object. Điều này phù hợp với per-class hoặc per-instance advice.

Per-class advice thường được sử dụng. Nó thích hợp với generic advice như các transaction advisors.

Per-instance advice thích hợp với introductions. Advice này sẽ thêm state vào proxied object.

Có thể trộn lẫn cả shared advice và per-instance advice trong cùng AOP proxy.

#### Advice types in Spring

Spring cung cấp nhiều advice types và có thể extensible. Bên dưới là một số standard advice types.

##### Interception around advice

Là loại advice cơ bản nhất. MethodInterceptors implementing around advice sẽ implement interface sau:

public interface MethodInterceptor extends Interceptor {

Object invoke(MethodInvocation invocation) throws Throwable;

}

Method invoke() sẽ trả về giá trị của joinpoint.

VD về một implement của MethodInterceptor:

public class DebugInterceptor implements MethodInterceptor {

public Object invoke(MethodInvocation invocation) throws Throwable {

System.out.println("Before: invocation=[" + invocation + "]");

Object rval = invocation.proceed();

System.out.println("Invocation returned");

return rval;

}

}

##### Before advice

Advice này không cần một MethodInvocation object, không gọi proceed() method. Chi tiết của MethodBeforeAdvice interface như sau:

public interface MethodBeforeAdvice extends BeforeAdvice {

void before(Method m, Object[] args, Object target) throws Throwable;

}

Before advice có thể chèn custom behavior trước khi thực thi joinpoint nhưng không thể thay đổi giá trị trả về. Nó cũng có thể throw một exception.

VD về before advice:

public class CountingBeforeAdvice implements MethodBeforeAdvice {

private int count;

public void before(Method m, Object[] args, Object target) throws Throwable {

++count;

}

public int getCount() {

return count;

}

}

Before advice có thể dùng được cho bất kỳ pointcut nào.

##### Throws advice

Throws advice được gọi nếu joinpoint throw một exception. Interface org.springframework.-aop.ThrowsAdvice không chứa bất kỳ method nào.

VD advice sẽ được gọi khi throw RemoteException:

public class RemoteThrowsAdvice implements ThrowsAdvice {

public void afterThrowing(RemoteException ex) throws Throwable {

// Do something with remote exception

}

}

VD advice sẽ được gọi khi throw ServletException, lưu ý tham số khác VD trước:

public static class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {

// Do something will all arguments

}

}

VD hai advice dạng trên trong cùng class:

public static class CombinedThrowsAdvice implements ThrowsAdvice {

public void afterThrowing(RemoteException ex) throws Throwable {

// Do something with remote exception

}

public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {

// Do something will all arguments

}

}

Throws advice có thể dùng được cho bất kỳ pointcut nào.

##### After Returning advice

Advice implement từ interface org.springframework.aop.AfterReturningAdvice sau:

public interface AfterReturningAdvice extends Advice {

void afterReturning(Object returnValue, Method m, Object[] args, Object target)

throws Throwable;

}

Advice này truy cập được: return value, invoked method, methods arguments và target.

VD advice trả về số lượng:

public class CountingAfterReturningAdvice implements AfterReturningAdvice {

private int count;

public void afterReturning(Object returnValue, Method m, Object[] args, Object target) throws Throwable {

++count;

}

public int getCount() {

return count;

}

}

Advice này không thay đổi được execution path. Nó dùng được cho bất kỳ pointcut nào.

##### Introduction advice

Introduction yêu cầu IntroductionAdvisor và IntroductionInterceptor.

public interface IntroductionInterceptor extends MethodInterceptor {

boolean implementsInterface(Class intf);

}

Introduction advice không thể dùng cho bất kỳ pointcut nào. Nó chỉ dùng cho mức class. Chỉ có thể dùng introduction advice với InterceptionIntroductionAdvisor. VD:

public interface InterceptionIntroductionAdvisor extends InterceptionAdvisor {

ClassFilter getClassFilter();

IntroductionInterceptor getIntroductionInterceptor();

Class[] getInterfaces();

}

Method getInterfaces() trả về interfaces do advisor này introduce.

### Advisors in Spring

Trong Spring, một Advisor là một modularization của một aspect. Advisors kết hợp cả advice và pointcut. Advisor dùng được cho bất kỳ advice nào như MethodInterceptor, BeforeAdvice hoặc ThrowsAdvice.

Trong Spring, có thể trộn lẫn advisor và advice types trong cùng AOP proxy.

### Using the ProxyFactoryBean to create AOP proxies

Nếu dùng Spring IoC container thì có thể cần dùng một Spring's AOP FactoryBeans.

Dùng org.springframework.aop.framework.ProxyFactoryBean để tạo một AOP proxy trong Spring. Nó cung cấp cách điểu khiển cho pointcuts và advice.

#### Basics

ProxyFactoryBean implement từ FactoryBean. Dùng ProxyFactoryBean để tạo ra AOP proxies, các advices và pointcuts có thể được IoC quản lý. Một advice có thể tự tham chiếu đến application objects, tận dụng tất cả pluggability do Dependency Injection cung cấp.

#### JavaBean properties

ProxyFactoryBean là một JavaBean. Các properties của nó dùng: chỉ định target cho proxy, chỉ định có dùng CGLIB không.

Một số thuộc tính kế thừa từ org.springframework.aop.framework.ProxyConfig là:

|  |  |
| --- | --- |
| **Thuộc tính** | **Giải thích** |
| proxyTargetClass | true/false khi proxy target là class/interface. Nếu đặt true thì dùng CGLIB. |
| optimize | Có dùng tối ưu hóa để tạo proxies không. Không sử dụng nếu không hiểu AOP proxy. Nó được dùng cho CGLIB proxies. |
| frozen | Mặc định false. Có cho phép advice thay đổi khi proxy factory đã được cấu hình không. |
| exposeProxy | Proxy hiện tại có được exposed trong một ThreadLocal để target truy cập không. Khi đặt true và target cần obtain proxy thì nó sẽ dùng method AopContext.currentProxy(). |
| aopProxyFactory | Đối tượng implement từ AopProxyFactory để sử dụng. Là cách để dùng dynamic proxy hay CGLIB proxy. |

Các thuộc tính khác chỉ định cho ProxyFactoryBean là:

|  |  |
| --- | --- |
| **Thuộc tính** | **Giải thích** |
| proxyInterfaces | Mảng String gồm các interface names. Nếu không chỉ định sẽ dùng CGLIB proxy. |
| interceptorNames | Mảng String của Advisor, interceptor hoặc các advice names khác. |
| singleton | Factory sẽ trả về một single object hay không. Mặc định true. |

#### Proxying interfaces

VD sau gồm: một target bean sẽ được proxied, một advisor và một Interceptor dùng cung cấp advice, một AOP proxy bean definition chỉ định target object và các interfaces cho proxy, các advices để áp dụng.

<bean id="personTarget" class="com.mycompany.PersonImpl">

<property name="name"><value>Tony</value></property>

<property name="age"><value>51</value></property>

</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">

<property name="someProperty"><value>Custom string property value</value></property>

</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">

</bean>

<bean id="person"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces"><value>com.mycompany.Person</value></property>

<property name="target"><ref local="personTarget"/></property>

<property name="interceptorNames">

<list>

<value>myAdvisor</value>

<value>debugInterceptor</value>

</list>

</property>

</bean>

Thuộc tính interceptorNames lấy một danh sách String là bean names của interceptor/advisors trong current factory. Bean person là một implement của Person:

Person person = (Person) factory.getBean("person");

Các beans khác trong cùng IoC context có thể tham chiếu đến bean trên:

<bean id="personUser" class="com.mycompany.PersonUser">

<property name="person"><ref local="person" /></property>

</bean>

#### Proxying classes

CGLIB proxying phát sinh một subclass của target class lúc runtime. Subclass được dùng để implement từ Decorator pattern.

Một số vấn đề: final methods không thể advised hay overridden, bạn cần CGLIB 2 binaries trong classpath.

### Convenient proxy creation

Chúng ta thường không cần full power của ProxyFactoryBean. Có một số convenience factories ta cần dùng để tạo AOP proxies khi muốn tập trung vào một aspect cụ thể.

#### TransactionProxyFactoryBean

TransactionProxyFactoryBean là một subclass của ProxyConfig. Nó yêu cầu một target và thông tin về transaction attributes như VD sau:

<bean id="petStoreTarget" class="org.springframework.samples.jpetstore.domain.logic.PetStoreImpl">

<property name="accountDao"><ref bean="accountDao"/></property>

<!-- Other dependencies omitted -->

</bean>

<bean id="petStore"

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

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

<property name="target"><ref local="petStoreTarget"/></property>

<property name="transactionAttributes">

<props>

<prop key="insert\*">PROPAGATION\_REQUIRED</prop>

<prop key="update\*">PROPAGATION\_REQUIRED</prop>

<prop key="\*">PROPAGATION\_REQUIRED,readOnly</prop>

</props>

</property>

</bean>

Ta phải dùng một inner bean để thiết lập thuộc tính cho target property.

TransactionProxyFactoryBean tự tạo một transaction advisor, gồm một pointcut dựa trên transaction attributes, vì vậy chỉ transactional methods được advised.

TransactionProxyFactoryBean cho phép chỉ rõ pre và post advice, dùng preInterceptors và postInterceptors properties. Các properties này lấy một mảng Object của các interceptors, các advice hoặc Advisors khác để đặt vào interception chain lúc before hoặc after transaction interceptor. VD:

<property name="preInterceptors">

<list>

<ref local="authorizationInterceptor"/>

<ref local="notificationBeforeAdvice"/>

</list>

</property>

<property name="postInterceptors">

<list>

<ref local="myAdvisor"/>

</list>

</property>

Các properties này có thể được thêm vào petStore bean definition phía trên.

Nếu chỉ khai báo transaction management, nên dùng TransactionProxyFactoryBean thay cho ProxyFactoryBean.

#### EJB proxies

Các dedicated proxies tạo proxies cho EJBs.

### Concise proxy definitions

Có thể dùng parent và child bean definitions, inner bean definitions trong quá trình định nghĩa proxy.

VD định nghĩa parent bean:

<bean id="txProxyTemplate" abstract="true"

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

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

<property name="transactionAttributes">

<props>

<prop key="\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

Định nghĩa child bean:

<bean id="myService" parent="txProxyTemplate">

<property name="target">

<bean class="org.springframework.samples.MyServiceImpl">

</bean>

</property>

</bean>

Có thể override các properties từ parent template như sau:

<bean id="mySpecialService" parent="txProxyTemplate">

<property name="target">

<bean class="org.springframework.samples.MySpecialServiceImpl">

</bean>

</property>

<property name="transactionAttributes">

<props>

<prop key="get\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="find\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="load\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="store\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

### Creating AOP proxies programmatically with the ProxyFactory

Dùng Spring để tạo AOP proxies programmatically cho phép dùng Spring AOP mà không phụ thuộc vào Spring IoC. VD tạo một proxy cho một target object, với một interceptor và một advisor, interfaces được target object implemented sẽ được proxied tự động:

ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);

factory.addInterceptor(myMethodInterceptor);

factory.addAdvisor(myAdvisor);

MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();

Đầu tiên là construct một đối tượng của type org.springframework.aop.framework.ProxyFactory. Bạn có thể thêm các interceptors hoặc advisors vào ProxyFactory. ProxyFactory (do kế thừa) cũng có phương thức để thêm các advice types khác như before và throws advice. AdvisedSupport là superclass của cả ProxyFactory và ProxyFactoryBean.

Khuyến khích dùng AOP để externalize các configuration từ Java code.

### Manipulating advised objects

Khi tạo các AOP proxies, có thể dùng interface org.springframework.aop.framework.Advised để thao tác chúng. Bất kỳ AOP proxy nào cũng có thể cast sang interface này. Nó gồm các methods sau:

Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();

Method getAdvisors() sẽ trả về một Advisor cho mọi advisor, interceptor hoặc các advice types khác mà được thêm vào factory.

Method addAdvisor() có thể dùng để thêm bất cứ Advisor nào.

VD về cast một AOP proxy sang Advised interface rồi kiểm tra và thao các advice của nó:

Advised advised = (Advised) myObject;

Advisor[] advisors = advised.getAdvisors();

int oldAdvisorCount = advisors.length;

System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut

// Will match all proxied methods

// Can use for interceptors, before, after returning or throws advice

advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut

advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors",

oldAdvisorCount + 2, advised.getAdvisors().length);

### Using the autoproxy facility

Spring cho phép dùng autoproxy bean definitions, có thể tự động proxy các bean definitions được chọn. Bạn cần thiết lập một số bean definitions đặc biệt trong file XML bean definition. Điều này cho phép khai báo các targets cho autoproxying không cần dùng ProxyFactoryBean.

Có 2 cách để thực hiện:

* Dùng một autoproxy creator tham chiếu đến các beans cụ thể trong current context.
* Một case đặc biệt của autoproxy creation là tạo autoproxy bằng thuộc tính source-level metadata.

#### Autoproxy bean definitions

Package org.springframework.aop.framework.autoproxy cung cấp các standard autoproxy creators.

##### BeanNameAutoProxyCreator

BeanNameAutoProxyCreator tự động tạo các AOP proxies cho các beans phù hợp với literal values hoặc wildcards.

<bean id="jdkBeanNameProxyCreator"

class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<property name="beanNames"><value>jdk\*,onlyJdk</value></property>

<property name="interceptorNames">

<list>

<value>myInterceptor</value>

</list>

</property>

</bean>

Thuộc tính interceptorNames chứa một danh sách các interceptor, interceptors có thể là các advisors hoặc bất cứ advice type nào. Dùng BeanNameAutoProxyCreator để áp dụng cùng configuration cho nhiều objects.

Các bean names phù hợp với VD trên có thể là jdkMyBean và onlyJdk.

##### DefaultAdvisorAutoProxyCreator

DefaultAdvisorAutoProxyCreator cũng dùng để tạo auto proxy. Nó tự động áp dụng các advisors trong current context, không cần include các bean names cụ thể trong bean definition của các autoproxy advisors.

<bean id="autoProxyCreator"

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">

</bean>

<bean id="txAdvisor"

autowire="constructor"

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

<property name="order"><value>1</value></property>

</bean>

<bean id="customAdvisor"

class="com.mycompany.MyAdvisor">

</bean>

<bean id="businessObject1"

class="com.mycompany.BusinessObject1">

<!-- Properties omitted -->

</bean>

<bean id="businessObject2"

class="com.mycompany.BusinessObject2">

</bean>

DefaultAdvisorAutoProxyCreator dùng áp dụng cùng advice cho nhiều business objects. Nó hỗ trợ filtering và ordering. Advisors có thể implement interface org.springframework.core.Ordered để đảm bảo ordering đúng.

##### AbstractAdvisorAutoProxyCreator

Đây là superclass của DefaultAdvisorAutoProxyCreator. Bạn có thể tạo custom autoproxy creators bằng cách tạo subclass của class này.

#### Using metadata-driven auto-proxying

Một loại autoproxying được driven bởi metadata. Thay vì dùng XML, cấu hình cho transaction management và các enterprise services khác được giữ trong thuộc tính source-level.

Dùng DefaultAdvisorAutoProxyCreator kết hợp với các Advisors mà hiểu được các thuộc tính metadata. Metadata được giữ trong pointcut part của các advisors.

VD về bean definitions:

<bean id="autoProxyCreator"

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">

</bean>

<bean id="transactionAttributeSource"

class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource"

autowire="constructor">

</bean>

<bean id="transactionInterceptor"

class="org.springframework.transaction.interceptor.TransactionInterceptor"

autowire="byType">

</bean>

<bean id="transactionAdvisor"

class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor"

autowire="constructor" >

</bean>

<bean id="attributes"

class="org.springframework.metadata.commons.CommonsAttributes"

/>

### Using TargetSources

Spring có khái niệm TargetSource (expressed in the org.springframework.aop.TargetSource). Interface này trả về target object đang implementing joinpoint. Mỗi lần AOP proxy xử lý thì TargetSource implementation được yêu cầu cho một target instance.

Các developers dùng Spring AOP thường không cần làm việc trực tiếp với TargetSources. Khi dùng một custom target source, target của bạn cần phải là một prototype.

#### Hot swappable target sources

org.springframework.aop.target.HotSwappableTargetSource cho phép target của một AOP proxy được switched trong khi các callers vẫn giữ được tham chiếu đến nó. Việc thay đổi target của target source có hiệu quả tức thì.

Dùng swap() method của HotSwappableTargetSource để thay đổi target:

HotSwappableTargetSource swapper =

(HotSwappableTargetSource) beanFactory.getBean("swapper");

Object oldTarget = swapper.swap(newTarget);

Yêu cầu về XML definitions như sau:

<bean id="initialTarget" class="mycompany.OldTarget">

</bean>

<bean id="swapper"

class="org.springframework.aop.target.HotSwappableTargetSource">

<constructor-arg><ref local="initialTarget"/></constructor-arg>

</bean>

<bean id="swappable"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="targetSource">

<ref local="swapper"/>

</property>

</bean>

#### Pooling target sources

Spring pooling có thể áp dụng được cho bất cứ POJO nào. Bạn cần gói jar commons-pool trong classpath để dùng đặc trưng này. Tạo các subclass của org.springframework.aop.target.-AbstractPoolingTargetSource để hỗ trợ các pooling API khác.

VD configuration:

<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"

singleton="false">

... properties omitted

</bean>

<bean id="poolTargetSource"

class="org.springframework.aop.target.CommonsPoolTargetSource">

<property name="targetBeanName"><value>businessObjectTarget</value></property>

<property name="maxSize"><value>25</value></property>

</bean>

<bean id="businessObject"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="targetSource"><ref local="poolTargetSource"/></property>

<property name="interceptorNames"><value>myInterceptor</value></property>

</bean>

Target object businessObjectTarget phải là một prototype. Điều này cho phép PoolingTarget-Source implementation tạo new instances của target đó.

Interceptor myInterceptor cần được định nghĩa trong cùng IoC context. Tuy nhiên, không cần chỉ định các interceptors để sử dụng pooling. Nếu chỉ muốn pooling thì không cần thiết lập interceptorNames property.

Có thể cấu hình Spring để cast bất kỳ pooled object nào sang interface org.springframework.-aop.target.PoolingConfig. Cần định nghĩa một advisor như sau:

<bean id="poolConfigAdvisor"

class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="targetObject"><ref local="poolTargetSource" /></property>

<property name="targetMethod"><value>getPoolingConfigMixin</value></property>

</bean>

poolConfigAdvisor phải đặt trong danh sách các interceptors names trong ProxyFactoryBean.

Việc cast trên giống như:

PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");

System.out.println("Max pool size is " + conf.getMaxSize());

#### Prototype target sources

Thiết lập một prototype target source giống với một pooling TargetSource. Một new instance của target được tạo ra cho mọi method invocation. Không nên dùng phương pháp này khi không thực sự cần thiết.

VD:

<bean id="prototypeTargetSource"

class="org.springframework.aop.target.PrototypeTargetSource">

<property name="targetBeanName"><value>businessObjectTarget</value></property>

</bean>

Chỉ có một property là name của target bean. Target bean cũng phải là một prototype bean definition.

### Defining new Advice types

Package org.springframework.aop.framework.adapter hỗ trợ custom advice types mới mà không làm thay đổi core framework. Ràng buộc đối với một custom Advice type là nó phải implement từ interface org.aopalliance.aop.Advice.

## AspectJ Integration

### Overview

AOP framework xử lý nhiều vấn đề của middleware và application-specific. Tuy nhiên, cần nhiều hơn AOP power để giải quyết các vấn đề: thêm additional fields vào một class, các advise objects không phải do Spring IoC container tạo ra.

Khuyến khích dùng AspectJ để giải quyết vấn đề này.

### Configuring AspectJ aspects using Spring IoC

Tương tác Spring/AspectJ cho phép Spring dùng Dependency Injection để cấu hình AspectJ aspects. Các benefits:

* Aspects không cần dùng ad-hoc configuration mechanisms.
* Aspects có thể depend vào application objects.
* Lấy được tham chiếu đến aspect thông qua Spring context. Cho phép dynamic reconfiguration cho aspect.

AspectJ aspects có thể expose các JavaBean properties cho Setter Injection. AspectJ aspects không thể dùng Constructor Injection hoặc Method Injection.

#### Singleton aspects

Đa số, AspectJ aspects là singletons, với một instance cho class loader.

Một Spring IoC container không thể instantiate một aspect, aspects không thể gọi constructors. Lấy một tham chiếu đến một aspect dùng aspectOf() method mà AspectJ định nghĩa cho tất cả aspects và nó có thể inject dependencies vào trong aspect đó.

##### Example

Xem một security aspect phụ thuộc vào security manager. Aspects này áp dụng cho mọi thay đổi giá trị của biến balance trong Account class. Mã AspectJ cho aspect này như sau:

public aspect BalanceChangeSecurityAspect {

private SecurityManager securityManager;

public void setSecurityManager(SecurityManager securityManager) {

this.securityManager = securityManager;

}

private pointcut balanceChanged() :

set(int Account.balance);

before() : balanceChanged() {

this.securityManager.checkAuthorizedToModify();

}

}

Dùng thuộc tính factory-method để chỉ định aspect sử dụng aspectOf() method.

<bean id="securityAspect"

class="org.springframework.samples.aspectj.bank.BalanceChangeSecurityAspect"

factory-method="aspectOf">

<property name="securityManager">

<ref local="securityManager"/>

</property>

</bean>

Không cần làm gì trong Spring configuration để target cho aspect này. Nó chứa thông tin pointcut trong mã AspectJ mà điều khiển khi nào áp dụng nó.

### Using AspectJ pointcuts to target Spring advice

Các biểu thức AspectJ pointcut có thể được dùng trong Spring XML hoặc các file bean definition khác thể target cho Spring advice. Cho phép áp dụng sức mạnh của AspectJ pointcut model vào Spring's proxy-based AOP framework. Chỉ có subset của AspectJ pointcuts liên quan đến method invocation là có thể sử dụng.

### Spring aspects for AspectJ

Có thể đóng gói một số Spring services, như khai báo transaction management, như các AspectJ aspects. Điều này cho phép AspectJ users sử dụng chúng mà không phụ thuộc vào Spring AOP framework, ngay cả không phụ thuộc vào Spring IoC container.

## 7. Transaction management

### 7.1. The Spring transaction abstraction

Spring provides a consistent abstraction for transaction management. This abstraction is one of the most important of Spring's abstractions, and delivers the following benefits:

Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, iBATIS Database Layer and JDO.

Provides a simpler, easier to use, API for programmatic transaction management than most of these transaction APIs

Integrates with the Spring data access abstraction

Supports Spring declarative transaction management

Traditionally, J2EE developers have had two choices for transaction management: to use global or local transactions. Global transactions are managed by the application server, using JTA. Local transactions are resource-specific: for example, a transaction associated with a JDBC connection. This choice had profound implications. Global transactions provide the ability to work with multiple transactional resources. (It's worth noting that most applications use a single transaction resource) With local transactions, the application server is not involved in transaction management, and cannot help ensure correctness across multiple resources.

Global transactions have a significant downside. Code needs to use JTA: a cumbersome API to use (partly due to its exception model). Furthermore, a JTA UserTransaction normally needs to be obtained from JNDI: meaning that we need to use both JNDI and JTA to use JTA. Obviously all use of global transactions limits the reusability of application code, as JTA is normally only available in an application server environment.

The preferred way to use global transactions was via EJB CMT (Container Managed Transaction): a form of declarative transaction management (as distinguished from programmatic transaction management). EJB CMT removes the need for transaction-related JNDI lookups--although of course the use of EJB itself necessitates the use of JNDI. It removes most--not all--need to write Java code to control transactions. The significant downside is that CMT is (obviously) tied to JTA and an application server environment; and that it's only available if we choose to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives around EJB in general are so great that this is not an attractive proposition, when there are alternatives for declarative transaction management.

Local transactions may be easier to use, but also have significant disadvantages: They cannot work across multiple transactional resources, and tend to invade the programming model. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction.

Spring resolves these problems. It enables application developers to use a consistent programming model in any environment. You write your code once, and it can benefit from different transaction management strategies in different environments. Spring provides both declarative and programmatic transaction management. Declarative transaction management is preferred by most users, and recommended in most cases.

With programmatic transaction management developers work with the Spring transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model developers typically write little or no code related to transaction management, and hence don't depend on Spring's or any other transaction API.

### 7.2. Transaction strategies

The key to the Spring transaction abstraction is the notion of a transaction strategy.

This is captured in the org.springframework.transaction.PlatformTransactionManager interface, shown below:

public interface PlatformTransactionManager {

TransactionStatus getTransaction(TransactionDefinition definition)

throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

}

This is primarily an SPI interface, although it can be used programmatically. Note that in keeping with Spring's philosophy, this is an interface. Thus it can easily be mocked or stubbed if necessary. Nor is it tied to a lookup strategy such as JNDI: PlatformTransactionManager implementations are defined like any other object in a Spring IoC container. This benefit alone makes this a worthwhile abstraction even when working with JTA: transactional code can be tested much more easily than if it directly used JTA.

In keeping with Spring's philosophy, TransactionException is unchecked. Failures of the transaction infrastructure are almost invariably fatal. In rare cases where application code can recover from them, the application developer can still choose to catch and handle TransactionException.

The getTransaction() method returns a TransactionStatus object, depending on a TransactionDefinition parameter. The returned TransactionStatus might represent a new or existing transaction (if there was a matching transaction in the current call stack).

As with J2EE transaction contexts, a TransactionStatus is associated with a thread of execution.

The TransactionDefinition interface specifies:

Transaction isolation: The degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?

Transaction propagation: Normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: For example, simply running in the existing transaction (the most common case); or suspending the existing transaction and creating a new transaction. Spring offers the transaction propagation options familiar from EJB CMT.

Transaction timeout: How long this transaction may run before timing out (automatically being rolled back by the underlying transaction infrastructure).

Read-only status: A read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

These settings reflect standard concepts. If necessary, please refer to a resource discussing transaction isolation levels and other core transaction concepts: Understanding such core concepts is essential to using Spring or any other transaction management solution.

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

public interface TransactionStatus {

boolean isNewTransaction();

void setRollbackOnly();

boolean isRollbackOnly();

}

However Spring transaction management is used, defining the PlatformTransactionManager implementation is essential. In good Spring fashion, this important definition is made using Inversion of Control.

PlatformTransactionManager implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate etc.

The following examples from dataAccessContext-local.xml from Spring's jPetStore sample application show how a local PlatformTransactionManager implementation can be defined. This will work with JDBC.

We must define a JDBC DataSource, and then use the Spring DataSourceTransactionManager, giving it a reference to the DataSource.

<bean id="dataSource"

class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName"><value>${jdbc.driverClassName}</value></property>

<property name="url"><value>${jdbc.url}</value></property>

<property name="username"><value>${jdbc.username}</value></property>

<property name="password"><value>${jdbc.password}</value></property>

</bean>

The PlatformTransactionManager definition will look like this:

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource"><ref local="dataSource"/></property>

</bean>

If we use JTA, as in the dataAccessContext-jta.xml file from the same sample application, we need to use a container DataSource, obtained via JNDI, and a JtaTransactionManager implementation. The JtaTransactionManager doesn't need to know about the DataSource, or any other specific resources, as it will use the container's global transaction management.

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName"><value>jdbc/jpetstore</value></property>

</bean>

<bean id="transactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager"/>

We can use Hibernate local transactions easily, as shown in the following examples from the Spring PetClinic sample application.

In this case, we need to define a Hibernate LocalSessionFactory, which application code will use to obtain Hibernate Sessions.

The DataSource bean definition will be similar to one of the above examples, and is not shown. (If it's a container DataSource it should be non-transactional as Spring, rather than the container, will manage transactions.)

The "transactionManager" bean in this case is of class HibernateTransactionManager. In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the session factory.

<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

<property name="dataSource"><ref local="dataSource"/></property>

<property name="mappingResources">

<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">${hibernate.dialect}</prop>

</props>

</property>

</bean>

<bean id="transactionManager"

class="org.springframework.orm.hibernate.HibernateTransactionManager">

<property name="sessionFactory"><ref local="sessionFactory"/></property>

</bean>

With Hibernate and JTA transactions we could simply use the JtaTransactionManager as with JDBC or any other resource strategy.

<bean id="transactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager"/>

Note that this is identical to JTA configuration for any resource, as these are global transactions, which can enlist any transactional resource.

In all these cases, application code won't need to change at all. We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

When not using global transactions, you do need to follow one special coding convention. Fortunately this is very simple. You need to obtain connection or session resources in a special way, to allow the relevant PlatformTransactionManager implementation to track connection usage, and apply transaction management as necessary.

For example, if using JDBC, you should not call the getConnection() method on a DataSource, but must use the Spring org.springframework.jdbc.datasource.DataSourceUtils class as follows:

Connection conn = DataSourceUtils.getConnection(dataSource);

This has the added advantage that any SQLException will be wrapped in a Spring CannotGetJdbcConnectionException--one of Spring's hierarchy of unchecked DataAccessExceptions. This gives you more information than can easily be obtained from the SQLException, and ensures portability across databases: even across different persistence technologies.

This will work fine without Spring transaction management, so you can use it whether or not you are using Spring for transaction management.

Of course, once you've used Spring's JDBC support or Hibernate support, you won't want to use DataSourceUtils or the other helper classes, because you'll be much happier working via the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring JdbcTemplate or jdbc.object package to simplify your use of JDBC, correct connection retrieval happens behind the scenes and you won't need to write any special code.

### 7.3. Programmatic transaction management

Spring provides two means of programmatic transaction management:

Using the TransactionTemplate

Using a PlatformTransactionManager implementation directly

We generally recommend the first approach.

The second approach is similar to using the JTA UserTransaction API (although exception handling is less cumbersome).

#### 7.3.1. Using the TransactionTemplate

The TransactionTemplate adopts the same approach as other Spring templates such as JdbcTemplate and HibernateTemplate. It uses a callback approach, to free application code from the working of acquiring and releasing resources. (No more try/catch/finally.) Like other templates, a TransactionTemplate is threadsafe.

Application code that must execute in a transaction context looks like this. Note that the TransactionCallback can be used to return a value:

Object result = tt.execute(new TransactionCallback() {

public Object doInTransaction(TransactionStatus status) {

updateOperation1();

return resultOfUpdateOperation2();

}

});

If there's no return value, use a TransactionCallbackWithoutResult like this:

tt.execute(new TransactionCallbackWithoutResult() {

protected void doInTransactionWithoutResult(TransactionStatus status) {

updateOperation1();

updateOperation2();

}

});

Code within the callback can roll the transaction back by calling the setRollbackOnly() method on the TransactionStatus object.

Application classes wishing to use the TransactionTemplate must have access to a PlatformTransactionManager: usually exposed as a JavaBean property or as a constructor argument.

It's easy to unit test such classes with a mock or stub PlatformTransactionManager. There's no JNDI lookup or static magic here: it's a simple interface. As usual, you can use Spring to simplify your unit testing.

#### 7.3.2. Using the PlatformTransactionManager

You can also use the org.springframework.transaction.PlatformTransactionManager directly to manage your transaction. Simply pass the implementation of the PlatformTransactionManager you're using to your bean via a bean reference. Then, using the TransactionDefinition and TransactionStatus objects you can initiate transactions, rollback and commit.

DefaultTransactionDefinition def = new DefaultTransactionDefinition()

def.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);

TransactionStatus status = transactionManager.getTransaction(def);

try {

// execute your business logic here

} catch (MyException ex) {

transactionManager.rollback(status);

throw ex;

}

transactionManager.commit(status);

### 7.4. Declarative transaction management

Spring also offers declarative transaction management. This is enabled by Spring AOP.

Most Spring users choose declarative transaction management. It is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with Spring declarative transaction management. The basic approach is similar: It's possible to specify transaction behavior (or lack of it) down to individual methods. It's possible to make a setRollbackOnly() call within a transaction context if necessary. The differences are:

Unlike EJB CMT, which is tied to JTA, Spring declarative transaction management works in any environment. It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only.

Spring enables declarative transaction management to be applied to any POJO, not just special classes such as EJBs.

Spring offers declarative rollback rules: a feature with no EJB equivalent, which we'll discuss below. Rollback can be controlled declaratively, not merely programmatically.

Spring gives you an opportunity to customize transactional behavior, using AOP. For example, if you want to insert custom behavior in the case of transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you have no way to influence the container's transaction management other than setRollbackOnly().

Spring does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, don't use this feature lightly. Normally we don't want transactions to span remote calls.

The concept of rollback rules is important: they enable us to specify which exceptions (and throwables) should cause automatic roll back. We specify this declaratively, in configuration, not in Java code. So, while we can still call setRollbackOnly() on the TransactionStatus object to roll the current transaction back programmatically, most often we can specify a rule that MyApplicationException should always result in roll back. This has the significant advantage that business objects don't need to depend on the transaction infrastructure. For example, they typically don't need to import any Spring APIs, transaction or other.

While the EJB default behavior is for the EJB container to automatically roll back the transaction on a system exception (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an application exception (checked exception other than java.rmi.RemoteException). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it's often useful to customize this.

On our benchmarks, the performance of Spring declarative transaction management exceeds that of EJB CMT.

The usual way of setting up transactional proxying in Spring is via the TransactionProxyFactoryBean. We need a target object to wrap in a transactional proxy. The target object is normally a POJO bean definition. When we define the TransactionProxyFactoryBean, we must supply a reference to the relevant PlatformTransactionManager, and transaction attributes. Transaction attributes contain the transaction definitions, discussed above. Consider the following sample:

<!-- this example is in verbose form, see note later about concise

for multiple proxies! -->

<!-- the target bean to wrap transactionally -->

<bean id="petStoreTarget">

...

</bean>

<bean id="petStore"

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

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

<property name="target"><ref bean="petStoreTarget"/></property>

<property name="transactionAttributes">

<props>

<prop key="insert\*">PROPAGATION\_REQUIRED,-MyCheckedException</prop>

<prop key="update\*">PROPAGATION\_REQUIRED</prop>

<prop key="\*">PROPAGATION\_REQUIRED,readOnly</prop>

</props>

</property>

</bean>

The transactional proxy will implement the interfaces of the target: in this case, the bean with id petStoreTarget. (Using CGLIB it's possible to transactionally proxy a target class. Set the "proxyTargetClass" property to true for this. It will happen automatically if the target doesn't implement any interfaces. In general, of course, we want to program to interfaces rather than classes.) It's possible (and usually a good idea) to restrict the transactional proxy to proxying only specific target interfaces, using the proxyInterfaces property. It's also possible to customize the behavior of a TransactionProxyFactoryBean via several properties inherited from org.springframework.aop.framework.ProxyConfig, and shared with all AOP proxy factories.

