java/j2ee Application Framework

Reference Documentation

Version 1.2.2

(Work in progress)

Table of Contents

서문

소프트웨어 애플리케이션을 개발하는것은 좋은 툴과 기술만으로는 충분하지 않다. 무겁지만 모든것을 약속하는 플랫폼을 사용하여 애플리케이션을 구현하는 것은 제어하기 힘들고 개발 주기가 더 어렵게 되는동안 효과적이지 않다. Spring은 기업용 애플리케이션을 빌드하기 위해 선언적인 트랜잭션 관리, RMI나 웹서비스를 사용하는 당신의 로직에 원격접근, 메일링 기능과 데이터베이스에 당신의 데이터를 지속하는 다양한 옵션을 사용하는 가능성을 지원하는 동안 가벼운 솔루션을 제공한다. Spring은 MVC프레임워크, AOP를 당신의 소프트웨어에 통합하는 일관적인 방법 그리고 선호하는 예외구조로부터 자동 맵핑을 포함한 잘 정의된 예외 구조를 제공한다.

Spring은 모든 당신의 기업용 애플리케이션을 위해 잠재적으로 one-stop-shop이 될수 있다. 어쨌든 Spring은 모듈적이고, 당신에게 나머지를 가져오는것 없이 이것의 일부를 사용하도록 허용한다. 당신은 가장 상위에 Struts를 사용하여 bean컨테이너를 사용할수 있다. 하지만 당신은 Hibernate통합이나 JDBC추상레이어를 사용하도록 선택할수 있다. Spring은 비침략적이고, 프레임워크에서 의존성이 의미하는것은 사용 영역에 의존해서 대개 아무것도 없거나 극도로 최소한적이다.

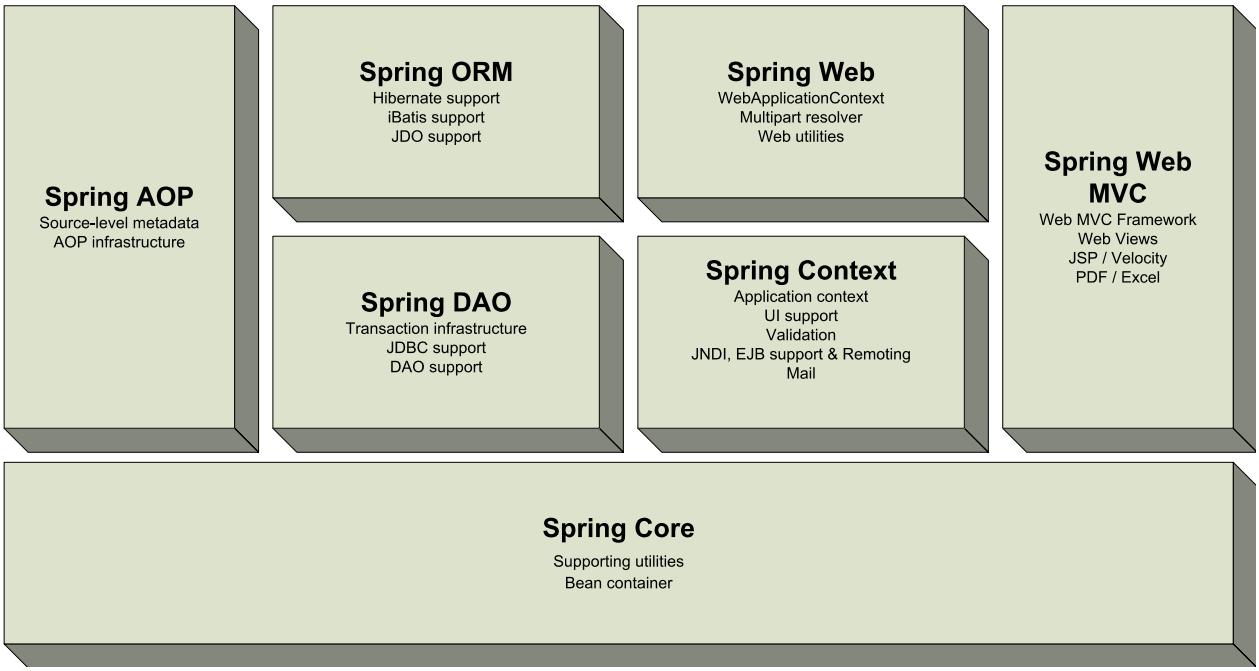
이 문서는 Spring의 기능에 대한 참조가이드를 제공한다. 이 문서는 여전히 작업중이기 때문에 만약 당신이 어떠한 요청이나 언급을 가진다면 사용자 메일링 리스트나 소스포지 프로젝트 페이지(http://www.sf.net/projects/springframework)의 포럼에 그것들을 올려달라.

시작하기 전 몇몇 감사의 말씀 : Chris Bauer(Hibernate팀의)이 준비하고 Hibernate의 참조가이트를 생성하는 것을 가능하도록 하기 위한 순서대로 DocBook-XSL 소프트웨어를 개작했다. 또한 우리에게 이것을 생성하도록 허용했다. 또한 자료의 몇몇 광대하고 가치있는 리뷰를 해 준 Russell Healy에게도 감사한다.

Chapter 1. 소개

1.1. 개요

Spring은 밑의 다이어그램에서 보여지는 7개의 모듈로 잘 조직된 많은 기능과 특성을 포함한다. 이 부분은 순서대로 각 모듈을 언급한다.



Spring 프레임워크의 개요

Core 패키지는 프레임워크의 가장 기본적인 부분이고 당신에게 bean컨테이너를 기능적으로 관리하는 것을 허용하는 의존성 삽입(Dependency Injection-DI)기능을 제공한다. 여기의 기본적인 개념은 프로그램에 따른 싱글톤의 필요성을 제거하는 factory패턴을 제공하고 당신의 실질적인 프로그램 로직으로부터 설정과 의존성 명시를 분리시키는 것을 당신에게 허용하는 BeanFactory이다.

Core 패키지의 가장 위에는 프레임워크 스타일의 방식으로 bean에 접근하기 위한 방법을 제공하는 다소 JNDI-등록기와 유사한 Context 패키지가 위치한다. context패키지는 bean패키지로부터 이 기능을 상속하고 예를 들어 resource bundle와 같은것을 사용하여 텍스트 메시지, 이벤트 위임, 자원-로딩 그리고 예를 들어 서블릿 컨테이너와 같은 것에 의해 투명한 컨텍스트 생성을 위한 지원을 추가한다.

DAO 패키지는 끔찍한 JDBC코딩과 데이터베이스 업체 특정 에러코드의 파싱을 할 필요를 제거하는 JDBC추상화 레이어를 제공한다. 또한 JDBC패키지는 특정 인터페이스를 구현하는 클래스를 위해서 뿐 아니라 당신의 모든 POJOs를 위해서도 선언적인 트랜잭션 관리만큼 프로그램에 따른 방식으로 할수 있는 방법을 제공한다.

ORM 패키지는 JDO, Hibernate 그리고 iBATIS를 포함하는 인기있는 객체-관계 맵핑 API를 위한 통합 레이어를 제공한다. ORM패키지는 사용하여 당신은 앞에서 언급된 간단한 선언적인 트랜잭션 관리와 같은 Spring이 제공하는 다른 모든 기능을 사용해서 혼합하여 모든 O/R매퍼를 사용할수 있다.