The transactionAttributes here are set using a Properties format defined in the org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource class. The mapping from method name, including wildcards, should be fairly intuitive. Note that the value for the insert\* mapping contains a rollback rule. Adding -MyCheckedException here specifies that if the method throws MyCheckedException or any subclasses, the transaction will automatically be rolled back. Multiple rollback rules can be specified here, comma-separated. A - prefix forces rollback; a + prefix specifies commit. (This allows commit even on unchecked exceptions, if you really know what you're doing!)

The TransactionProxyFactoryBean allows you to set "pre" and "post" advice, for additional interception behavior, using the "preInterceptors" and "postInterceptors" properties. Any number of pre and post advices can be set, and their type may be Advisor (in which case they can contain a pointcut), MethodInterceptor or any advice type supported by the current Spring configuration (such as ThrowsAdvice, AfterReturningtAdvice or BeforeAdvice, which are supported by default.) These advices must support a shared-instance model. If you need transactional proxying with advanced AOP features such as stateful mixins, it's normally best to use the generic org.springframework.aop.framework.ProxyFactoryBean, rather than the TransactionProxyFactoryBean convenience proxy creator.

It's also possible to set up autoproxying: that is, to configure the AOP framework so that classes are automatically proxied without needing individual proxy definitions.

Please see the chapter on AOP for more information and examples.

Note: Using TransactionProxyFactoryBean definitions in the form above can seem overly verbose when many almost identical transaction proxies need to be created. You will almost always want to take advantage of parent and child bean definitions, along with inner bean definitions, to significantly reduce the verbosity of your transaction proxy definitions, as described in Section 5.7, “Concise proxy definitions”.

You don't need to be an AOP expert--or indeed, to know much at all about AOP--to use Spring's declarative transaction management effectively. However, if you do want to become a "power user" of Spring AOP, you will find it easy to combine declarative transaction management with powerful AOP capabilities.

#### 7.4.1. BeanNameAutoProxyCreator, another declarative approach

TransactionProxyFactoryBean is very useful, and gives you full control when wrapping objects with a transactional proxy. Used with parent/child bean definitions and inner beans holding the target, it is generally the best choice for transactional wrapping. In the case that you need to wrap a number of beans in a completely identical fashion (for example, a boilerplate, 'make all methods transactional', using a BeanFactoryPostProcessor called BeanNameAutoProxyCreator can offer an alternative approach which can end up being even less verbose for this simplified use case.

To recap, once the ApplicationContext has read its initialization information, it instantiates any beans within it which implement the BeanPostProcessor interface, and gives them a chance to post-process all other beans in the ApplicationContext. So using this mechanism, a properly configured BeanNameAutoProxyCreator can be used to postprocess any other beans in the ApplicationContext (recognizing them by name), and wrap them with a transactional proxy. The actual transaction proxy produced is essentially identical to that produced by the use of TransactionProxyFactoryBean, so will not be discussed further.

Let us consider a sample configuration:

<!-- Transaction Interceptor set up to do PROPAGATION\_REQUIRED on all methods -->

<bean id="matchAllWithPropReq"

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

<property name="transactionAttribute"><value>PROPAGATION\_REQUIRED</value></property>

</bean>

<bean id="matchAllTxInterceptor"

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

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

<property name="transactionAttributeSource"><ref bean="matchAllWithPropReq"/></property>

</bean>

<!-- One BeanNameAutoProxyCreator handles all beans where we want all methods to use

PROPAGATION\_REQUIRED -->

<bean id="autoProxyCreator"

class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<property name="interceptorNames">

<list>

<idref local="matchAllTxInterceptor"/>

<idref bean="hibInterceptor"/>

</list>

</property>

<property name="beanNames">

<list>

<idref local="core-services-applicationControllerSevice"/>

<idref local="core-services-deviceService"/>

<idref local="core-services-authenticationService"/>

<idref local="core-services-packagingMessageHandler"/>

<idref local="core-services-sendEmail"/>

<idref local="core-services-userService"/>

</list>

</property>

</bean>

Assuming that we already have a TransactionManager instance in our ApplicationContext, the first thing we need to do is create a TransactionInterceptor instance to use. The TransactionInterceptor decides which methods to intercept based on a TransactionAttributeSource implementing object passed to it as a property. In this case, we want to handle the very simple case of matching all methods. This is not necessarily the most efficient approach, but it's very quick to set up, because we can use the special pre-defined MatchAlwaysTransactionAttributeSource, which simply matches all methods. If we wanted to be more specific, we could use other variants such as MethodMapTransactionAttributeSource, NameMatchTransactionAttributeSource, or AttributesTransactionAttributeSource.

Now that we have the transaction interceptor, we simply feed it to a BeanNameAutoProxyCreator instance we define, along with the names of 6 beans in the ApplicationContext that we want to wrap in an identical fashion. As you can see, the net result is significantly less verbose than it would have been to wrap 6 beans identically with TransactionProxyFactoryBean. Wrapping a 7th bean would add only one more line of config.

You may notice that we are able to apply multiple interceptors. In this case, we are also applying a HibernateInterceptor we have previously defined (bean id=hibInterceptor), which will manage Hibernate Sessions for us.

There is one thing to keep in mind, with regards to bean naming, when switching back and forth between the use of TransactionProxyFactoryBean, and BeanNameAutoProxyCreator. For the former, if the target bean is not defined as an inner bean, you normally give the target bean you want to wrap an id similar in form to myServiceTarget, and then give the proxy object an id of myService; then all users of the wrapped object simply refer to the proxy, i.e. myService. (These are just sample naming conventions, the point is that the target object has a different name than the proxy, and both are available from the ApplicationContext). However, when using BeanNameAutoProxyCreator, you name the target object something like myService. Then, when BeanNameAutoProxyCreator postprocesses the target object and create the proxy, it causes the proxy to be inserted into the Application context under the name of the original bean. From that point on, only the proxy (the wrapped object) is available from the ApplicationContext. When using TransactionProxyFactoryBean with the target specified as an inner bean, this naming issue is not a concern, since the inner bean is not normally given a name.

### 7.5. Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. Using the TransactionTemplate may be a good approach.

On the other hand, if your applications has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure in Spring. Using Spring, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

### 7.6. Do you need an application server for transaction management?

Spring's transaction management capabilities--and especially its declarative transaction management--significantly changes traditional thinking as to when a J2EE application requires an application server.

In particular, you don't need an application server just to have declarative transactions via EJB. In fact, even if you have an application server with powerful JTA capabilities, you may well decide that Spring declarative transactions offer more power and a much more productive programming model than EJB CMT.

You need an application server's JTA capability only if you need to enlist multiple transactional resources. Many applications don't face this requirement. For example, many high-end applications use a single, highly scalable, database such as Oracle 9i RAC.

Of course you may need other application server capabilities such as JMS and JCA. However, if you need only JTA, you could also consider an open source JTA add-on such as JOTM. (Spring integrates with JOTM out of the box.) However, as of early 2004, high-end application servers provide more robust support for XA transactions.

The most important point is that with Spring you can choose when to scale your application up to a full-blown application server. Gone are the days when the only alternative to using EJB CMT or JTA was to write coding using local transactions such as those on JDBC connections, and face a hefty rework if you ever needed that code to run within global, container-managed transactions. With Spring only configuration needs to change: your code doesn't.

### 7.7. Common problems

Developers should take care to use the correct PlatformTransactionManager implementation for their requirements.

It's important to understand how the Spring transaction abstraction works with JTA global transactions. Used properly, there is no conflict here: Spring merely provides a simplifying, portable abstraction.

If you are using global transactions, you must use the Spring org.springframework.transaction.jta.JtaTransactionManager for all your for all your transactional operations. Otherwise Spring will attempt to perform local transactions on resources such as container DataSources. Such local transactions don't make sense, and a good application server will treat them as errors.

## 8. Source Level Metadata Support

### 8.1. Source-level metadata

Source-level metadata is the addition of attributes or annotations to program elements: usually, classes and/or methods.

For example, we might add metadata to a class as follows:

/\*\*

\* Normal comments

\* @@org.springframework.transaction.interceptor.DefaultTransactionAttribute()

\*/

public class PetStoreImpl implements PetStoreFacade, OrderService {

We could add metadata to a method as follows:

/\*\*

\* Normal comments

\* @@org.springframework.transaction.interceptor.RuleBasedTransactionAttribute ()

\* @@org.springframework.transaction.interceptor.RollbackRuleAttribute (Exception.class)

\* @@org.springframework.transaction.interceptor.NoRollbackRuleAttribute ("ServletException")

\*/

public void echoException(Exception ex) throws Exception {

....

}

Both these examples use Jakarta Commons Attributes syntax.

Source-level metadata was introduced to the mainstream with the release of Microsoft's .NET platform, which uses source-level attributes to control transactions, pooling and other behaviour.

The value in this approach has been recognized in the J2EE community. For example, it's much less verbose than the traditional XML deployment descriptors exclusively used by EJB. While it is desirable to externalize some things from program source code, some important enterprise settings--notably transaction characteristics--belong in program source. Contrary to the assumptions of the EJB spec, it seldom makes sense to modify the transactional characteristics of a method.

Although metadata attributes are typically used mainly by framework infrastructure to describe the services application classes require, it should also be possible for metadata attributes to be queried at runtime. This is a key distinction from solutions such as XDoclet, which primarily view metadata as a way of generating code such as EJB artefacts.

There are a number of solutions in this space, including:

JSR-175: the standard Java metadata implementation, available in Java 1.5. But we need a solution now and may always want a facade

XDoclet: well-established solution, primarily intended for code generation

Various open source attribute implementations, pending the release of JSR-175, of which Commons Attributes appears to be the most promising. All these require a special pre- or post-compilation step.

### 8.2. Spring's metadata support

In keeping with its provision of abstractions over important concepts, Spring provides a facade to metadata implementations, in the form of the org.springframework.metadata.Attributes interface.

Such a facade adds value for several reasons:

There is currently no standard metadata solution. Java 1.5 will provide one, but it is still in beta as of Spring 1.0. Furthermore, there will be a need for metadata support in 1.3 and 1.4 applications for at least two years. Spring aims to provide working solutions now; waiting for 1.5 is not an option in such an important area.

Current metadata APIs, such as Commons Attributes (used by Spring 1.0) are hard to test. Spring provides a simple metadata interface that is much easier to mock.

Even when Java 1.5 provides metadata support at language level, there will still be value in providing such an abstraction:

JSR-175 metadata is static. It is associated with a class at compile time, and cannot be changed in a deployed environment. There is a need for hierarchical metadata, providing the ability to override certain attribute values in deployment--for example, in an XML file.

JSR-175 metadata is returned through the Java reflection API. This makes it impossible to mock during test time. Spring provides a simple interface to allow this.

Thus Spring will support JSR-175 before Java 1.5 reaches GA, but will continue to offer an attribute abstraction API.

The Spring Attributes interface looks like this:

public interface Attributes {

Collection getAttributes(Class targetClass);

Collection getAttributes(Class targetClass, Class filter);

Collection getAttributes(Method targetMethod);

Collection getAttributes(Method targetMethod, Class filter);

Collection getAttributes(Field targetField);

Collection getAttributes(Field targetField, Class filter);

}

This is a lowest common denominator interface. JSR-175 offers more capabilities than this, such as attributes on method arguments. As of Spring 1.0, Spring aims to provide the subset of metadata required to provide effective declarative enterprise services a la EJB or .NET. Beyond Spring 1.0, it is likely that Spring will provide further metadata methods.

Note that this interface offers Object attributes, like .NET. This distinguishes it from attribute systems such as that of Nanning Aspects and JBoss 4 (as of DR2), which offer only String attributes. There is a significant advantage in supporting Object attributes. It enables attributes to participate in class hierarchies and enables attributes to react intelligently to their configuration parameters.

In most attribute providers, attribute classes will be configured via constructor arguments or JavaBean properties. Commons Attributes supports both.

As with all Spring abstraction APIs, Attributes is an interface. This makes it easy to mock attribute implementations for unit tests.

### 8.3. Integration with Jakarta Commons Attributes

Presently Spring supports only Jakarta Commons Attributes out of the box, although it is easy to provide implementations of the org.springframework.metadata.Attributes interface for other metadata providers.

Commons Attributes 2.1 (http://jakarta.apache.org/commons/attributes/) is a capable attributes solution. It supports attribute configuration via constructor arguments and JavaBean properties, which offers better self-documentation in attribute definitions. (Support for JavaBean properties was added at the request of the Spring team.)

We've already seen two examples of Commons Attributes attributes definitions. In general, we will need to express:

The name of the attribute class. This can be an FQN, as shown above. If the relevant attribute class has already been imported, the FQN isn't required. It's also possible to specify "attribute packages" in attribute compiler configuration.

Any necessary parameterization, via constructor arguments or JavaBean properties

Bean properties look as follows:

/\*\*

\* @@MyAttribute(myBooleanJavaBeanProperty=true)

\*/

It's possible to combine constructor arguments and JavaBean properties (as in Spring IoC).

Because, unlike Java 1.5 attributes, Commons Attributes is not integrated with the Java language, it is necessary to run a special attribute compilation step as part of the build process.

To run Commons Attributes as part of the build process, you will need to do the following.

1. Copy the necessary library Jars to $ANT\_HOME/lib. Four Jars are required, and all are distributed with Spring:

The Commons Attributes compiler Jar and API Jar

xjavadoc.jar, from XDoclet

commons-collections.jar, from Jakarta Commons

2. Import the Commons Attributes ant tasks into your project build script, as follows:

<taskdef resource="org/apache/commons/attributes/anttasks.properties"/>

3. Next, define an attribute compilation task, which will use the Commons Attributes attribute-compiler task to "compile" the attributes in the source. This process results in the generation of additional sources, to a location specified by the destdir attribute. Here we show the use of a temporary directory:

<target name="compileAttributes" >

<attribute-compiler

destdir="${commons.attributes.tempdir}"

>

<fileset dir="${src.dir}" includes="\*\*/\*.java"/>

</attribute-compiler>

</target>

The compile target that runs Javac over the sources should depend on this attribute compilation task, and must also compile the generated sources, which we output to our destination temporary directory. If there are syntax errors in your attribute definitions, they will normally be caught by the attribute compiler. However, if the attribute definitions are syntactically plausible, but specify invalid types or class names, the compilation of the generated attribute classes may fail. In this case, you can look at the generated classes to establish the cause of the problem.

Commons Attributes also provides Maven support. Please refer to Commons Attributes documentation for further information.

While this attribute compilation process may look complex, in fact it's a one-off cost. Once set up, attribute compilation is incremental, so it doesn't usually noticeably slow the build process. And once the compilation process is set up, you may find that use of attributes as described in this chapter can save you a lot of time in other areas.

If you require attribute indexing support (only currently required by Spring for attribute-targeted web controllers, discussed below), you will need an additional step, which must be performed on a Jar file of your compiled classes. In this, optional, step, Commons Attributes will create an index of all the attributes defined on your sources, for efficient lookup at runtime. This step looks as follows:

<attribute-indexer jarFile="myCompiledSources.jar">

<classpath refid="master-classpath"/>

</attribute-indexer>

See the /attributes directory of the Spring jPetStore sample application for an example of this build process. You can take the build script it contains and modify it for your own projects.

If your unit tests depend on attributes, try to express the dependency on the Spring Attributes abstraction, rather than Commons Attributes. Not only is this more portable--for example, your tests will still work if you switch to Java 1.5 attributes in future--it simplifies testing. Commons Attributes is a static API, while Spring provides a metadata interface that you can easily mock.

### 8.4. Metadata and Spring AOP autoproxying

The most important uses of metadata attributes are in conjunction with Spring AOP. This provides a .NET-like programming model, where declarative services are automatically provided to application objects that declare metadata attributes. Such metadata attributes can be supported out of the box by the framework, as in the case of declarative transaction management, or can be custom.

There is widely held to be a synergy between AOP and metadata attributes.

#### 8.4.1. Fundamentals

This builds on the Spring AOP autoproxy functionality. Configuration might look like this:

<bean id="autoproxy"

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">

</bean>

<bean id="transactionAttributeSource"

class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource"

autowire="constructor">

</bean>

<bean id="transactionInterceptor"

class="org.springframework.transaction.interceptor.TransactionInterceptor"

autowire="byType">

</bean>

<bean id="transactionAdvisor"

class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor"

autowire="constructor" >

</bean>

<bean id="attributes"

class="org.springframework.metadata.commons.CommonsAttributes"

/>

The basic concepts here should be familiar from the discussion of autoproxying in the AOP chapter.

The most important bean definitions are those named autoproxy and transactionAdvisor. Note that the actual bean names are not important; what matters is their class.

The autoproxy bean definition of class org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator will automatically advise ("autoproxy") all bean instances in the current factory based on matching Advisor implementations. This class knows nothing about attributes, but relies on Advisors' pointcuts matching. The pointcuts do know about attributes.

Thus we simply need an AOP advisor that will provide declarative transaction management based on attributes.

It's possible to add arbitrary custom Advisor implementations as well, and they will also be evaluated and applied automatically. (You can use Advisors whose pointcuts match on criteria besides attributes in the same autoproxy configuration, if necessary.)

Finally, the attributes bean is the Commons Attributes Attributes implementation. Replace with another implementation of org.springframework.metadata.Attributes to source attributes from a different source.

#### 8.4.2. Declarative transaction management

The commonest use of source-level attributes it to provide declarative transaction management a la .NET. Once the bean definitions shown above are in place, you can define any number of application objects requiring declarative transactions. Only those classes or methods with transaction attributes will be given transaction advice. You need to do nothing except define the required transaction attributes.

Unlike in .NET, you can specify transaction attributes at either class or method level. Class-level attributes, if specified, will be "inherited" by all methods. Method attributes will wholly override any class-level attributes.

#### 8.4.3. Pooling

Again, as with .NET, you can enable pooling behavior via class-level attributes. Spring can apply this behavior to any POJO. You simply need to specify a pooling attribute, as follows, in the business object to be pooled:

/\*\*

\* @@org.springframework.aop.framework.autoproxy.target.PoolingAttribute (10)

\*

\* @author Rod Johnson

\*/

public class MyClass {

You'll need the usual autoproxy infrastructure configuration. You then need to specify a pooling TargetSourceCreator, as follows. Because pooling affects the creation of the target, we can't use a regular advice. Note that pooling will apply even if there are no advisors applicable to the class, if that class has a pooling attribute.

<bean id="poolingTargetSourceCreator"

class="org.springframework.aop.framework.autoproxy.metadata.AttributesPoolingTargetSourceCreator"

autowire="constructor" >

</bean>

The relevant autoproxy bean definition needs to specify a list of "custom target source creators", including the Pooling target source creator. We could modify the example shown above to include this property as follows:

<bean id="autoproxy"

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">

>

<property name="customTargetSourceCreators">

<list>

<ref local="poolingTargetSourceCreator" />

</list>

</property>

</bean>

As with the use of metadata in Spring in general, this is a one-off cost: once setup is out of the way, it's very easy to use pooling for additional business objects.

It's arguable that the need for pooling is rare, so there's seldom a need to apply pooling to a large number of business objects. Hence this feature does not appear to be used often.

Please see the Javadoc for the org.springframework.aop.framework.autoproxy package for more details. It's possible to use a different pooling implementation than Commons Pool with minimal custom coding.

#### 8.4.4. Custom metadata

We can even go beyond the capabilities of .NET metadata attributes, because of the flexibility of the underlying autoproxying infrastructure.

We can define custom attributes, to provide any kind of declarative behavior. To do this, you need to:

Define your custom attribute class

Define a Spring AOP Advisor with a pointcut that fires on the presence of this custom attribute.

Add that Advisor as a bean definition to an application context with the generic autoproxy infrastructure in place.

Add attributes to your POJOs.

There are several potential areas you might want to do this, such as custom declarative security, or possibly caching.

This is a powerful mechanism which can significantly reduce configuration effort in some projects. However, remember that it does rely on AOP under the covers. The more Advisors you have in play, the more complex your runtime configuration will be.(If you want to see what advice applies to any object, try casting a reference to org.springframework.aop.framework.Advised. This will enable you to examine the Advisors.)

### 8.5. Using attributes to minimize MVC web tier configuration

The other main use of Spring metadata as of 1.0 is to provide an option to simplify Spring MVC web configuration.

Spring MVC offers flexible handler mappings: mappings from incoming request to controller (or other handler) instance. Normally handler mappings are configured in the xxxx-servlet.xml file for the relevant Spring DispatcherServlet.

Holding these mappings in the DispatcherServlet configuration file is normally A Good Thing. It provides maximum flexibility. In particular:

The controller instance is explicitly managed by Spring IoC, through an XML bean definition

The mapping is external to the controller, so the same controller instance could be given multiple mappings in the same DispatcherServlet context or reused in a different configuration.

Spring MVC is able to support mappings based on any criteria, rather than merely the request URL-to-controller mappings available in most other frameworks.

However, this does mean that for each controller we typically need both a handler mapping (normally in a handler mapping XML bean definition) and an XML mapping for the controller itself.

Spring offers a simpler approach based on source-level attributes, which is an attractive option in simpler scenarios.

The approach described in this section is best suited to relatively simple MVC scenarios. It sacrifices some of the power of Spring MVC, such as the ability to use the same controller with different mappings, and the ability to base mappings on something other than request URL.

In this approach, controllers are marked with one or more class-level metadata attributes, each specifying one URL they should be mapped to.

The following examples show the approach. In each case, we have a controller that depends on a business object of type Cruncher. As usual, this dependency will be resolved by Dependency Injection. The Cruncher must be available through a bean definition in the relevant DispatcherServlet XML file, or a parent context.

We attach an attribute to the controller class specifying the URL that should map to it. We can express the dependency through a JavaBean property or a constructor argument. This dependency must be resolvable by autowiring: that is, there must be exactly one business object of type Cruncher available in the context.

/\*\*

\* Normal comments here

\* @author Rod Johnson

\* @@org.springframework.web.servlet.handler.metadata.PathMap("/bar.cgi")

\*/

public class BarController extends AbstractController {

private Cruncher cruncher;

public void setCruncher(Cruncher cruncher) {

this.cruncher = cruncher;

}

protected ModelAndView handleRequestInternal(

HttpServletRequest arg0, HttpServletResponse arg1)

throws Exception {

System.out.println("Bar Crunching c and d =" +

cruncher.concatenate("c", "d"));

return new ModelAndView("test");

}

}

For this auto-mapping to work, we need to add the following to the relevant xxxx-servlet.xml file, specifying the attributes handler mapping. This special handler mapping can handle any number of controllers with attributes as shown above. The bean id ("commonsAttributesHandlerMapping") is not important. The type is what matters:

<bean id="commonsAttributesHandlerMapping"

class="org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping"

/>

We do not currently need an Attributes bean definition, as in the above example, because this class works directly with the Commons Attributes API, not via the Spring metadata abstraction.

We now need no XML configuration for each controller. Controllers are automatically mapped to the specified URL(s). Controllers benefit from IoC, using Spring's autowiring capability. For example, the dependency expressed in the "cruncher" bean property of the simple controller shown above is automatically resolved in the current web application context. Both Setter and Constructor Dependency Injection are available, each with zero configuration.

An example of Constructor Injection, also showing multiple URL paths:

/\*\*

\* Normal comments here

\* @author Rod Johnson

\*

\* @@org.springframework.web.servlet.handler.metadata.PathMap("/foo.cgi")

\* @@org.springframework.web.servlet.handler.metadata.PathMap("/baz.cgi")

\*/

public class FooController extends AbstractController {

private Cruncher cruncher;

public FooController(Cruncher cruncher) {

this.cruncher = cruncher;

}

protected ModelAndView handleRequestInternal(

HttpServletRequest arg0, HttpServletResponse arg1)

throws Exception {

return new ModelAndView("test");

}

}

This approach has the following benefits:

Significantly reduced volume of configuration. Each time we add a controller we need add no XML configuration. As with attribute-driven transaction management, once the basic infrastructure is in place, it is very easy to add more application classes.

We retain much of the power of Spring IoC to configure controllers.

This approach has the following limitations:

One-off cost in more complex build process. We need an attribute compilation step and an attribute indexing step. However, once in place, this should not be an issue.

Currently Commons Attributes only, although support for other attribute providers may be added in future.

Only "autowiring by type" dependency injection is supported for such controllers. However, this still leaves them far in advance of Struts Actions (with no IoC support from the framework) and, arguably, WebWork Actions (with only rudimentary IoC support) where IoC is concerned.

Reliance on automagical IoC resolution may be confusing.

Because autowiring by type means there must be exactly one dependency of the specified type, we need to be careful if we use AOP. In the common case using TransactionProxyFactoryBean, for example, we end up with two implementations of a business interface such as Cruncher: the original POJO definition, and the transactional AOP proxy. This won't work, as the owning application context can't resolve the type dependency unambiguously. The solution is to use AOP autoproxying, setting up the autoproxy infrastructure so that there is only one implementation of Cruncher defined, and that implementation is automatically advised. Thus this approach works well with attribute-targeted declarative services as described above. As the attributes compilation process must be in place to handle the web controller targeting, this is easy to set up.

Unlike other metadata functionality, there is currently only a Commons Attributes implementation available: org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping. This limitation is due to the fact that not only do we need attribute compilation, we need attribute indexing: the ability to ask the attributes API for all classes with the PathMap attribute. Indexing is not currently offered on the org.springframework.metadata.Attributes abstraction interface, although it may be in future. (If you want to add support for another attributes implementation--which must support indexing--you can easily extend the AbstractPathMapHandlerMapping superclass of CommonsPathMapHandlerMapping, implementing the two protected abstract methods to use your preferred attributes API.)

Thus we need two additional steps in the build process: attribute compilation and attribute indexing. Use of the attribute indexer task was shown above. Note that Commons Attributes presently requires a Jar file as input to indexing.

If you begin with a handler metadata mapping approach, it is possible to switch at any point to a classic Spring XML mapping approach. So you don't close off this option. For this reason, I find that I often start a web application using metadata mapping.

### 8.6. Other uses of metadata attributes

Other uses of metadata attributes appear to be growing in popularity. As of March 2004, an attribute-based validation package for Spring is in development. The one-off setup cost of attribute parsing looks more attractive, when the potential for multiple uses is considered.

### 8.7. Adding support for additional metadata APIs

Should you wish to provide support for another metadata API it is easy to do so.

Simply implement the org.springframework.metadata.Attributes interface as a facade for your metadata API. You can then include this object in your bean definitions as shown above.

All framework services that use metadata, such as AOP metadata-driven autoproxying, will then automatically be able to use your new metadata provider.

We expect to add support for Java 1.5 attributes--probably as an add-on to the Spring core--in Q2 2004.

## 9. DAO support

### 9.1. Introduction

The DAO (Data Access Object) support in Spring is primarily aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way. This allows you to switch between them fairly easily and it also allows you to code without worrying about catching exceptions that are specific to each technology.

### 9.2. Consistent Exception Hierarchy

Spring provides a convenient translation from technology specific exceptions like SQLException to its own exception hierarchy with the DataAccessException as the root exception. These exceptions wrap the original exception so there is never any risk that you would lose any information as to what might have gone wrong.

In addition to JDBC exceptions, Spring can also wrap Hibernate exceptions, converting them from proprietary, checked exceptions, to a set of abstracted runtime exceptions. The same is true for JDO exceptions. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. As we mentioned above, JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

The above is true for the Template versions of the ORM access framework. If you use the Interceptor based classes then the application must care about handling HibernateExceptions and JDOExceptions itself, preferably via delegating to SessionFactoryUtils' convertHibernateAccessException or convertJdoAccessException methods respectively. These methods converts the exceptions to ones that are compatible with the org.springframework.dao exception hierarchy. As JDOExceptions are unchecked, they can simply get thrown too, sacrificing generic DAO abstraction in terms of exceptions though.

The exception hierarchy that Spring uses is outlined in the following graph:

### 9.3. Consistent Abstract Classes for DAO Support

To make it easier to work with a variety of data access technologies like JDBC, JDO and Hibernate in a consistent way, Spring provides a set of abstract DAO classes that you can extend. These abstract classes has methods for setting the data source and any other configuration settings that are specific to the technology you currently are using.

Dao Support classes:

JdbcDaoSupport - super class for JDBC data access objects. Requires a DataSource to be set, providing a JdbcTemplate based on it to subclasses.

HibernateDaoSupport - super class for Hibernate data access objects. Requires a SessionFactory to be set, providing a HibernateTemplate based on it to subclasses. Can alternatively be initialized directly via a HibernateTemplate, to reuse the latter's settings like SessionFactory, flush mode, exception translator, etc.

JdoDaoSupport - super class for JDO data access objects. Requires a PersistenceManagerFactory to be set, providing a JdoTemplate based on it to subclasses.

## 10. Data Access using JDBC

### 10.1. Introduction

The JDBC abstraction framework provided by Spring consists of four different packages core, datasource, object, and support.

The org.springframework.jdbc.core package contains the JdbcTemplate class and its various callback interfaces, plus a variety of related classes.

The org.springframework.jdbc.datasource package contains a utility class for easy DataSource access, and various simple DataSource implementations that can be used for testing and running unmodified JDBC code outside of a J2EE container. The utility class provides static methods to obtain connections from JNDI and to close connections if necessary. It has support for thread-bound connections, e.g. for use with DataSourceTransactionManager.

Next, the org.springframework.jdbc.object package contains classes that represent RDBMS queries, updates, and stored procedures as thread safe, reusable objects. This approach is modeled by JDO, although of course objects returned by queries are “disconnected” from the database. This higher level of JDBC abstraction depends on the lower-level abstraction in the org.springframework.jdbc.core package.

Finally the org.springframework.jdbc.support package is where you find the SQLException translation functionality and some utility classes.

Exceptions thrown during JDBC processing are translated to exceptions defined in the org.springframework.dao package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated exceptions are unchecked giving you the option of catching the exceptions that you can recover from while allowing other exceptions to be propagated to the caller.

### 10.2. Using the JDBC Core classes to control basic JDBC processing and error handling

#### 10.2.1. JdbcTemplate

This is the central class in the JDBC core package. It simplifies the use of JDBC since it handles the creation and release of resources. This helps to avoid common errors like forgetting to always close the connection. It executes the core JDBC workflow like statement creation and execution, leaving application code to provide SQL and extract results. This class executes SQL queries, update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Code using this class only need to implement callback interfaces, giving them a clearly defined contract. The PreparedStatementCreator callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The same is true for the CallableStatementCreateor interface which creates callable statement. The RowCallbackHandler interface extracts values from each row of a ResultSet.

This class can be used within a service implementation via direct instantiation with a DataSource reference, or get prepared in an application context and given to services as bean reference. Note: The DataSource should always be configured as a bean in the application context, in the first case given to the service directly, in the second case to the prepared template. Because this class is parameterizable by the callback interfaces and the SQLExceptionTranslator interface, it isn't necessary to subclass it. All SQL issued by this class is logged.

#### 10.2.2. DataSource

In order to work with data from a database, we need to obtain a connection to the database. The way Spring does this is through a DataSource. A DataSource is part of the JDBC specification and can be seen as a generalized connection factory. It allows a container or a framework to hide connection pooling and transaction management issues from the application code. As a developer, you don't need to know any details about how to connect to the database, that is the responsibility for the administrator that sets up the datasource. You will most likely have to fulfill both roles while you are developing and testing you code though, but you will not necessarily have to know how the production data source is configured.

When using Spring's JDBC layer, you can either obtain a data source from JNDI or you can configure your own, using an implementation that is provided in the Spring distribution. The latter comes in handy for unit testing outside of a web container. We will use the DriverManagerDataSource implementation for this section but there are several additional implementations that will be covered later on. The DriverManagerDataSource works the same way that you probably are used to work when you obtain a JDBC connection. You have to specify the fully qualified class name of the JDBC driver that you are using so that the DriverManager can load the driver class. Then you have to provide a url that varies between JDBC drivers. You have to consult the documentation for your driver for the correct value to use here. Finally you must provide a username and a password that will be used to connect to the database. Here is an example of how to configure a DriverManagerDataSource:

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName( "org.hsqldb.jdbcDriver");

dataSource.setUrl( "jdbc:hsqldb:hsql://localhost:");

dataSource.setUsername( "sa");

dataSource.setPassword( "");

#### 10.2.3. SQLExceptionTranslator

SQLExceptionTranslator is an interface to be implemented by classes that can translate between SQLExceptions and our data access strategy-agnostic org.springframework.dao.DataAccessException.

Implementations can be generic (for example, using SQLState codes for JDBC) or proprietary (for example, using Oracle error codes) for greater precision.

SQLErrorCodeSQLExceptionTranslator is the implementation of SQLExceptionTranslator that is used by default. This implementation uses specific vendor codes. More precise than SQLState implementation, but vendor specific. The error code translations are based on codes held in a JavaBean type class named SQLErrorCodes. This class is created and populated by an SQLErrorCodesFactory which as the name suggests is a factory for creating SQLErrorCodes based on the contents of a configuration file named "sql-error-codes.xml". This file is populated with vendor codes and based on the DatabaseProductName taken from the DatabaseMetaData, the codes for the current database are used.

The SQLErrorCodeSQLExceptionTranslator applies the following matching rules:

Try custom translation implemented by any subclass. Note that this class is concrete and is typically used itself, in which case this rule doesn't apply.

Apply error code matching. Error codes are obtained from the SQLErrorCodesFactory by default. This looks up error codes from the classpath and keys into them from the database name from the database metadata.

Use the fallback translator. SQLStateSQLExceptionTranslator is the default fallback translator.

SQLErrorCodeSQLExceptionTranslator can be extended the following way:

public class MySQLErrorCodesTranslator extends SQLErrorCodeSQLExceptionTranslator {

protected DataAccessException customTranslate(String task, String sql, SQLException sqlex) {

if (sqlex.getErrorCode() == -12345)

return new DeadlockLoserDataAccessException(task, sqlex);

return null;

}

}

In this example the specific error code '-12345' is translated and any other errors are simply left to be translated by the default translator implementation. To use this custom translator, it is necessary to pass it to the JdbcTemplate using the method setExceptionTranslator and to use this JdbcTemplate for all of the data access processing where this translator is needed. Here is an example of how this custom translator can be used:

// create a JdbcTemplate and set data source

JdbcTemplate jt = new JdbcTemplate();

jt.setDataSource(dataSource);

// create a custom translator and set the datasource for the default translation lookup

MySQLErrorCodesTransalator tr = new MySQLErrorCodesTransalator();

tr.setDataSource(dataSource);

jt.setExceptionTranslator(tr);

// use the JdbcTemplate for this SqlUpdate

SqlUpdate su = new SqlUpdate();

su.setJdbcTemplate(jt);

su.setSql("update orders set shipping\_charge = shipping\_charge \* 1.05");

su.compile();

su.update();

The custom translator is passed a data source because we still want the default translation to look up the error codes in sql-error-codes.xml.

#### 10.2.4. Executing Statements

To execute an SQL statement, there is very little code needed. All you need is a DataSource and a JdbcTemplate. Once you have that, you can use a number of convenience methods that are provided with the JdbcTemplate. Here is a short example showing what you need to include for a minimal but fully functional class that creates a new table.

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {

private JdbcTemplate jt;

private DataSource dataSource;

public void doExecute() {

jt = new JdbcTemplate(dataSource);

jt.execute("create table mytable (id integer, name varchar(100))");

}

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

}

#### 10.2.5. Running Queries

In addition to the execute methods, there is a large number of query methods. Some of these methods are intended to be used for queries that return a single value. Maybe you want to retrieve a count or a specific value from one row. If that is the case then you can use queryForInt,queryForLong or queryForObject. The latter will convert the returned JDBC Type to the Java class that is passed in as an argument. If the type conversion is invalid, then an InvalidDataAccessApiUsageException will be thrown. Here is an example that contains two query methods, one for an int and one that queries for a String.

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

private JdbcTemplate jt;

private DataSource dataSource;

public int getCount() {

jt = new JdbcTemplate(dataSource);

int count = jt.queryForInt("select count(\*) from mytable");

return count;

}

public String getName() {

jt = new JdbcTemplate(dataSource);

String name = (String) jt.queryForObject("select name from mytable", java.lang.String.class);

return name;

}

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

}

In addition to the singe results query methods there are several methods that return a List with an entry for each row that the query returned. The most generic one is queryForList which returns a List where each entry is a Map with each entry in the map representing the column value for that row. If we add a method to the above example to retrieve a list of all the rows, it would look like this:

public List getList() {

jt = new JdbcTemplate(dataSource);

List rows = jt.queryForList("select \* from mytable");

return rows;

}

The list returned would look something like this: [{name=Bob, id=1}, {name=Mary, id=2}].

#### 10.2.6. Updating the database

There are also a number of update methods that you can use. I will show an example where we update a column for a certain primary key. In this example I am using an SQL statement that has place holders for row parameters. Most of the query and update methods have this functionality. The parameter values are passed in as an array of objects.

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

private JdbcTemplate jt;

private DataSource dataSource;

public void setName(int id, String name) {

jt = new JdbcTemplate(dataSource);

jt.update("update mytable set name = ? where id = ?", new Object[] {name, new Integer(id)});

}

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

}

### 10.3. Controlling how we connect to the database

#### 10.3.1. DataSourceUtils

Helper class that provides static methods to obtain connections from JNDI and close connections if necessary. Has support for thread-bound connections, e.g. for use with DataSourceTransactionManager.

Note: The getDataSourceFromJndi methods are targeted at applications that do not use a BeanFactory resp. an ApplicationContext. With the latter, it is preferable to preconfigure your beans or even JdbcTemplate instances in the factory: JndiObjectFactoryBean can be used to fetch a DataSource from JNDI and give the DataSource bean reference to other beans. Switching to another DataSource is just a matter of configuration then: You can even replace the definition of the FactoryBean with a non-JNDI DataSource!

#### 10.3.2. SmartDataSource

Interface to be implemented by classes that can provide a connection to a relational database. Extends the javax.sql.DataSource interface to allow classes using it to query whether or not the connection should be closed after a given operation. This can sometimes be useful for efficiency, if we know that we want to reuse a connection.

#### 10.3.3. AbstractDataSource

Abstract base class for Spring's DataSource implementations, taking care of the "uninteresting" glue. This is the class you would extend if you are writing your own DataSource implementation.

#### 10.3.4. SingleConnectionDataSource

Implementation of SmartDataSource that wraps a single connection which is not closed after use. Obviously, this is not multi-threading capable.

If client code will call close in the assumption of a pooled connection, like when using persistence tools, set suppressClose to true. This will return a close-suppressing proxy instead of the physical connection. Be aware that you will not be able to cast this to a native Oracle Connection or the like anymore.

This is primarily a test class. For example, it enables easy testing of code outside an application server, in conjunction with a simple JNDI environment. In contrast to DriverManagerDataSource, it reuses the same connection all the time, avoiding excessive creation of physical connections.

#### 10.3.5. DriverManagerDataSource

Implementation of SmartDataSource that configures a plain old JDBC Driver via bean properties, and returns a new connection every time.

Useful for test or standalone environments outside of a J2EE container, either as a DataSource bean in a respective ApplicationContext, or in conjunction with a simple JNDI environment. Pool-assuming Connection.close() calls will simply close the connection, so any DataSource-aware persistence code should work.

#### 10.3.6. DataSourceTransactionManager

PlatformTransactionManager implementation for single JDBC data sources. Binds a JDBC connection from the specified data source to the thread, potentially allowing for one thread connection per data source.

Application code is required to retrieve the JDBC connection via DataSourceUtils.getConnection(DataSource) instead of J2EE's standard DataSource.getConnection. This is recommended anyway, as it throws unchecked org.springframework.dao exceptions instead of checked SQLException. All framework classes like JdbcTemplate use this strategy implicitly. If not used with this transaction manager, the lookup strategy behaves exactly like the common one - it can thus be used in any case.

Supports custom isolation levels, and timeouts that get applied as appropriate JDBC statement query timeouts. To support the latter, application code must either use JdbcTemplate or call DataSourceUtils.applyTransactionTimeout method for each created statement.

This implementation can be used instead of JtaTransactionManager in the single resource case, as it does not require the container to support JTA. Switching between both is just a matter of configuration, if you stick to the required connection lookup pattern. Note that JTA does not support custom isolation levels!

### 10.4. Modeling JDBC operations as Java objects

The org.springframework.jdbc.object package contains the classes that allow you to access the database in a more object oriented manner. You can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. You can also execute stored procedures and run update, delete and insert statements.

#### 10.4.1. SqlQuery

Reusable thread safe object to represent an SQL query. Subclasses must implement the newResultReader() method to provide an object that can save the results while iterating over the ResultSet. This class is rarely used directly since the MappingSqlQuery, that extends this class, provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend SqlQuery are MappingSqlQueryWithParameters and UpdatableSqlQuery.

#### 10.4.2. MappingSqlQuery

MappingSqlQuery is a reusable query in which concrete subclasses must implement the abstract mapRow(ResultSet, int) method to convert each row of the JDBC ResultSet into an object.

Of all the SqlQuery implementations, this is the one used most often and it is also the one that is the easiest to use.

Here is a brief example of a custom query that maps the data from the customer table to a Java object called Customer.

private class CustomerMappingQuery extends MappingSqlQuery {

public CustomerMappingQuery(DataSource ds) {

super(ds, "SELECT id, name FROM customer WHERE id = ?");

super.declareParameter(new SqlParameter("id", Types.INTEGER));

compile();

}

public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {

Customer cust = new Customer();

cust.setId((Integer) rs.getObject("id"));

cust.setName(rs.getString("name"));

return cust;

}

}

We provide a constructor for this customer query that takes the DataSource as the only parameter. In this constructor we call the constructor on the superclass with the DataSource and the SQL that should be executed to retrieve the rows for this query. This SQL will be used to create a PreparedStatement so it may contain place holders for any parameters to be passed in during execution. Each parameter must be declared using the declareParameter method passing in an SqlParameter. The SqlParameter takes a name and the JDBC type as defined in java.sql.Types. After all parameters have been defined we call the compile method so the statement can be prepared and later be executed.

Let's take a look at the code where this custom query is instantiated and executed:

public Customer getCustomer(Integer id) {

CustomerMappingQuery custQry = new CustomerMappingQuery(dataSource);

Object[] parms = new Object[1];

parms[0] = id;

List customers = custQry.execute(parms);

if (customers.size() > 0)

return (Customer) customers.get(0);

else

return null;

}

The method in this example retrieves the customer with the id that is passed in as the only parameter. After creating an instance of the CustomerMappingQuery class we create an array of objects that will contain all parameters that are passed in. In this case there is only one parameter and it is passed in as an Integer. Now we are ready to execute the query using this array of parameters and we get a List that contains a Customer object for each row that was returned for our query. In this case it will only be one entry if there was a match.

#### 10.4.3. SqlUpdate

RdbmsOperation subclass representing a SQL update. Like a query, an update object is reusable. Like all RdbmsOperation objects, an update can have parameters and is defined in SQL.

This class provides a number of update() methods analogous to the execute() methods of query objects.

This class is concrete. Although it can be subclassed (for example to add a custom update method) it can easily be parameterized by setting SQL and declaring parameters.

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;

import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating extends SqlUpdate {

public UpdateCreditRating(DataSource ds) {

setDataSource(ds);

setSql("update customer set credit\_rating = ? where id = ?");

declareParameter(new SqlParameter(Types.NUMERIC));

declareParameter(new SqlParameter(Types.NUMERIC));

compile();

}

/\*\*

\* @param id for the Customer to be updated

\* @param rating the new value for credit rating

\* @return number of rows updated

\*/

public int run(int id, int rating) {

Object[] params =

new Object[] {

new Integer(rating),

new Integer(id)};

return update(params);

}

}

#### 10.4.4. StoredProcedure

Superclass for object abstractions of RDBMS stored procedures. This class is abstract and its execute methods are protected, preventing use other than through a subclass that offers tighter typing.

The inherited sql property is the name of the stored procedure in the RDBMS. Note that JDBC 3.0 introduces named parameters, although the other features provided by this class are still necessary in JDBC 3.0.

Here is an example of a program that calls a function sysdate() that comes with any Oracle database. To use the stored procedure functionality you have to create a class that extends StoredProcedure. There are no input parameters, but there is an output parameter that is declared as a date using the class SqlOutParameter. The execute() method returns a map with an entry for each declared output parameter using the parameter name as the key.

import java.sql.Types;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlOutParameter;

import org.springframework.jdbc.datasource.\*;

import org.springframework.jdbc.object.StoredProcedure;

public class TestStoredProcedure {

public static void main(String[] args) {

TestStoredProcedure t = new TestStoredProcedure();

t.test();

System.out.println("Done!");

}

void test() {

DriverManagerDataSource ds = new DriverManagerDataSource();

ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");

ds.setUrl("jdbc:oracle:thin:@localhost:1521:mydb");

ds.setUsername("scott");

ds.setPassword("tiger");

MyStoredProcedure sproc = new MyStoredProcedure(ds);

Map res = sproc.execute();

printMap(res);

}

private class MyStoredProcedure extends StoredProcedure {

public static final String SQL = "sysdate";

public MyStoredProcedure(DataSource ds) {

setDataSource(ds);

setFunction(true);

setSql(SQL);

declareParameter(new SqlOutParameter("date", Types.DATE));

compile();

}

public Map execute() {

Map out = execute(new HashMap());

return out;

}

}

private static void printMap(Map r) {

Iterator i = r.entrySet().iterator();

while (i.hasNext()) {

System.out.println((String) i.next().toString());

}

}

}

#### 10.4.5. SqlFunction

SQL "function" wrapper for a query that returns a single row of results. The default behavior is to return an int, but that can be overridden by using the methods with an extra return type parameter. This is similar to using the queryForXxx methods of the JdbcTemplate. The advantage with SqlFunction is that you don't have to create the JdbcTemplate, it is done behind the scenes.

This class is intended to use to call SQL functions that return a single result using a query like "select user()" or "select sysdate from dual". It is not intended for calling more complex stored functions or for using a CallableStatement to invoke a stored procedure or stored function. Use StoredProcedure or SqlCall for this type of processing.

This is a concrete class, which there is normally no need to subclass. Code using this package can create an object of this type, declaring SQL and parameters, and then invoke the appropriate run method repeatedly to execute the function. Here is an example of retrieving the count of rows from a table:

public int countRows() {

SqlFunction sf = new SqlFunction(dataSource, "select count(\*) from mytable");

sf.compile();

return sf.run();

}

## 11. Data Access using O/R Mappers

### 11.1. Introduction

Spring provides integration with Hibernate, JDO, and iBATIS SQL Maps in terms of resource management, DAO implementation support, and transaction strategies. For Hibernate there is first-class support with lots of IoC convenience features, addressing many typical Hibernate integration issues. All of these comply with Spring's generic transaction and DAO exception hierarchies.

Spring's adds significant support when using the O/R mapping layer of your choice to create data-access applications. First of all you should know that once you started using Spring's support for O/R mapping, you don't have to go all the way. No matter to what extent, you're invited to review and leverage the Spring approach, before deciding to take the effort and risk of building a similar infrastructure in-house. Much of the O/R mapping support, no matter what technology you're using may be used in a library style, as everything is designed as a set of reusable JavaBeans. Usage inside an ApplicationContext or BeanFactory does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside an ApplicationContext.

Some of the the benefits of using Spring to create your O/R mapping applications include:

To avoid vendor lock-in, and allow mix-and-match implementation strategies. While Hibernate is powerful, flexible, open source and free, it still uses a proprietary API. Furthermore one could argue that iBatis is a bit lightweight, although it's excellent for use in application that don't require complex O/R mapping strategies. Given the choice, it's usually desirable to implement major application functionality using standard or abstracted APIs, in case you need to switch to another implementation for reasons of functionality, performance, or any other concerns. For example, Spring's abstraction of Hibernate Transactions and Exceptions, along with its IoC approach which allows you to easily swap in mapper/DAO objects implementing data-access functionality, makes it easy to isolate all Hibernate-specific code in one area of your application, without sacrificing any of the power of Hibernate. Higher level service code dealing with the DAOs has no need to know anything about their implementation. This approach has the additional benefit of making it easy to intentionally implement data-access with a mix-and-match approach (i.e. some data-access performed using Hibernate, and some using JDBC, others using iBatis) in a non-intrusive fashion, potentially providing great benefits in terms of continuing to use legacy code or leveraging the strength of each technology.

Ease of testing. Spring's inversion of control approach makes it easy to swap the implementations and locations of Hibernate session factories, datasources, transaction managers, and mapper object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.

General resource management. Spring application contexts can handle the location and configuration of Hibernate SessionFactories, JDBC datasources, iBatis SQLMaps configuration objects, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy and safe handling of Hibernate Sessions. Related code using Hibernate generally needs to use the same Hibernate Session object for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a session to the current thread, using either a declarative, AOP method interceptor approach, or by using an explicit, template wrapper class at the Java code level. Thus Spring solves many of the usage issues that repeatedly arise on the Hibernate forums.

Exception wrapping. Spring can wrap exceptions from you O/R mapping tool of choice, converting them from proprietary, checked exceptions, to a set of abstracted runtime exceptions. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. Remember that JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

Integrated transaction management. Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the Java code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc.) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your Hibernate related code being affected. As an added benefit, JDBC-related code can fully integrate transactionally with the code you use to do O/R mapping. This is useful for handling functionality not implemented in, for example, Hibernate or iBatis.

### 11.2. Hibernate

#### 11.2.1. Resource Management

Typical business applications are often cluttered with repetitive resource management code. Many projects try to invent their own solutions for this issue, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates strikingly simple solutions for proper resource handling: Inversion of control via templating, i.e. infrastructure classes with callback interfaces, or applying AOP interceptors. The infrastructure cares for proper resource handling, and for appropriate conversion of specific API exceptions to an unchecked infrastructure exception hierarchy. Spring introduces a DAO exception hierarchy, applicable to any data access strategy. For direct JDBC, the JdbcTemplate class mentioned in a previous section cares for connection handling, and for proper conversion of SQLException to the DataAccessException hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. It supports both JTA and JDBC transactions, via respective Spring transaction managers. Spring also offers Hibernate and JDO support, consisting of a HibernateTemplate / JdoTemplate analogous to JdbcTemplate, a HibernateInterceptor / JdoInterceptor, and a Hibernate / JDO transaction manager. The major goal is to allow for clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business object dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. One simple and consistent approach to wiring up application objects, keeping them as reusable and free from container dependencies as possible. All the individual data access features are usable on their own but integrate nicely with Spring's application context concept, providing XML-based configuration and cross-referencing of plain JavaBean instances that don't need to be Spring-aware. In a typical Spring app, many important objects are JavaBeans: data access templates, data access objects (that use the templates), transaction managers, business objects (that use the data access objects and transaction managers), web view resolvers, web controllers (that use the business objects), etc.

#### 11.2.2. Resource Definitions in an Application Context

To avoid tying application objects to hard-coded resource lookups, Spring allows you to define resources like a JDBC DataSource or a Hibernate SessionFactory as beans in an application context. Application objects that need to access resources just receive references to such pre-defined instances via bean references (the DAO definition in the next section illustrates this). The following excerpt from an XML application context definition shows how to set up a JDBC DataSource and a Hibernate SessionFactory on top of it:

<beans>

<bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName">

<value>java:comp/env/jdbc/myds</value>

</property>

</bean>

<bean id="mySessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

<property name="mappingResources">

<list>

<value>product.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>

</props>

</property>

<property name="dataSource">

<ref bean="myDataSource"/>

</property>

</bean>

...

</beans>

Note that switching from a JNDI-located DataSource to a locally defined one like a Jakarta Commons DBCP BasicDataSource is just a matter of configuration:

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName">

<value>org.hsqldb.jdbcDriver</value>

</property>

<property name="url">

<value>jdbc:hsqldb:hsql://localhost:9001</value>

</property>

<property name="username">

<value>sa</value>

</property>

<property name="password">

<value></value>

</property>

</bean>

You can also use a JNDI-located SessionFactory, but that's typically not necessary outside an EJB context (see the "container resources vs local resources" section for a discussion).

#### 11.2.3. Inversion of Control: Template and Callback

The basic programming model for templating looks as follows, for methods that can be part of any custom data access object or business object. There are no restrictions on the implementation of the surrounding object at all, it just needs to provide a Hibernate SessionFactory. It can get the latter from anywhere, but preferably as bean reference from a Spring application context - via a simple setSessionFactory bean property setter. The following snippets show a DAO definition in a Spring application context, referencing the above defined SessionFactory, and an example for a DAO method implementation.

<beans>

<bean id="myProductDao" class="product.ProductDaoImpl">

<property name="sessionFactory">

<ref bean="mySessionFactory"/>

</property>

</bean>

...

</beans>

public class ProductDaoImpl implements ProductDao {

private SessionFactory sessionFactory;

public void setSessionFactory(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

public List loadProductsByCategory(final String category) {

HibernateTemplate hibernateTemplate =

new HibernateTemplate(this.sessionFactory);

return (List) hibernateTemplate.execute(

new HibernateCallback() {

public Object doInHibernate(Session session) throws HibernateException {

List result = session.find(

"from test.Product product where product.category=?",

category, Hibernate.STRING);

// do some further stuff with the result list

return result;

}

}

);

}

}

A callback implementation can effectively be used for any Hibernate data access. HibernateTemplate will ensure that Sessions 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 like a single find, load, saveOrUpdate, or delete call, HibernateTemplate offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient HibernateDaoSupport base class that provides a setSessionFactory method for receiving a SessionFactory, and getSessionFactory and getHibernateTemplate for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

public List loadProductsByCategory(String category) {

return getHibernateTemplate().find(

"from test.Product product where product.category=?", category,

Hibernate.STRING);

}

}

#### 11.2.4. Applying an AOP Interceptor Instead of a Template

An alternative to using a HibernateTemplate is Spring's AOP HibernateInterceptor, replacing the callback implementation with straight Hibernate code within a delegating try/catch block, and a respective interceptor configuration in the application context. The following snippets show respective DAO, interceptor, and proxy definitions in a Spring application context, and an example for a DAO method implementation.

<beans>

...

<bean id="myHibernateInterceptor"

class="org.springframework.orm.hibernate.HibernateInterceptor">

<property name="sessionFactory">

<ref bean="mySessionFactory"/>

</property>

</bean>

<bean id="myProductDaoTarget" class="product.ProductDaoImpl">

<property name="sessionFactory">

<ref bean="mySessionFactory"/>

</property>

</bean>

<bean id="myProductDao" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces">

<value>product.ProductDao</value>

</property>

<property name="interceptorNames">

<list>

<value>myHibernateInterceptor</value>

<value>myProductDaoTarget</value>

</list>

</property>

</bean>

...

</beans>

public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

public List loadProductsByCategory(final String category) throws MyException {

Session session = SessionFactoryUtils.getSession(getSessionFactory(), false);

try {

List result = session.find(

"from test.Product product where product.category=?",

category, Hibernate.STRING);

if (result == null) {

throw new MyException("invalid search result");

}

return result;

}

catch (HibernateException ex) {

throw SessionFactoryUtils.convertHibernateAccessException(ex);

}

}

}

This method will only work with a HibernateInterceptor for it, caring for opening a thread-bound Session before and closing it after the method call. The "false" flag on getSession makes sure that the Session must already exist; otherwise SessionFactoryUtils would create a new one if none was found. If there is already a SessionHolder bound to the thread, e.g. by a HibernateTransactionManager transaction, SessionFactoryUtils automatically takes part in it in any case. HibernateTemplate uses SessionFactoryUtils internally - it's all the same infrastructure. The major advantage of HibernateInterceptor is that it allows any checked application exception to be thrown within the data access code, while HibernateTemplate is restricted to unchecked exceptions within the callback. Note that one can often defer the respective checks and throwing of application exceptions to after the callback, though. The interceptor's major drawback is that it requires special setup in the context. HibernateTemplate's convenience methods offers simpler means for many scenarios.

#### 11.2.5. Programmatic Transaction Demarcation

On top of such lower-level data access services, transactions can be demarcated in a higher level of the application, spanning any number of operations. There are no restrictions on the implementation of the surrounding business object here too, it just needs a Spring PlatformTransactionManager. Again, the latter can come from anywhere, but preferably as bean reference via a setTransactionManager method - just like the productDAO should be set via a setProductDao method. The following snippets show a transaction manager and a business object definition in a Spring application context, and an example for a business method implementation.

<beans>

...

<bean id="myTransactionManager"

class="org.springframework.orm.hibernate.HibernateTransactionManager">

<property name="sessionFactory">

<ref bean="mySessionFactory"/>

</property>

</bean>

<bean id="myProductService" class="product.ProductServiceImpl">

<property name="transactionManager">

<ref bean="myTransactionManager"/>

</property>

<property name="productDao">

<ref bean="myProductDao"/>

</property>

</bean>

</beans>

public class ProductServiceImpl implements ProductService {

private PlatformTransactionManager transactionManager;

private ProductDao productDao;

public void setTransactionManager(PlatformTransactionManager transactionManager) {

this.transactionManager = transactionManager;

}

public void setProductDao(ProductDao productDao) {

this.productDao = productDao;

}

public void increasePriceOfAllProductsInCategory(final String category) {

TransactionTemplate transactionTemplate = new TransactionTemplate(this.transactionManager);

transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);

transactionTemplate.execute(

new TransactionCallbackWithoutResult() {

public void doInTransactionWithoutResult(TransactionStatus status) {

List productsToChange = productDAO.loadProductsByCategory(category);

...

}

}

);

}

}

#### 11.2.6. Declarative Transaction Demarcation

Alternatively, one can use Spring's AOP TransactionInterceptor, replacing the transaction demarcation code with an interceptor configuration in the application context. This allows you to keep business objects free of repetitive transaction demarcation code in each business method. Furthermore, transaction semantics like propagation behavior and isolation level can be changed in a configuration file and do not affect the business object implementations.

<beans>

...

<bean id="myTransactionManager"

class="org.springframework.orm.hibernate.HibernateTransactionManager">

<property name="sessionFactory">

<ref bean="mySessionFactory"/>

</property>

</bean>

<bean id="myTransactionInterceptor"

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

<property name="transactionManager">

<ref bean="myTransactionManager"/>

</property>

<property name="transactionAttributeSource">

<value>

product.ProductService.increasePrice\*=PROPAGATION\_REQUIRED

product.ProductService.someOtherBusinessMethod=PROPAGATION\_MANDATORY

</value>

</property>

</bean>

<bean id="myProductServiceTarget" class="product.ProductServiceImpl">

<property name="productDao">

<ref bean="myProductDao"/>

</property>

</bean>

<bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces">

<value>product.ProductService</value>

</property>

<property name="interceptorNames">

<list>

<value>myTransactionInterceptor</value>

<value>myProductServiceTarget</value>

</list>

</property>

</bean>

</beans>

public class ProductServiceImpl implements ProductService {

private ProductDao productDao;

public void setProductDao(ProductDao productDao) {

this.productDao = productDao;

}

public void increasePriceOfAllProductsInCategory(final String category) {

List productsToChange = this.productDAO.loadProductsByCategory(category);

...

}

...

}

As with HibernateInterceptor, TransactionInterceptor allows any checked application exception to be thrown with the callback code, while TransactionTemplate is restricted to unchecked exceptions within the callback. TransactionTemplate will trigger a rollback in case of an unchecked application exception, or if the transaction has been marked rollback-only by the application (via TransactionStatus). TransactionInterceptor behaves the same way by default but allows configurable rollback policies per method. A convenient alternative way of setting up declarative transactions is TransactionProxyFactoryBean, particularly if there are no other AOP interceptors involved. TransactionProxyFactoryBean combines the proxy definition itself with transaction configuration for a particular target bean. This reduces the configuration effort to one target bean plus one proxy bean. Furthermore, you do not need to specify which interfaces or classes the transactional methods are defined in.

<beans>

...

<bean id="myTransactionManager"

class="org.springframework.orm.hibernate.HibernateTransactionManager">

<property name="sessionFactory">

<ref bean="mySessionFactory"/>

</property>

</bean>

<bean id="myProductServiceTarget" class="product.ProductServiceImpl">

<property name="productDao">

<ref bean="myProductDao"/>

</property>

</bean>

<bean id="myProductService"

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

<property name="transactionManager">

<ref bean="myTransactionManager"/>

</property>

<property name="target">

<ref bean="myProductServiceTarget"/>

</property>

<property name="transactionAttributes">

<props>

<prop key="increasePrice\*">PROPAGATION\_REQUIRED</prop>

<prop key="someOtherBusinessMethod">PROPAGATION\_MANDATORY</prop>

</props>

</property>

</bean>

</beans>

#### 11.2.7. Transaction Management Strategies

Both TransactionTemplate and TransactionInterceptor delegate the actual transaction handling to a PlatformTransactionManager instance, which can be a HibernateTransactionManager (for a single Hibernate SessionFactory, using a ThreadLocal Session under the hood) or a JtaTransactionManager (delegating to the JTA subsystem of the container) for Hibernate applications. You could even use a custom PlatformTransactionManager implementation. So switching from native Hibernate transaction management to JTA, i.e. when facing distributed transaction requirements for certain deployments of your application, is just a matter of configuration. Simply replace the Hibernate transaction manager with Spring's JTA transaction implementation. Both transaction demarcation and data access code will work without changes, as they just use the generic transaction management APIs. For distributed transactions across multiple Hibernate session factories, simply combine JtaTransactionManager as a transaction strategy with multiple LocalSessionFactoryBean definitions. Each of your DAOs then gets one specific SessionFactory reference passed into its respective bean property. If all underlying JDBC data sources are transactional container ones, a business object can demarcate transactions across any number of DAOs and any number of session factories without special regard, as long as it is using JtaTransactionManager as the strategy.

<beans>

<bean id="myDataSource1" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName">

<value>java:comp/env/jdbc/myds1</value>

</property>

</bean>

<bean id="myDataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName">

<value>java:comp/env/jdbc/myds2</value>

</property>

</bean>

<bean id="mySessionFactory1" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

<property name="mappingResources">

<list>

<value>product.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>

</props>

</property>

<property name="dataSource">

<ref bean="myDataSource1"/>

</property>

</bean>

<bean id="mySessionFactory2" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

<property name="mappingResources">

<list>

<value>inventory.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect</prop>

</props>

</property>

<property name="dataSource">

<ref bean="myDataSource2"/>

</property>

</bean>

<bean id="myTransactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager"/>

<bean id="myProductDao" class="product.ProductDaoImpl">

<property name="sessionFactory">

<ref bean="mySessionFactory1"/>

</property>

</bean>

<bean id="myInventoryDao" class="product.InventoryDaoImpl">

<property name="sessionFactory">

<ref bean="mySessionFactory2"/>

</property>

</bean>

<bean id="myProductServiceTarget" class="product.ProductServiceImpl">

<property name="productDao">

<ref bean="myProductDao"/>

</property>

<property name="inventoryDao">

<ref bean="myInventoryDao"/>

</property>

</bean>

<bean id="myProductService"

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

<property name="transactionManager">

<ref bean="myTransactionManager"/>

</property>

<property name="target">

<ref bean="myProductServiceTarget"/>

</property>

<property name="transactionAttributes">

<props>

<prop key="increasePrice\*">PROPAGATION\_REQUIRED</prop>

<prop key="someOtherBusinessMethod">PROPAGATION\_MANDATORY</prop>

</props>

</property>

</bean>

</beans>

Both HibernateTransactionManager and JtaTransactionManager allow for proper JVM-level cache handling with Hibernate - without container-specific transaction manager lookup or JCA connector (as long as not using EJB to initiate transactions). Additionally, HibernateTransactionManager can export the JDBC Connection used by Hibernate to plain JDBC access code. This allows for high level transaction demarcation with mixed Hibernate/JDBC data access completely without JTA, as long as just accessing one database!

Note, for an alternative approach to using TransactionProxyFactoryBean to declaratively demarcate transactions, please see Section 7.4.1, “BeanNameAutoProxyCreator, another declarative approach”.

#### 11.2.8. Container Resources versus Local Resources

Spring's resource management allows for simple switching between a JNDI SessionFactory and a local one, same for a JNDI DataSource, without having to change a single line of application code. Whether to keep the resource definitions in the container or locally within the application, is mainly a matter of the transaction strategy being used. Compared to a Spring-defined local SessionFactory, a manually registered JNDI SessionFactory does not provide any benefits. If registered via Hibernate's JCA connector, there is the added value of transparently taking part in JTA transactions, especially within EJBs. An important benefit of Spring's transaction support is that it isn't bound to a container at all. Configured to any other strategy than JTA, it will work in a standalone or test environment too. Especially for the typical case of single-database transactions, this is a very lightweight and powerful alternative to JTA. When using local EJB Stateless Session Beans to drive transactions, you depend both on an EJB container and JTA - even if you just access a single database anyway, and just use SLSBs for declarative transactions via CMT. The alternative of using JTA programmatically requires a J2EE environment too. JTA does not just involve container dependencies in terms of JTA itself and of JNDI DataSources. For non-Spring JTA-driven Hibernate transactions, you have to use the Hibernate JCA connector, or extra Hibernate transaction code with JTATransaction being configured, for proper JVM-level caching. Spring-driven transactions can work with a locally defined Hibernate SessionFactory nicely, just like with a local JDBC DataSource - if accessing a single database, of course. Therefore you just have to fall back to Spring's JTA transaction strategy when actually facing distributed transaction requirements. Note that a JCA connector needs container-specific deployment steps, and obviously JCA support in the first place. This is far more hassle than deploying a simple web app with local resource definitions and Spring-driven transactions. And you often need the Enterprise Edition of your container, as e.g. WebLogic Express does not provide JCA. A Spring app with local resources and transactions spanning one single database will work in any J2EE web container (without JTA, JCA, or EJB) - like Tomcat, Resin, or even plain Jetty. Additionally, such a middle tier can be reused in desktop applications or test suites easily. All things considered: If you do not use EJB, stick with local SessionFactory setup and Spring's HibernateTransactionManager or JtaTransactionManager. You will get all benefits including proper transactional JVM-level caching and distributed transactions, without any container deployment hassle. JNDI registration of a Hibernate SessionFactory via the JCA connector only adds value for use within EJBs.

#### 11.2.9. Samples

The Petclinic sample in the Spring distribution offers alternative DAO implementations and application context configurations for Hibernate, JDBC, and Apache OJB. Petclinic can therefore serve as working sample app that illustrates the use of Hibernate in a Spring web app. It also leverages declarative transaction demarcation with different transaction strategies.

### 11.3. JDO

ToDo

### 11.4. iBATIS

Through the org.springframework.orm.ibatis package, Spring supports iBATIS SqlMaps 1.3.x and 2.0. The iBATIS support much resembles Hibernate support in that it supports the same template style programming and just as with Hibernate, iBatis support works with Spring's exception hierarchy and let's you enjoy the all IoC features Spring has.

#### 11.4.1. Overview and differences between 1.3.x and 2.0

Spring supports both iBATIS SqlMaps 1.3 and 2.0. First let's have a look at the differences between the two.

Table 11.1. iBATIS SqlMaps supporting classes for 1.3 and 2.0

Feature 1.3.x 2.0

Creation of SqlMap SqlMapFactoryBean SqlMapClientFactoryBean

Template-style helper class SqlMapTemplate SqlMapClientTemplate

Callback to use MappedStatement SqlMapCallback SqlMapClientCallback

Super class for DAOs SqlMapDaoSupport SqlMapClientDaoSupport

#### 11.4.2. Setting up the SqlMap

Using iBATIS SqlMaps involves creating SqlMap configuration files containing statements and result maps. Spring takes care of loading those using the SqlMapFactoryBean or SqlMapClientFactoryBean where the latter is to be used in combination with SqlMaps 2.0.

public class Account {

private String name;

private String email;

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;

}

}

Suppose we would want to map this class. We'd have to create the following SqlMap. Using the query, we can later on retrieve users through their email addresses. Account.xml:

<sql-map name="Account">

<result-map name="result" class="examples.Account">

<property name="name" column="NAME" columnIndex="1"/>

<property name="email" column="EMAIL" columnIndex="2"/>

</result-map>

<mapped-statement name="getAccountByEmail" result-map="result">

select

ACCOUNT.NAME,

ACCOUNT.EMAIL

from ACCOUNT

where ACCOUNT.EMAIL = #value#

</mapped-statement>

<mapped-statement name="insertAccount">

insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)

</mapped-statement>

</sql-map>

After having defined the Sql Map, we have to create a configuration file for iBATIS (sqlmap-config.xml):

<sql-map-config>

<sql-map resource="example/Account.xml"/>

</sql-map-config>

iBATIS loads resources from the classpath so be sure to add the Account.xml file to the classpath somewhere.