Spring의 AOP 패키지는 당신이 정의하는것을 허용하는 AOP 제휴 호환 aspect-지향 프로그래밍 구현물을 제공한다. 예를 들어 코드를 명백하게 분리하기 위한 메소드-인터셉터와 pointcut은 논리적으로 구별되어야 할 기능을 구현한다. 소스레벨 메터데이타 기능을 사용하여 당신은 .NET속성과 다소 비슷한

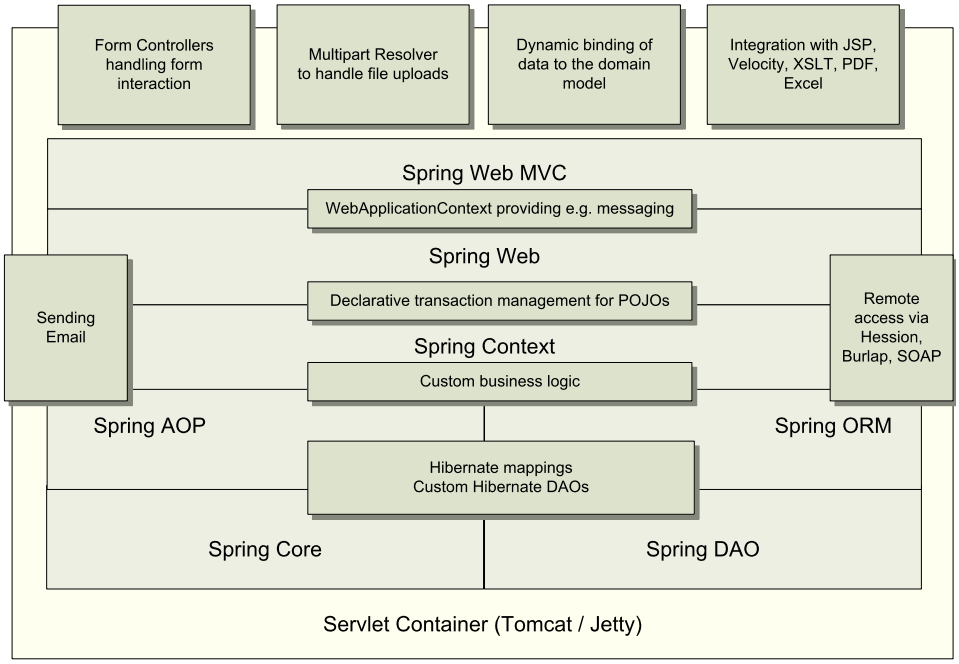
모든 종류의 행위적 정보를 당신의코드로 결합한다.

Spring의 Web 패키지는 멀티파트기능, 서블릿 리스너를 사용한 컨텍스트 초기화 그리고 웹-기반 애플리케이션 컨텍스트와같은 기본적인 웹-기반 통합 기능들을 제공한다. WebWork나 Struts와 함께 Spring을 사용할때 이것은 그것들과 통합할 패키지이다.

Spring의 웹 MVC 패키지는 웹 애플리케이션을 위한 Model-View-Controller구현물을 제공한다. Spring의 MVC구현물은 어떠한 구현물이 아니다. 이것은 도메인 모델 코드와 웹폼(Web forms)사이의 분명한 구분을 제공하고 유효성체크와 같은 Spring프레임워크의 다른 모든 기능을 사용하도록 당신에게 허용한다.

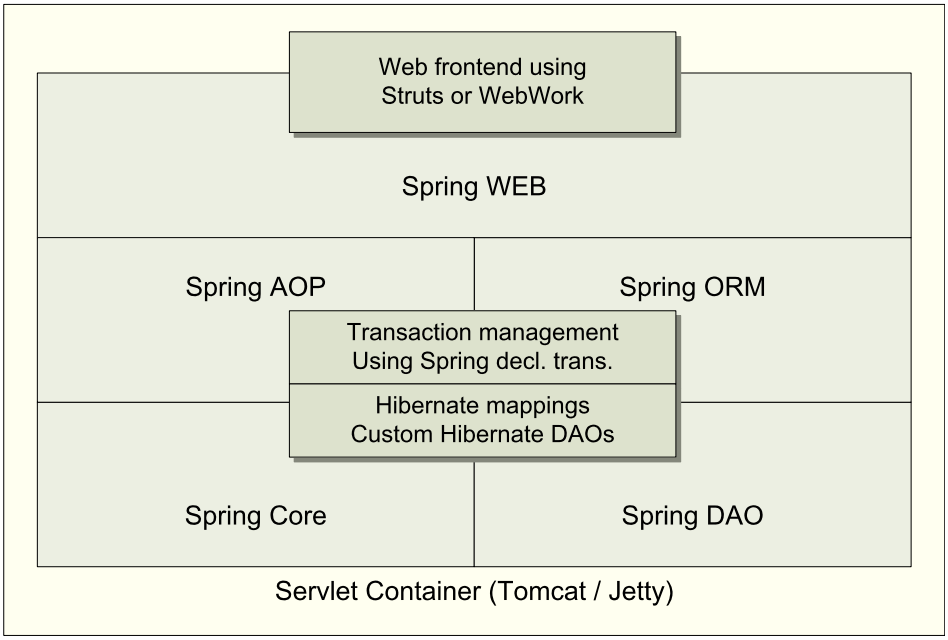
1.2. 사용 시나리오

위에서 언급된 빌드단위로 당신은 애플릿에서부터 Spring의 트랜잭션 관리 기능과 웹 프레임워크를 사용하는 완전한 기업용 애플리케이션까지 모든 종류의 시나리오로 Spring을 사용할수 있다.



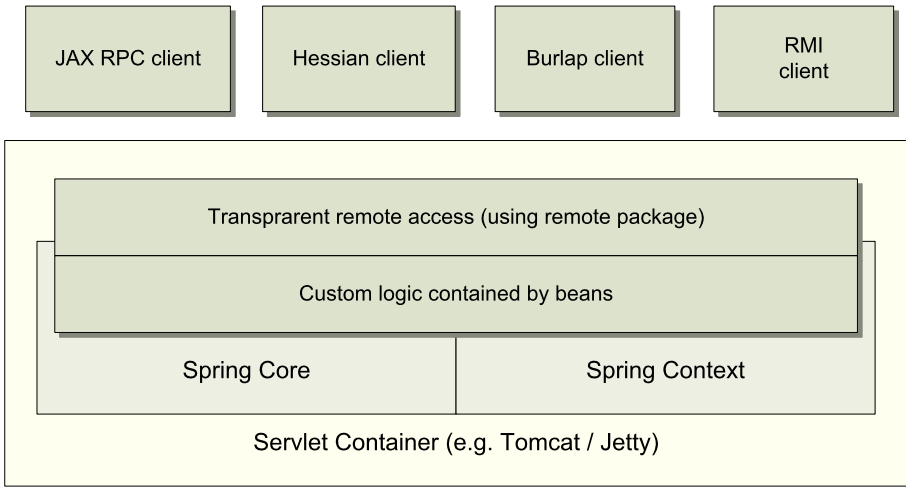
전형적으로 완전한 Spring웹 애플리케이션

대부분의 Spring기능을 사용한 전형적인 웹 애플리케이션. TransactionProxyFactoryBeans를 사용하면 웹 애플리케이션은 EJB에 의해 제공되는것과 같은 컨테이너 관리 트랜잭션을 사용할때 되는것처럼 완벽하게 트랜잭션적이다. 당신의 모든 사용자 지정 비지니스 로직은 Spring의 의존성 삽입 컨테이너에 의해 관리되는 간단한 POJO를 사용해서 구현될수 있다. 메일을 보내거나 유효성체크와 같은 추가적인 서비스, 웹 레이어의 비의존성은 당신에게 유효성체크 규칙을 수행하기 위한 위치를 선택하도록 허용한다. Spring의 ORM지원은 Hibernate, JDO 그리고 iBATIS와 통합된다. 예를 들어 HibernateDaoSupport를 사용하면 당신은 존재하는 Hibernate맵핑을 재사용할수 있다. 폼 컨트롤러는 ActionForms이나 HTTP파라미터를 당신의 도메인 모델을 위한 값에 이동시키는 다른 클래스의 필요성을 제거하는 도메인모델을 가진 웹레이어와 유사하게 통합한다.