Using Spring, we can now very easily set up the SqlMap, using the SqlMapFactoryBean:

<bean id="sqlMap" class="org.springframework.orm.ibatis.SqlMapFactoryBean">

<property name="configLocation"><value>WEB-INF/sqlmap-config.xml</value></property>

</bean>

#### 11.4.3. Using SqlMapDaoSupport

The SqlMapDaoSupport class offers a supporting class similar to the HibernateDaoSupport and the JdbcDaoSupport types. Let's implement a DAO:

public class SqlMapAccountDao extends SqlMapDaoSupport implements AccountDao {

public Account getAccount(String email) throws DataAccessException {

return (Account) getSqlMapTemplate().executeQueryForObject("getAccountByEmail", email);

}

public void insertAccount(Account account) throws DataAccessException {

getSqlMapTemplate().executeUpdate("insertAccount", account);

}

}

As you can see, we're using the SqlMapTemplate to execute the query. Spring has initialized the SqlMap for us using the SqlMapFactoryBean and when setting up the SqlMapAccountDao as follows, you're all set to go:

<!-- for more information about using datasource, have a look at the JDBC chapter -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName"><value>${jdbc.driverClassName}</value></property>

<property name="url"><value>${jdbc.url}</value></property>

<property name="username"><value>${jdbc.username}</value></property>

<property name="password"><value>${jdbc.password}</value></property>

</bean>

<bean id="accountDao" class="example.SqlMapAccountDao">

<property name="dataSource"><ref local="dataSource"/></property>

<property name="sqlMap"><ref local="sqlMap"/></property>

</bean>

#### 11.4.4. Transaction management

It's pretty easy to add declarative transaction management to applications using iBATIS. Basically the only thing you need to do is adding a transaction manager to you application context and declaratively set your transaction boundaries using for example the TransactionProxyFactoryBean. More on this can be found in Chapter 7, Transaction management

TODO elaborate!

## 12. Web framework

### 12.1. Introduction to the web framework

Spring's web framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well support for upload files. The default handler is a very simple Controller interface, just offering a ModelAndView handleRequest(request,response) method. This can already be used for application controllers, but you will prefer the included implementation hierarchy, consisting of for example AbstractController, AbstractCommandController and SimpleFormController. Application controllers will typically be subclasses of those. Note that you can choose an appropriate base class: If you don't have a form, you don't need a FormController. This is a major difference to Struts.

You can take any object as command or form object: There's no need to implement an interface or derive from a base class. Spring's data binding is highly flexible, e.g. it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. So you don't need to duplicate your business objects' properties as Strings in your form objects, just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it's often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes like Action and ActionForm - for every type of action.

Compared to WebWork, Spring has more differentiated object roles: It supports the notion of a Controller, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data. Instead, a WebWork Action combines all those roles into one single object. WebWork does allow you to use existing business objects as part of your form, but just by making them bean properties of the respective Action class. Finally, the same Action instance that handles the request gets used for evaluation and form population in the view. Thus, reference data needs to be modeled as bean properties of the Action too. These are arguably too many roles in one object.

Regarding views: Spring's view resolution is extremely flexible. A Controller implementation can even write a view directly to the response, returning null as ModelAndView. In the normal case, a ModelAndView instance consists of a view name and a model Map, containing bean names and corresponding objects (like a command or form, reference data, etc). View name resolution is highly configurable, either via bean names, via a properties file, or via your own ViewResolver implementation. The abstract model Map allows for complete abstraction of the view technology, without any hassle: Be it JSP, Velocity, or anything else - every renderer can be integrated directly. The model Map simply gets transformed into an appropriate format, like JSP request attributes or a Velocity template model.

#### 12.1.1. Pluggability of MVC implementation

Many teams will try to leverage their investments in terms of know-how and tools, both for existing projects and for new ones. Concretely, there are not only a large number of books and tools for Struts but also a lot of developers that have experience with it. Thus, if you can live with Struts's architectural flaws, it can still be a viable choice for the web layer. The same applies to WebWork and other web frameworks.

If you don't want to use Spring's web MVC but intend to leverage other solutions that Spring offers, you can integrate the web framework of your choice with Spring easily. Simply start up a Spring root application context via its ContextLoaderListener, and access it via its ServletContext attribute (or Spring's respective helper method) from within a Struts or WebWork action. Note that there aren't any "plugins" involved, therefore no dedicated integration: From the view of the web layer, you'll simply use Spring as a library, with the root application context instance as entry point.

All your registered beans and all of Spring's services can be at your fingertips even without Spring's web MVC. Spring doesn't compete with Struts or WebWork in this usage, it just addresses the many areas that the pure web frameworks don't, from bean configuration to data access and transaction handling. So you are able to enrich your application with a Spring middle tier and/or data access tier, even if you just want to use e.g. the transaction abstraction with JDBC or Hibernate.

#### 12.1.2. Features of Spring MVC

If just focusing on the web support, some of the Spring's unique features are:

Clear separation of roles: controller vs validator vs command object vs form object vs model object, DispatcherServlet vs handler mapping vs view resolver, etc.

Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy in-between referencing via an application context, e.g. from web controllers to business objects and validators.

Adaptability, non-intrusiveness: Use whatever Controller subclass you need (plain, command, form, wizard, multi action, or a custom one) for a given scenario instead of deriving from Action/ActionForm for everything.

Reusable business code, no need for duplication: You can use existing business objects as command or form objects instead of mirroring them in special ActionForm subclasses.

Customizable binding and validation: type mismatches as application-level validation errors that keep the offending value, localized date and number binding, etc instead of String-only form objects with manual parsing and conversion to business objects.

Customizable handler mapping, customizable view resolution: flexible model transfer via name/value Map, handler mapping and view resolution strategies from simple to sophisticated instead of one single way.

Customizable locale and theme resolution, support for JSPs with and without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, etc.

Simple but powerful tag library that avoids HTML generation at any cost, allowing for maximum flexibility in terms of markup code.

### 12.2. The DispatcherServlet

Spring's web framework is - like many other web frameworks - a request driven web framework, designed around a servlet that dispatches requests to controllers and offers other functionality facilitating the development of web applications. Spring's DispatcherServlet however, does more than just that. It is completely integrated with the Spring ApplicationContext and allows you to use every other feature Spring has.

Servlets are declared in the web.xml of your web application, so is the DispatcherServlet. Requests that you want the DispatcherServlet to handle, will have to be mapped, using a URL mapping in the same web.xml file.

<web-app>

...

<servlet>

<servlet-name>example</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>example</servlet-name>

<url-pattern>\*.form</url-pattern>

</servlet-mapping>

</web-app>

In the example above, all requests ending with .form will be handled by the DispatcherServlet. Then, the DispatcherServlet needs to be configured. As illustrated in Section 3.10, “Introduction to the ApplicationContext”, ApplicationContexts in Spring can be scoped. In the web framework, each DispatcherServlet has its own WebApplicationContext, which contains the DispatcherServlet configuration beans. The default BeanFactory used by the DispatcherServlet is the XmlBeanFactory and the DispatcherServlet will on initialization look for a file named [servlet-name]-servlet.xml in the WEB-INF directory of your web application. The default values used by the DispatcherServlet can be modified by using the servlet initialization parameters (see below for more information).

The WebApplicationContext is just an ordinary ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes (see Section 12.7, “Using themes”), and that is knows to which servlet it is associated (by having a link to the ServletContext). The WebApplicationContext is bound in the ServletContext, and using RequestContextUtils you can always lookup the WebApplicationContext in case you need it.

The Spring DispatcherServlet has a couple of special beans it uses, in order to be able to process requests and render the appropriate views. Those beans are included in the Spring framework and (optionally) have to be configured in the WebApplicationContext, just as any other bean would have to be configured. Each of those beans, is described in more detail below. Right now, we'll just mention them, just to let you know they exist and to enable us to go on talking about the DispatcherServlet. For most of the beans, defaults are provided so you don't have to worry about those.

Table 12.1. Special beans in the WebApplicationContext

Expression Explanation

handler mapping(s) (Section 12.4, “Handler mappings”) a list of pre- and postprocessors and controllers that will be executed if they match certain criteria (for instance a matching URL specified with the controller)

controller(s) (Section 12.3, “Controllers”) the beans providing the actual functionality (or at least, access to the functionality) as part of the MVC triad

view resolver (Section 12.5, “Views and resolving them”) capable of resolving view names and needed by the DispatcherServlet to resolves those views with

locale resolver (Section 12.6, “Using locales”) capable of resolves the locale a client is using, in order to be able to offer internationalized views

theme resolver (Section 12.7, “Using themes”) capable of resolving themes your web application can use e.g. to offer personalized layouts

multipart resolver (Section 12.8, “Spring's multipart (fileupload) support”) offers functionality to process file uploads from HTML forms

handlerexception resolver (Section 12.9, “Handling exceptions”) offers functionality to map exceptions to views or implement other more complex exception handling code

When a DispatcherServlet is setup for use and a request comes in for that specific DispatcherServlet it starts processing it. The list below describes the complete process a request goes through if a DispatcherServlet is supposed to handle it:

The WebApplicationContext is searched for and bound in the request as an attribute in order for controller and other elements in the chain of process to use it. It is bound by default under the key DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE

The locale resolver is bound to the request to let elements in the chain resolve the locale to use when processing the request (rendering the view, preparing data, etcetera). If you don't use the resolver, it won't affect anything, so if you don't need locale resolving, just don't bother

The theme resolver is bound to the request to let e.g. views determine which theme to use (if you don't needs themes, don't bother, the resolver is just bound and does not affect anything if you don't use it)

If a multipart resolver is specified, the request is inspected for multiparts and if so, it is wrapped in a MultipartHttpServletRequest for further processing by other elements in the chain (more information about multipart handling is provided below)

An appropriate handler is searched for. If a handler is found, it execution chain associated to the handler (preprocessors, postprocessors, controllers) will be executed in order to prepare a model

If a model is returned, the view is rendered, using the view resolver that has been configured with the WebApplicationContext. If no model was returned (which could be the result of a pre- or postprocessor intercepting the request because of for instance security reasons), no view is rendered as well, since the request could already have been fulfilled

Exceptions that might be thrown during processing of the request get picked up by any of the handlerexception resolvers that are declared in the WebApplicationContext. Using those exception resolvers you can define custom behavior in case such exceptions get thrown.

The Spring DispatcherServlet also has support for returning the last-modification-date, as specified by the Servlet API. The process of determining the last modification date for a specific request, is simple. The DispatcherServlet will first of all lookup an appropriate handler mapping and test if the handler that matched implements the interface LastModified and if so, the value the of long getLastModified(request) is returned to the client.

You can customize Spring's DispatcherServlet by adding context parameters in the web.xml file or servlet init parameters. The possibilities are listed below.

Table 12.2. DispatcherServlet initialization parameters

Parameter Explanation

contextClass Class that implements WebApplicationContext, which will be used to instantiate the context used by this servlet. If this parameter isn't specified, the XmlWebApplicationContext will be used

contextConfigLocation String which is passed to the context instance (specified by contextClass) to indicate where context(s) can be found. The String is potentially split up into multiple strings (using a comma as a delimiter) to support multiple contexts (in case of multiple context locations, of beans that are defined twice, the latest takes precedence)

namespace the namespace of the WebApplicationContext. Defaults to [server-name]-servlet

### 12.3. Controllers

The notion of controller is part of the MVC design pattern. Controllers define application behavior, or at least provide users with access to the application behavior. Controllers interpret user input and transform the user input into a sensible model which will be represented to the user by the view. Spring has implemented the notion of a controller in a very abstract way enabling a wide variety of different kinds of controllers to be created. Spring contains formcontroller, commandcontroller, controllers that execute wizard-style logic and more.

Spring's basis for the controller architecture is the org.springframework.mvc.Controller interface, which is listed below.

public interface Controller {

/\*\*

\* Process the request and return a ModelAndView object which the DispatcherServlet

\* will render.

\*/

ModelAndView handleRequest(

HttpServletRequest request,

HttpServletResponse response)

throws Exception;

}

As you can see, the Controller interface just states one single method that should be capable of handling a request and return an appropriate model and view. Those three concepts are the basis for the Spring MVC implementation; ModelAndView and Controller. While the Controller interface is quite abstract, Spring offers a lot of controllers that already contain a lot of functionality you might need. The controller interface just define the most commons functionality offered by every controller: the functionality of handling a request and returning a model and a view.

#### 12.3.1. AbstractController and WebContentGenerator

Of course, just a controller interface isn't enough. To provide a basic infrastructure, all of Spring's Controllers inherit from AbstractController, a class offering caching support and for instance the setting of the mimetype.

Table 12.3. Features offered by the AbstractController

Feature Explanation

supportedMethods indicates what methods this controller should accept. Usually this is set to both GET and POST, but you can modify this to reflect the method you want to support. If a request is received with a method that is not supported by the controller, the client will be informed of this (using a ServletException))

requiresSession indicates whether or not this controller requires a session to do its work. This feature is offered to all controllers. If a session is not present when such a controller receives a request, the user is informed using a ServletException

synchronizeSession use this if you want handling by this controller to be synchronized on the user's session. To be more specific, extending controller will override the handleRequestInternal method, which will be synchronized if you specify this variable

cacheSeconds when you want a controller to generate caching directive in the HTTP response, specify a positive integer here. By default it is set to -1 so no caching directives will be included

useExpiresHeader tweaking of your controllers specifying the HTTP 1.0 compatible "Expires" header. By default it's set to true, so you won't have to touch it

useCacheHeader tweaking of your controllers specifying the HTTP 1.1 compatible "Cache-Control" header. By default this is set to true so you won't really have to touch it

the last two properties are actually part of the WebContentGenerator which is the superclass of AbstractController but to keeps things clear...

When using the AbstractController as a baseclass for your controllers (which is not recommended since there are a lot of other controller that might already do the job for your) you only have to override the handleRequestInternal(HttpServletRequest, HttpServletResponse)-method and implement your logic code and return a ModelAndView object there. A short example consisting of a class and a declaration in the web application context.

package samples;

public class SampleController extends AbstractController {

public ModelAndView handleRequestInternal(

HttpServletRequest request,

HttpServletResponse response)

throws Exception {

ModelAndView mav = new ModelAndView("foo", new HashMap());

}

}

<bean id="sampleController" class="samples.SampleController">

<property name="cacheSeconds"><value>120</value</property>

</bean>

The class above and the declaration in the web application context is all you need to do besides setting up a handler mapping (see Section 12.4, “Handler mappings”) to get this very simple controller working. This controller will generates caching directives telling the client to cache things for 2 minutes before rechecking. This controller furthermore returns an hard-coded view (hmm, not so nice), named index (see Section 12.5, “Views and resolving them” for more information about views).

#### 12.3.2. Other simple controllers

Besides the AbstractController - which you could of course extend, although a more concrete controller might offer you more functionality - there are a couple of other simple controllers that might ease the pain of developing simple MVC applications. The ParameterizableViewController basically is the same as the one in the example above, except for the fact that you can specify its view name that it'll be returning in the web application context (ahhh, no need to hard-code the viewname).

The FileNameViewController inspects the URL and retrieves the filename of the file request (the filename of http://www.springframework.org/index.html is index) and uses that as a viewname. Nothing more to it.

#### 12.3.3. The MultiActionController

Spring offers a multi-action controller with which you aggregate multiple actions into one controller, grouping functionality together. The multi-action controller lives in a separate package - org.springframework.web.mvc.multiaction - and is capable of mapping requests to method names and then invoking the right method name. Using the multi-action controller is especially handy when you're having a lot of commons functionality in one controller, but want to have multiple entry points to the controller to tweak behavior for instance.

Table 12.4. Features offered by the MultiActionController

Feature Explanation

delegate there's two usage-scenarios for the MultiActionController. Either you subclass the MultiActionController and specify the methods that will be resolved by the MethodNameResolver on the subclass (in case you don't need this configuration parameter), or you define a delegate object, on which methods resolved by the Resolver will be invoked. If you choose to enter this scenario, you will have to define the delegate using this configuration parameter as a collaborator

methodNameResolver somehow, the MultiActionController will need to resolve the method it has to invoke, based on the request that came in. You can define a resolver that is capable of doing that using this configuration parameter

Methods defined for a multi-action controller will need to conform to the following signature:

// actionName can be replaced by any methodname

ModelAndView actionName(HttpServletRequest, HttpServletResponse);

Method overloading is not allowed since it'll confuse the MultiActionController. Furthermore, you can define exception handlers capable of handling exception that will be thrown form a method you specify. Exception handler methods need to return a ModelAndView object, just as any other action method and will need to conform to the following signature:

// anyMeaningfulName can be replaced by any methodname

ModelAndView anyMeaningfulName(HttpServletRequest, HttpServletResponse, ExceptionClass);

The ExceptionClass can be any exception, as long as it's a subclass of java.lang.Exception or java.lang.RuntimeException.

The MethodNameResolver is supposed to resolve method names based on the request coming in. There are three resolver to your disposal, but of course you can implement more of them yourself if you want.

ParameterMethodNameResolver - capable of resolving a request parameter and using that as the method name (http://www.sf.net/index.view?testParam=testIt will result in a method testIt(HttpServletRequest, HttpServletResponse) being called). Use the paramName configuration parameter to tweak the parameter that's inspected)

InternalPathMethodNameResolver - retrieves the filename from the path and uses that as the method name (http://www.sf.net/testing.view will result in a method testing(HttpServletRequest, HttpServletResponse) being called)

PropertiesMethodNameResolver - uses a user-defined properties object with request URLs mapped to methodnames. When the properties contain /index/welcome.html=doIt and a request to /index/welcome.html comes in, the doIt(HttpServletRequest, HttpServletResponse) method is called. This method name resolver works with the PathMatcher (see ???) so if the properties contained /\*\*/welcom?.html it would also have worked!

A couple of examples. First of all one showing the ParameterMethodNameResolver and the delegate property, which will accept requests to urls with the parameter method included and set to retrieveIndex:

<bean id="paramResolver" class="org....mvc.multiaction.ParameterMethodNameResolver">

<property name="paramName"><value>method</value></property>

</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">

<property name="methodNameResolver"><ref bean="paramResolver"/></property>

<property name="delegate"><ref bean="sampleDelegate"/>

</bean>

<bean id="sampleDelegate" class="samples.SampleDelegate"/>

## together with

public class SampleDelegate {

public ModelAndView retrieveIndex(

HttpServletRequest req,

HttpServletResponse resp) {

return new ModelAndView("index", "date", new Long(System.currentTimeMillis()));

}

}

When using the delegates shown above, we could also use the PropertiesMethodNameResolver to match a couple of URLs to the method we defined:

<bean id="propsResolver" class="org....mvc.multiaction.PropertiesMethodNameResolver">

<property name="mappings">

<props>

<prop key="/index/welcome.html">retrieveIndex</prop>

<prop key="/\*\*/notwelcome.html">retrieveIndex</prop>

<prop key="/\*/user?.html">retrieveIndex</prop>

</props>

</property>

</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">

<property name="methodNameResolver"><ref bean="propsResolver"/></property>

<property name="delegate"><ref bean="sampleDelegate"/>

</bean>

#### 12.3.4. CommandControllers

Spring's CommandControllers are a fundamental part of the Spring MVC package. Command controllers provide a way to interact with data objects and dynamically bind parameters from the HttpServletRequest to the data object you're specifying. This compares to Struts's actionforms, where in Spring, you don't have to implement any interface of superclasses to do data binding. First, let's examine what command controllers available, just to get clear picture of what you can do with them:

AbstractCommandController - a command controller you can use to create your own command controller, capable of binding request parameters to a data object you're specifying. This class does not offer form functionality, it does however, offer validation features and lets you specify in the controller itself what to do with the data object that has been filled with the parameters from the request.

AbstractFormController - an abstract controller offering form submission support. Using this controller you can model forms and populate them using a data object you're retrieving in the controller. After a user has filled the form, the AbstractFormController binds the fields, validates and hands the object back to you - the controller - to take appropriate action. Supported features are invalid form submission (resubmission), validation, and the right workflow a form always has. What views you tie to your AbstractFormController you decide yourself. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the applicationcontext

SimpleFormController - an even more concrete FormController that helps you creating a form with corresponding data object even more. The SimpleFormController let's you specify a command object, a viewname for the form, a viewname for page you want to show the user when formsubmission has succeeded, and more

AbstractWizardFormController - as the class name suggests, this is an abstract class--your WizardController should extend it. This means you have to implement both the validatePage(), processFinish as well as processCancel methods.

Probably you also want to write a contractor, which should at the very least call setPages() and setCommandName(). The former takes as its argument an array of type String. This array is the list of views which comprise your wizard. The latter takes as its argument a String, which will be used to refer to your Command object from within your views.

As with any instance of AbstractFormController, you are required to use a Command object - a JavaBean which will be populated with the data from your forms. You can do this in one of two ways: either call setCommandClass() from the constructor with the class of your command object, or implement the formBackingObject() method.

AbstractWizardFormController has a number of concrete methods that you may wish to override. Of these, the ones you are likely to find most useful are: referenceData which you can use to pass model data to your view in the form of a Map; getTargetPage if your wizard needs to change page order or omit pages dynamically; and onBindAndValidate if you want to override the built-in binding and validation workflow.

Finally, it is worth pointing out the setAllowDirtyBack and setAllowDirtyForward, which you can call from getTargetPage to allow users to move backwards and forwards in the wizard even if validation fails for the current page.

For a full list of methods, see the JavaDoc for AbstractWizardFormController. There is an implemented example of this wizard in the jPetStore included in the Spring distribution: org.springframework.samples.jpetstore.web.spring.OrderFormController.java

### 12.4. Handler mappings

Using a handler mapping you can map incoming web requests to appropriate handlers. There are some handler mapping you can use, for example the SimpleUrlHandlerMapping or the BeanNameUrlHandlerMapping, but let's first examine the general concept of a HandlerMapping.

The functionality a basic HandlerMapping provides is the delivering of a HandlerExecutionChain, first of all containing one handler that matched the incoming request. The second (but optional) element a handler execution chain will contain is a list of handler interceptor that should be applied to the request. When a request comes in, the DispatcherServlet will hand it over to the handler mapping to let it inspect the request and come up with an appropriate HandlerExecutionChain. When done, the DispatcherServlet will execute the handler and interceptors in the chain (if any).

The concept of configurable handler mappings that can optionally contain interceptors (executed before or after the actual handler was executed, or both) is extremely powerful. A lot of supporting functionality can be built-in in custom HandlerMappings. Think of a custom handler mapping that chooses a handler not only based on the URL of the request coming in, but also on a specific state of the session associated with the request.

This section describes two of Spring's most often used handler mapping. They both extend the AbstractHandlerMapping and share the following properties

interceptors: the list of interceptors to use. HandlerInterceptors are discussed further ahead

defaultHandler: the default handler to use, when this handler mapping does not result in a matching handler

order: based on the value of the order property (see the org.springframework.core.Ordered interface), Spring will sort all handler mapping available in the context and apply them in them one after the other.

alwaysUseFullPath: based on this setting, Spring will either use the full path within the current servlet context (if set to true) or the path within the current servlet mapping (false, the default value). If for example a servlet is mapped using /testing/\* and you've set the alwaysUseFullPath property to true, /testing/viewPage.html will match, whereas /viewPage.html will only match if you leave the default value (false) in place (Note: this property is only available for the org.springframework.web.servlet.handler.AbstractUrlHandlerMapping and its subclasses)

urlPathHelper: using this property, you can tweak the UrlPathHelper used when inspecting URLs. Normally, you shouldn't have to change the default value. (Note: this property is only available for the org.springframework.web.servlet.handler.AbstractUrlHandlerMapping and its subclasses)

urlDecode: the default value for this property is false. The HttpServletRequest returns request URLs and URIs that are not decoded. If you do want them to be decoded before a HandlerMapping will use them to find an appropriate handler, you have to set this to true (this requires JDK 1.4 however). Uses either the encoding specified by the request or the default ISO-8859-1 encoding scheme. (Note: this property is only available for the org.springframework.web.servlet.handler.AbstractUrlHandlerMapping and its subclasses)

lazyInitHandlers: allows for lazy initialization of singleton handlers (prototype handlers are always lazily initialized). Default value is false (Note: this property is only available for the org.springframework.web.servlet.handler.AbstractUrlHandlerMapping and its subclasses)

#### 12.4.1. BeanNameUrlHandlerMapping

A very simple, but very powerful handler mapping is the BeanNameUrlHandlerMapping, which maps incoming HTTP requests to names of beans, defined in the web application context. Let's say we want to enable a user to insert an account and we've already provided an appropriate FormController (see Section 12.3.4, “CommandControllers” for more information on Command- and FormControllers) and a JSP view (or Velocity template) that renders the form. When using the BeanNameUrlHandlerMapping, we could map the HTTP request with URL http://samples.com/editaccount.form to the appropriate FormController as follows:

<beans>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean name="/editaccount.form"

class="org.springframework.web.servlet.mvc.SimpleFormController">

<property name="formView"><value>account</value></property>

<property name="successView"><value>account-created</value></property>

<property name="commandName"><value>Account</value></property>

<property name="commandClass"><value>samples.Account</value></property>

</bean>

<beans>

All incoming requests for the URL /editaccount.form will now be handled by the FormController in the source listing above. Of course we have to define a servlet-mapping in web.xml as well, to let through all the requests ending with .form.

<web-app>

...

<servlet>

<servlet-name>sample</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Maps the sample dispatcher to /\*.form -->

<servlet-mapping>

<servlet-name>sample</servlet-name>

<url-pattern>\*.form</url-pattern>

</servlet-mapping>

...

</web-app>

NOTE: if you want to use the BeanNameUrlHandlerMapping, you don't necessarily have to define it in the web application context (as indicated above). By default, if no handler mapping can be found in the context, the DispatcherServlet creates a BeanNameUrlHandlerMapping for you!

#### 12.4.2. SimpleUrlHandlerMapping

A further - and much more powerful handler mapping - is the SimpleUrlHandlerMapping. This mapping is configurable in the application context and has Ant-style path matching capabilities (see ???). A couple of example will probably makes thing clear enough:

<web-app>

...

<servlet>

<servlet-name>sample</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Maps the sample dispatcher to /\*.form -->

<servlet-mapping>

<servlet-name>sample</servlet-name>

<url-pattern>\*.form</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>sample</servlet-name>

<url-pattern>\*.html</url-pattern>

</servlet-mapping>

...

</web-app>

Allows all requests ending with .html and .form to be handled by the sample dispatcher servlet.

<beans>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<prop key="/\*/account.form">editAccountFormController</prop>

<prop key="/\*/editaccount.form">editAccountFormController</prop>

<prop key="/ex/view\*.html">someViewController</prop>

<prop key="/\*\*/help.html">helpController</prop>

</props>

</property>

</bean>

<bean id="someViewController"

class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

<bean id="editAccountFormController"

class="org.springframework.web.servlet.mvc.SimpleFormController">

<property name="formView"><value>account</value></property>

<property name="successView"><value>account-created</value></property>

<property name="commandName"><value>Account</value></property>

<property name="commandClass"><value>samples.Account</value></property>

</bean>

<beans>

This handler mapping first of all reroutes all requests in all directories for a file named help.html to the someViewController, which is a UrlFilenameViewController (more about that can be found in Section 12.3, “Controllers”). Also, all requests for a resource beginning with view, ending with .html, in the directory ex, will be rerouted to that specific controller. Furthermore, two mappings have been defined that will match with the editAccountFormController.

#### 12.4.3. Adding HandlerInterceptors

The handler mapping also has a notion of handler interceptors, that can be extremely useful when you want to apply specific functionality to all requests, for example the checking for a principal or something alike.

Interceptors located in the handler mapping must implement HandlerInterceptor from the org.springframework.web.servlet-package. This interface defines three methods, one that will be called before the actual handler will be executed, one that will be called after the handler is executed, and one that is called after the complete request has finished. Those three methods should provide you with enough flexibility to do all kinds of pre- and post-processing.

The preHandle method has a boolean return value. Using this value, you can tweak the behavior of the execution chain. When returning true, the handler execution chain will continue, when returning false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and for instance rendered an appropriate view) and does not continue with executing the other interceptors and the actual handler in the execution chain.

The following example provides an interceptor that intercepts all requests and reroutes the user to a specific page if the time is not between 9 a.m. and 6 p.m.

<beans>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="interceptors">

<list>

<ref bean="officeHoursInterceptor"/>

</list>

</property>

<property name="mappings">

<props>

<prop key="/\*.form">editAccountFormController</prop>

<prop key="/\*.view">editAccountFormController</prop>

</props>

</property>

</bean>

<bean id="officeHoursInterceptor"

class="samples.TimeBasedAccessInterceptor">

<property name="openingTime"><value>9</value></property>

<property name="closingTime"><value>18</value></property>

</bean>

<beans>

package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

private int openingTime;

private int closingTime;

public void setOpeningTime(int openingTime) {

this.openingTime = openingTime;

}

public void setClosingTime(int closingTime) {

this.closingTime = closingTime;

}

public boolean preHandle(

HttpServletRequest request,

HttpServletResponse response,

Object handler)

throws Exception {

Calendar cal = Calendar.getInstance();

int hour = cal.get(HOUR\_OF\_DAY);

if (openingTime <= hour < closingTime) {

return true;

} else {

response.sendRedirect("http://host.com/outsideOfficeHours.html");

return false;

}

}

}

Any request coming in, will be intercepted by the TimeBasedAccessInterceptor, and if the current time is outside office hours, the user will be redirect to a static html file, saying for instance he can only access the website during office hours.

As you can see, Spring has an adapter to make it easy for you to extend the HandlerInterceptor.

### 12.5. Views and resolving them

No MVC framework for web applications is without a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying yourself to a specific view technology. Out-of-the-box, Spring enables you to use Java Server Pages, Velocity templates and XSLT views, for example. Chapter 13, Integrating view technologies has details of integrating various view technologies.

The two classes which are important to the way Spring handles views are the ViewResolver and the View. The View interface addresses the preparation of the request and hands the request over to one of the view technologies. The ViewResolver provides a mapping between view names and actual views.

#### 12.5.1. ViewResolvers

As discussed before, all controllers in the Spring web framework, return a ModelAndView instance. Views in Spring are addressed by a view name and are resolved by a viewresolver. Spring comes with quite a few view resolvers. We'll list most of them and then provide a couple of examples.

Table 12.5. View resolvers

ViewResolver Description

AbstractCachingViewResolver Abstract view resolver taking care of caching views. Lots of views need preparation before they can be used, extending from this viewresolver enables caching of views

XmlViewResolver Implementation of ViewResolver that accepts a config file written in XML to the same DTD as Spring's bean factories

ResourceBundleViewResolver Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath

UrlBasedViewResolver Simple implementation of ViewResolver that allows for direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings

InternalResourceViewResolver Convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses like JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass. See UrlBasedViewResolver's javadocs for details

VelocityViewResolver / FreeMarkerViewResolver Convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them

As an example, when using JSP for a view technology you can use the the UrlBasedViewResolver. This view resolver translates view names to a URL and hands the request over the RequestDispatcher to render the view.

<bean id="viewResolver"

class="org.springframework.web.servlet.view.UrlBasedViewResolver">

<property name="prefix"><value>/WEB-INF/jsp/</value></property>

<property name="suffix"><value>.jsp</value></property>

</bean>

When returning test as a viewname, this view resolver will hand the request over to the RequestDispatcher that'll send the request to /WEB-INF/jsp/test.jsp.

When mixing different view technologies in a web application, you can use the ResourceBundleViewResolver:

<bean id="viewResolver"

class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename"><value>views</value></property>

<property name="defaultParentView"><value>parentView</value></property>

</bean>

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename and for each view it is supposed to resolve, it uses the value of the property [viewname].class as the view class and the value of the property [viewname].url as the view url. As you can see, you can identify a parent view, from which all view in the properties file sort of extend. This way you can specify a default view class for instance.

A note on caching: subclasses of AbstractCachingViewResolver cache view instances they've resolved. This greatly improves performance when using certain view technology. It's possible to turn off the cache, by setting the cache property to false. Furthermore, if you have the requirement to be able to refresh a certain view at runtime (for example when a Velocity template has been modified), you can use the removeFromCache(String viewName, Locale loc) method.

#### 12.5.2. Chaining ViewResolvers

Spring supports more than just one view resolver. This allows you to chain resolvers and for example override specific views in certain circumstances. Chaining is view resolvers is pretty straightforward: just add more than one resolver to your application context and if necessary, set the order property to specify an order. Remember: the higher the order property, the later the view resolver will be positioned in the chain.

A chain of view resolvers for example could consist of two, one InternalResourceViewResolver (positioned as the last resolver) and an XmlViewResolver specifying Excel views (those are not supported by the InternalResourceViewResolver):

<bean id="jspViewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="order"><value>2</value></property>

<property name="viewClass">

<value>org.springframework.web.servlet.view.JstlView</value>

</property>

<property name="prefix"><value>/WEB-INF/jsp/</value></property>

<property name="suffix"><value>.jsp</value></property>

</bean>

<bean id="excelViewResolver">

class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="order"><value>1</value></property>

<property name="location"><value>/WEB-INF/views.xml</value></property>

</bean>

### views.xml

<beans>

<bean name="report" class="org.springframework.example.ReportExcelView"/>

</beans>

If a specific view resolver does not result in a view, Spring will inspect the context to see if other view resolvers are configured. If not, it'll throw an Exception. If there are additional view resolvers, it'll continue to inspect those.

You have to keep something else in mind: the contract of a view resolver mentions that a view resolver can return null to indicate the view could not be found. Not all view resolvers do this however! This is because the InternalResourceViewResolver for example simply cannot detect whether or not the view exists (it uses the RequestDispatcher internally, and dispatching is the only way to figure out if for example a JSP exists--this can only be done once). The same holds for the VelocityViewResolver and some others. Check the JavaDoc for the view resolver to see if you're dealing with a view resolver that does not report non-existing views. As a result of this, putting an InternalResourceViewResolver in the chain on a place other than the last, will result in the chain not being fully inspected, since the InternalResourceViewResolver will always return a view!

### 12.6. Using locales

Most parts of Spring's architecture support internationalization, just as the Spring web framework does. SpringWEB enables you to automatically resolve messages using the client's locale. This is done with LocaleResolver objects.

When a request comes in, the DispatcherServlet looks for a locale resolver and if it finds one it tries to use it and set the locale. Using the RequestContext.getLocale() method, you can always retrieve the locale that was resolved by the locale resolver.

Besides the automatic locale resolution, you can also attach an interceptor to the handler mapping (see Section 12.4.3, “Adding HandlerInterceptors” for more info on that), to change the locale under specific circumstances, based on a parameter occurring in the request for example.

Locale resolvers and interceptors are all defined in the org.springframework.web.servlet.i18n package, and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

#### 12.6.1. AcceptHeaderLocaleResolver

This locale resolver inspects the accept-language header in the request that was sent by the browser of the client. Usually this header field contains the locale of the client's operating system.

#### 12.6.2. CookieLocaleResolver

This locale resolver inspects a Cookie that might exist on the client, to see if there's a locale specified. If so, it uses that specific locale. Using the properties of this locale resolver, you can specify the name of the cookie, as well as the maximum age.

<bean id="localeResolver">

<property name="cookieName"><value>clientlanguage</value></property>

<!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->

<property name="cookieMaxAge"><value>100000</value></property>

</bean>

This is an example of defining a CookieLocaleResolver.

Table 12.6. Special beans in the WebApplicationContext

Property Default Description

cookieName classname + LOCALE The name of the cookie

cookieMaxAge Integer.MAX\_INT The maximum time a cookie will stay persistent on the client. If -1 is specified, the cookie will not be persisted, at least, only until the client shuts down his or her browser

cookiePath / Using this parameter, you can limit the visibility of the cookie to a certain part of your site. When cookiePath is specified, the cookie will only be visible to that path, and the paths below

#### 12.6.3. SessionLocaleResolver

The SessionLocaleResolver allows you to retrieve locales from the session that might be associated to the user's request.

#### 12.6.4. LocaleChangeInterceptor

You can build in changing of locales using the LocaleChangeInterceptor. This interceptor needs to be added to one of the handler mappings (see Section 12.4, “Handler mappings”) and it will detect a parameter in the request and change the locale (it calls setLocale() on the LocaleResolver that also exists in the context).

<bean id="localeChangeInterceptor"

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">

<property name="paramName"><value>siteLanguage</value></property>

</bean>

<bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="interceptors">

<list>

<ref local="localeChangeInterceptor"/>

</list>

</property>

<property name="mappings">

<props>

<prop key="/\*\*/\*.view">someController</prop>

</props>

</property>

</bean>

All calls to all \*.view resources containing a parameter named siteLanguage will now change the locale. So a call to http://www.sf.net/home.view?siteLanguage=nl will change the site language to Dutch.

### 12.7. Using themes

Dummy paragraph

### 12.8. Spring's multipart (fileupload) support

#### 12.8.1. Introduction

Spring has built-in multipart support to handle fileuploads in web applications. The design for the multipart support is done with pluggable MultipartResovler objects, defined in the org.springframework.web.multipart package. Out of the box, Spring provides MultipartResolver for use with Commons FileUpload (http://jakarta.apache.org/commons/fileupload) and COS FileUpload (http://www.servlets.com/cos). How uploading files is supported will be described in the rest of this chapter.

By default, no multipart handling will be done by Spring, as some developers will want to handle multiparts themselves. You'll have to enable it yourself by adding a multipart resolver to the web application's context. After you've done that, each request will be inspected for a multipart that it might contain. If no such multipart is found, the request will continue as expected. However, if a multipart is found in the request, the MultipartResolver that has been declared in your context will resolve. After that, the multipart attribute in your request will be treated as any other attributes.

#### 12.8.2. Using the MultipartResolver

The following example shows how to use the CommonsMultipartResolver:

<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

<!-- one of the properties available; the maximum file size in bytes -->

<property name="maxUploadSize">

<value>100000</value>

</property>

</bean>

This is an example using the CosMultipartResolver:

<bean id="multipartResolver"

class="org.springframework.web.multipart.cos.CosMultipartResolver">

<!-- one of the properties available; the maximum file size in bytes -->

<property name="maxUploadSize">

<value>100000</value>

</property>

</bean>

Of course you need to stick the appropriate jars in your classpath for the multipart resolver to work. In the case of the CommonsMultipartResolver, you need to use commons-fileupload.jar, while in the case of the CosMultipartResolver, use cos.jar.

Now that you have seen how to set Spring up to handle multipart requests, let's talk about how to actually use it. When the Spring DispatcherServlet detects a Multipart request, it activates the resolver that has been declared in your context and hands over the request. What it basically does is wrap the current HttpServletRequest into a MultipartHttpServletRequest that has support for multiparts. Using the MultipartHttpServletRequest you can get information about the multiparts contained by this request and actually get the multiparts themselves in your controllers.

#### 12.8.3. Handling a fileupload in a form

After the MultipartResolver has finished doing its job, the request will be processed like any other. To use it, you create a form with an upload field, then let Spring bind the file on your form. Just as with any other property that's not automagically convertible to a String or primitive type, to be able to put binary data in your beans you have to register a custom editor with the ServletRequestDatabinder. There are a couple of editors available for handling files and setting the results on a bean. There's a StringMultipartEditor capable of converting files to Strings (using a user-defined character set) and there's a ByteArrayMultipartEditor which converts files to byte arrays. They function just as the CustomDateEditor does.

So, to be able to upload files using a form in a website, declare the resolver, a url mapping to a controller that will process the bean, and the controller itself.

<beans>

...

<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<prop key="/upload.form">fileUploadController</prop>

</props>

</property>

</bean>

<bean id="fileUploadController" class="examples.FileUploadController">

<property name="commandClass"><value>examples.FileUploadBean</value></property>

<property name="formView"><value>fileuploadform</value></property>

<property name="successView"><value>confirmation</value></property>

</bean>

</beans>

After that, create the controller and the actual bean holding the file property

// snippet from FileUploadController

public class FileUploadController extends SimpleFormController {

protected ModelAndView onSubmit(

HttpServletRequest request,

HttpServletResponse response,

Object command,

BindException errors)

throws ServletException, IOException {

// cast the bean

FileUploadBean bean = (FileUploadBean)command;

// let's see if there's content there

byte[] file = bean.getFile();

if (file == null) {

// hmm, that's strange, the user did not upload anything

}

// well, let's do nothing with the bean for now and return:

return super.onSubmit(request, response, command, errors);

}

protected void initBinder(

HttpServletRequest request,

ServletRequestDataBinder binder)

throws ServletException {

// to actually be able to convert Multipart instance to byte[]

// we have to register a custom editor (in this case the

// ByteArrayMultipartEditor

binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());

// now Spring knows how to handle multipart object and convert them

}

}

// snippet from FileUploadBean

public class FileUploadBean {

private byte[] file;

public void setFile(byte[] file) {

this.file = file;

}

public byte[] getFile() {

return file;

}

}

As you can see, the FileUploadBean has a property typed byte[] that holds the file. The controller registers a custom editor to let Spring know how to actually convert the multipart objects the resolver has found to properties specified by the bean. In these examples, nothing is done with the byte[] property of the bean itself, but in practice you can do whatever you want (save it in a database, mail it to somebody, etcetera).

But we're still not finished. To actually let the user upload something, we have to create a form:

<html>

<head>

<title>Upload a file please</title>

</head>

<body>

<h1>Please upload a file</h1>

<form method="post" action="upload.form" enctype="multipart/form-data">

<input type="file" name="file"/>

<input type="submit"/>

</form>

</body>

</html>

As you can see, we've created a field named after the property of the bean that holds the byte[]. Furthermore we've added the encoding attribute which is necessary to let the browser know how to encode the multipart fields (do not forget this!). Right now everything should work.

### 12.9. Handling exceptions

Spring provides HandlerExceptionResolvers to ease the pain of unexpected exceptions occurring while your request is being handled by a controller which matched the request. HandlerExceptionResolvers somewhat resemble the exception mappings you can define in the web application descriptor web.xml. However, they provide a more flexible way to handle exceptions. They provide information about what handler was executing when the exception was thrown. Furthermore, a programmatic way of handling exception gives you many more options for how to respond appropriately before the request is forwarded to another URL (the same end result as when using the servlet specific exception mappings).

Besides implementing the HandlerExceptionResolver, which is only a matter of implementing the resolveException(Exception, Handler) method and returning a ModelAndView, you may also use the SimpleMappingExceptionResolver. This resolver enables you to take the class name of any exception that might be thrown and map it to a view name. This is functionally equivalent to the exception mapping feature from the Servlet API, but it's also possible to implement more fine grained mappings of exceptions from different handlers.

## 13. Integrating view technologies

### 13.1. Introduction

One of the areas in which Spring excels is in the separation of view technologies from the rest of the MVC framework. For example, deciding to use Velocity or XSLT in place of an existing JSP is primarily a matter of configuration. This chapter covers the major view technologies that work with Spring and touches briefly on how to add new ones. This chapter assumes you are already familiar with Section 12.5, “Views and resolving them” which covers the basics of how views in general are coupled to the MVC framework.

### 13.2. JSP & JSTL

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views. Using JSP or JSTL is done using a normal viewresolver defined in the WebApplicationContext. Furthermore, of course you need to write some JSPs that will actually render the view. This part describes some of the additional features Spring provides to facilitate JSP development.

#### 13.2.1. View resolvers

Just as with any other view technology you're integrating with Spring, for JSPs you'll need a view resolver that will resolve your views. The most commonly used view resolvers when developing with JSPs are the InternalResourceViewResolver and the ResourceBundleViewResolver. Both are declared in the WebApplicationContext:

# The ResourceBundleViewResolver:

<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename"><value>views</value></property>

</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):

welcome.class=org.springframework.web.servlet.view.JstlView

welcome.url=/WEB-INF/jsp/welcome.jsp

productList.class=org.springframework.web.servlet.view.JstlView

productList.url=/WEB-INF/jsp/productlist.jsp

As you can see, the ResourceBundleViewResolver needs a properties file defining the view names mapped to 1) a class and 2) a URL. With a ResourceBundleViewResolver you can mix different types of views using only one resolver.

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></property>

<property name="prefix"><value>/WEB-INF/jsp/</value></property>

<property name="suffix"><value>.jsp</value></property>

</bean>

The InternalResourceBundleViewResolver can be configured for using JSPs as described above. As a best practice, we strongly encourage placing your JSP files in a a directory under the WEB-INF directory, so there can be no direct access by clients.

#### 13.2.2. 'Plain-old' JSPs versus JSTL

When using Java Standard Tag Library you must use a special view class, the JstlView, as JSTL needs some preparation before things such as the i18N features will work.

#### 13.2.3. Additional tags facilitating development

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags have html escaping features to enable or disable escaping of characters.

The tag library descriptor (TLD) is included in the spring.jar as well in the distribution itself. More information about the individual tags can be found online: http://www.springframework.org/docs/taglib/index.html.

### 13.3. Tiles

It is possible to integrate Tiles - just as any other view technology - in web applications using Spring. The following describes in a broad way how to do this.

#### 13.3.1. Dependencies

To be able to use Tiles you have to have a couple of additional dependencies included in your project. The following is the list of dependencies you need.

struts version 1.1

commons-beanutils

commons-digester

commons-logging

commons-lang

The dependencies are all available in the Spring distribution.

#### 13.3.2. How to integrate Tiles

To be able to use Tiles, you have to configure it using files containing definitions (for basic information on definitions and other Tiles concepts, please have a look at http://jakarta.apache.org/struts). In Spring this is done using the TilesConfigurer. Have a look at the following piece of example ApplicationContext configuration:

<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles.TilesConfigurer">

<property name="factoryClass">

<value>org.apache.struts.tiles.xmlDefinition.I18nFactorySet</value>

</property>

<property name="definitions">

<list>

<value>/WEB-INF/defs/general.xml</value>

<value>/WEB-INF/defs/widgets.xml</value>

<value>/WEB-INF/defs/administrator.xml</value>

<value>/WEB-INF/defs/customer.xml</value>

<value>/WEB-INF/defs/templates.xml</value>

</list>

</property>

</bean>

As you can see, there are five files containing definitions, which are all located in the WEB-INF/defs directory. At initialization of the WebApplicationContext, the files will be loaded and the definitionsfactory defined by the factoryClass-property is initialized. After that has been done, the tiles includes in the definition files can be used as views within your Spring web application. To be able to use the views you have to have a ViewResolver just as with any other view technology used with Spring. Below you can find two possibilities, the InternalResourceViewResolver and the ResourceBundleViewResolver.

##### 13.3.2.1. InternalResourceViewResolver

The InternalResourceViewResolver instantiates the given viewClass for each view it has to resolve.

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="requestContextAttribute"><value>requestContext</value></property>

<property name="viewClass">

<value>org.springframework.web.servlet.view.tiles.TilesView</value>

</property>

</bean>

##### 13.3.2.2. ResourceBundleViewResolver

The ResourceBundleViewResolver has to be provided with a property file containing viewnames and viewclasses the resolver can use:

<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename"><value>views</value></property>

</bean>

...

welcomeView.class=org.springframework.web.servlet.view.tiles.TilesView

welcomeView.url=welcome (<b>this is the name of a definition</b>)

vetsView.class=org.springframework.web.servlet.view.tiles.TilesView

vetsView.url=vetsView (<b>again, this is the name of a definition</b>)

findOwnersForm.class=org.springframework.web.servlet.view.JstlView

findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp

...

As you can see, when using the ResourceBundleViewResolver, you can mix view using different view technologies.

### 13.4. Velocity & FreeMarker

Velocity and FreeMarker are two templating languages that can both be used as view technologies within Spring MVC applications. The languages are quite similar and serve similar needs and so are considered together in this section. For semantic and syntactic differences between the two languages, see the FreeMarker web site.

#### 13.4.1. Dependencies

Your web application will need to include velocity-1.x.x.jar or freemarker-2.x.jar in order to work with Velocity or FreeMarker respectively and commons-collections.jar needs also to be available for Velocity. Typically they are included in the WEB-INF/lib folder where they are guaranteed to be found by a J2EE server and added to the classpath for your application. It is of course assumed that you already have the spring.jar in your WEB-INF/lib folder too! The latest stable velocity, freemarker and commons collections jars are supplied with the Spring framework and can be copied from the relevant /lib/ sub-directories. If you make use of Spring's dateToolAttribute or numberToolAttribute in your Velocity views, you will also need to include the velocity-tools-generic-1.x.jar

#### 13.4.2. Context configuration

A suitable configuration is initialized by adding the relevant configurer bean definition to your \*-servlet.xml as shown below:

<!--

This bean sets up the Velocity environment for us based on a root path for templates.

Optionally, a properties file can be specified for more control over the Velocity

environment, but the defaults are pretty sane for file based template loading.

-->

<bean

id="velocityConfig"

class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">

<property name="resourceLoaderPath"><value>/WEB-INF/velocity/</value></property>

</bean>

<!--

View resolvers can also be configured with ResourceBundles or XML files. If you need

different view resolving based on Locale, you have to use the resource bundle resolver.

-->

<bean

id="viewResolver"

class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">

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

<property name="prefix"><value></value></property>

<property name="suffix"><value>.vm</value></property>

</bean>

<!-- freemarker config -->

<bean

id="freemarkerConfig"

class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">

<property name="templateLoaderPath"><value>/WEB-INF/freemarker/</value></property>

</bean>

<!--

View resolvers can also be configured with ResourceBundles or XML files. If you need

different view resolving based on Locale, you have to use the resource bundle resolver.

-->

<bean

id="viewResolver"

class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">

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

<property name="prefix"><value></value></property>

<property name="suffix"><value>.ftl</value></property>

</bean>

NB: For non web-apps add a VelocityConfigurationFactoryBean or a FreeMarkerConfigurationFactoryBean to your application context definition file.

#### 13.4.3. Creating templates

Your templates need to be stored in the directory specified by the \*Configurer bean shown above in Section 13.4.2, “Context configuration” This document does not cover details of creating templates for the two languages - please see their relevant websites for information. If you use the view resolvers highlighted, then the logical view names relate to the template file names in similar fashion to InternalResourceViewResolver for JSP's. So if your controller returns a ModelAndView object containing a view name of "welcome" then the resolvers will look for the /WEB-INF/freemarker/welcome.ftl or /WEB-INF/velocity/welcome.vm template as appropriate.

#### 13.4.4. Advanced configuration

The basic configurations highlighted above will be suitable for most application requirements, however additional configuration options are available for when unusual or advanced requirements dictate.

##### 13.4.4.1. velocity.properties

This file is completely optional, but if specified, contains the values that are passed to the Velocity runtime in order to configure velocity itself. Only required for advanced configurations, if you need this file, specify its location on the VelocityConfigurer bean definition above.

<bean

id="velocityConfig"

class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">

<property name="configLocation">

<value>/WEB-INF/velocity.properties</value>

</property>

</bean>

Alternatively, you can specify velocity properties directly in the bean definition for the Velocity config bean by replacing the "configLocation" property with the following inline properties.

<bean

id="velocityConfig"

class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">

<property name="velocityProperties">

<props>

<prop key="resource.loader">file</prop>

<prop key="file.resource.loader.class">

org.apache.velocity.runtime.resource.loader.FileResourceLoader

</prop>

<prop key="file.resource.loader.path">${webapp.root}/WEB-INF/velocity</prop>

<prop key="file.resource.loader.cache">false</prop>

</props>

</property>

</bean>

Refer to the API documentation for Spring configuration of Velocity, or the Velocity documentation for examples and definitions of the velocity.properties file itself.

##### 13.4.4.2. FreeMarker

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker Configuration object managed by Spring by setting the appropriate bean properties on the FreeMarkerConfigurer bean. The freemarkerSettings property requires a java.util.Properties object and the freemarkerVariables property requires a java.util.Map.

<bean

id="freemarkerConfig"

class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">

<property name="templateLoaderPath"><value>/WEB-INF/freemarker/</value></property>

<property name="freemarkerVariables">

<map>

<entry key="xml\_escape"><ref local="fmXmlEscape"/></entry>

</map>

</property>

</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>

See the FreeMarker documentation for details of settings and variables as they apply to the Configuration object.

#### 13.4.5. Bind support and form handling

Spring provides a tag library for use in JSP's that contains (amongst other things) a <spring:bind> tag. This tag primarily enables forms to display values from form backing objects and to show the results of failed validations from a Validator in the web or business tier. From version 1.1, Spring now has support for the same functionality in both Velocity and FreeMarker, with additional convenience macros for generating form input elements themselves.

##### 13.4.5.1. the bind macros

A standard set of macros are maintained within the spring.jar file for both languages, so they are always available to a suitably configured application. However they can only be used if your view sets the bean property exposeSpringMacroHelpers to true . The same property can be set on VelocityViewResolver or FreeMarkerViewResolver too if you happen to be using it, in which case all of your views will inherit the value from it. Note that this property is not required for any aspect of HTML form handling except where you wish to take advantage of the Spring macros. Below is an example of a view.properties file showing correct configuration of such a view for either language;

personFormV.class=org.springframework.web.servlet.view.velocity.VelocityView

personFormV.url=personForm.vm

personFormV.exposeSpringMacroHelpers=true

personFormF.class=org.springframework.web.servlet.view.freemarker.FreeMarkerView

personFormF.url=personForm.ftl

personFormF.exposeSpringMacroHelpers=true

Some of the macros defined in the Spring libraries are considered internal (private) but no such scoping exists in the macro definitions making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to be directly calling from within your templates. If you wish to view the macro code directly, the files are called spring.vm / spring.ftl and are in the packages org.springframework.web.servlet.view.velocity or org.springframework.web.servlet.view.freemarker respectively.

##### 13.4.5.2. simple binding

In your html forms (vm / ftl templates) that act as the 'formView' for a Spring form controller, you can use code similar to the following to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. Note that the name of the command object is "command" by default, but can be overridden in your MVC configuration by setting the 'commandName' bean property on your form controller. Example code is shown below for the personFormV and personFormF views configured earlier;

<!-- velocity macros are automatically available -->

<html>

...

<form action="" method="POST">

Name:

#springBind( "command.name" )

<input type="text"

name="${status.expression}"

value="$!status.value" /><br>

#foreach($error in $status.errorMessages) <b>$error</b> <br> #end

<br>

...

<input type="submit" value="submit"/>

</form>

...

</html>

<!-- freemarker macros have to be imported into a namespace. We strongly

recommend sticking to 'spring' -->

<#import "spring.ftl" as spring />

<html>

...

<form action="" method="POST">

Name:

<@spring.bind "command.name" />

<input type="text"

name="${spring.status.expression}"

value="${spring.status.value?default("")}" /><br>

<#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>

<br>

...

<input type="submit" value="submit"/>

</form>

...

</html>

#springBind / <@spring.bind> requires a 'path' argument which consists of the name of your command object (it will be 'command' unless you changed it in your FormController properties) followed by a period and the name of the field on the command object you wish to bind to. Nested fields can be used too such as "command.address.street". The bind macro assumes the default HTML escaping behavior specified by the ServletContext parameter defaultHtmlEscape in web.xml

The optional form of the macro called #springBindEscaped / <@spring.bindEscaped> takes a second argument and explicitly specifies whether HTML escaping should be used in the status error messages or values. Set to true or false as required. Additional form handling macros simplify the use of HTML escaping and these macros should be used wherever possible. They are explained in the next section.

##### 13.4.5.3. form input generation macros

Additional convenience macros for both languages simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and they can be mixed and matched with simple HTML or calls direct to the spring bind macros highlighted previously.

The following table of available macros show the VTL and FTL definitions and the parameter list that each takes.

Table 13.1. table of macro definitions

macro VTL definition FTL definition

message (output a string from a resource bundle based on the code parameter) #springMessage($code) <@spring.message code/>

messageText (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter) #springMessageText($code $default) <@spring.messageText code, default/>

url (prefix a relative URL with the application's context root) #springUrl($relativeUrl) <@spring.url relativeUrl/>

formInput (standard input field for gathering user input) #springFormInput($path $attributes) <@spring.formInput path, attributes, fieldType/>

formHiddenInput \* (hidden input field for submitting non-user input) #springFormHiddenInput($path $attributes) <@spring.formHiddenInput path, attributes/>

formPasswordInput \* (standard input field for gathering passwords. Note that no value will ever be populated in fields of this type) #springFormPasswordInput($path $attributes) <@spring.formPasswordInput path, attributes/>

formTextarea (large text field for gathering long, freeform text input) #springFormTextarea($path $attributes) <@spring.formTextarea path, attributes/>

formSingleSelect (drop down box of options allowing a single required value to be selected) #springFormSingleSelect( $path $options $attributes) <@spring.formSingleSelect path, options, attributes/>

formMultiSelect (a list box of options allowing the user to select 0 or more values) #springFormMultiSelect($path $options $attributes) <@spring.formMultiSelect path, options, attributes/>

formRadioButtons (a set of radio buttons allowing a single selection to be made from the available choices) #springFormRadioButtons($path $options $separator $attributes) <@spring.formRadioButtons path, options separator, attributes/>

formCheckboxes (a set of checkboxes allowing 0 or more values to be selected) #springFormCheckboxes($path $options $separator $attributes) <@spring.formCheckboxes path, options, separator, attributes/>

showErrors (simplify display of validation errors for the bound field) #springShowErrors($separator $classOrStyle) <@spring.showErrors separator, classOrStyle/>

\* In FTL (FreeMarker), these two macros are not actually required as you can use the normal formInput macro, specifying 'hidden' or 'password' as the value for the fieldType parameter.

The parameters to any of the above macros have consistent meanings:

path: the name of the field to bind to (ie "command.name")

options: a Map of all the available values that can be selected from in the input field. The keys to the map represent the values that will be POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from the corresponding values posted back by the form. Usually such a map is supplied as reference data by the controller. Any Map implementation can be used depending on required behavior. For strictly sorted maps, a SortedMap such as a TreeMap with a suitable Comparator may be used and for arbitrary Maps that should return values in insertion order, use a LinkedHashMap or a LinkedMap from commons-collections.

separator: where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (ie "<br>").

attributes: an additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field you may supply attributes as 'rows="5" cols="60"' or you could pass style information such as 'style="border:1px solid silver"'.

classOrStyle: for the showErrors macro, the name of the CSS class that the span tag wrapping each error will use. If no information is supplied (or the value is empty) then the errors will be wrapped in <b></b> tags.

Examples of the macros are outlined below some in FTL and some in VTL. Where usage differences exist between the two languages, they are explained in the notes.

13.4.5.3.1. Input Fields

<!-- the Name field example from above using form macros in VTL -->

...

Name:

#springFormInput("command.name" "")<br>

#springShowErrors("<br>" "")<br>

The formInput macro takes the path parameter (command.name) and an additional attributes parameter which is empty in the example above. The macro, along with all other form generation macros, performs an implicit spring bind on the path parameter. The binding remains valid until a new bind occurs so the showErrors macro doesn't need to pass the path parameter again - it simply operates on whichever field a bind was last created for.

The showErrors macro takes a separator parameter (the characters that will be used to separate multiple errors on a given field) and also accepts a second parameter, this time a class name or style attribute. Note that FreeMarker is able to specify default values for the attributes parameter, unlike Velocity, and the two macro calls above could be expressed as follows in FTL:

<@spring.formInput "command.name"/>

<@spring.showErrors "<br>"/>

Output is shown below of the form fragment generating the name field, and displaying a validation error after the form was submitted with no value in the field. Validation occurs through Spring's Validation framework.

The generated HTML looks like this:

Name:

<input type="text" name="name" value=""

>

<br>

<b>required</b>

<br>

<br>

The formTextarea macro works the same way as the formInput macro and accepts the same parameter list. Commonly, the second parameter (attributes) will be used to pass style information or rows and cols attributes for the textarea.

13.4.5.3.2. Selection Fields

Four selection field macros can be used to generate common UI value selection inputs in your HTML forms.

formSingleSelect

formMultiSelect

formRadioButtons

formCheckboxes

Each of the four macros accepts a Map of options containing the value for the form field, and the label corresponding to that value. The value and the label can be the same.

An example of radio buttons in FTL is below. The form backing object specifies a default value of 'London' for this field and so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'.

...

Town:

<@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>

This renders a line of radio buttons, one for each value in cityMap using the separator "". No additional attributes are supplied (the last parameter to the macro is missing). The cityMap uses the same String for each key-value pair in the map. The map's keys are what the form actually submits as POSTed request parameters, map values are the labels that the user sees. In the example above, given a list of three well known cities and a default value in the form backing object, the HTML would be

Town:

<input type="radio" name="address.town" value="London"

>

London

<input type="radio" name="address.town" value="Paris"

checked="checked"

>

Paris

<input type="radio" name="address.town" value="New York"

>

New York

If your application expects to handle cities by internal codes for example, the map of codes would be created with suitable keys like the example below.

protected Map referenceData(HttpServletRequest request) throws Exception {

Map cityMap = new LinkedHashMap();

cityMap.put("LDN", "London");

cityMap.put("PRS", "Paris");

cityMap.put("NYC", "New York");

Map m = new HashMap();

m.put("cityMap", cityMap);

return m;

}

The code would now produce output where the radio values are the relevant codes but the user still sees the more user friendly city names.

Town:

<input type="radio" name="address.town" value="LDN"

>

London

<input type="radio" name="address.town" value="PRS"

checked="checked"

>

Paris

<input type="radio" name="address.town" value="NYC"

>

New York

##### 13.4.5.4. overriding HTML escaping and making tags XHTML compliant

Default usage of the form macros above will result in HTML tags that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your web.xml as used by Spring's bind support. In order to make the tags XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model where they will be visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

To switch to XHTML compliance for your tags, specify a value of 'true' for a model/context variable named xhtmlCompliant:

## for Velocity..

#set($springXhtmlCompliant = true)

<#-- for FreeMarker -->

<#assign xhtmlCompliant = true in spring>

Any tags generated by the Spring macros will now be XHTML compliant after processing this directive.

In similar fashion, HTML escaping can be specified per field:

<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>

<#-- next field will use HTML escaping -->

<@spring.formInput "command.name" />

<#assign htmlEscape = false in spring>

<#-- all future fields will be bound with HTML escaping off -->

### 13.5. XSLT

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring application.

#### 13.5.1. My First Words

This example is a trivial Spring application that creates a list of words in the Controller and adds them to the model map. The map is returned along with the view name of our XSLT view. See Section 12.3, “Controllers” for details of Spring Controllers. The XSLT view will turn the list of words into a simple XML document ready for transformation.

##### 13.5.1.1. Bean definitions

Configuration is standard for a simple Spring application. The dispatcher servlet config file contains a reference to a ViewResolver, URL mappings and a single controller bean..

<bean id="homeController"class="xslt.HomeController"/>

..that implements our word generation 'logic'.

##### 13.5.1.2. Standard MVC controller code

The controller logic is encapsulated in a subclass of AbstractController, with the handler method being defined like so..

protected ModelAndView handleRequestInternal(

HttpServletRequest req,

HttpServletResponse resp)

throws Exception {

Map map = new HashMap();

List wordList = new ArrayList();

wordList.add("hello");

wordList.add("world");

map.put("wordList", wordList);

return new ModelAndView("home", map);

}

So far we've done nothing that's XSLT specific. The model data has been created in the same way as you would for any other Spring MVC application. Depending on the configuration of the application now, that list of words could be rendered by JSP/JSTL by having them added as request attributes, or they could be handled by Velocity by adding the object to the VelocityContext. In order to have XSLT render them, they of course have to be converted into an XML document somehow. There are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

##### 13.5.1.3. Convert the model data to XML

In order to create a DOM document from our list of words or any other model data, we subclass org.springframework.web.servlet.view.xslt.AbstractXsltView. In doing so, we must implement the abstract method createDomNode(). The first parameter passed to this method is our model Map. Here's the complete listing of the HomePage class in our trivial word application - it uses JDOM to build the XML document before converting it to the required W3C Node, but this is simply because I find JDOM (and Dom4J) easier API's to handle than the W3C API.

package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

protected Node createDomNode(

Map model, String rootName, HttpServletRequest req, HttpServletResponse res

) throws Exception {

org.jdom.Document doc = new org.jdom.Document();

Element root = new Element(rootName);

doc.setRootElement(root);

List words = (List) model.get("wordList");

for (Iterator it = words.iterator(); it.hasNext();) {

String nextWord = (String) it.next();

Element e = new Element("word");

e.setText(nextWord);

root.addContent(e);

}

// convert JDOM doc to a W3C Node and return

return new DOMOutputter().output( doc );

}

}

13.5.1.3.1. Adding stylesheet parameters

A series of parameter name/value pairs can optionally be defined by your subclass which will be added to the transformation object. The parameter names must match those defined in your XSLT template declared with <xsl:param name="myParam">defaultValue</xsl:param> To specify the parameters, override the method getParameters() from AbstractXsltView and return a Map of the name/value pairs. If your parameters need to derive information from the current request, you can (from version 1.1) override the getParameters(HttpServletRequest request) method instead.

13.5.1.3.2. Formatting dates and currency

Unlike JSTL and Velocity, XSLT has relatively poor support for locale based currency and date formatting. In recognition of the fact, Spring provides a helper class that you can use from within your createDomNode() methods to get such support. See the javadocs for org.springframework.web.servlet.view.xslt.FormatHelper

##### 13.5.1.4. Defining the view properties

The views.properties file (or equivalent xml definition if you're using an XML based view resolver as we did in the Velocity examples above) looks like this for the one-view application that is 'My First Words'..

home.class=xslt.HomePage

home.stylesheetLocation=/WEB-INF/xsl/home.xslt

home.root=words

Here, you can see how the view is tied in with the HomePage class just written which handles the model domification in the first property '.class'. The stylesheetLocation property obviously points to the XSLT file which will handle the XML transformation into HTML for us and the final property '.root' is the name that will be used as the root of the XML document. This gets passed to the HomePage class above in the second parameter to the createDomNode method.

##### 13.5.1.5. Document transformation

Finally, we have the XSLT code used for transforming the above document. As highlighted in the views.properties file, it is called home.xslt and it lives in the war file under WEB-INF/xsl.

<?xml version="1.0"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="text/html" omit-xml-declaration="yes"/>

<xsl:template match="/">

<html>

<head><title>Hello!</title></head>

<body>

<h1>My First Words</h1>

<xsl:for-each select="wordList/word">

<xsl:value-of select="."/><br />

</xsl:for-each>

</body>

</html>

</xsl:template>

</xsl:stylesheet>

#### 13.5.2. Summary

A summary of the files discussed and their location in the WAR file is shown in the simplified WAR structure below.

ProjectRoot

|

+- WebContent

|

+- WEB-INF

|

+- classes

| |

| +- xslt

| | |

| | +- HomePageController.class

| | +- HomePage.class

| |

| +- views.properties

|

+- lib

| |

| +- spring.jar

|

+- xsl

| |

| +- home.xslt

|

+- frontcontroller-servlet.xml

You will also need to ensure that an XML parser and an XSLT engine are available on the classpath. JDK 1.4 provides them by default, and most J2EE containers will also make them available by default, but it's a possible source of errors to be aware of.

### 13.6. Document views (PDF/Excel)

#### 13.6.1. Introduction

Returning an HTML page isn't always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the correct content type to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.

In order to use Excel views, you need to add the 'poi' library to your classpath, and for PDF generation, the iText.jar. Both are included in the main Spring distribution.

#### 13.6.2. Configuration and setup

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

##### 13.6.2.1. Document view definitions

Firstly, let's amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier..

home.class=xslt.HomePage

home.stylesheetLocation=/WEB-INF/xsl/home.xslt

home.root=words

xl.class=excel.HomePage

pdf.class=pdf.HomePage

If you want to start with a template spreadsheet to add your model data to, specify the location as the 'url' property in the view definition

##### 13.6.2.2. Controller code

The controller code we'll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

##### 13.6.2.3. Subclassing for Excel views

Exactly as we did for the XSLT example, we'll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass of org.springframework.web.servlet.view.document.AbstractExcelView and implementing the buildExcelDocument

Here's the complete listing for our Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet..

package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

protected void buildExcelDocument(

Map model,

HSSFWorkbook wb,

HttpServletRequest req,

HttpServletResponse resp)

throws Exception {

HSSFSheet sheet;

HSSFRow sheetRow;

HSSFCell cell;

// Go to the first sheet

// getSheetAt: only if wb is created from an existing document

//sheet = wb.getSheetAt( 0 );

sheet = wb.createSheet("Spring");

sheet.setDefaultColumnWidth((short)12);

// write a text at A1

cell = getCell( sheet, 0, 0 );

setText(cell,"Spring-Excel test");

List words = (List ) model.get("wordList");

for (int i=0; i < words.size(); i++) {

cell = getCell( sheet, 2+i, 0 );

setText(cell, (String) words.get(i));

}

}

}

If you now amend the controller such that it returns xl as the name of the view (return new ModelAndView("xl", map);) and run your application again, you should find that the Excel spreadsheet is created and downloaded automagically when you request the same page as before.