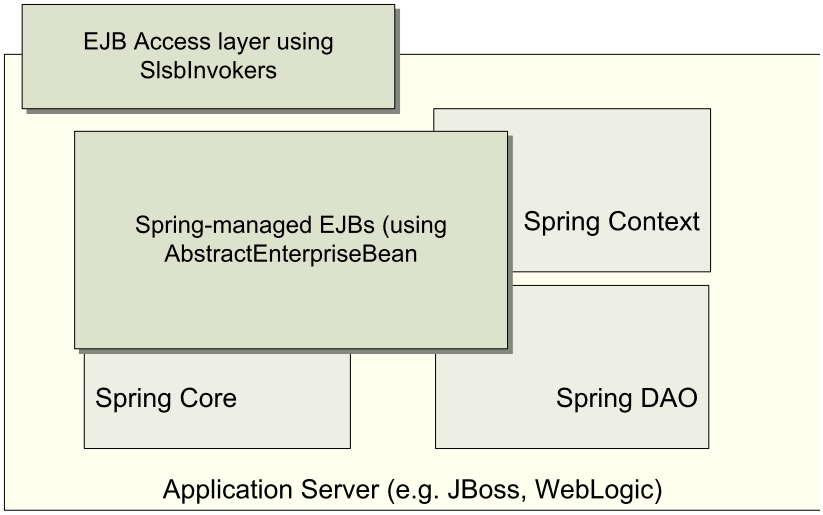
3자(third-party)의 웹 프레임워크를 사용한 Spring 미들티어

때때로 현재의 환경은 당신에게 다른 프레임워크로의 완벽한 교체를 허용하지 않는다. Spring은 이것내 모든것을 사용하도록 당신에게 강요하지 않는다. 이것은 모든것 또는 아무것도 아닌것(all-or-nothing)인 솔루션이 아니다. WebWork, Struts, Tapestry 또는 다른 UI프레임워크를 사용한 존재하는 앞부분은 당신에게 Spring이 제공하는 트랜잭션 기능을 사용하도록 허용하는 Spring기반의 미들티어와 완벽하게 통합될수 있다. 당신이 해야할 필요가 있는 오직 한가지는 ApplicationContext를 사용하여 당신의 비지니스 로직을 묶고 WebApplicationContext를 사용하여 당신의 웹 UI레이어를 통합하는 것이다.



원격 사용 시나리오

당신이 웹서비스를 통해 존재하는 코드에 접근할 필요가 있을 때, 당신은 Spring의 Hessian-, Burlap-, Rmi-나 JaxRpcProxyFactory클래스를 사용할수 있다. 존재하는 애플리케이션에 원격 접근을 가능하게 하는 것은 최근에는 어려운 일이 아니다.



EJB - 존재하는 POJO를 포장하기

Spring은 POJO를 재사용하는것을 당신에게 허용하고 그것들을 비상태유지(stateless) 세션빈으로 포장하고 선언적인 보안이 필요한 측정가능한 실패에 안전한(failsafe) 웹 애플리케이션내 사용하기 위한 EJB를 위해 존재하는 접근 레이어와 추상 레이어를 제공한다.

Chapter 2. 배경 지식

2.1. 제어의 역행 / 의존 주입(Inversion of Control / Dependency Injection)

2004년초, Martin Fowler는 그의 사이트 독자들에거 물었다 : Inversion of Control에 관해 이야기할때 "질문은, 제어의 측면에서 무엇이 역행하는가?" 였다. Martin은 Inversion of Control 용어에 대해서 설명한후 패턴 이름을 바꾸거나 적어도 더 나은 스스로-해석되는 이름을 가지길 제안했고, Dependency Injection 용어를 사용하기 시작한다. 그의 기사는 Inversion of Control 또는 Dependency Injection 뒤에 숨겨진 굉장한 아이디어들을 설명하기위해 계속된다. 어느정도 통찰력이 필요하다면 : http://martinfowler.com/articles/injection.html 을 방문하라.

Chapter 3. Beans, BeanFactory 그리고 ApplicationContext

3.1. 소개

Spring내에서 가장 기초적이고 가장 중요한 두개의 패키지는 org.springframework.beans 와 org.springframework.context패키지이다. 이 패키지내 코드는 Spring의 Inversion of Control(대안으로 Dependency Injection으로 불리는)기능의 기초를 제공한다. BeanFactory [http://www.springframework.org/docs/api/org/springframework/beans/factory/BeanFactory.html]는 잠재적으로 어떤 종류의 저장 기능을 사용하여 어떤 성질의 bean을 관리하는 향상된 설정 기법을 제공한다. ApplicationContext [http://www.springframework.org/docs/api/org/springframework/context/ApplicationContext.html]는 BeanFactory(또는 하위클래스)의 가장 상위에 빌드하고 향상된 점 중에서도 Spring AOP기능의 좀더 쉬운 통합, 메시지 자원 핸들링(국제화내에서 사용하기 위한), 이벤트 위임, ApplicationContext와 옵션적으로 부모 컨텍스트를 생성하기 위한 선언적인 기법, WebApplicationContext와 같은 애플리케이션 레이어 특정 컨텍스트를 사용하는 것과 같은 다른 기능을 추가한다.

짧게 말하면 ApplicationContext가 그것들중 몇몇 좀더 J2EE이고 기업중심인 것을 위한 향상된 기능을 추가하는 반면에 BeanFactory는 설정 프레임워크와 기본적인 기능을 제공한다. 일반적으로 ApplicationContext는 BeanFactory의 완벽한 수퍼셋(superset)이고 BeanFactory기능과 행위의 어떤 설명은 ApplicationContexts에 잘 적용이 되도록 검토되어야 한다.

사용자들은 때때로 BeanFactory나 ApplicationContext중 어느것이 특정 상황에서 사용하기 위해 가장 적합한지 확신하지 못한다. 대개 J2EE환경내 대부분의 애플리케이션을 빌드할때 ApplicationContext는 일반적으로 선호하는 몇몇 기능의 사용을 위한 좀더 선언적인 접근법을 허용하는 동안 BeanFactory의 모든 기능을 제공하고 기능적인 면에서 이것을 추가하였기 때문에 가장 좋은 선택은 ApplicationContext를 사용하는 것이다.. 당신이 BeanFactory를 사용하기 위해 선택할 중요한 사용 시나리오는 메모리 사용이 가장 큰 관심사항일때(가장 최근 킬로바이트가 계산하는 애플릿과 같은 경우)이고 당신이 ApplicationContext의 모든 기능을 필요로 하지 않을때이다.

이 장은 BeanFactory와 ApplicationContext에 둘다 관련되는 사항들을 다룬다. BeanFactory를 언급할때 당신은 언제나 ApplicaitonContext에도 적용이 된다고 가정할것이다. 기능이 ApplicaitonContext에만 적용될때는 명시적으로 이것만 언급한다.

3.2. BeanFactory 와 BeanDefinitions - 기초

3.2.1. BeanFactory

BeanFactory [http://www.springframework.org/docs/api/org/springframework/beans/factory/BeanFactory.html] 는 인스턴스를 생성하고 설정하고 많은 수의 bean을 관리하는 실질적인 컨테이너이다. 그 bean들은 일반적으로 서로 협력하고 그들 사이의 의존성을 가진다. 그 의존성들은 BeanFactory에 의해 사용된 설정 데이타에 반영된다(비록 몇몇 의존성은 설정 데이타에서 보이지 않을수도 있지만 실행시 bean사이의 프로그램마다 다른 상호작용의 기능이 될것이다.).

BeanFactory는 다중 구현물을 위한 org.springframework.beans.factory.BeanFactory인터페이스에 의해

나타난다. 가장 공통적으로 사용되는 간단한 BeanFactory구현물은 org.springframework.beans.factory.xml.XmlBeanFactory이다. (이것은 ApplicationContexts가 BeanFactory의 하위 클래스임을 알리는 자격을 갖추어야 한다. 그리고 대부분의 사용자는 결국에 ApplicationContext의 각각 다른 형태의 XML을 사용해야만 한다.)

비록 대부분의 시나리오를 위해 BeanFactory에 의해 관리되는 대부분의 모든 사용자 코드가 BeanFactory를 인식하지 못하더라도 BeanFactory는 어떻게 해서든지 인스턴스화되어야 한다. 이것은

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

또는

```
ClassPathResource res = new ClassPathResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

또는

```
ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml", "applicationContext-part2.xml"});
// of course, an ApplicationContext is just a BeanFactory
BeanFactory factory = (BeanFactory) appContext;
```

과 같은 명시적인 사용자 코드를 통해 발생할수 있다.

많은 사용 시나리오를 위한 사용자 코드는 Spring프레임워크가 인스턴스화 한 이후 BeanFactory를 인스턴스화 하지 않을것이다. 예를 들면 웹 레이어는 J2EE웹 애플리케이션의 일반적인 시작 프로세스의 일부처럼 Spring ApplicationContext을 자동적으로 로드하기 위한 지원 코드를 제공한다. 이 선언적인 프로세스는 here에서 언급된다.

BeanFactory의 프로그램마다 다른 조작이 이후에 언급될때까지 다음 부분은 BeanFactory의 설정을 언급하는데 집중할것이다.

BeanFactory설정은 가장 기초적인 레벨, BeanFactory가 관리해야만 하는 하나 이상의 bean의 정의로 구성된다. XmlBeanFactory에서 가장 상위레벨의 beans 요소내 하나 이상의 beans요소로 설정된다.

```
<?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>
```

3.2.2. BeanDefinition

각각의 다른 형태(XmlBeanFactory처럼)의 DefaultListableBeanFactory내 Bean정의는 다음의 상세사항을 포함하는 BeanDefinition객체처럼 표현된다.

☒ 클래스명 : 이것은 bean정의내 서술되는 bean의 실질적인 구현 클래스이다. 어쨌든 bean이 일반적인 생성자를 사용하는 대신에 정적인 factory메소드를 호출함으로써 생성된다면, 이것은 factory클래스의 클래스명이 될것이다.
☒ bean행위적 설정 요소, bean이 컨테이너내에서 어떻게 행위를 하는지의 상태(이를 테면 프로터타입 또는 싱글톤, autowiring모드, 의존성 체크 모드, 초기화(initialization)와 제거(destruction)메소드.)
☒ 새롭게 생성되는 bean내에 셋팅할 생성자 인자와 프라퍼티 값. 예제는 Connection pool(프라퍼티나 생성자 인자처럼 명시되는)이나 pool크기 제한을 관리하는 bean내에서 사용하는 Connection의 수가 될것이다.
☒ 이것의 작업을 하기 위해 필요한 bean. 이를 테면 collaborators(프라퍼티나 생성자 인자처럼 명시되는). 의존성에 의해 호출될수 있다.

위에서 나열된 개념은 bean정의를 구성하는 요소의 집합을 직접적으로 번역한다. 그러한 요소그룹의 몇몇은 각각에 대한 좀더 상세한 문서 링크를 포함해서 밑에 나열했다.

Table 3.1. Bean 정의 설명

특징	좀더 상세한 정보
class	Section 3.2.3, “bean 클래스”
id 와 name	Section 3.2.4, “bean 구분자 (id 와 name)”
싱글톤 또는 프로토타입	Section 3.2.5, “싱글톤이나 비-싱글톤(non-singleton)”
생성자 인자들	Section 3.3.1, “bean프라퍼티와 협력자(collaborators) 셋팅하기”
bean 프라퍼티	Section 3.3.1, “bean프라퍼티와 협력자(collaborators) 셋팅하기”
autowiring 모드	Section 3.3.6, “Autowiring 협력자”
의존성 체크 모드	Section 3.3.7, “의존성을 위한 체크”
초기화 메소드	Section 3.4.1, “Lifecycle 인터페이스”
제거(destruction) 메소드	Section 3.4.1, “Lifecycle 인터페이스”

bean정의는 실질적인 org.springframework.beans.factory.config.BeanDefinition인터페이스와 다양한 하위 인터페이스 그리고 구현물들에 의해 표현된다. 어쨌든 이것은 대부분의 사용자 코드가 BeanDefinition와 함께 작동할 것 같지는 않다.

bean을 생성하는 방법에 대한 정보를 포함하느 bean정의 외에도 bean factory는 존재하는 bean인스턴스를 등록하는것을 허용할수 있다. DefaultListableBeanFactory는 org.springframework.beans.factory.config.ConfigurableBeanFactory인터페이스에 의해 정의된 registerSingleton메소드를 통해 이것을 지원한다. 전형적인 애플리케이션은 단순히 bean 정의를 가지고 작동한다.

3.2.3. bean 클래스

class 속성은 일반적으로 필수(두 예외를 위한 Section 3.2.3.3, “인스턴스 factory메소드를 통한 Bean 생성” 와 Section 3.5, “추상 그리고 자식 bean정의” 을 보라.)이고 두가지 목적중 하나를 위해

사용된다. BeanFactory자체가 생성자를 호출함(new를 호출하는 자바코드와 같은)으로써 bean을 직접적으로 생성하는 많은 공통적인 상황에서 class속성은 생성될 bean의 클래스를 명시한다. BeanFactory가 bean을 생성하기 위한 클래스의 정적인 factory 메소드를 호출하는 다소 적은 공통적인 상황에서는 class속성은 정적 factory메소드를 포함하는 실질적인 클래스를 명시한다. (정적 factory메소드로 부터 반환되는 bean의 타입은 같은 클래스이거나 완전히 다른 클래스일것이다. 이것은 문제가 아니다.)

3.2.3.1. 생성자를 통한 Bean 생성

생성자를 이용한 접근법을 사용하여 bean을 생성할때 모든 일반적인 클래스는 Spring과 Spring에 호환되는 것에 의해 사용가능하다. 생성된 클래스는 어떤 특정 인터페이스를 구현하거나 특정 형태로 작성될 필요가 없다. bean클래스를 명시하는 것만으로도 충분하다.어쨌든 특정 bean을 위해 사용하기 위한 어떤 타입의 IoC에 의존한다. 당신은 아마도 디폴트(빈) 생성자가 필요할것이다.

추가적으로 BeanFactory는 자바빈을 관리하는데 제한을 두지 않는다. 이것은 또한 당신이 관리하고자 하는 사실상의 어떤 클래스를 관리하는것이 가능하다. Spring을 사용하는 대부분의 사람들은 BeanFactory내 실질적인 자바빈(프라퍼티둬 디폴트 생성자와 선호하는 ssetter와 getter메소드를 가지는)을 가지는 것을 선호한다. 하지만 이것은 BeanFactory내 색다른 bean스타일이 아닌 클래스를 가지는것이 가능하다. 만약 예를 들어, 당신이 자바빈 애플리케이션을 따르지 않는 예전(legacy) connection pool을 사용할 필요가 있다면 걱정하지 말라. Spring은 이것을 잘 관리할수 있다.

XmlBeanFactory을 사용하면 당신은 다음처럼 당신의 bean클래스를 명시할수 있다.

```
<bean id="exampleBean"
      class="examples.ExampleBean"/>
<bean name="anotherExample"
      class="examples.ExampleBeanTwo"/>
```

생성자를 위한 인자, 또는 객체 인스턴스가 생성된후 객체 인스턴스의 프라퍼티를 셋팅하기 위한 기법은 짧게 언급될것이다.

3.2.3.2. 정적 factory메소드를 통한 Bean 생성

정적인 factory메소드를 포함하는 클래스를 명시하는 class 속성에 따라 정적인 factory메소드를 사용하여 생성되기 위한 bean을 정의할때 factory-method라는 이름의 다른 속성은 factory메소드 자체의 이름을 명시할 필요가 있다. Spring은 만약 생성자를 통해 일반적으로 생성되는 것처럼 처리되는 시점으로 부터 이 메소드(나중에 언급되는 것처럼 인자의 목록을 가지는)를 호출하고 시스템에서 살아있는(live)객체를 돌려받는 것이 가능하도록 기대한다. bean정의와 같은 것을 위해 사용하는것은 예전(legacy)코드내 정적 factory를 호출하는것이다.

다음은 factory메소드를 호출함으로써 생성되기 위한 bean을 명시하는 bean정의의 예제이다. 정의는 반환되는 객체의 타입(클래스)를 명시하지 않을뿐 아니라 클래스는 factory메소드를 포함한다는것을 알라. 예를 들어 createInstance는 정적 메소드가 되어야만 한다.

```
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance"/>
```

factory메소드를 위한 인자나 factory로 부터 반환된 뒤 객체 인스턴스의 프라퍼티를 셋팅하는 것을 지원하기 위한 기법은 짧게 언급될것이다.

3.2.3.3. 인스턴스 factory메소드를 통한 Bean 생성

bean을 생성하기 위한 정적 factory메소드를 사용하는 것과 흡사한것은 factory로 부터 존재하는 bean의 factory메소드가 새로운 bean을 생성하기 위해 호출되는 인스턴스 (정적이 아닌) factory메소드 사용이다.

이 기법을 사용하는것은 class 속성이 빈값으로 남아야만 하고 factory-bean 속성은 최근 bean의 이름이나 factory메소드를 포함하는 조상(ancestor) bean factory를 명시해야만 한다. factory메소드 자체는 factory-method 속성을 통해 셋팅될것이다.

다음은 예제이다.