##### 13.6.2.4. Subclassing for PDF views

The PDF version of the word list is even simpler. This time, the class extends org.springframework.web.servlet.view.document.AbstractPdfView and implements the buildPdfDocument() method as follows..

package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

protected void buildPdfDocument(

Map model,

Document doc,

PdfWriter writer,

HttpServletRequest req,

HttpServletResponse resp)

throws Exception {

List words = (List) model.get("wordList");

for (int i=0; i<words.size(); i++)

doc.add( new Paragraph((String) words.get(i)));

}

}

Once again, amend the controller to return the pdf view with a return new ModelAndView("pdf", map); and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

### 13.7. JasperReports

JasperReports (http://jasperreports.sourceforge.net) is a powerful, open-source reporting engine that supports the creation of report designs using an easily understood XML file formats. JasperReports is capable of rendering reports output into four different formats: CSV, Excel, HTML and PDF.

#### 13.7.1. Dependencies

Your application will need to include the latest release of JasperReports, which at the time of writing was 0.6.1. JasperReports itself depends on the following projects:

BeanShell

Commons BeanUtils

Commons Collections

Commons Digester

Commons Logging

iText

POI

JasperReports also requires a JAXP compliant XML parser.

#### 13.7.2. Configuration

To configure JasperReports views in your ApplicationContext you have to define a ViewResolver to map view names to the appropriate view class depending on which format you want your report rendered in.

##### 13.7.2.1. Configuring the ViewResolver

Typically, you will use the ResourceBundleViewResolver to map view names to view classes and files in a properties file

<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename">

<value>views</value>

</property>

</bean>

Here we've configured an instance of ResourceBundleViewResolver which will look for view mappings in the resource bundle with base name views. The exact contents of this file is described in the next section.

##### 13.7.2.2. Configuring the Views

Spring contains five different View implementations for JasperReports four of which corresponds to one of the four output formats supported by JasperReports and one that allows for the format to be determined at runtime:

Table 13.2. JasperReports View Classes

Class Name Render Format

JasperReportsCsvView CSV

JasperReportsHtmlView HTML

JasperReportsPdfView PDF

JasperReportsXlsView Microsoft Excel

JasperReportsMultiFormatView Decided at runtime (see Section 13.7.2.4, “Using JasperReportsMultiFormatView”)

Mapping one of these classes to a view name and a report file is simply a matter of adding the appropriate entries into the resource bundle configured in the previous section as shown here:

simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView

simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper

Here you can see that the view with name, simpleReport, is mapped to the JasperReportsPdfView class. This will cause the output of this report to be rendered in PDF format. The url property of the view is set to the location of the underlying report file.

##### 13.7.2.3. About Report Files

JasperReports has two distinct types of report file: the design file, which has a .jrxml extension, and the compiled report file, which has a .jasper extension. Typically, you use the JasperReports Ant task to compile your .jrxml design file into a .jasper file before deploying it into your application. With Spring you can map either of these files to your report file and Spring will take care of compiling the .jrxml file on the fly for you. You should note that after a .jrxml file is compiled by Spring, the compiled report is cached for the life of the application. To make changes to the file you will need to restart your application.

##### 13.7.2.4. Using JasperReportsMultiFormatView

The JasperReportsMultiFormatView allows for report format to be specified at runtime. The actual rendering of the report is delegated to one of the other JasperReports view classes - the JasperReportsMultiFormatView class simply adds a wrapper layer that allows for the exact implementation to be specified at runtime.

The JasperReportsMultiFormatView class introduces two concepts: the format key and the discriminator key. The JasperReportsMultiFormatView class uses the mapping key to lookup the actual view implementation class and uses the format key to lookup up the mapping key. From a coding perspective you add an entry to your model with the formay key as the key and the mapping key as the value, for example:

public ModelAndView handleSimpleReportMulti(HttpServletRequest request,

HttpServletResponse response) throws Exception {

String uri = request.getRequestURI();

String format = uri.substring(uri.lastIndexOf(".") + 1);

Map model = getModel();

model.put("format", format);

return new ModelAndView("simpleReportMulti", model);

}

In this example, the mapping key is determined from the extension of the request URI and is added to the model under the default format key: format. If you wish to use a different format key then you can configure this using the formatKey property of the JasperReportsMultiFormatView class.

By default the following mapping key mappings are configured in JasperReportsMultiFormatView:

Table 13.3. JasperReportsMultiFormatView Default Mapping Key Mappings

Mapping Key View Class

csv JasperReportsCsvView

html JasperReportsHtmlView

pdf JasperReportsPdfView

xls JasperReportsXlsView

So in the example above a request to URI /foo/myReport.pdf would be mapped to the JasperReportsPdfView class. You can override the mapping key to view class mappings using the formatMappings property of JasperReportsMultiFormatView.

#### 13.7.3. Populating the ModelAndView

In order to render your report correctly in the format you have chosen, you must supply Spring with all of the data needed to populate your report. For JasperReports this means you must pass in all report parameters along with the report datasource. Report parameters are simple name/value pairs and can be added be to the Map for your model as you would add any name/value pair.

When adding the datasource to the model you have two approaches to choose from. The first approach is to add an instance of JRDataSource or Collection to the model Map under any arbitrary key. Spring will then locate this object in the model and treat it as the report datasource. For example, you may populate your model like this:

private Map getModel() {

Map model = new HashMap();

Collection beanData = getBeanData();

model.put("myBeanData", beanData);

return model;

}

The second approach is to add the instance of JRDataSource or Collection under a specific key and then configure this key using the reportDataKey property of the view class. In both cases Spring will instances of Collection in a JRBeanCollectionDataSource instance. For example:

private Map getModel() {

Map model = new HashMap();

Collection beanData = getBeanData();

Collection someData = getSomeData();

model.put("myBeanData", beanData);

model.put("someData", someData);

return model;

}

Here you can see that two Collection instances are being added to the model. To ensure that the correct one is used, we simply modify our view configuration as appropriate:

simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView

simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper

simpleReport.reportDataKey=myBeanData

Be aware that when using the first approach, Spring will use the first instance of JRDataSource or Collection that it encounters. If you need to place multiple instances of JRDataSource or Collection into the model then you need to use the second approach.

#### 13.7.4. Working with Sub-Reports

JasperReports provides support for embedded sub-reports within your master report files. There are a wide variety of mechanisms for including sub-reports in your report files. The easiest way is to hard code the report path and the SQL query for the sub report into your design files. The drawback of this approach is obvious - the values are hard-coded into your report files reducing reusability and making it harder to modify and update report designs. To overcome this you can configure sub-reports declaratively and you can include additional data for these sub-reports directly from your controllers.

##### 13.7.4.1. Configuring Sub-Report Files

To control which sub-report files are included in a master report using Spring, your report file must be configured to accept sub-reports from an external source. To do this you declare a parameter in your report file like this:

<parameter name="ProductsSubReport" class="net.sf.jasperreports.engine.JasperReport"/>

Then, you define your sub-report to use this sub-report parameter:

<subreport>

<reportElement isPrintRepeatedValues="false" x="5" y="25" width="325"

height="20" isRemoveLineWhenBlank="true" backcolor="#ffcc99"/>

<subreportParameter name="City">

<subreportParameterExpression><![CDATA[$F{city}]]></subreportParameterExpression>

</subreportParameter>

<dataSourceExpression><![CDATA[$P{SubReportData}]]></dataSourceExpression>

<subreportExpression class="net.sf.jasperreports.engine.JasperReport">

<![CDATA[$P{ProductsSubReport}]]></subreportExpression>

</subreport>

This defines a master report file that expects the sub-report to be passed in as an instance of net.sf.jasperreports.engine.JasperReports under the parameter ProductsSubReport. When configuring your Jasper view class, you can instruct Spring to load a report file and pass into the JasperReports engine as a sub-report using the subReportUrls property:

<property name="subReportUrls">

<map>

<entry key="ProductsSubReport">

<value>/WEB-INF/reports/subReportChild.jrxml</value>

</entry>

</map>

</property>

Here, the key of the Map corresponds to the name of the sub-report parameter in th report design file, and the entry is the URL of the report file. Spring will load this report file, compiling it if necessary, and will pass into the JasperReports engine under the given key.

##### 13.7.4.2. Configuring Sub-Report Data Sources

This step is entirely optional when using Spring configure your sub-reports. If you wish, you can still configure the data source for your sub-reports using static queries. However, if you want Spring to convert data returned in your ModelAndView into instances of JRDataSource then you need to specify which of the parameters in your ModelAndView Spring should convert. To do this configure the list of parameter names using the subReportDataKeys property of the your chosen view class:

<property name="subReportDataKeys">

<value>SubReportData</value>

</property>

Here, the key you supply MUST correspond to both the key used in your ModelAndView and the key used in your report design file.

#### 13.7.5. Configuring Exporter Parameters

If you have special requirements for exporter configuration - perhaps you want a specific page size for your PDF report, then you can configure these exporter parameters declaratively in your Spring configuration file using the exporterParameters property of the view class. The exporterParameters property is typed as Map and in your configuration the key of an entry should be the fully-qualified name of a static field that contains the exporter parameter definition and the value of an entry should be the value you want to assign to the parameter. An example of this is shown below:

<bean id="htmlReport"

class="org.springframework.web.servlet.view.jasperreports.JasperReportsHtmlView">

<property name="url">

<value>/WEB-INF/reports/simpleReport.jrxml</value>

</property>

<property name="exporterParameters">

<map>

<entry key="net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML\_FOOTER">

<value>Footer by Spring!

&lt;/td&gt;&lt;td width="50%"&gt;&amp;nbsp; &lt;/td&gt;&lt;/tr&gt;

&lt;/table&gt;&lt;/body&gt;&lt;/html&gt;

</value>

</entry>

</map>

</property>

</bean>

Here you can see that the JasperReportsHtmlView is being configured with an exporter parameter for net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML\_FOOTER which will output a footer in the resulting HTML.

## 14. Integrating with other web frameworks

### 14.1. Introduction

Spring can be easily integrated into any Java-based web framework. All you need to do is to declare the ContextLoaderListener in your web.xml and use a contextConfigLocation <context-param> to set which context files to load.

The <context-param>:

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/applicationContext\*.xml</param-value>

</context-param>

The <listener>:

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

NOTE: Listeners were added to the Servlet API in version 2.3. If you have a Servlet 2.2 container, you can use the ContextLoaderServlet to achieve this same functionality.

If you don't specify the contextConfigLocation context parameter, the ContextLoaderListener will look for a /WEB-INF/applicationContext.xml file to load. Once the context files are loaded, Spring creates a WebApplicationContext object based on the bean definitions and puts it into the ServletContext.

All Java web frameworks are built on top of the Servlet API, so you can use the following code to get the ApplicationContext that Spring created.

WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);

The WebApplicationContextUtils class is for convenience, so you don't have to remember the name of the ServletContext attribute. Its getWebApplicationContext() method will return null if an object doesn't exist under the WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE key. Rather than risk getting NullPointerExceptions in your application, it's better to use the getRequiredWebApplicationContext() method. This method throws an Exception when the ApplicationContext is missing.

Once you have a reference to the WebApplicationContext, you can retrieve beans by their name or type. Most developers retrieve beans by name, then cast them to one of their implemented interfaces.

Fortunately, most of the frameworks in this section have simpler ways of looking up beans. Not only do they make it easy to get beans from the BeanFactory, but they also allow you to use dependency injection on their controllers. Each framework section has more detail on its specific integration strategies.

### 14.2. JavaServer Faces

JavaServer Faces (JSF) is a component-based, event-driven web framework. According to Sun Microsystem's JSF Overview, JSF technology includes:

A set of APIs for representing UI components and managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility.

A JavaServer Pages (JSP) custom tag library for expressing a JavaServer Faces interface within a JSP page.

#### 14.2.1. DelegatingVariableResolver

The easiest way to integrate your Spring middle-tier with your JSF web layer is to use the DelegatingVariableResolver class. To configure this variable resolver in your application, you'll need to edit your faces-context.xml. After the opening <faces-config> element, add an <application> element and a <variable-resolver> element within it. The value of the variable resolver should reference Spring's DelegatingVariableResolver:

<faces-config>

<application>

<variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>

<locale-config>

<default-locale>en</default-locale>

<supported-locale>en</supported-locale>

<supported-locale>es</supported-locale>

</locale-config>

<message-bundle>messages</message-bundle>

</application>

By specifying Spring's variable resolver, you can configure Spring beans as managed properties of your managed beans. The DelegatingVariableResolver will first delegate value lookups to the default resolver of the underlying JSF implementation, and then to Spring's root WebApplicationContext. This allows you to easily inject dependencies into your JSF-managed beans.

Managed beans are defined in your faces-config.xml file. Below is an example where #{userManager} is a bean that's retrieved from Spring's BeanFactory.

<managed-bean>

<managed-bean-name>userList</managed-bean-name>

<managed-bean-class>com.whatever.jsf.UserList</managed-bean-class>

<managed-bean-scope>request</managed-bean-scope>

<managed-property>

<property-name>userManager</property-name>

<value>#{userManager}</value>

</managed-property>

</managed-bean>

The DelegatingVariableResolver is the recommended strategy for integrating JSF and Spring. If you're looking for more robust integration features, you might take a look at the JSF-Spring project.

#### 14.2.2. FacesContextUtils

A custom VariableResolver works well when mapping your properties to beans in faces-config.xml, but at times you may need to grab a bean explicitly. The FacesContextUtils class makes this easy. It's similar to WebApplicationContextUtils, except that it takes a FacesContext parameter rather than a ServletContext parameter.

ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());

### 14.3. Struts

Struts is the de facto web framework for Java applications, mainly because it was one of the first to be released (June 2001). Invented by Craig McClanahan, Struts is an open source project hosted by the Apache Software Foundation. At the time, it greatly simplified the JSP/Servlet programming paradigm and won over many developers who were using proprietary frameworks. It simplified the programming model, it was open source, and it had a large community, which allowed the project to grow and become popular among Java web developers.

To integrate your Struts application with Spring, you have two options:

Configure Spring to manage your Actions as beans, using the ContextLoaderPlugin, and set their dependencies in a Spring context file.

Subclass Spring's ActionSupport classes and grab your Spring-managed beans explicitly using a getWebApplicationContext() method.

#### 14.3.1. ContextLoaderPlugin

The ContextLoaderPlugin is a Struts 1.1+ plug-in that loads a Spring context file for the Struts ActionServlet. This context refers to the root WebApplicationContext (loaded by the ContextLoaderListener) as its parent. The default name of the context file is the name of the mapped servlet, plus -servlet.xml. If ActionServlet is defined in web.xml as <servlet-name>action</servlet-name>, the default is /WEB-INF/action-servlet.xml.

To configure this plug-in, add the following XML to the plug-ins section near the bottom of your struts-config.xml file:

<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>

The location of the context configuration files can be customized using the "contextConfigLocation" property.

<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">

<set-property property="contextConfigLocation"

value="/WEB-INF/action-servlet.xml.xml,

/WEB-INF/applicationContext.xml"/>

</plug-in>

It is possible to use this plugin to load all your context files, which can be useful when using testing tools like StrutsTestCase. StrutsTestCase's MockStrutsTestCase won't initialize Listeners on startup so putting all your context files in the plugin is a workaround. A bug has been filed for this issue.

After configuring this plug-in in struts-config.xml, you can configure your Action to be managed by Spring. Spring 1.1.3 provides two ways to do this:

Override Struts' default RequestProcessor with Spring's DelegatingRequestProcessor.

Use the DelegatingActionProxy class in the type attribute of your <action-mapping>.

Both of these methods allow you to manage your Actions and their dependencies in the action-context.xml file. The bridge between the Action in struts-config.xml and action-servlet.xml is built with the action-mapping's "path" and the bean's "name". If you have the following in your struts-config.xml file:

<action path="/users" .../>

You must define that Action's bean with the "/users" name in action-servlet.xml:

<bean name="/users" .../>

##### 14.3.1.1. DelegatingRequestProcessor

To configure the DelegatingRequestProcessor in your struts-config.xml file, override the "processorClass" property in the <controller> element. These lines follow the <action-mapping> element.

<controller>

<set-property property="processorClass"

value="org.springframework.web.struts.DelegatingRequestProcessor"/>

</controller>

After adding this setting, your Action will automatically be looked up in Spring's context file, no matter what the type. In fact, you don't even need to specify a type. Both of the following snippets will work:

<action path="/user" type="com.whatever.struts.UserAction"/>

<action path="/user" type="com.whatever.struts.UserAction"/>

If you're using Struts' modules feature, your bean names must contain the module prefix. For example, an action defined as <action path="/user"/> with module prefix "admin" requires a bean name with <bean name="/admin/user"/>.

NOTE: If you're using Tiles in your Struts application, you must configure your <controller> with the DelegatingTilesRequestProcessor.

##### 14.3.1.2. DelegatingActionProxy

If you have a custom RequestProcessor and can't use the DelegatingTilesRequestProcessor, you can use the DelegatingActionProxy as the type in your action-mapping.

<action path="/user" type="org.springframework.web.struts.DelegatingActionProxy"

name="userForm" scope="request" validate="false" parameter="method">

<forward name="list" path="/userList.jsp"/>

<forward name="edit" path="/userForm.jsp"/>

</action>

The bean definition in action-servlet.xml remains the same, whether you use a custom RequestProcessor or the DelegatingActionProxy.

Defining your Action in a context file enables you to use Spring's IoC features, as well as instantiate new Actions for ach request. To use this feature, add singleton="false" to your action's bean definition.

<bean name="/user" singleton="false" autowire="byName"

class="org.appfuse.web.UserAction"/>

#### 14.3.2. ActionSupport Classes

As previously mentioned, you can retrieve the WebApplicationContext from the ServletContext using the WebApplicationContextUtils class. An easier way is to extend Spring's Action classes for Struts. For example, instead of subclassing Struts' Action class, you can subclass Spring's ActionSupport class.

The ActionSupport class provides additional convenience methods, like getWebApplicationContext(). Below is an example of how you might use this in an Action:

public class UserAction extends DispatchActionSupport {

public ActionForward execute(ActionMapping mapping,

ActionForm form,

HttpServletRequest request,

HttpServletResponse response)

throws Exception {

if (log.isDebugEnabled()) {

log.debug("entering 'delete' method...");

}

WebApplicationContext ctx = getWebApplicationContext();

UserManager mgr = (UserManager) ctx.getBean("userManager");

// talk to manager for business logic

return mapping.findForward("success");

}

}

Spring includes subclasses for all of the standard Struts Actions - the Spring versions merely have Support appended to the name:

ActionSupport,

DispatchActionSupport,

LookupDispatchActionSupport and

MappingDispatchActionSupport.

The recommended strategy is to use the approach that best suits your project. Subclassing makes your code more readable, and you know exactly how your dependencies are resolved. However, using the ContextLoaderPlugin allow you to easily add new dependencies in your context XML file. Either way, Spring provides some nice options for integrating the two frameworks.

### 14.4. Tapestry

Tapestry is a powerful, component-oriented web application framework from Apache's Jakarta project (http://jakarta.apache.org/tapestry). While Spring has its own powerful web ui layer, there are a number of unique advantages to building a J2EE application using a combination of Tapestry for the web ui, and the Spring container for the lower layers. This document attempts to detail a few best practices for combining these two frameworks. It is expected that you are relatively familiar with both Tapestry and Spring Framework basics, so they will not be explained here. General introductory documentation for both Tapestry and Spring Framework are available on their respective web sites.

#### 14.4.1. Architecture

A typical layered J2EE application built with Tapestry and Spring will consist of a top UI layer built with Tapestry, and a number of lower layers, hosted out of one or more Spring Application Contexts.

User Interface Layer:

- concerned with the user interface

- contains some application logic

- provided by Tapestry

- aside from providing UI via Tapestry, code in this layer does its work via objects which implement interfaces from the Service Layer. The actual objects which implement these service layer interfaces are obtained from a Spring Application Context.

Service Layer:

- application specific 'service' code

- works with domain objects, and uses the Mapper API to get those domain objects into and out of some sort of repository (database)

- hosted in one or more Spring contexts

- code in this layer manipulates objects in the domain model, in an application specific fashion. It does its work via other code in this layer, and via the Mapper API. An object in this layer is given the specific mapper implementations it needs to work with, via the Spring context.

- since code in this layer is hosted in the Spring context, it may be transactionally wrapped by the Spring context, as opposed to managing its own transactions

Domain Model:

- domain specific object hierarchy, which deals with data and logic specific to this domain

- although the domain object hierarchy is built with the idea that it is persisted somehow and makes some general concessions to this (for example, bidirectional relationships), it generally has no knowledge of other layers. As such, it may be tested in isolation, and used with different mapping implementations for production vs. testing.

- these objects may be standalone, or used in conjunction with a Spring application context to take advantage of some of the benefits of the context, e.g., isolation, inversion of control, different strategy implementations, etc.

Data Source Layer:

- Mapper API (also called Data Access Objects): an API used to persist the domain model to a repository of some sort (generally a DB, but could be the filesystem, memory, etc.)

- Mapper API implementations: one or more specific implementations of the Mapper API, for example, a Hibernate-specific mapper, a JDO-specific mapper, JDBC-specific mapper, or a memory mapper.

- mapper implementations live in one or more Spring Application Contexts. A service layer object is given the mapper objects it needs to work with via the context.

Database, filesystem, or other repositories:

- objects in the domain model are stored into one or more repositories via one or more mapper implementations

- a repository may be very simple (e.g. filesystem), or may have its own representation of the data from the domain model (i.e. a schema in a db). It does not know about other layers howerver.

#### 14.4.2. Implementation

The only real question (which needs to be addressed by this document), is how Tapestry pages get access to service implementations, which are simply beans defined in an instance of the Spring Application Context.

##### 14.4.2.1. Sample application context

Assume we have the following simple Application Context definition, in xml form:

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

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-- ========================= GENERAL DEFINITIONS ========================= -->

<!-- ========================= PERSISTENCE DEFINITIONS ========================= -->

<!-- the DataSource -->

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName"><value>java:DefaultDS</value></property>

<property name="resourceRef"><value>false</value></property>

</bean>

<!-- define a Hibernate Session factory via a Spring LocalSessionFactoryBean -->

<bean id="hibSessionFactory"

class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

<property name="dataSource"><ref bean="dataSource"/></property>

</bean>

<!--

- Defines a transaction manager for usage in business or data access objects.

- No special treatment by the context, just a bean instance available as reference

- for business objects that want to handle transactions, e.g. via TransactionTemplate.

-->

<bean id="transactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager">

</bean>

<bean id="mapper"

class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">

<property name="sessionFactory"><ref bean="hibSessionFactory"/></property>

</bean>

<!-- ========================= BUSINESS DEFINITIONS ========================= -->

<!-- AuthenticationService, including tx interceptor -->

<bean id="authenticationServiceTarget"

class="com.whatever.services.service.user.AuthenticationServiceImpl">

<property name="mapper"><ref bean="mapper"/></property>

</bean>

<bean id="authenticationService"

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

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

<property name="target"><ref bean="authenticationServiceTarget"/></property>

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

<property name="transactionAttributes">

<props>

<prop key="\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

<!-- UserService, including tx interceptor -->

<bean id="userServiceTarget"

class="com.whatever.services.service.user.UserServiceImpl">

<property name="mapper"><ref bean="mapper"/></property>

</bean>

<bean id="userService"

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

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

<property name="target"><ref bean="userServiceTarget"/></property>

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

<property name="transactionAttributes">

<props>

<prop key="\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

</beans>

Inside the Tapestry application, we need to load this application context, and allow Tapestry pages to get the authenticationService and userService beans, which implement the AuthenticationService and UserService interfaces, respectively.

##### 14.4.2.2. Obtaining beans in Tapestry pages

At this point, the application context is available to a web application by calling Spring's static utility function WebApplicationContextUtils.getApplicationContext(servletContext), where servletContext is the standard ServletContext from the J2EE Servlet specification. As such, one simple mechanism for a page to get an instance of the UserService, for example, would be with code such as:

WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(

getRequestCycle().getRequestContext().getServlet().getServletContext());

UserService userService = (UserService) appContext.getBean("userService");

... some code which uses UserService

This mechanism does work. It can be made a lot less verbose by encapsulating most of the functionality in a method in the base class for the page or component. However, in some respects it goes against the Inversion of Control approach which Spring encourages, which is being used in other layers of this app, in that ideally you would like the page to not have to ask the context for a specific bean by name, and in fact, the page would ideally not know about the context at all.

Luckily, there is a mechanism to allow this. We rely upon the fact that Tapestry already has a mechanism to declaratively add properties to a page, and it is in fact the preferred approach to manage all properties on a page in this declarative fashion, so that Tapestry can properly manage their lifecycle as part of the page and component lifecycle.

##### 14.4.2.3. Exposing the application context to Tapestry

First we need to make the ApplicationContext available to the Tapestry page or Component without having to have the ServletContext; this is because at the stage in the page's/component's lifecycle when we need to access the ApplicationContext, the ServletContext won't be easily available to the page, so we can't use WebApplicationContextUtils.getApplicationContext(servletContext) directly. One way is by defining a custom version of the Tapestry IEngine which exposes this for us:

package com.whatever.web.xportal;

...

import ...

...

public class MyEngine extends org.apache.tapestry.engine.BaseEngine {

public static final String APPLICATION\_CONTEXT\_KEY = "appContext";

/\*\*

\* @see org.apache.tapestry.engine.AbstractEngine#setupForRequest(org.apache.tapestry.request.RequestContext)

\*/

protected void setupForRequest(RequestContext context) {

super.setupForRequest(context);

// insert ApplicationContext in global, if not there

Map global = (Map) getGlobal();

ApplicationContext ac = (ApplicationContext) global.get(APPLICATION\_CONTEXT\_KEY);

if (ac == null) {

ac = WebApplicationContextUtils.getWebApplicationContext(

context.getServlet().getServletContext()

);

global.put(APPLICATION\_CONTEXT\_KEY, ac);

}

}

}

This engine class places the Spring Application Context as an attribute called "appContext" in this Tapestry app's 'Global' object. Make sure to register the fact that this special IEngine instance should be used for this Tapestry application, with an entry in the Tapestry application definition file. For example:

file: xportal.application:

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

<!DOCTYPE application PUBLIC

"-//Apache Software Foundation//Tapestry Specification 3.0//EN"

"http://jakarta.apache.org/tapestry/dtd/Tapestry\_3\_0.dtd">

<application

name="Whatever xPortal"

engine-class="com.whatever.web.xportal.MyEngine">

</application>

##### 14.4.2.4. Component definition files

Now in our page or component definition file (\*.page or \*.jwc), we simply add property-specification elements to grab the beans we need out of the ApplicationContext, and create page or component properties for them. For example:

<property-specification name="userService"

type="com.whatever.services.service.user.UserService">

global.appContext.getBean("userService")

</property-specification>

<property-specification name="authenticationService"

type="com.whatever.services.service.user.AuthenticationService">

global.appContext.getBean("authenticationService")

</property-specification>

The OGNL expression inside the property-specification specifies the initial value for the property, as a bean obtained from the context. The entire page definition might look like this:

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

<!DOCTYPE page-specification PUBLIC

"-//Apache Software Foundation//Tapestry Specification 3.0//EN"

"http://jakarta.apache.org/tapestry/dtd/Tapestry\_3\_0.dtd">

<page-specification class="com.whatever.web.xportal.pages.Login">

<property-specification name="username" type="java.lang.String"/>

<property-specification name="password" type="java.lang.String"/>

<property-specification name="error" type="java.lang.String"/>

<property-specification name="callback" type="org.apache.tapestry.callback.ICallback" persistent="yes"/>

<property-specification name="userService"

type="com.whatever.services.service.user.UserService">

global.appContext.getBean("userService")

</property-specification>

<property-specification name="authenticationService"

type="com.whatever.services.service.user.AuthenticationService">

global.appContext.getBean("authenticationService")

</property-specification>

<bean name="delegate" class="com.whatever.web.xportal.PortalValidationDelegate"/>

<bean name="validator" class="org.apache.tapestry.valid.StringValidator" lifecycle="page">

<set-property name="required" expression="true"/>

<set-property name="clientScriptingEnabled" expression="true"/>

</bean>

<component id="inputUsername" type="ValidField">

<static-binding name="displayName" value="Username"/>

<binding name="value" expression="username"/>

<binding name="validator" expression="beans.validator"/>

</component>

<component id="inputPassword" type="ValidField">

<binding name="value" expression="password"/>

<binding name="validator" expression="beans.validator"/>

<static-binding name="displayName" value="Password"/>

<binding name="hidden" expression="true"/>

</component>

</page-specification>

##### 14.4.2.5. Adding abstract accessors

Now in the Java class definition for the page or component itself, all we need to do is add an abstract getter method for the properties we have defined, to access them. When the page or component is actually loaded by Tapestry, it performs runtime code instrumentation on the classfile to add the properties which have been defined, and hook up the abstract getter methods to the newly created fields. For example:

// our UserService implementation; will come from page definition

public abstract UserService getUserService();

// our AuthenticationService implementation; will come from page definition

public abstract AuthenticationService getAuthenticationService();

For completeness, the entire Java class, for a login page in this example, might look like this:

package com.whatever.web.xportal.pages;

/\*\*

\* Allows the user to login, by providing username and password.

\* After successfully logging in, a cookie is placed on the client browser

\* that provides the default username for future logins (the cookie

\* persists for a week).

\*/

public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

/\*\* the key under which the authenticated user object is stored in the visit as \*/

public static final String USER\_KEY = "user";

/\*\*

\* The name of a cookie to store on the user's machine that will identify

\* them next time they log in.

\*\*/

private static final String COOKIE\_NAME = Login.class.getName() + ".username";

private final static int ONE\_WEEK = 7 \* 24 \* 60 \* 60;

// --- attributes

public abstract String getUsername();

public abstract void setUsername(String username);

public abstract String getPassword();

public abstract void setPassword(String password);

public abstract ICallback getCallback();

public abstract void setCallback(ICallback value);

public abstract UserService getUserService();

public abstract AuthenticationService getAuthenticationService();

// --- methods

protected IValidationDelegate getValidationDelegate() {

return (IValidationDelegate) getBeans().getBean("delegate");

}

protected void setErrorField(String componentId, String message) {

IFormComponent field = (IFormComponent) getComponent(componentId);

IValidationDelegate delegate = getValidationDelegate();

delegate.setFormComponent(field);

delegate.record(new ValidatorException(message));

}

/\*\*

\* Attempts to login.

\*

\* <p>If the user name is not known, or the password is invalid, then an error

\* message is displayed.

\*

\*\*/

public void attemptLogin(IRequestCycle cycle) {

String password = getPassword();

// Do a little extra work to clear out the password.

setPassword(null);

IValidationDelegate delegate = getValidationDelegate();

delegate.setFormComponent((IFormComponent) getComponent("inputPassword"));

delegate.recordFieldInputValue(null);

// An error, from a validation field, may already have occurred.

if (delegate.getHasErrors())

return;

try {

User user = getAuthenticationService().login(getUsername(), getPassword());

loginUser(user, cycle);

}

catch (FailedLoginException ex) {

this.setError("Login failed: " + ex.getMessage());

return;

}

}

/\*\*

\* Sets up the {@link User} as the logged in user, creates

\* a cookie for their username (for subsequent logins),

\* and redirects to the appropriate page, or

\* a specified page).

\*

\*\*/

public void loginUser(User user, IRequestCycle cycle) {

String username = user.getUsername();

// Get the visit object; this will likely force the

// creation of the visit object and an HttpSession.

Map visit = (Map) getVisit();

visit.put(USER\_KEY, user);

// After logging in, go to the MyLibrary page, unless otherwise

// specified.

ICallback callback = getCallback();

if (callback == null)

cycle.activate("Home");

else

callback.performCallback(cycle);

// I've found that failing to set a maximum age and a path means that

// the browser (IE 5.0 anyway) quietly drops the cookie.

IEngine engine = getEngine();

Cookie cookie = new Cookie(COOKIE\_NAME, username);

cookie.setPath(engine.getServletPath());

cookie.setMaxAge(ONE\_WEEK);

// Record the user's username in a cookie

cycle.getRequestContext().addCookie(cookie);

engine.forgetPage(getPageName());

}

public void pageBeginRender(PageEvent event) {

if (getUsername() == null)

setUsername(getRequestCycle().getRequestContext().getCookieValue(COOKIE\_NAME));

}

}

#### 14.4.3. Summary

In this example, we've managed to allow service beans defined in the Spring ApplicationContext to be provided to the page in a declarative fashion. The page class does not know where the service implementations are coming from, and in fact it is easy to slip in another implementation, for example, during testing. This inversion of control is one of the prime goals and benefits of the Spring Framework, and we have managed to extend it all the way up the J2EE stack in this Tapestry application.

### 14.5. WebWork

WebWork is a web framework designed with simplicity in mind. It's built on top of XWork, which is a generic command framework. XWork also has an IoC container, but it isn't as full-featured as Spring and won't be covered in this section. WebWork controllers are called Actions, mainly because they must implement the Action interface. The ActionSupport class implements this interface, and it is most common parent class for WebWork actions.

WebWork maintains its own Spring integration project, located on java.net in the xwork-optional project. Currently, three options are available for integrating WebWork with Spring:

SpringObjectFactory: override XWork's default ObjectFactory so XWork will look for Spring beans in the root WebApplicationContext.

ActionAutowiringInterceptor: use an interceptor to automatically wire an Action's dependencies as they're created.

SpringExternalReferenceResolver: look up Spring beans based on the name defined in an <external-ref> element of an <action> element.

All of these strategies are explained in further detail in WebWork's Documentation.

## 15. JMS

### 15.1. Introduction

Spring provides a JMS abstraction framework that simplifies the use of the JMS API and shields the user from differences between the JMS 1.0.2 and 1.1 APIs.

JMS can be roughly divided into two areas of functionality, production and consumption of messages. In a J2EE environment, the ability to consume messages asynchronously is provided for by message-driven beans while in a standalone application this is provided for by the creation of MessageListeners or ConnectionConsumers. The functionality in JmsTemplate is focused on producing messages. Future releases of Spring will address asynchronous message consumption in a standalone environment.

The package org.springframework.jms.core provides the core functionality for using JMS. It contains JMS template classes that simplifies the use of the JMS by handling the creation and release of resources, much like the JdbcTemplate does for JDBC. The design principal common to Spring template classes is to provide helper methods to perform common operations and for more sophisticated usage, delegate the essence of the processing task to user implemented callback interfaces. The JMS template follows the same design. The classes offer various convenience methods for the sending of messages, consuming a message synchronously, and exposing the JMS session and message producer to the user.