```
<!-- 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"/>
```

비록 bean프라퍼티를 셋팅하기 위한 기법이 여전히 논의되어도 이 접근법의 하나의 함축은 factory bean자체가 컨테이너에 의해 의존성 삽입(Dependency Injection)을 통해 관리되고 설정될수 있다.

3.2.4. bean 구분자 (id 와 name)

모든 bean은 하나 이상의 id(구분자, 이름이라고 불리는:그 용어들은 같은것을 지칭한다.)를 가진다. 그 id들은 bean을 생성한 BeanFactory 나 ApplicationContext내에서 유일해야만 한다. bean은 거의 대부분 하나의 id만을 가진다. 하지만 bean이 한개의 id보다 많은수를 가진다면 나머지 것들은 기본적으로 별칭으로 간주될수 있다..

XmlBeanFactory(여러가지의 ApplicationContext를 포함해서)에서 당신은 bean id를 명시하기 위해 id 나 name속성을 사용하고 적어도 하나의 id는 그 속성중 하나나 둘다에 명시해야만 한다. id 속성은 당신에게 하나의 id를 명시하는것을 허용하고 실제 XML요소의 ID속성처럼 XML DTD(정의문서)내 표시된다. 파서는 다른 요소점이 이 요소로 돌아갈때 몇몇 별개의 유효성체크를 하는것이 가능하다. 그러한 것처럼 bean id를 명시하기 위한 선호되는 방법이 있다. 어쨌든 XML스펙은 XML ID들내 지역적인(legal) 문자들을 제한한다. 이것은 언제나 제약인것은 아니지만 만약 당신이 그러한 문사들중 하나를 사용할 필요가 있거나 bean에 대한 다른 별칭을 소개하길(introduce) 원한다면 당신은 그렇게 하거나 name 속성을 통해 하나이상의 bean id(,으로 구분되는) 또는 세미콜론(;)을 명시하는것으로 대신한다.

3.2.5. 싱글톤이나 비-싱글톤(non-singleton)

bean은 두가지 모드(싱글톤또는 비-싱글톤)중 하나로 배치되기 위해 정의된다. 후자(비-싱글톤)는 프로토타입이라도도 불린다. 비록 용어가 정확하게 적합한것이 아닌 느슨하게 대충 사용이 되더라도). bean이 싱글톤일때 bean의 오직 하나의 공유인스턴스만이 관리될것이고 id를 가진 bean을 위한 모든 요청이나 bean정의에 되는 id가 반환되는 하나의 특정 bean인스턴스를 야기한다.

비-싱글톤, bean배치의 프로토타입 모드는 특정 bean이 수행하기 위한 요청에 대해 매번 새로운 bean인스턴스의 생성이라는 결과를 만든다. 이것은 예를 들어 각각의 사용자가 비의존적인 사용자 객체나

유사한 어떤것이 필요한 상황에 이상적이다.

bean은 당신이 디폴트로 싱글톤모드로 배치된다. 타입을 비-싱글톤으로 변경함으로써 그것을 잊지말라. bean을 위한 각각의 요청은 새롭게 생성된 bean을 초래하고 이것은 당신이 실질적으로 원하는것이 되지는 않을것이다. 그래서 오직 절대적으로 필요할때 모드를 프로터타입으로 변경하라.

아래의 예제에서, 두개의 bean은 하나는 싱글톤으로 정의되고 다른 하나는 비-싱글톤(프로토타입)으로 선언되었다. yetAnotherExample이 오직 한번만 생성되는 동안 exampleBean은 각각 생성되고 매번 클라이언트는 이 bean을 위해 BeanFactory를 요청한다. 정확히 같은 인스턴스를 위한 참조는 이 bean을 위한 각각의 요청에 반환된다.

```
<bean id="exampleBean"
      class="examples.ExampleBean" singleton="false"/>
<bean name="yetAnotherExample"
      class="examples.ExampleBeanTwo" singleton="true"/>
```

메모: bean을 프로토타입모드내에서 배치할때 bean의 생명주기는 미세하게 변경한다. 정의에 의해 Spring은 이것이 생성된 후에는 비-싱글톤/프로토타입 bean의 완벽한 생명주기를 관리할수 없다. 이것은 더 이상 놓치지 않는 클라이언트와 컨테이너에게 주어진다.당신은 'new' 연산(operator)을 위한 교체품처럼 비-싱글톤/프로토타입 bean에 대하여 얘기할때 Spring의 역활에 대해 생각할수있다. 어떠한 생명주기 양상은 클라이언트에 의해 다루어질 시점을 지난다. BeanFactory내 bean의 생명주기는 Section 3.4.1, "Lifecycle 인터페이스" 에서 좀더 상세하게 언급된다.

3.3. 프라퍼티, 협력자(collaborators), autowiring 과 의존성 체크

3.3.1. bean프라퍼티와 협력자(collaborators) 셋팅하기

IoC(Inversion of Control)는 이미 Dependency Injection처럼 간주되고 있다. 기본적인 원칙은 오직 생성자 인자, factory메소드를 위한 인자 또는 factory메소드로부터 생성되거나 반환된 후 객체 인스턴스에 셋팅하는 프라퍼티를 통해 bean이 그것들의 의존성(이를 테면, 그것들이 함께 작동하는 다른 객체들)을 명시한다는 것이다. 그 다음, 이것이 bean을 생성할때 그러한 의존성을 실질적으로 삽입하기 위한 컨테이너의 책임이다. 이것은 근본적으로 bean인스턴스화의 역전(inverse - 나아가 Inversion of Control)이거나 클래스의 직접적인 생성을 사용하여 그것의 의존성을 위치시키는 것이다. 또는 서비스 위치자(Locator) 패턴처럼 어떤것이다. 우리가 의존성 삽입의 장점에 정성을 들이지 않을동안 이것은 코드가 좀더 깔끔하게 되고 좀더 높은 디커플링 등급에 도달하는 것이 bean이 그것들의 의존성을 보지 않을때 좀더 쉽게 되도록 하는 명백한 사용법이 된다. 하지만 그것들과 함께 제공되고 추가적으로 의존성이 위치하는 곳과 실질적인 타입이 무엇인지 알지 않는다.

앞 단락에서 알아본것처럼 Inversion of Control/의존성 삽입은 두가지 큰 종류가 존재한다.

- ☒ setter-기반 의존성 삽입은 당신의 bean을 인스턴스화 하기 위한 인자 없는 생성자나 인자 없는 정적 factory메소드를 호출한 후에 당신의 bean의 setter를 호출함으로써 구체화된다. BeanFactory내 명시된 bean은 setter-기반 의존성 삽입을 사용하는 것이 신뢰할수 있는(true) 자바빈이다. Spring은 많은 수의 생성자 인자가 다루기 어렵고, 몇몇 프라퍼티가 선택사항일때 유별나기 때문에 대개 setter-기반 의존성 삽입의 사용을 지지한다.
- ☒ 생성자-기반 의존성 삽입은 각각 협력자(collaborator)나 프라퍼티를 표현하는 많은 수의 인자를 가진 생성자를 호출함으로써 구체화된다. 추가적으로 bean을 생성하기 위해 특정 인자를 가진 정적 factory메소드를 호출하는 것은 대부분 동일하게 간주될수 있고 이 글의 나머지는 생성자를 위한 인자와 정적 factory메소드를 위한 인자를 눈여겨 볼것이다. 비록 Spring이 대부분의 경우를 위해

setter-기반 의존성 삽입의 사용을 지지한다고 하더라도 당신이 setter없이 다중 인자를 가진 생성자만 제공하는 이미 존재하는 bean을 사용하길 바랄지도 모르기 때문에 생성자-기반 접근법또한 완벽하게 지원한다. 추가적으로 좀더 간단한 bean을 위해 몇몇 사람들은 bean이 유효하지 않은 상태에서 생성이 될수 없다는 것을 확신하는 의미처럼 생성자 접근법을 선호한다.

BeanFactory는 이것이 관리하는 bean으로 의존성을 삽입하기 위한 이러한 형태 둘다 지원한다(이것은 사실 몇몇 의존성이 생성자 접근법을 통해 이미 제공된 후 setter-기반으로 의존성을 삽입하는 것들 지원한다.). BeanDefinition의 형태로 들어오는 의존성을 위한 설정은 프라퍼티를 하나의 포맷에서 다른 포맷으로 변환하는 방법을 알기 위한 자바빈 PropertyEditors와 같이 사용된다. 여기저기 전달되는 실제 값들은 PropertyValue객체의 형태로 수행된다. 어쨌든 Spring의 대부분의 사용자는 직접적으로 이 클래스를 다루지는 않을것이지만(이를테면 프로그램에 따라 다르게) XML정의 파일은 내부적으로 그러한 클래스의 인스턴스로 변환될것이고 전체 BeanFactory 나 ApplicationContext를 로드하기 위해 사용된다.

Bean의존성 해석은 일반적으로 다음처럼 발생한다.

1. BeanFactory는 모든 bean을 서술하는 설정으로 생성되고 초기화된다. 대부분의 Spring사용자는 XML형태의 설정 파일을 지원하는 BeanFactory 나 ApplicationContext 종류를 사용한다.
2. 각각의 bean은 프라퍼티, 생성자 인자, 또는 보통의 생성자 대신에 사용되는 정적 factory메소드를 위한 인자의 형태로 표현되는 의존성을 가진다. 이러한 의존성은 bean이 실질적으로 생성되었을때 bean에 제공될것이다.
3. 각각의 프라퍼티또는 생성자의 인자는 셋팅하기 위한 값의 실질적인 정의이거나 BeanFactory내에서 다른 bean에 대한 참조이다. ApplicationContext의 경우 참조는 부모 ApplicationContext내 bean에 대한 것이 될수 있다.
4. 각각의 프라퍼티와 생성자의 인자는 이것이 명시하는 어떠한 타입에서 프라퍼티나 생성자의 인자의 실질적인 타입으로 변환될수 있어야만 하는 값이다. 디폴트에 의해 Spring은 문자열로 제공되는 값에서 int, long, String, boolean, 등등과 같은 모든 내장 타입으로 변환할수 있다. 추가적으로 XML기반 BeanFactory 형태에 대해서 이야기 할때 그것들은 List, Map, Set 그리고 프라퍼티 collection타입들을 정의하기 위한 내장 지원을 가진다. 추가적으로 Spring은 문자열 값에서 다른, 임의의 타입으로 변환될수 있는 자바빈 PropertyEditor정의를 사용한다.(당신은 당신 자신의 사용자 지정 타입으로 변환할수 있도록 하는 당신 자신의 PropertyEditor정의를 가진 BeanFactory를 제공할수 있다. PropertyEditors에 대한 좀더 상세한 정보와 사용자 지정 PropertyEditors를 직접 추가하는 방법은 Section 3.9, “추가적인 사용자지정 PropertyEditors 등록하기” 에서 찾을수 있다.). bean프라퍼티 타입이 자바클래스 타입일때 Spring은 당신에게 클래스명인 문자열값 같은 프라퍼티를 위한 값을 명시하도록 허용하고 내장된 ClassEditor PropertyEditor는 실질적인 클래스 인스턴스를 위한 클래스명을 변환하는것을 다룰것이다.
5. Spring이 BeanFactory가 생성될때 bean참조가 유효한 bean에 실질적으로 참조(이를 테면 참조될 bean은 BeanFactory내 명시되거나 ApplicationContext의 경우 부모 컨텍스트)하는 프라퍼티를 체크하는것을 포함해서 BeanFactory내 각각의 bean의 설정을 체크하는 것을 구체화하는것은 중요하다. 어쨌든 bean프라퍼티 자체는 bean이 실질적으로 생성이 될때까지 셋팅되지 않는다. 싱글톤이고 미리 인스턴스화(ApplicationContext내 싱글톤 bean과 같은)되기 위한 bean을 위해 생성은 BeanFactory가 생성될때마다 발생하지만 반면에 이것은 bean이 요청될때만 발생한다. bean이 실질적으로 생성되었다면 이것은 의존성과 의존성의 의존성이 생성되고 할당되는것처럼 생성되기 위한 다른 bean의 그래프(graph)를 잠재적으로 야기할 것이다.
6. 당신은 일반적으로 적절한것을 하기 위해 Spring을 신뢰할수 있다. 이것은 BeanFactory가 로드되는 시점에 존재하지 않는 bean과 순환적인 의존성에 대한 참조를 포함해서 설정사항을 다룰것이다. 이것은 bean이 실질적으로 생성될때 가능한 늦게 실질적으로 프라퍼티를 셋팅하고 의존성을 해석(이를 테면 필요하다면 그러한 의존적인것을 생성한다.)한다. 이것은 정확하게 로드되는 BeanFactory가 bean이나 이것의 의존성중 하나를 생성할때 문제가 발생한다면 당신이 bean을 요청할때 나중에 예외를 생성할수 있다는 것을 의미한다. 이것은 예를 들어 bean이 잃어버리거나 유효하지 않은 프라퍼티의 결과처럼 예외를 던진다면 발생할수 있다. 몇몇 설정사항의 잠재적으로

늦은(delayed) 가시성(visibility)은 디폴트에 의해 ApplicationContext이 싱글톤 bean을 미리 인스턴스화하기 때문이다. 그것들이 실질적으로 필요하기 전에 그 bean을 생성하기 위한 몇몇 선행 시간과 메모리의 비용에서 당신은 나중이 아닌 ApplicationContext이 생성될때 설정사항에 대해 찾는다. 만약 당신이 바란다면 당신은 이 디폴트 행위를 오버라이드 할수 있고 늦은 로드(미리 인스턴스화되는것이 아닌)를 위한 싱글톤 bean을 셋팅할수 있다.

몇몇 예제들

첫번째 setter-기반 의존성 삽입을 위해 BeanFactory를 사용하는 예제. 아래는 몇몇 bean정의를 명시하는 XmlBeanFactory설정 파일의 작은 일부이다. 다음은 실질적으로 핵심적인 bean자체를 위한 코드이다. 선호하는 setter가 선언된것을 보자.

```
<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"/>
```

```
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;
    }
}
```

당신이 볼수 있는 것처럼 setter는 XML파일내 명시되는 프라퍼티에 대응하기 위해 선언된다. (XML파일의 프라퍼티는 RootBeanDefinition로부터 PropertyValues객체와 직접적으로 관련된다.)

IoC타입 3(생성자-기반 의존성 삽입)을 위해 BeanFactory를 사용하는 예제. 아래는 생성자의 인자와 실질적인 bean코드를 명시하는 XML설정의 작은 조각이다. 생성자를 보자.

```
<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"/>
```

```
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;
    }
}
```

당신이 볼수 있는것처럼 bean정의내 명시되는 생성자의 인자는 ExampleBean의 생성자를 위한 인자처럼 전달될것이다.

지금 생성자를 사용하는것 대신에 사용되는 것들의 다양한 종류를 검토해보자. Spring은 객체의 인스턴스를 반환하기 위해 정적 factory메소드를 호출하는것을 말한다.

```
<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"/>
```

```
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;
    }
}
```

정적 factory메소드를 위한 인자는 constructor-arg요소를 통해 제공된다. 생성자가 실질적으로 사용되는 것처럼 정확하게 같다. 그 인자들은 선택사항이다. 물론 정적 factory메소드를 포함하는 클래스와 같은 타입이 되지않을 factory메소드에 의해 반환될 클래스의 타입을 구체화하는것은 중요하다. 앞에서 언급된 인스턴스(정적이 아닌) factory메소드는 기본적으로 동일한 형태(class속성 대신에 factory-bean속성의 사용을 제외하고)로 사용되기 때문에 여기서는 상세하기 다루지 않을것이다.

3.3.2. 생성자의 인자 분석

생성자의 인자 분석 대응(matching)은 인자타입을 사용할때 발생한다. 다른 bean이 참조될때 타입은 알려지고 대응은 발생한다. <value>true<value>와 같은 간단한 타입이 사용될때 Spring은 값의 타입을 결정할수 없고 도움 없이는 타입에 의해 대응(match)할수 없다. 두개의 부분을 위해 사용된 다음의 클래스를 검토하라.