The package org.springframework.jms.support provides JMSException translation functionality. The translation converts the checked JMSException hierarchy to a mirrored hierarchy of unchecked exceptions. If there are any provider specific subclasses of the checked javax.jms.JMSException, this exception is wrapped in the unchecked UncategorizedJmsException. The package org.springframework.jms.support.converter provides a MessageConverter abstraction to convert between Java objects and JMS messages. The package org.springframework.jms.support.destination provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI.

Finally, the package org.springframework.jms.connection provides an implementation of the ConnectionFactory suitable for use in standalone applications. It also contains an implementation of Spring's PlatformTransactionManager for JMS. This allows for integration of JMS as a transactional resource into Spring's transaction management mechanisms.

### 15.2. Domain unification

There are two major releases of the JMS specification, 1.0.2 and 1.1. JMS 1.0.2 defined two types of messaging domains, point-to-point (Queues) and publish/subscribe (Topics). The 1.0.2 API reflected these two messaging domains by providing a parallel class hierarchy for each domain. Consequentially, a client application was domain specific in the use of the JMS API. JMS 1.1 introduced the concept of domain unification that minimized both the functional differences and client API differences between the two domains. As an example of a functional difference that was removed, if you use a JMS 1.1 provider you can transactionally consume a message from one domain and produce a message on the other using the same Session.

The JMS 1.1 specification was released in April 2002 and incorporated as part of J2EE 1.4 in November 2003. As a result, most application servers that are currently in use are only required to support JMS 1.0.2.

### 15.3. JmsTemplate

Two implementations of the JmsTemplate are provided. The class JmsTemplate uses the JMS 1.1 API and the subclass JmsTemplate102 uses the JMS 1.0.2 API.

Code that uses the JmsTemplate only needs to implement callback interfaces giving them a clearly defined contract. The MessageCreator callback interface creates a message given a Session provided by the calling code in JmsTemplate. In order to allow for more complex usage of the JMS API, the callback SessionCallback provides the user with the JMS session and the callback ProducerCallback exposes a Session and MessageProducer pair.

The JMS API exposes two types of send methods, one that takes delivery mode, priority, and time-to-live as quality of service (QOS) parameters and one that takes no QOS parameters which uses default values. Since there are many send methods in JmsTemplate, the setting of the QOS parameters have been exposed as bean properties to avoid duplication in the number of send methods. Similarly, the timeout value for synchronous receive calls is set using the property setReceiveTimeout.

Some JMS providers allow the setting of default QOS values administratively through the configuration of the ConnectionFactory. This has the effect that a call to MessageProducer's send method send(Destination destination, Message message) will use QOS different default values than those specified in the JMS specification. Therefore, in order to provide consistent management of QOS values, the JmsTemplate must be specifically enabled to use its own QOS values by setting the boolean property isExplicitQosEnabled to true.

#### 15.3.1. ConnectionFactory

The JmsTemplate requires a reference to a ConnectionFactory. The ConnectionFactory is part of the JMS specification and serves as the entry point for working with JMS. It is used by the client application as a factory to create connections with the JMS provider and encapsulates various configuration parameters, many of which are vendor specific such as SSL configuration options.

When using JMS inside an EJB the vendor provides implementations the JMS interfaces so that they can participate in declarative transaction management and perform pooling of connections and session. In order to use this implementation, J2EE containers typically require that you declare a JMS connection factory as a resource-ref inside the EJB or servlet deployment descriptors. To ensure the use of these features with the JmsTemplate inside an EJB, the client application should ensure that it references the managed implementation of the ConnectionFactory.

Spring provides an implementation of the ConnectionFactory interface, SingleConnectionFactory, that will return the same Connection on all createConnection calls and ignore calls to close. This is useful for testing and standalone environments so that the same connection can be used for multiple JmsTemplate calls that may span any number of transactions. SingleConnectionFactory takes a reference to a standard ConnectionFactory that would typically comes from JNDI.

#### 15.3.2. Transaction Management

Spring provides a JmsTransactionManager that manages transactions for a single JMS ConnectionFactory. This allows JMS applications to leverage the managed transaction features of Spring as described in Chapter 7. The JmsTransactionManager binds a Connection/Session pair from the specified ConnectionFactory to the thread. However, in a J2EE environment the ConnectionFactory will pool connections and sessions, so the instances that are bound to the thread depend on the pooling behavior. In a standalone environment, using Spring's SingleConnectionFactory will result in a using a single JMS Connection and each transaction having its own Session. The JmsTemplate can also be used with the JtaTransactionManager and an XA-capable JMS ConnectionFactory for performing distributed transactions.

Reusing code across a managed and unmanaged transactional environment can be confusing when using JMS API to create a Session from a Connection. This is because the JMS API only has only one factory method to create a Session and it requires values for the transaction and acknowledgement modes. In a managed environment, setting these values in the responsibility of the environments transactional infrastructure, so these values are ignored by the vendor's wrapper to the JMS Connection. When using the JmsTemplate in an unmanaged environment you can specify these values though the use of the properties SessionTransacted and SessionAcknowledgeMode. When using a PlatformTransactionManager with JmsTemplate, the template will always be given a transactional JMS Session.

#### 15.3.3. Destination Management

Destinations, like ConnectionFactories, are JMS administered objects that can be stored and retrieved in JNDI. When configuring a Spring application context one can use the JNDI factory class JndiObjectFactoryBean to perform dependency injection on your object's references to JMS destinations. However, often this strategy is cumbersome if there are a large number of destinations in the application or if there are advanced destination management features unique to the JMS provider. Examples of such advanced destination management would be the creation of dynamic destinations or support for a hierarchical namespace of destinations. The JmsTemplate delegates the resolution of a destination name to a JMS destination object to an implementation of the interface DestinationResolver. DynamicDestinationResolver is the default implementation used by JmsTemplate and accommodates resolving dynamic destinations. A JndiDestinationResolver is also provided that acts as a service locator for destinations contained in JNDI and optionally falls back to the behavior contained in DynamicDestinationResolver.

Quite often the destinations used in a JMS application are only known at runtime and therefore can not be administratively created when the application is deployed. This is often because there is shared application logic between interacting system components that create destinations at runtime according to a well known naming convention. Even though the creation of dynamic destinations are not part of the JMS specification, most vendors have provided this functionality. Dynamic destinations are created with a name defined by the user which differentiates them from temporary destinations and are often not registered in JNDI. The API used to create dynamic destinations varies from provider to provider since the properties associated with the destination are vendor specific. However, a simple implementation choice that is sometimes made by vendors is to disregard the warnings in the JMS specification and to use the TopicSession method createTopic(String topicName) or the QueueSession method createQueue(String queueName) to create a new destination with default destination properties. Depending on the vendor implementation, DynamicDestinationResolver may then also create a physical destination instead of only resolving one.

The boolean property PubSubDomain is used to configure the JmsTemplate with knowledge of what JMS domain is being used. By default the value of this property is false, indicating that the point-to-point domain, Queues, will be used. In the 1.0.2 implementation the value of this property determines if the JmsTemplate's send operations will send a message to a Queue or to a Topic. This flag has no effect on send operations for the 1.1 implementation. However, in both implementations, this property determines the behavior of resolving dynamic destination via implementations of DestinationResolver.

You can also configure the JmsTemplate with a default destination via the property DefaultDestination. The default destination will be used with send and receive operations that do not refer to a specific destination.

### 15.4. Using the JmsTemplate

To get started using the JmsTemplate you need to select either the JMS 1.0.2 implementation JmsTemplate102 or the JMS 1.1 implementation JmsTemplate. Check your JMS provider to determine what version is supported.

#### 15.4.1. Sending a message

The JmsTemplate contains many convenience methods to send a message. There are send methods that specify the destination using a javax.jms.Destination object and those that specify the destination using a string for use in a JNDI lookup. The send method that takes no destination argument uses the default destination. Here is an example that sends a message to a queue using the 1.0.2 implementation.

import javax.jms.ConnectionFactory;

import javax.jms.JMSException;

import javax.jms.Message;

import javax.jms.Queue;

import javax.jms.Session;

import org.springframework.jms.core.JmsTemplate;

import org.springframework.jms.core.JmsTemplate102;

import org.springframework.jms.core.MessageCreator;

public class JmsQueueSender {

private JmsTemplate jt;

private ConnectionFactory connFactory;

private Queue queue;

public void simpleSend() {

jt = new JmsTemplate102(connFactory, false);

jt.send(queue, new MessageCreator() {

public Message createMessage(Session session) throws JMSException {

return session.createTextMessage("hello queue world");

}

});

}

public void setConnectionFactory(ConnectionFactory cf) {

connFactory = cf;

}

public void setQueue(Queue q) {

queue = q;

}

}

This example uses the MessageCreator callback to create a text message from the supplied Session object and the JmsTemplate is constructed by passing a reference to a ConnectionFactory and a boolean specifying the messaging domain. A zero argument constructor and a setConnectionFactory/Queue method are also provided and can be used for constructing the instance using a BeanFactory. The method simpleSend modified to send to a topic instead of a queue is shown below

public void simpleSend() {

jt = new JmsTemplate102(connFactory, true);

jt.send(topic, new MessageCreator() {

public Message createMessage(Session session) throws JMSException {

return session.createTextMessage("hello topic world");

}

});

}

When configuring the 1.0.2 in an application context it is important to remember setting the value of the boolean property PubSubDomain property in order to indicate if you want to send to Queues or Topics.

The method send(String destinationName, MessageCreator c) lets you send to a message using the string name of the destination. If these names are registered in JNDI, you should set the DestinationResolver property of the template to an instance of JndiDestinationResolver.

If you created the JmsTemplate and specified a default destination, the send(MessageCreator c) sends a message to that destination.

#### 15.4.2. Synchronous Receiving

While JMS is typically associated with asynchronous processing, it is possible to consume messages synchronously. The overloaded receive methods provide this functionality. During a synchronous receive the calling thread blocks until a message becomes available. This can be a dangerous operation since the calling thread can potentially be blocked indefinitely. The property receiveTimeout specifies how long the receiver should wait before giving up waiting for a message.

#### 15.4.3. Using Message Converters

In order to facilitate the sending of domain model objects the JmsTemplate has various send methods that take a Java object as an argument for a message's data content. The overloaded methods convertAndSend and receiveAndConvert in JmsTemplate delegate the conversion process to an instance of the MessageConverter interface. This interface defines a simple contract to convert between Java objects and JMS messages. The default implementation, SimpleMessageConverter supports conversion between String and TextMessage, byte[] and BytesMesssage, and java.util.Map and MapMessage. By using the converter, you your application code can focus on the business object that is being sent or received via JMS and not bother with the details of how it is represented as a JMS message.

The sandbox currently includes a MapMessageConverter which uses reflection to convert between a JavaBean and a MapMessage. Other popular implementations choices you might implement yourself are Converters that bust an existing XML marshalling packages, such as JAXB, Castor, XMLBeans, or XStream, to create a TextMessage representing the object.

To accommodate the setting of a message's properties, headers, and body that can not be generically encapsulated inside a converter class, the interface MessagePostProcessor gives you access to the message after it has been converted, but before it is sent. The example below shows how to modify a message header and a property after a java.util.Map is converted to a message.

public void sendWithConversion() {

Map m = new HashMap();

m.put("Name", "Mark");

m.put("Age", new Integer(35));

jt.convertAndSend("testQueue", m, new MessagePostProcessor() {

public Message postProcessMessage(Message message)

throws JMSException {

message.setIntProperty("AccountID", 1234);

message.setJMSCorrelationID("123-00001");

return message;

}

});

}

This results in a message of the form

MapMessage={

Header={

... standard headers ...

CorrelationID={123-00001}

}

Properties={

AccountID={Integer:1234}

}

Fields={

Name={String:Mark}

Age={Integer:35}

}

}

#### 15.4.4. SessionCallback and ProducerCallback

While the send operations cover many common usage scenarios, there are cases when you want to perform multiple operations on a JMS Session or MessageProducer. The SessionCallback and ProducerCallback expose the JMS Session and Session/MessageProducer pair respectfully. The execute() methods on JmsTemplate execute these callback methods.

## 16. Accessing and implementing EJBs

As a lightweight container, Spring is often considered an EJB replacement. We do believe that for many if not most applications and use cases, Spring as a container, combined with its rich supporting functionality in the area of transactions, ORM and JDBC access, is a better choice than implementing equivalent functionality via an EJB container and EJBs.

However, it is important to note that using Spring does not prevent you from using EJBs. In fact, Spring makes it much easier to access EJBs and implement EJBs and functionality within them. Additionally, using Spring to access services provided by EJBs allows the implementation of those services to later transparently be switched between local EJB, remote EJB, or POJO (plain java object) variants, without the client code client code having to be changed.

In this chapter, we look at how Spring can help you access and implement EJBs. Spring provides particular value when accessing stateless session beans (SLSBs), so we'll begin by discussing this.

### 16.1. Accessing EJBs

#### 16.1.1. Concepts

To invoke a method on a local or remote stateless session bean, client code must normally perform a JNDI lookup to obtain the (local or remote) EJB Home object, then use a 'create' method call on that object to obtain the actual (local or remote) EJB object. One or more methods are then invoked on the EJB.

To avoid repeated low-level code, many EJB applications use the Service Locator and Business Delegate patterns. These are better than spraying JNDI lookups throughout client code, but their usual implementations have significant disadvantages. For example:

Typically code using EJBs depends on Service Locator or Business Delegate singletons, making it hard to test

In the case of the Service Locator pattern used without a Business Delegate, application code still ends up having to invoke the create() method on an EJB home, and deal with the resulting exceptions. Thus it remains tied to the EJB API and the complexity of the EJB programming model.

Implementing the Business Delegate pattern typically results in significant code duplication, where we have to write numerous methods that simply call the same method on the EJB.

The Spring approach is to allow the creation and use of proxy objects, normally configured inside a Spring ApplicationContext or BeanFactory, which act as code-less business delegates. You do not need to write another Service Locator, another JNDI lookup, or duplicate methods in a hand-coded Business Delegate unless you’re adding real value.

#### 16.1.2. Accessing local SLSBs

Assume that we have a web controller that needs to use a local EJB. We’ll follow best practice and use the EJB Business Methods Interface pattern, so that the EJB’s local interface extends a non EJB-specific business methods interface. Let’s call this business methods interface MyComponent.

public interface MyComponent {

...

}

(One of the main reasons to the Business Methods Interface pattern is to ensure that synchronization between method signatures in local interface and bean implementation class is automatic. Another reason is that it later makes it much easier for us to switch to a POJO (plain java object) implementation of the service if it makes sense to do so) Of course we’ll also need to implement the local home interface and provide a bean implementation class that implements SessionBean and the MyComponent business methods interface. Now the only Java coding we’ll need to do to hook up our web tier controller to the EJB implementation is to expose a setter method of type MyComponent on the controller. This will save the reference as an instance variable in the controller:

private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {

this.myComponent = myComponent;

}

We can subsequently use this instance variable in any business method in the controller. Now assuming we are obtaining our controller object out of a Spring ApplicationContext or BeanFactory, we can in the same context configure a LocalStatelessSessionProxyFactoryBean instance, which will be EJB proxy object. The configuration of the proxy, and setting of the myComponent property of the controller is done with a configuration entry such as:

<bean id="myComponent"

class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">

<property name="jndiName"><value>myComponent</value></property>

<property name="businessInterface"><value>com.mycom.MyComponent</value></property>

</bean>

<bean id="myController" class = "com.mycom.myController">

<property name="myComponent"><ref bean="myComponent"/></property>

</bean>

There’s a lot of magic happening behind the scenes, courtesy of the Spring AOP framework, although you aren’t forced to work with AOP concepts to enjoy the results. The myComponent bean definition creates a proxy for the EJB, which implements the business method interface. The EJB local home is cached on startup, so there’s only a single JNDI lookup. Each time the EJB is invoked, the proxy invokes the create() method on the local EJB and invokes the corresponding business method on the EJB.

The myController bean definition sets the myController property of the controller class to this proxy.

This EJB access mechanism delivers huge simplification of application code: The web tier code (or other EJB client code) has no dependence on the use of EJB. If we want to replace this EJB reference with a POJO or a mock object or other test stub, we could simply change the myComponent bean definition without changing a line of Java code. Additionally, we haven’t had to write a single line of JNDI lookup or other EJB plumbing code as part of our application.

Benchmarks and experience in real applications indicate that the performance overhead of this approach (which involves reflective invocation of the target EJB) is minimal, and undetectable in typical use. Remember that we don’t want to make fine-grained calls to EJBs anyway, as there’s a cost associated with the EJB infrastructure in the application server.

There is one caveat with regards to the JNDI lookup. In a bean container, this class is normally best used as a singleton (there simply is no reason to make it a prototype). However, if that bean container pre-instantiates singletons (as do the XML ApplicationContext variants) you may have a problem if the bean container is loaded before the EJB container loads the target EJB. That is because the JNDI lookup will be performed in the init method of this class and cached, but the EJB will not have been bound at the target location yet. The solution is to not pre-instantiate this factory object, but allow it to be created on first use. In the XML containers, this is controlled via the lazy-init attribute.

Although this will not be of interest to the majority of Spring users, those doing programmatic AOP work with EJBs may want to look at LocalSlsbInvokerInterceptor.

#### 16.1.3. Accessing remote SLSBs

Accessing remote EJBs is essentially identical to accessing local EJBs, except that the SimpleRemoteStatelessSessionProxyFactoryBean is used. Of course, with or without Spring, remote invocation semantics apply; a call to a method on an object in another VM in another computer does sometimes have to be treated differently in terms of usage scenarios and failure handling.

Spring's EJB client support adds one more advantage over the non-Spring approach. Normally it is problematic for EJB client code to be easily switched back and forth between calling EJBs locally or remotely. This is because the remote interface methods must declare that they throw RemoteException, and client code must deal with this, while the local interface methods don't. Client code written for local EJBs which needs to be moved to remote EJBs typically has to be modified to add handling for the remote exceptions, and client code written for remote EJBs which needs to be moved to local EJBs, can either stay the same but do a lot of unnecessary handling of remote exceptions, or needs to be modified to remove that code. With the Spring remote EJB proxy, you can instead not declare any thrown RemoteException in your Business Method Interface and implementing EJB code, have a remote interface which is identical except that it does throw RemoteException, and rely on the proxy to dynamically treat the two interfaces as if they were the same. That is, client code does not have to deal with the checked RemoteException class. Any actual RemoteException that is thrown during the EJB invocation will be re-thrown as the non-checked RemoteAccessException class, which is a subclass of RuntimeException. The target service can then be switched at will between a local EJB or remote EJB (or even plain Java object) implementation, without the client code knowing or caring. Of course, this is optional; there is nothing stopping you from declaring RemoteExceptions in your business interface.

### 16.2. Using Spring convenience EJB implementation classes

Spring also provides convenience classes to help you implement EJBs. These are designed to encourage the good practice of putting business logic behind EJBs in POJOs, leaving EJBs responsible for transaction demarcation and (optionally) remoting.

To implement a Stateless or Stateful session bean, or Message Driven bean, you derive your implementation class from AbstractStatelessSessionBean, AbstractStatefulSessionBean, and AbstractMessageDrivenBean/AbstractJmsMessageDrivenBean, respectively.

Consider an example Stateless Session bean which actually delegates the implementation to a plain java service object. We have the business interface:

public interface MyComponent {

public void myMethod(...);

...

}

We have the plain java implementation object:

public class MyComponentImpl implements MyComponent {

public String myMethod(...) {

...

}

...

}

And finally the Stateless Session Bean itself:

public class MyComponentEJB extends AbstractStatelessSessionBean

implements MyComponent {

MyComponent \_myComp;

/\*\*

\* Obtain our POJO service object from the BeanFactory/ApplicationContext

\* @see org.springframework.ejb.support.AbstractStatelessSessionBean#onEjbCreate()

\*/

protected void onEjbCreate() throws CreateException {

\_myComp = (MyComponent) getBeanFactory().getBean(

ServicesConstants.CONTEXT\_MYCOMP\_ID);

}

// for business method, delegate to POJO service impl.

public String myMethod(...) {

return \_myComp.myMethod(...);

}

...

}

The Spring EJB support base classes will by default create and load a BeanFactory (or in this case, its ApplicationContext subclass) as part of their lifecycle, which is then available to the EJB (for example, as used in the code above to obtain the POJO service object). The loading is done via a strategy object which is a subclass of BeanFactoryLocator. The actual implementation of BeanFactoryLocator used by default is ContextJndiBeanFactoryLocator, which creates the ApplicationContext from a resource locations specified as a JNDI environment variable (in the case of the EJB classes, at java:comp/env/ejb/BeanFactoryPath). If there is a need to change the BeanFactory/ApplicationContext loading strategy, the default BeanFactoryLocator implementation used may be overridden by calling the setBeanFactoryLocator() method, either in setSessionContext(), or in the actual constructor of the EJB. Please see the JavaDocs for more details.

As described in the JavaDocs, Stateful Session beans expecting to be passivated and reactivated as part of their lifecycle, and which use a non-serializable BeanFactory/ApplicationContext instance (which is the normal case) will have to manually call unloadBeanFactory() and loadBeanFactory from ejbPassivate and ejbActivate, respectively, to unload and reload the BeanFactory on passivation and activation, since it can not be saved by the EJB container.

The default usage of ContextJndiBeanFactoryLocator to load an ApplicationContext for the use of the EJB is adequate for some situations. However, it is problematic when the ApplicationContext is loading a number of beans, or the initialization of those beans is time consuming or memory intensive (such as a Hibernate SessionFactory initialization, for example), since every EJB will have their own copy. In this case, the user may want to override the default ContextJndiBeanFactoryLocator usage and use another BeanFactoryLocator variant, such as ContextSingletonBeanFactoryLocatore, which can load and use a shared BeanFactory or ApplicationContext to be used by multiple EJBs or other clients. Doing this is relatively simple, by adding code similar to this to the EJB:

/\*\*

\* Override default BeanFactoryLocator implementation

\*

\* @see javax.ejb.SessionBean#setSessionContext(javax.ejb.SessionContext)

\*/

public void setSessionContext(SessionContext sessionContext) {

super.setSessionContext(sessionContext);

setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());

setBeanFactoryLocatorKey(ServicesConstants.PRIMARY\_CONTEXT\_ID);

}

Please see the respective JavaDocs for BeanFactoryLocator and ContextSingletonBeanFactoryLocatore for more information on their usage.

## 17. Remoting and web services using Spring

### 17.1. Introduction

Spring features integration classes for remoting support using various technologies. The remoting support eases the development of remote-enabled services, implemented by your usual (Spring) POJOs. Currently, Spring supports four remoting technologies:

Remote Method Invocation (RMI). Through the use of the RmiProxyFactoryBean and the RmiServiceExporter Spring supports both traditional RMI (with java.rmi.Remote interfaces and java.rmi.RemoteException) and transparent remoting via RMI invokers (with any Java interface).

Spring's HTTP invoker. Spring provides a special remoting strategy which allows for Java serialization via HTTP, supporting any Java interface (just like the RMI invoker). The corresponding support classes are HttpInvokerProxyFactoryBean and HttpInvokerServiceExporter.

Hessian. By using the HessianProxyFactoryBean and the HessianServiceExporter you can transparently expose your services using the lightweight binary HTTP-based protocol provided by Caucho.

Burlap. Burlap is Caucho's XML-based alternative for Hessian. Spring provides support classes such as BurlapProxyFactoryBean and BurlapServiceExporter.

JAX RPC. Spring provides remoting support for Web Services via JAX-RPC.

JMS (TODO).

While discussing the remoting capabilities of Spring, we'll use the following domain model and corresponding services:

// Account domain object

public class Account implements Serializable{

private String name;

public String getName();

public void setName(String name) {

this.name = name;

}

}

// Account service

public interface AccountService {

public void insertAccount(Account acc);

public List getAccounts(String name);

}

// Remote Account service

public interface RemoteAccountService extends Remote {

public void insertAccount(Account acc) throws RemoteException;

public List getAccounts(String name) throws RemoteException;

}

// ... and corresponding implement doing nothing at the moment

public class AccountServiceImpl implements AccountService {

public void insertAccount(Account acc) {

// do something

}

public List getAccounts(String name) {

// do something

}

}

We will start exposing the service to a remote client by using RMI and talk a bit about the drawbacks of using RMI. We'll then continue to show an example for Hessian.

### 17.2. Exposing services using RMI

Using Spring's support for RMI, you can transparently expose your services through the RMI infrastructure. After having this set up, you basically have a configuration similar to remote EJBs, except for the fact that there is no standard support for security context propagation or remote transaction propagation. Spring does provide hooks for such additional invocation context when using the RMI invoker, so you can for example plug in security frameworks or custom security credentials here.

#### 17.2.1. Exporting the service using the RmiServiceExporter

Using the RmiServiceExporter, we can expose the interface of our AccountService object as RMI object. The interface can be accessed by using RmiProxyFactoryBean, or via plain RMI in case of a traditional RMI service. The RmiServiceExporter explicitly supports the exposing of any non-RMI services via RMI invokers.

Of course, we first have to set up our service in the Spring BeanFactory:

<bean id="accountService" class="example.AccountServiceImpl">

<!-- any additional properties, maybe a DAO? -->

</bean>

Next we'll have to expose our service using the RmiServiceExporter:

<bean class="org.springframework.remoting.rmi.RmiServiceExporter">

<!-- does not necessarily have to be the same name as the bean to be exported -->

<property name="serviceName"><value>AccountService</value></property>

<property name="service"><ref bean="accountService"/></property>

<property name="serviceInterface"><value>example.AccountService</value></property>

<!-- defaults to 1099 -->

<property name="registryPort"><value>1199</value></property>

</bean>

As you can see, we're overriding the port for the RMI registry. Often, your application server also maintains an RMI registry and it is wise to not interfere with that one. Furthermore, the service name is used to bind the service under. So right now, the service will be bound at rmi://HOST:1199/AccountService. We'll use the URL later on to link in the service at the client side.

Note: We've left out one property, i.e. the servicePort property, which is 0 by default. This means an anonymous port will be used to communicate with the service. You can specify a different port if you like.

#### 17.2.2. Linking in the service at the client

Our client is a simple object using the AccountService to manage accounts:

public class SimpleObject {

private AccountService accountService;

public void setAccountService(AccountService accountService) {

this.accountService = accountService;

}

}

To link in the service on the client, we'll create a separate bean factory, containing the simple object and the service linking configuration bits:

<bean class="example.SimpleObject">

<property name="accountService"><ref bean="accountService"/></bean>

</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">

<property name="serviceUrl"><value>rmi://HOST:1199/AccountService</value></property>

<property name="serviceInterface"><value>example.AccountService</value></property>

</bean>

That's all we need to do to support the remote account service on the client. Spring will transparently create an invoker and remotely enable the account service through the RmiServiceExporter. At the client we're linking it in using the RmiProxyFactoryBean.

### 17.3. Using Hessian or Burlap to remotely call services via HTTP

Hessian offers a binary HTTP-based remoting protocol. It's created by Caucho and more information about Hessian itself can be found at http://www.caucho.com.

#### 17.3.1. Wiring up the DispatcherServlet for Hessian

Hessian communicates via HTTP and does so using a custom servlet. Using Spring's DispatcherServlet principles, you can easily wire up such a servlet exposing your services. First we'll have to create a new servlet in your application (this an excerpt from web.xml):

<servlet>

<servlet-name>remote</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

You're probably familiar with Spring's DispatcherServlet principles and if so, you know that know you'll have to create an application context named remote-servlet.xml (after the name of your servlet) in the WEB-INF directory. The application context will be used in the next section.

#### 17.3.2. Exposing your beans by using the HessianServiceExporter

In the newly created application context called remote-servlet.xml we'll create a HessianServiceExporter exporting your services:

<bean id="accountService" class="example.AccountServiceImpl">

<!-- any additional properties, maybe a DAO? -->

</bean>

<bean name="/AccountService" class="org.springframework.remoting.caucho.HessianServiceExporter">

<property name="service"><ref bean="accountService"/></property>

<property name="serviceInterface">

<value>example.AccountService</value>

</property>

</bean>

Now we're ready to link in the service at the client. No handler mapping is specified mapping requests (urls) onto services and that's why the BeanNameUrlHandlerMapping will be used, hence the service will be exported at the URL http://HOST:8080/AccountService.

#### 17.3.3. Linking in the service on the client

Using the HessianProxyFactoryBean we can link in the service at the client. The same principles apply as with the RMI example. We'll create a separate bean factory or application context and mention the following beans where the SimpleObject is using the AccountService to manage accounts:

<bean class="example.SimpleObject">

<property name="accountService"><ref bean="accountService"/></property>

</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">

<property name="serviceUrl"><value>http://remotehost:8080/AccountService</value></property>

<property name="ServiceInterface"><value>example.AccountService</value></property>

</bean>

That's all there is to it.

#### 17.3.4. Using Burlap

We won't discuss Burlap, the XML-based equivalent of Hessian, in detail here, since it is configured and set up in exactly the same way as the Hessian variant explained above. Just replace the word Hessian with Burlap and you're all set to go.

#### 17.3.5. Applying HTTP basic authentication to a service exposed through Hessian or Burlap

One of the advantages of Hessian and Burlap is that we can easily apply HTTP basic authentication, because both protocols are HTTP-based. Your normal HTTP server security mechanism can easily be applied through using the web.xml security features, for example. Usually, you don't use per-user security credentials here, but rather shared credentials defined at the Hessian/BurlapProxyFactoryBean level (similar to a JDBC DataSource).

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">

<property name="interceptors">

<list>

<ref bean="authorizationInterceptor"/>

</list>

</property>

</bean>

<bean id="authorizationInterceptor"

class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">

<property name="authorizedRoles">

<list>

<value>administrator</value>

<value>operator</value>

</list>

</property>

</bean>

This an example where we explicitly mention the BeanNameUrlHandlerMapping and set an interceptor allowing only administrators and operators to call the beans mentioned in this application context.

Note: Of course, this example doesn't show a flexible kind of security infrastructure. For more options as far as security is concerned, have a look at the Acegi Security System for Spring, to be found at http://acegisecurity.sourceforge.net.

### 17.4. Exposing services using HTTP invokers

As opposed to Burlap and Hessian, which are both lightweight protocols using their own slim serialization mechanisms, Spring Http invokers use the standard Java serialization mechanism to expose services through HTTP. This has a huge advantage if your arguments and return types are complex types that cannot be serialized using the serialization mechanisms Hessian and Burlap use (refer to the next section for more considerations when choosing a remoting technology).

Under the hood, Spring uses either the standard facilities provided by J2SE to perform HTTP calls or Commons HttpClient. Use the latter if you need more advanced and easy-to-use functionality. Refer to jakarta.apache.org/commons/httpclient for more info.

#### 17.4.1. Exposing the service object

Setting up the HTTP invoker infrastructure for a service objects much resembles the way you would do using Hessian or Burlap. Just as Hessian support provides the HessianServiceExporter, Spring Http invoker support provides the so-called org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter. To expose the AccountService (mentioned above), the following configuration needs to be in place:

<bean name="/AccountService" class="org.sprfr.remoting.httpinvoker.HttpInvokerServiceExporter">

<property name="service"><ref bean="accountService"/></property>

<property name="serviceInterface">

<value>example.AccountService</value>

</property>

</bean>

#### 17.4.2. Linking in the service at the client

Again, linking in the service from the client much resembles the way you would do it when using Hessian or Burlap. Using a proxy, Spring will be able to translate your calls to HTTP POST requests to the URL pointing to the exported service.

<bean id="httpInvokerProxy" class="org.sprfr.remoting.httpinvoker.HttpInvokerProxyFactoryBean">

<property name="serviceUrl">

<value>http://remotehost:8080/AccountService</value>

</property>

<property name="serviceInterface">

<value>example.AccountService</value>

</property>

</bean>

As mentioned before, you can choose what HTTP client you want to use. By default, the HttpInvokerProxy uses the J2SE HTTP functionality, but you can also use the Commons HttpClient by setting the httpInvokerRequestExecutor property:

<property name="httpInvokerRequestExecutor">

<bean class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerRequestExecutor"/>

</property>

### 17.5. Web Services

Spring has support for:

Exposing services using JAX-RPC

Accessing Web Services

#### 17.5.1. Exposing services using JAX-RPC

Spring has a convenience base class for JAX-RPC servlet endpoint implementations - ServletEndpointSupport. To expose our AccountService we extend Spring's ServletEndpointSupport class and implement our business logic here, usually delegating the call to the business layer.

/\*\*

\* JAX-RPC compliant RemoteAccountService implementation that simply delegates

\* to the AccountService implementation in the root web application context.

\*

\* This wrapper class is necessary because JAX-RPC requires working with

\* RMI interfaces. If an existing service needs to be exported, a wrapper that

\* extends ServletEndpointSupport for simple application context access is

\* the simplest JAX-RPC compliant way.

\*

\* This is the class registered with the server-side JAX-RPC implementation.

\* In the case of Axis, this happens in "server-config.wsdd" respectively via

\* deployment calls. The Web Service tool manages the life-cycle of instances

\* of this class: A Spring application context can just be accessed here.

\*/

public class AccountServiceEndpoint extends ServletEndpointSupport implements RemoteAccountService {

private AccountService biz;

protected void onInit() {

this.biz = (AccountService) getWebApplicationContext().getBean("accountService");

}

public void insertAccount(Account acc) throws RemoteException {

biz.insertAccount(acc);

}

public Account[] getAccounts(String name) throws RemoteException {

return biz.getAccounts(name);

}

}

Our AccountServletEndpoint needs to run in the same web application as the Spring context to allow for access to Spring's facilities. In case of Axis, copy the AxisServlet definition into your web.xml, and set up the endpoint in "server-config.wsdd" (or use the deploy tool). See the sample application JPetStore where the OrderService is exposed as a Web Service using Axis.