```
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;
    }
}
```

3.3.2.1. 생성자의 인자 타입 대응(match)

위 시나리오는 type 속성을 사용하여 생성자의 인자 타입을 명확하게 명시함으로써 간단한 타입으로의 타입 매치를 사용할수 있다. 예를 들면.

```
<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>
```

3.3.2.2. 생성자의 인자 인덱스

생성자의 인자는 index 속성을 사용하여 명확하게 명시된 인덱스를 가질수 있다. 예를 들면.

```
<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>
```

여러개의 간단한 값들의 모호한 문제를 푸는것에 더하여 인덱스를 명시하는 것은 생성자가 같은 타입의 두개의 인자를 가지는 모호함의 문제도 해결한다. 인덱스는 0 부터 시작된다는것에 주의하라.

생성자의 인자 인덱스를 명시하는것은 생성자 IoC를 수행하는 방법이 선호된다.

3.3.3. bean프라퍼티와 상세화된 생성자의 인자

앞 부분에서 언급된것처럼 bean프라퍼티와 생성자의 인자는 다른 관리빈(협력자), 또는 인라인으로 명시된 값의 참조처럼 명시될수 있다. XmlBeanFactory는 이러한 목적을 위해 property 와 constructor-arg내 많은 수의 하위요소타입을 지원한다.

value 요소는 사람이 읽을수 있는 문자열 표현처럼 프라퍼티나 생성자의 인자를 명시한다. 앞서 상세하게 언급된것처럼 자바빈 PropertyEditors는 java.lang.String로 부터 문자열값을 실질적인 프라퍼티나 인자타입으로 변환하기 위해 사용된다.

```
<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>
```

```
        <property name="username">
            <value>root</value>
        </property>
    </bean>
</beans>
```

null 요소는 null값을 다루기 위해 사용된다. Spring은 프라퍼티를 위한 빈 인자를 빈 문자열처럼 처리한다. 다음은 XmlBeanFactory설정이다.

```
<bean class="ExampleBean">
    <property name="email"><value></value></property>
</bean>
```

email프라퍼티내 결과는 ""으로 셋팅되고 자바코드 exampleBean.setEmail("") 와 동일하다. 특별한 <null> 요소는 아마도 null값을 표시하는데 사용될것이다.

```
<bean class="ExampleBean">
    <property name="email"><null/></property>
</bean>
```

이것은 자바코드 exampleBean.setEmail(null) 와 동일하다.

list, set, map, 그리고 props 요소는 명시하고 셋팅되기 위한 프라퍼티와 자바타입 List, Set, Map, 그리고 Properties 의 인자를 허용한다.

```
<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>
```

map의 값이나 set값은 어느 요소도 될수 있다는 것을 알라.

```
(bean | ref | idref | list | set | map | props | value | null)
```

property 요소내 bean 요소는 BeanFactory내 명시되는 bean을 위한 참조대신에 bean값을 인라인으로 명시하기 위해 사용된다. 인라인 bean정의는 명시되는 어떠한 id로 필요로 하지 않는다.

```
<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>
```

idref 요소는 컨테이너내 다른 bean의 문자열 id 나 name으로 프라퍼티를 셋팅하기 위한 짧고(shorthand) 에러를 검사(error-proof)하는 방법이다.

```
<bean id="theTargetBean" class="...">
</bean>
<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean"/>
    </property>
</bean>
```

이것은 수행시 다음의 조각들과 동일하다.

```
<bean id="theTargetBean" class="...">
</bean>
<bean id="theClientBean" class="...">
    <property name="targetName">
        <value>theTargetBean</value>
    </property>
</bean>
```

첫번째 형태가 두번째를 선호하는 가장 중요한 이유는 idref 태그를 사용하는 것이 Spring이 다른 bean이 실질적으로 존재하는 배치시점에 유효하도록 허용한다는것이다. 두번째 에서 targetName프라퍼티의 클래스는 자기 자신이 유효하도록 강제하고 클래스가 Spring에 의해 가능하면 컨테이너가 실질적으로 배치한 후에 실질적으로 인스턴스화될때 발생할것이다.

추가적으로 만약 참조되는 bean이 같은 XML파일내에 있고 bean이름이 bean id라면 사용될 local 속성은 XML문서를 파싱하는 시점에 좀더 일찍 XML파서가 bean이름을 자체적으로 체크하는것을 허용할것이다.

```
    <property name="targetName">
        <idref local="theTargetBean"/>
    </property>
```

ref 요소는 property정의 요소내에서 허용되는 마지막 요소이다. 이것은 컨테이너나 협력자에 의해 관리되는 다른 bean을 위해 참조되기 위한 명시된 프라퍼티의 값을 셋팅하는데 사용된다. 앞 부분에서 언급했던것처럼 참조되는 bean은 프라퍼티가 셋팅되는 bean의 의존성이 되는것이 검토되고 프라퍼티가 셋팅되기 전에 필요(만약 이것이 싱글톤 bean이라면 이것은 컨테이너에 의해 이미 초기화되었을것이다.)하다면

요구에 의해 초기화될것이다. 모든 참조는 궁극적으로 다른 객체에 대한 참조이지만 다른 객체의 id/name을 명시하는 방법은 3가지가 있다.

ref 태그의 bean 속성을 사용하여 대상 bean을 명시하는것이 가장 일반적인 형태이고 같은 BeanFactory/ApplicationContext(같은 XML파일이든 아니든)나 부모 BeanFactory/ApplicationContext내에서 어떠한 bean에 대한 참조를 생성하는 것을 허용할것이다. bean 속성의 값은 대상 bean의 id 속성이나 name 속성의 값중 하나처럼 같은것이 될것이다.

```
<ref bean="someBean"/>
```

local 속성을 사용하여 대상 bean을 명시하는것은 같은 파일내 타당한 XML id 참조를 위한 XML파서의 기능에 영향을 미친다. local 속성의 값은 대상 bean의 id 속성과 같아야만 한다. XML파서는 대응되는 요소가 같은 파일내에 발견되지 않는다면 에러를 발생시킬것이다. 그런것처럼 만약 대상 bean이 같은 XML파일내 있다면 local 형태를 사용하는 것이 가장 좋은 선택(가능한한 빨리 에러에 대해 알기 위해)이다.

```
<ref local="someBean"/>
```

parent 속성을 사용하여 대상 bean을 명시하는 것은 현재 BeanFactory(나 ApplicationContext)의 부모 BeanFactory(나 ApplicationContext)내 있을 bean을 위해 생성될 참조를 허용한다. parent 속성의 값은 아마 대상 bean의 id 속성이나 name 속성내 값중에 하나와 같을것이고 대상 bean은 최근것을 위해 부모 BeanFactory 나 ApplicationContext내 있어야만 한다. bean참조 형태의 가장 중요한 사용은 몇몇 프록시의 순서대로 부모 컨텍스트내 존재하는 bean을 포장하기 위해 필요할때이고 초기의 객체는 그것을 포장하기 위해 필요하다.

```
<ref parent="someBean"/>
```

3.3.3.1. value와 ref 간략화한(shortcut) 폼

이것은 value이나 bean참조를 설정하기 위해 필요한 공통사항이다. 완전한 형태의 value 와 ref를 사용하는것보다 다소 덜 장황하게 간략화한 몇가지 형태가 존재한다. property, constructor-arg, 그리고 entry 요소 모두 완전한 형태의 value 요소 대신에 사용된 value 속성을 지원한다. 결과로써 다음과 같다.

```
<property name="myProperty">
    <value>hello</value>
</property
```

```
<constructor-arg>
    <value>hello</value>
</constructor-arg>
```

```
<entry key="myKey">
    <value>hello</value>
</entry>
```

는 다음과 동일하다.