#### 17.5.2. Accessing Web Services

Spring has two factory beans to create web service proxies LocalJaxRpcServiceFactoryBean and JaxRpcPortProxyFactoryBean. The former can only return a JAX-RPC Service class for us to work with. The latter is the full fledged version that can return a proxy that implements our business service interface. In this example we use the later to create a proxy for the AccountService Endpoint we exposed in the previous paragraph. You will see that Spring has great support for Web Services requiring little coding efforts - most of the magic is done in the spring configuration file as usual:

<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">

<property name="serviceInterface">

<value>example.RemoteAccountService</value>

</property>

<property name="wsdlDocumentUrl">

<value>http://localhost:8080/account/services/accountService?WSDL</value>

</property>

<property name="namespaceUri">

<value>http://localhost:8080/account/services/accountService</value>

</property>

<property name="serviceName">

<value>AccountService</value>

</property>

<property name="portName">

<value>AccountPort</value>

</property>

</bean>

Where serviceInterface is our remote business interface the clients will use. wsdlDocumentUrl is the URL for the WSDL file. Spring needs this a startup time to create the JAX-RPC Service. namespaceUri corresponds to the targetNamespace in the .wsdl file. serviceName corresponds to the serivce name in the .wsdl file. portName corresponds to the port name in the .wsdl file.

Accessing the Web Service is now very easy as we have a bean factory for it that will expose it as RemoteAccountService interface. We can wire this up in Spring:

<bean id="client" class="example.AccountClientImpl">

...

<property name="service">

<ref bean="accountWebService"/>

</property>

</bean>

And from the client code we can access the Web Service just as if it was a normal class, except that it throws RemoteException.

public class AccountClientImpl {

private RemoteAccountService service;

public void setService(RemoteAccountService service) {

this.service = service;

}

public void foo() {

try {

service.insertAccount(...);

} catch (RemoteException e) {

// ouch

...

}

}

}

We can get rid of the checked RemoteException since Spring supports automatic conversion to its corresponding unchecked RemoteAccessException. This requires that we provide a non RMI interface also. Our configuration is now:

<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">

<property name="serviceInterface">

<value>example.AccountService</value>

</property>

<property name="portInterface">

<value>example.RemoteAccountService</value>

</property>

...

</bean>

Where serviceInterface is changed to our non RMI interface. Our RMI interface is now defined using the property portInterface. Our client code can now avoid handling java.rmi.RemoteException:

public class AccountClientImpl {

private AccountService service;

public void setService(AccountService service) {

this.service = service;

}

public void foo() {

service.insertAccount(...);

}

}

#### 17.5.3. Register Bean Mappings

To transfer complex objects over the wire such as Account we must register bean mappings on the client side.

[Note] Note

On the server side using Axis registering bean mappings is usually done in server-config.wsdd.

We will use Axis to register bean mappings on the client side. To do this we need to subclass Spring Bean factory and register the bean mappings programmatic:

public class AxisPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {

protected void postProcessJaxRpcService(Service service) {

TypeMappingRegistry registry = service.getTypeMappingRegistry();

TypeMapping mapping = registry.createTypeMapping();

registerBeanMapping(mapping, Account.class, "Account");

registry.register("http://schemas.xmlsoap.org/soap/encoding/", mapping);

}

protected void registerBeanMapping(TypeMapping mapping, Class type, String name) {

QName qName = new QName("http://localhost:8080/account/services/accountService", name);

mapping.register(type, qName,

new BeanSerializerFactory(type, qName),

new BeanDeserializerFactory(type, qName));

}

}

#### 17.5.4. Registering our own Handler

In this section we will register our own javax.rpc.xml.handler.Handler to the Web Service Proxy where we can do custom code before the SOAP message is sent over the wire. The javax.rpc.xml.handler.Handler is a callback interface. There is a convenience base class provided in jaxrpc.jar - javax.rpc.xml.handler.GenericHandler that we will extend:

public class AccountHandler extends GenericHandler {

public QName[] getHeaders() {

return null;

}

public boolean handleRequest(MessageContext context) {

SOAPMessageContext smc = (SOAPMessageContext) context;

SOAPMessage msg = smc.getMessage();

try {

SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();

SOAPHeader header = envelope.getHeader();

...

} catch (SOAPException e) {

throw new JAXRPCException(e);

}

return true;

}

}

What we need to do now is to register our AccountHandler to JAX-RPC Service so it would invoke handleRequest before the message is sent over the wire. Spring has at this time of writing no declarative support for registering handlers. So we must use the programmatic approach. However Spring has made it very easy for us to do this as we can extend its bean factory and override its postProcessJaxRpcService method that is designed for this:

public class AccountHandlerJaxRpcPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {

protected void postProcessJaxRpcService(Service service) {

QName port = new QName(this.getNamespaceUri(), this.getPortName());

List list = service.getHandlerRegistry().getHandlerChain(port);

list.add(new HandlerInfo(AccountHandler.class, null, null));

logger.info("Registered JAX-RPC Handler [" + AccountHandler.class.getName() + "] on port " + port);

}

}

And the last thing we must remember to do is to change the Spring configuration to use our factory bean:

<bean id="accountWebService" class="example.AccountHandlerJaxRpcPortProxyFactoryBean">

...

</bean>

### 17.6. Auto-detection is not implemented for remote interfaces

The main reason why auto-detection of implemented interfaces does not occur for remote interfaces is to avoid opening too many doors to remote callers. The target object might implement internal callback interfaces like InitializingBean or DisposableBean which one would not want to expose to callers.

Offering a proxy with all interfaces implemented by the target usually does not matter in the local case. But when exporting a remote service, you should expose a specific service interface, with specific operations intended for remote usage. Besides internal callback interfaces, the target might implement multiple business interfaces, with just one of them intended for remote exposure. For these reasons, we require such a service interface to be specified.

This is a trade-off between configuration convenience and the risk of accidental exposure of internal methods. Always specifying a service interface is not too much effort, and puts you on the safe side regarding controlled exposure of specific methods.

### 17.7. Considerations when choosing a technology

Each and every technology presented here has its drawbacks. You should carefully consider you needs, the services your exposing and the objects you'll be sending over the wire when choosing a technology.

When using RMI, it's not possible to access the objects through the HTTP protocol, unless you're tunneling the RMI traffic. RMI is a fairly heavy-weight protocol in that it support full-object serialization which is important when using a complex data model that needs serialization over the wire. However, RMI-JRMP is tied to Java clients: It is a Java-to-Java remoting solution.

Spring's HTTP invoker is a good choice if you need HTTP-based remoting but also rely on Java serialization. It shares the basic infrastructure with RMI invokers, just using HTTP as transport. Note that HTTP invokers are not only limited to Java-to-Java remoting but also to Spring on both the client and server side. (The latter also applies to Spring's RMI invoker for non-RMI interfaces.)

Hessian and/or Burlap might provide significant value when operating in a heterogeneous environment, because they explicitly allow for non-Java clients. However, non-Java support is still limited. Known problems include the serialization of Hibernate objects in combination with lazily initializing collections. If you have such a data model, consider using RMI or HTTP invokers instead of Hessian.

JMS can be useful for providing clusters of services and allowing the JMS broker to take care of load balancing, discovery and auto-failover. By default Java serialization is used when using JMS remoting but the JMS provider could use a different mechanism for the wire formatting, such as XStream to allow servers to be implemented in other technologies.

Last but not least, EJB has an advantage over RMI in that it supports standard role-based authentication and authorization and remote transaction propagation. It is possible to get RMI invokers or HTTP invokers to support security context propagation as well, although this is not provided by core Spring: There are just appropriate hooks for plugging in third-party or custom solutions here.

## 18. Sending Email with Spring mail abstraction layer

### 18.1. Introduction

Spring provides a higher level of abstraction for sending electronic mail which shields the user from the specifics of underlying mailing system and is responsible for a low level resource handling on behalf of the client.

### 18.2. Spring mail abstraction structure

The main package of Spring mail abstraction layer is org.springframework.mail package. It contains central interface for sending emails called MailSender and the value object which encapsulates properties of a simple mail such as from, to, cc, subject, text called SimpleMailMessage. This package also contains a hierarchy of checked exceptions which provide a higher level of abstraction over the lower level mail system exceptions with the root exception being MailException.Please refer to JavaDocs for more information on mail exception hierarchy.

Spring also provides a sub-interface of MailSender for specialized JavaMail features such as MIME messages, namely org.springframework.mail.javamail.JavaMailSender It also provides a callback interface for preparation of JavaMail MIME messages, namely org.springframework.mail.javamail.MimeMessagePreparator

MailSender:

public interface MailSender {

/\*\*

\* Send the given simple mail message.

\* @param simpleMessage message to send

\* @throws MailException in case of message, authentication, or send errors

\*/

public void send(SimpleMailMessage simpleMessage) throws MailException;

/\*\*

\* Send the given array of simple mail messages in batch.

\* @param simpleMessages messages to send

\* @throws MailException in case of message, authentication, or send errors

\*/

public void send(SimpleMailMessage[] simpleMessages) throws MailException;

}

JavaMailSender:

public interface JavaMailSender extends MailSender {

/\*\*

\* Create a new JavaMail MimeMessage for the underlying JavaMail Session

\* of this sender. Needs to be called to create MimeMessage instances

\* that can be prepared by the client and passed to send(MimeMessage).

\* @return the new MimeMessage instance

\* @see #send(MimeMessage)

\* @see #send(MimeMessage[])

\*/

public MimeMessage createMimeMessage();

/\*\*

\* Send the given JavaMail MIME message.

\* The message needs to have been created with createMimeMessage.

\* @param mimeMessage message to send

\* @throws MailException in case of message, authentication, or send errors

\* @see #createMimeMessage

\*/

public void send(MimeMessage mimeMessage) throws MailException;

/\*\*

\* Send the given array of JavaMail MIME messages in batch.

\* The messages need to have been created with createMimeMessage.

\* @param mimeMessages messages to send

\* @throws MailException in case of message, authentication, or send errors

\* @see #createMimeMessage

\*/

public void send(MimeMessage[] mimeMessages) throws MailException;

/\*\*

\* Send the JavaMail MIME message prepared by the given MimeMessagePreparator.

\* Alternative way to prepare MimeMessage instances, instead of createMimeMessage

\* and send(MimeMessage) calls. Takes care of proper exception conversion.

\* @param mimeMessagePreparator the preparator to use

\* @throws MailException in case of message, authentication, or send errors

\*/

public void send(MimeMessagePreparator mimeMessagePreparator) throws MailException;

/\*\*

\* Send the JavaMail MIME messages prepared by the given MimeMessagePreparators.

\* Alternative way to prepare MimeMessage instances, instead of createMimeMessage

\* and send(MimeMessage[]) calls. Takes care of proper exception conversion.

\* @param mimeMessagePreparators the preparator to use

\* @throws MailException in case of message, authentication, or send errors

\*/

public void send(MimeMessagePreparator[] mimeMessagePreparators) throws MailException;

}

MimeMessagePreparator:

public interface MimeMessagePreparator {

/\*\*

\* Prepare the given new MimeMessage instance.

\* @param mimeMessage the message to prepare

\* @throws MessagingException passing any exceptions thrown by MimeMessage

\* methods through for automatic conversion to the MailException hierarchy

\*/

void prepare(MimeMessage mimeMessage) throws MessagingException;

}

### 18.3. Using Spring mail abstraction

Let's assume there is a business interface called OrderManager

public interface OrderManager {

void placeOrder(Order order);

}

and there is a use case that says that an email message with order number would need to be generated and sent to a customer placing that order. So for this purpose we want to use MailSender and SimpleMailMessage

Please note that as usual, we work with interfaces in the business code and let Spring IoC container take care of wiring of all the collaborators for us.

Here is the implementation of OrderManager

import org.springframework.mail.MailException;

import org.springframework.mail.MailSender;

import org.springframework.mail.SimpleMailMessage;

public class OrderManagerImpl implements OrderManager {

private MailSender mailSender;

private SimpleMailMessage message;

public void setMailSender(MailSender mailSender) {

this.mailSender = mailSender;

}

public void setMessage(SimpleMailMessage message) {

this.message = message;

}

public void placeOrder(Order order) {

//... \* Do the business calculations....

//... \* Call the collaborators to persist the order

//Create a thread safe "sandbox" of the message

SimpleMailMessage msg = new SimpleMailMessage(this.message);

msg.setTo(order.getCustomer().getEmailAddress());

msg.setText(

"Dear "

+ order.getCustomer().getFirstName()

+ order.getCustomer().getLastName()

+ ", thank you for placing order. Your order number is "

+ order.getOrderNumber());

try{

mailSender.send(msg);

}

catch(MailException ex) {

//log it and go on

System.err.println(ex.getMessage());

}

}

}

Here is what the bean definitions for the code above would look like:

<bean id="mailSender"

class="org.springframework.mail.javamail.JavaMailSenderImpl">

<property name="host"><value>mail.mycompany.com</value></property>

</bean>

<bean id="mailMessage"

class="org.springframework.mail.SimpleMailMessage">

<property name="from"><value>customerservice@mycompany.com</value></property>

<property name="subject"><value>Your order</value></property>

</bean>

<bean id="orderManager"

class="com.mycompany.businessapp.support.OrderManagerImpl">

<property name="mailSender"><ref bean="mailSender"/></property>

<property name="message"><ref bean="mailMessage"/></property>

</bean>

Here is the implementation of OrderManager using MimeMessagePreparator callback interface. Please note that the mailSender property is of type JavaMailSender in this case in order to be able to use JavaMail MimeMessage:

import javax.mail.Message;

import javax.mail.MessagingException;

import javax.mail.internet.InternetAddress;

import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;

import org.springframework.mail.MailException;

import org.springframework.mail.javamail.JavaMailSender;

import org.springframework.mail.javamail.MimeMessagePreparator;

public class OrderManagerImpl implements OrderManager {

private JavaMailSender mailSender;

public void setMailSender(JavaMailSender mailSender) {

this.mailSender = mailSender;

}

public void placeOrder(final Order order) {

//... \* Do the business calculations....

//... \* Call the collaborators to persist the order

MimeMessagePreparator preparator = new MimeMessagePreparator() {

public void prepare(MimeMessage mimeMessage) throws MessagingException {

mimeMessage.setRecipient(Message.RecipientType.TO,

new InternetAddress(order.getCustomer().getEmailAddress()));

mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));

mimeMessage.setText(

"Dear "

+ order.getCustomer().getFirstName()

+ order.getCustomer().getLastName()

+ ", thank you for placing order. Your order number is "

+ order.getOrderNumber());

}

};

try{

mailSender.send(preparator);

}

catch(MailException ex) {

//log it and go on

System.err.println(ex.getMessage());

}

}

}

If you want to use JavaMail MimeMessage to the full power, the MimeMessagePreparator is available at your fingertips.

Please note that the mail code is a crosscutting concern and is a perfect candidate for refactoring into a custom Spring AOP advice, which then could easily be applied to OrderManager target. Please see the AOP chapter.

#### 18.3.1. Pluggable MailSender implementations

Spring comes with two MailSender implementations out of the box - the JavaMail implementation and the implementation on top of Jason Hunter's MailMessage class that's included in http://servlets.com/cos (com.oreilly.servlet). Please refer to JavaDocs for more information.

### 18.4. Using the JavaMail MimeMessageHelper

One of the components that comes in pretty handy when dealing with JavaMail messages is the org.springframework.mail.javamail.MimeMessageHelper. It prevents you from having to use the nasty APIs the the javax.mail.internet classes. A couple of possible scenarios:

#### 18.4.1. Creating a simple MimeMessage and sending it

Using the MimeMessageHelper it's pretty easy to setup and send a MimeMessage:

// of course you would setup the mail sender using

// DI in any real-world cases

JavaMailSenderImpl sender = new JavaMailSenderImpl();

sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMesage();

MimeMessageHelper helper = new MimeMessageHelper(message);

helper.setTo("test@host.com");

helper.setText("Thank you for ordering!");

sender.send(message);

#### 18.4.2. Sending attachments and inline resources

Email allow for attachments, but also for inline resources in multipart messages. Inline resources could for example be images or stylesheet you want to use in your message, but don't want displayed as attachment. The following shows you how to use the MimeMessageHelper to send an email along with an inline image.

JavaMailSenderImpl sender = new JavaMailSenderImpl();

sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMesage();

// use the true flag to indicate you need a multipart message

MimeMessageHelper helper = new MimeMessageHelper(message, true);

helper.setTo("test@host.com");

// use the true flag to indicate the text included is HTML

helper.setText(

"<html><body><img src='cid:identifier1234'></body></html>"

true);

// let's include the infamous windows Sample file (this time copied to c:/)

FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));

helper.addInline("identifier1234", res);

// if you would need to include the file as an attachment, use

// addAttachment() methods on the MimeMessageHelper

sender.send(message);

Inline resources are added to the mime message using the Content-ID specified as you've seen just now (identifier1234 in this case). The order in which you're adding the text and the resource are VERY important. First add the text and after that the resources. If you're doing it the other way around, it won't work!

## 19. Scheduling jobs using Quartz or Timer

### 19.1. Introduction

Spring features integration classes for scheduling support. Currently, Spring supports the Timer, part of the JDK since 1.3, and the Quartz Scheduler (http://www.quartzscheduler.org). Both schedulers are set up using a FactoryBean with optional references to Timers or Triggers, respectively. Furthermore, a convenience class for both the Quartz Scheduler and the Timer is available that allows you to invoke a method of an existing target object (analogous to normal MethodInvokingFactoryBeans).

### 19.2. Using the OpenSymphony Quartz Scheduler

Quartz uses Triggers, Jobs and JobDetail ro realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, have a look at http://www.opensymphony.com/quartz. For convenience purposes, Spring offers a couple of classes that simplify usage of Quartz within Spring-based applications.

#### 19.2.1. Using the JobDetailBean

JobDetail objects contain all information needed to run a job. Spring provides a so-called JobDetailBean that makes the JobDetail more of an actual JavaBean with sensible defaults. Let's have a look at an example:

<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailBean">

<property name="jobClass">

<value>example.ExampleJob</value>

</property>

<property name="jobDataAsMap">

<map>

<entry key="timeout"><value>5</value></entry>

</map>

</property>

</bean>

The job detail bean has all information it needs to run the job (ExampleJob). The timeout is specified as the job data map. The job data map is available through the JobExecutionContext (passed to you at execution time), but the JobDetailBean also maps the properties from the job data map to properties of the actual job. So in this case, if the ExampleJob contains a property named timeout, the JobDetailBean will automatically apply it:

package example;

public class ExampleJob extends QuartzJobBean {

private int timeout;

/\*\*

\* Setter called after the ExampleJob is instantiated

\* with the value from the JobDetailBean (5)

\*/

public void setTimeout(int timeout) {

this.timeout = timeout;

}

protected void executeInternal(JobExecutionContext ctx)

throws JobExecutionException {

// do the actual work

}

}

All additional settings from the job detail bean are of course available to you as well.

Note: Using the name and group properties, you can modify the name and the group of the job, respectively. By default the name of the job equals the bean name of the job detail bean (in the example above, this is exampleJob).

#### 19.2.2. Using the MethodInvokingJobDetailFactoryBean

Often you just need to invoke a method on a specific object. Using the MethodInvokingJobDetailFactoryBean you can do exactly this:

<bean id="methodInvokingJobDetail"

class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject"><ref bean="exampleBusinessObject"/></property>

<property name="targetMethod"><value>doIt</value></property>

</bean>

The above example will result in the doIt being called on the exampleBusinessObject (see below):

public class BusinessObject {

// properties and collaborators

public void doIt() {

// do the actual work

}

}

<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>

Using the MethodInvokingJobDetailFactoryBean you don't need to create one-line jobs that just invoke a method, and you only need to create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same JobDetail, it might be possible that before the first job has finished, the second one will start. If JobDetail objects implement the Stateful interface, this won't happen. The second job will not start before the first one has finished. To make jobs resulting from the MethodInvokingJobDetailFactoryBean non-concurrent, set the concurrent flag to false.

<bean id="methodInvokingJobDetail"

class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject"><ref bean="exampleBusinessObject"/></property>

<property name="targetMethod"><value>doIt</value></property>

<property name="concurrent"><value>false</value></property>

</bean>

Note: By default, jobs will run in a concurrent fashion.

#### 19.2.3. Wiring up jobs using triggers and the SchedulerFactoryBean

We've created job details, jobs and we've reviewed the convenience bean that allows to you invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done using triggers and a SchedulerFactoryBean. Several triggers are available within Quartz. Spring offers two subclassed triggers with convenient defaults: CronTriggerBean and SimpleTriggerBean.

Triggers need to be scheduled. Spring offers a SchedulerFactoryBean exposing properties to set the triggers. SchedulerFactoryBean schedules the actual jobs with those triggers.

A couple of examples:

<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerBean">

<property name="jobDetail">

<!-- see the example of method invoking job above -->

<ref bean="methodInvokingJobDetail"/>

</property>

<property name="startDelay">

<!-- 10 seconds -->

<value>10000</value>

</property>

<property name="repeatInterval">

<!-- repeat every 50 seconds -->

<value>50000</value>

</property>

</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">

<property name="jobDetail">

<ref bean="exampleJob"/>

</property>

<property name="cronExpression">

<!-- run every morning at 6 AM -->

<value>0 0 6 \* \* ?</value>

</property>

</bean>

OK, now we've set up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one every morning at 6 AM. To finalize everything, we need to set up the SchedulerFactoryBean:

<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="triggers">

<list>

<ref local="cronTrigger"/>

<ref local="simpleTrigger"/>

</list>

</property>

</bean>

More properties are available for the SchedulerFactoryBean for you to set, such as the calendars used by the job details, properties to customize Quartz with, etc. Have a look at the JavaDoc (http://www.springframework.org/docs/api/org/springframework/scheduling/quartz/SchedulerFactoryBean.html) for more information.

### 19.3. Using JDK Timer support

The other way to schedule jobs in Spring is using JDK Timer objects. More information about Timers themselves can be found at http://java.sun.com/docs/books/tutorial/essential/threads/timer.html. The concepts discussed above also apply to the Timer support. You can create custom timers or use the timer that invokes methods. Wiring timers has to be done using the TimerFactoryBean.

#### 19.3.1. Creating custom timers

Using the TimerTask you can create customer timer tasks, similar to Quartz jobs:

public class CheckEmailAddresses extends TimerTask {

private List emailAddresses;

public void setEmailAddresses(List emailAddresses) {

this.emailAddresses = emailAddresses;

}

public void run() {

// iterate over all email addresses and archive them

}

}

Wiring it up is simple:

<bean id="checkEmail" class="examples.CheckEmailAddress">

<property name="emailAddresses">

<list>

<value>test@springframework.org</value>

<value>foo@bar.com</value>

<value>john@doe.net</value>

</list>

</property>

</bean>

<bean id="scheduledTask" class="org.springframework.scheduling.timer.ScheduledTimerTask">

<!-- wait 10 seconds before starting repeated execution -->

<property name="delay">

<value>10000</value>

</property>

<!-- run every 50 seconds -->

<property name="period">

<value>50000</value>

</property>

<property name="timerTask">

<ref local="checkEmail"/>

</property>

</bean>

Letting the task only run once can be done by changing the period property to -1 (or some other negative value)

#### 19.3.2. Using the MethodInvokingTimerTaskFactoryBean

Similar to the Quartz support, the Timer support also features a component that allows you to periodically invoke a method:

<bean id="methodInvokingTask"

class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">

<property name="targetObject"><ref bean="exampleBusinessObject"/></property>

<property name="targetMethod"><value>doIt</value></property>

</bean>

The above example will result in the doIt being called on the exampleBusinessObject (see below):

public class BusinessObject {

// properties and collaborators

public void doIt() {

// do the actual work

}

}

Changing the reference of the above example in which the ScheduledTimerTask is mentioned to the methodInvokingTask will result in this task being executed.

#### 19.3.3. Wrapping up: setting up the tasks using the TimerFactoryBean

The TimerFactoryBean is similar to the Quartz SchedulerFactoryBean in that it serves the same purpose: setting up the actual scheduling. The TimerFactoryBean sets up an actual Timer and schedules the tasks it has references to. You can specify whether or not daemon threads should be used.

<bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">

<property name="scheduledTimerTasks">

<list>

<!-- see the example above -->

<ref local="scheduledTask"/>

</list>

</property>

</bean>

That's all!

## 20. Testing

### 20.1. Unit testing

You don't need this manual to help you write effective unit tests for Spring-based applications.

One of the main benefits of Dependency Injection is that your code should depend far less on the container than in traditional J2EE development.

The POJOs that comprise your application should be testable in JUnit tests, with objects simply instantiated using the new operator, without Spring or any other container. You can use mock objects or many other valuable testing techniques, to test your code in isolation. If you follow the architecture recommendations around Spring--for example, those in J2EE without EJB--you will find that the resulting clean layering will also greatly facilitate testing. For example, you will be able to test service layer objects by stubbing or mocking DAO interfaces, without any need to access persistent data while running unit tests.

True unit tests will run extremely quickly, as there is no runtime infrastructure to set up, whether application server, database, ORM tool etc. Thus emphasizing true unit tests will boost your productivity.

### 20.2. Integration testing

However, it's also important to be able to perform some integration testing without deployment to your application server. This will test things such as:

Correct wiring of your Spring contexts.

Data access using JDBC or ORM tool--correctness of SQL statements. For example, you can test your DAO implementation classes.

Thus Spring provides valuable support for integration testing, in the spring-mock.jar. This can be thought of as a significantly superior alternative to in-container testing using tools such as Cactus.

The org.springframework.test package provides valuable superclasses for integration tests using a Spring container, but not dependent on an application server or other deployed environment. Such tests can run in JUnit--even in an IDE--without any special deployment step. They will be slower to run than unit tests, but much faster to run than Cactus tests or remote tests relying on deployment to an application server.

The superclasses in this package provide the following functionality:

Context caching.

Dependency Injection for test classes.

Transaction management appropriate to tests.

Inherited instance variables useful for testing.

Numerous Interface21 and other projects since late 2004 have demonstrated the power and utility of this approach. Let's look at some of the important areas of functionality in detail.

#### 20.2.1. Context management and caching

The org.springframework.test package provides support for consistent loading of Spring contexts, and caching of loaded contexts. The latter is important, because if you are working on a large project startup time may become an issue--not because of the overhead of Spring itself, but because the objects instantiated by the Spring container will themselves take time to instantiate. For example, a project with 50-100 Hibernate mapping files might take 10-20 seconds to load them, and incurring that cost before running every test case will greatly reduce productivity.

Thus, AbstractDependencyInjectionSpringContextTests has an abstract protected method that subclasses must implement, to provide the location of contexts:

protected abstract String[] getConfigLocations();

This should provide a list of the context locations--typically on the classpath--used to configure the application. This will be the same, or nearly the same, as the list of config locations specified in web.xml or other deployment configuration.

By default, once loaded, the set of configs will be reused for each test case. Thus the setup cost will be incurred only once, and subsequent test execution will be much faster.

In the unlikely case that a test may "dirty" the config location, requiring reloading--for example, by changing a bean definition or the state of an application object--you can call the setDirty() method on AbstractDependencyInjectionSpringContextTests to cause it to reload the configurations and rebuild the application context before executing the next test case.

#### 20.2.2. Dependency Injection of test class instances

When AbstractDependencyInjectionSpringContextTests (and subclasses) load your application context, they can optionally configure instances of yourr test classes by Setter Injection. All you need to do is to define instance variables and the corresponding setters. AbstractDependencyInjectionSpringContextTests will automatically locate the corresponding object in the set of configuration files specified in the getConfigLocations() method.

The superclasses use autowire by type. Thus if you have multiple bean definitions of the same type, you cannot rely on this approach for those particular beans. In that case, you can use the inherited applicationContext instance variable, and explicit lookup using getBean().

If you don't want Setter Injection applied to your test cases, don't declare any setters. Or extend AbstractSpringContextTests--the root of the class hierarchy in the org.springframework.test package. It merely contains convenience methods to load Spring contexts, and performs no Dependency Injection.

#### 20.2.3. Transaction management

One common problem in tests that access a real database is their effect on the state of the persistence store. Even when you're using a development database, changes to the state may affect future tests.

Also, many operations--such as inserting to or modifying persistence data--can't be done (or verified) outside a transaction.

The org.springframework.test.AbstractTransactionalDataSourceSpringContextTests superclass (and subclasses) exist to meet this need. By default, they create and roll back a transaction for each test case. You simply write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they will behave correctly, according to their transactional semantics.

AbstractTransactionalSpringContextTests depends on a PlatformTransactionManager bean being defined in the application context. The name doesn't matter, due to the use of autowire by type.

Typically you will extend the subclass, AbstractTransactionalDataSourceSpringContextTests. This also requires a DataSource bean definition--again, with any name--is present in the configurations. It creates a JdbcTemplate instance variable that is useful for convenient querying, and provides handy methods to delete the contents of selected tables. (Remember that the transaction will roll back by default, so this is safe.)

If you want a transaction to commit--unusual, but useful if you want a particular test to populate the database, for example--you can call the setComplete() method inherited from AbstractTransactionalSpringContextTests. This will cause the transaction to commit instead of roll back.

There is also convenient ability to end a transaction before the test case ends, through calling the endTransaction() method. This will roll back the transaction by default, and commit it only if setComplete() had previously been called. This functionality is useful if you want to test the behaviour of "disconnected" data objects, such as Hibernate-mapped objects that will be used in a web or remoting tier outside a transaction. Often, lazy loading errors are discovered only through UI testing; if you call endTransaction() you can ensure correct operation of the UI through your JUnit test suite.

Note that these test support classes are designed to work with a single database.

#### 20.2.4. Convenience variables

When you extend org.springframework.test package you will have access to the following protected instance variables:

applicationContext (ConfigurableApplicationContext): inherited from AbstractDependencyInjectionSpringContextTests. Use this to perfom explicit bean lookup, or test the state of the context as a whole.

jdbcTemplate: inherited from AbstractTransactionalDataSourceSpringContextTests. Useful for querying to confirm state. For example, you might query before and after testing application code that creates an object and persists it using an ORM tool, to verify that the data appears in the database. (Spring will ensure that the query runs in the scope of the same transaction.) You will need to tell your ORM tool to "flush" its changes for this to work correctly, for example using the flush() method on Hibernate's Session interface.

Often you will provide an application-wide superclass for integration tests that provides further useful instance variables used in many tests.

#### 20.2.5. Example

The PetClinic sample application included with the Spring distribution illustrates the use of these test superclasses (Spring 1.1.5 and above).

Most test functionality is included in AbstractClinicTests, for which a partial listing is shown belong:

public abstract class AbstractClinicTests extends AbstractTransactionalDataSourceSpringContextTests {

protected Clinic clinic;

public void setClinic(Clinic clinic) {

this.clinic = clinic;

}

public void testGetVets() {

Collection vets = this.clinic.getVets();

assertEquals("JDBC query must show the same number of vets",

jdbcTemplate.queryForInt("SELECT COUNT(0) FROM VETS"),

vets.size());

Vet v1 = (Vet) EntityUtils.getById(vets, Vet.class, 2);

assertEquals("Leary", v1.getLastName());

assertEquals(1, v1.getNrOfSpecialties());

assertEquals("radiology", ((Specialty) v1.getSpecialties().get(0)).getName());

Vet v2 = (Vet) EntityUtils.getById(vets, Vet.class, 3);

assertEquals("Douglas", v2.getLastName());

assertEquals(2, v2.getNrOfSpecialties());

assertEquals("dentistry", ((Specialty) v2.getSpecialties().get(0)).getName());

assertEquals("surgery", ((Specialty) v2.getSpecialties().get(1)).getName());

}

Notes:

This test case extends org.springframework.AbstractTransactionalDataSourceSpringContextTests, from which it inherits Dependency Injection and transactional behaviour.

The clinic instance variable--the application object being tested--is set by Dependency Injection through the setClinic() method.

The testGetVets() method illustrates how the inherited JdbcTemplate variable can be used to verify correct behaviour of the application code being tested. This allows for stronger tests, and lessens dependency on the exact test data. For example, you can add additional rows in the database without breaking tests.

Like many integration tests using a database, most of the tests in AbstractClinicTests depend on a minimum amount of data already in the database before the test cases run. You might, however, choose to populate the database in your test cases also--again, within the one transaction.

The PetClinic application supports three data access technologies--JDBC, Hibernate and Apache OJB. Thus AbstractClinicTests does not specify the context locations--this is deferred to subclasses, that implement the necessary protected abstract method from AbstractDependencyInjectionSpringContextTests.

For example, the JDBC implementation of the PetClinic tests contains the following method:

public class HibernateClinicTests extends AbstractClinicTests {

protected String[] getConfigLocations() {

return new String[] {

"/org/springframework/samples/petclinic/hibernate/applicationContext-hibernate.xml"

};

}

}

As the PetClinic is a very simple application, there is only one Spring configuration file. Of course, more complex applications will typically break their Spring configuration across multiple files.

Instead of being defined in a leaf class, config locations will often be specified in a common base class for all application-specific integration tests. This may also add useful instance variables--populated by Dependency Injection, naturally--such as a HibernateTemplate, in the case of an application using Hibernate.

As far as possible, you should have exactly the same Spring configuration files in your integration tests as in the deployed environment. One likely point of difference concerns database connection pooling and transaction infrastructure. If you are deploying to a full-blown application server, you will probably use its connection pool (available through JNDI) and JTA implementation. Thus in production you will use a JndiObjectFactoryBean for the DataSource, and JtaTransactionManager. JNDI and JTA will not be available in out-of-container integration tests, so you should use a combination like the Commons DBCP BasicDataSource and DataSourceTransactionManager or HibernateTransactionManager for them. You can factor out this variant behaviour into a single XML file, having the choice between application server and "local" configuration separated from all other configuration, which will not vary between the test and production environments.

#### 20.2.6. Running integration tests

Integration tests naturally have more environmental dependencies than plain unit tests. Such integration testing is an additional form of testing, not a substitute for unit testing.

The main dependency will typically be on a development database containing a complete schema used by the application. This may also contain test data, set up by a a tool such as a DBUnit, or an import using your database's tool set.

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

# Heading 1 fdfjsal fsjflas fs fjsdfla lewlf ưefjlw jwelfw wfwjl ẻwrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

## Heading 1.1 fjsdlas jrlwe flwefjlw flwjfwf ừlwfjewlf ừlwjlefw ưlw

### fdfa

### dfjsalfds fdfsa rưejrlw rưl sfjalf ưlflw fwlf w flwjrlw sdls fwefw

#### ffdslffsfsla fsjdfal lư flw fwjwl flwf lựlspa ầ fjslaf ừljflwefjw flwf ưlfwlfwl

#### fdjfslaf sfjsf fjdsflsfsa fjsdlf sfarw dfla fjdsla ljrwe lfjwl fwwrwl

### fjdfka fdjsal djfsla sjflsf álf ậlrjlw lằeopaf fsjdfapruw fajfowf ẻw

#### dslfas fjdsaljflew fwfwf jwfelw

### dfjsal jewlrewrw

fdsfakl fsfda

fdsfla

fdsa

fsa coding title coding example