```
<property name="myProperty" value="hello"/>
```

```
<constructor-arg value="hello"/>
```

```
<entry key="myKey" value="hello"/>
```

대개 손으로 정의를 작성할때 당신은 다소 덜 장황한 간략화된 형태를 사용하는것을 선호할것이다.

property 와 constructor-arg 요소는 완전한 형태의 내포된 ref 요소 대신에 사용될 유사한 간략화된 ref 속성을 지원한다. 다음과 같다.

```
<property name="myProperty">
    <ref bean="myBean">
</property
```

```
<constructor-arg>
    <ref bean="myBean">
</constructor-arg>
```

는 다음과 동일하다.

```
<property name="myProperty" ref="myBean"/>
```

```
<constructor-arg value="myBean"/>
```

어쨌든 간략화된 형태는 <ref bean="xxx"> 요소와 동일하다. <ref local="xxx"> 를 위한 간략화된 형태는 없다. local ref를 위해 당신은 긴 형태를 사용해야만 한다.

마지막으로 entry 요소는 key-ref 와 value-ref속성의 형태로 map의 키 그리고/또는 값을 명시하기 위한 간략화된 형태를 허용한다. 다음과 같다.

```
<entry>
  <key><ref bean="myKeyBean"/></key>
  <ref bean="myValueBean"/>
</entry>
```

는 다음과 동일하다.

```
<entry key-ref="myKeyBean" value-ref="myValueBean"/>
```

다시 간략화된 형태는 <ref bean="xxx"> 요소와 동일하다. <ref local="xxx"> 를 위한 간략화된 형태는 없다.

3.3.4. 메소드 삽입

대부분의 사용자를 위해 컨테이너내 대부분의 bean은 싱글톤일것이다. 싱글톤 bean이 다른 싱글톤 bean과 협력할 필요가 있거나 비-싱글톤 bean이 다른 비-싱글톤 bean과 협력할 필요가 있을때 다른 것의 프라퍼티가 되기 위한 하나의 bean을 명시하여 이 의존성을 다루는 전형적이고 공통적인 접근법은 꽤 충분하다. 어쨌든 bean생명주기가 다를때 문제가 있다. 비-싱글톤(프로토타입) bean B를 사용할 필요가 있는 싱글톤 bean A가 A의 각각의 메소드 호출을 한다고 해보자. 컨테이너는 싱글톤 bean A를

단지 한번만 생성할것이고 그것의 프러퍼티를 셋팅하기 위한 기회를 오직 한번만 가진다. 그것이 필요할때마다 bean B의 새로운 인스턴스를 가진 bean A를 제공하기 위한 컨테이너를 위한 기회는 없다.

이 문제를 해결하기 위한 하나의 해결법은 몇몇 Inversion of Control을 버리는 것이다. bean A는 BeanFactoryAware를 구현해서 컨테이너(여기에서 언급된것처럼)를 인식할수 있고 이것이 필요할때마다 (새로운) bean B를 위한 getBean("B") 호출을 통해 컨테이너에게 요청하기 위한 프로그램마다 다른 방법을 사용한다. 이것은 bean코드가 인식을 하고 Spring에 커플링이 되기 때문에 대개 바람직한 해결법은 아니다.

BeanFactory의 향상된 기능인 메소드 삽입은 몇몇 다른 시나리오에 따라 깔끔한 형태로 다루어지는 사용 상황을 허용한다.

3.3.4.1. 룩업(Lookup) 메소드 삽입

룩업 메소드 삽입은 컨테이너내 다른 명명된 bean를 룩업하는 결과를 반환하는 컨테이너내 관리빈의 추상및 구현된 메소드를 오버라이드하기 위해 컨테이너의 기능을 적용한다. 룩업은 위에서 언급(비록 싱글톤이 될수 있더라도)된 시나리오마다 비-싱글톤 bean이 될것이다. Spring은 CGLIB 라이브러리를 통해 바이트코드 생성을 사용하여 동적으로 생성된 하위클래스가 메소드를 오버라이드하는것을 통해 이것을 구현한다.

삽입되어야 할 메소드를 포함하는 클라이언트 클래스내에서 메소드 정의는 이 형태로 추상(또는 구현된) 정의가 되어야만 한다.

```
protected abstract SingleShotHelper createSingleShotHelper();
```

만약 메소드가 추상적이지 않다면 Spring은 존재하는 구현물을 간단하게 오버라이드 할것이다. XmlBeanFactory의 경우 당신은 bean정의내 lookup-method요소를 사용해서 컨테이너로부터 특정 bean을 반환하기 위한 이 메소드를 삽입/오버라이드 하도록 Spring에게 지시한다. 예를 들면

```
<!-- 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>
```

myBean처럼 구분되는 bean은 이것이 singleShotHelper bean의 새로운 인스턴스를 필요로 할때마다 그것 자신의 메소드인 createSingleShotHelper을 호출할것이다. bean을 배치하는 사람은 비-싱글톤(그것이 실질적으로 필요한 것이라면)처럼 singleShotHelper를 배치하기 위해 주의해야 한다고 알리는것이 중요하다. 만약 싱글톤(명시적이거나 이 플래그를 위해 디폴트로 true셋팅한 것에 의존하여)처럼 배치되었다면 singleShotHelper의 같은 인스턴스는 매번 반환될것이다 !

룩업 메소드 삽입은 생성자 삽입(생성되는 bean을 위한 선택사항인 생성자의 인자를 제공하는)과 함께 조합될수 있고 또한 setter삽입(생성되는 bean의 프라퍼티를 셋팅하는)과도 조합될수 있다.

3.3.4.2. 임의의 메소드 교체

룩업 메소드 삽입보다 메소드 삽입의 다소 덜 공통적으로 유용한 형태는 다른 메소드 구현물을 가진 관리빈내에 임의의 메소드를 교체하는 기능이다. 사용자는 이 기능이 실질적으로 필요할때까지 이 부분의 나머지(향상된 기능에 대해 언급하는)를 생략할수 있다.

XmlBeanFactory에서 replaced-method 요소는 배치된 bean위해 다른것을 가진 존재하는 메소드 구현물을 교체하기 위해 사용된다. 우리가 오버라이드하길 원하는 메소드 computeValue를 가진 다음의 클래스를 검토하라.

```
...
public class MyValueCalculator {
  public String computeValue(String input) {
    ... some real code
  }

  ... some other methods
}
```

org.springframework.beans.factory.support.MethodReplacer인터페이스를 구현하는 클래스는 새로운 메소드 정의를 제공할 필요가 있다.

```
/** 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 ...;
}
```

원래의 클래스를 배치하고 오버라이드할 메소드를 명시하기 위한 BeanFactory배치 정의는 다음처럼 보일것이다.

```
<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>
```

replaced-method 요소내 하나 이상이 포함된 arg-type 요소는 오버라이드된 메소드의 메소드 시그너처를 표시하기 위해 사용된다. 인자를 위한 시그너처는 메소드가 실질적으로 오버로드되고 클래스내 몇가지 종류가 되는 경우에만 필요하다. 편의상 인자를 위한 문자열 타입은 완전한 형태의 타입명의 일부가 된다. 예를 들면 다음은 java.lang.String과 대응된다.

```
    java.lang.String
    String
    Str
```

많은 수의 인자가 종종 각각의 가능한 선택들 사이에 구별하기 충분하기 때문에 이 간략화된 형태는 인자에 대응될 가장 짧은 문자열을 사용하여 많은 타입을 저장할수 있다.

3.3.5. depends-on 사용하기

대부분의 상황을 위해 bean이 다른 것들의 의존성이라는 사실은 하나의 bean이 다른것의 프라퍼티처럼 셋팅한다는 사실에 의해 간단하게 표현된다. 이것은 XmlBeanFactory내 ref 요소를 가지고 수행한다. 이것의 다양한 종류에서 때때로 컨테이너를 인식하는 bean은 간단하게 주어진 의존성(문자열 값이나 문자열 값과 같은것을 평가하는 idref 요소의 대안을 사용하여)의 id이다. 첫번째 bean은 이것의 의존성을 위해 컨테이너에 프로그램마다 다른 방식으로 요청한다. 어느 경우에나 의존성은 의존적인 bean이전에 초기화된다.

다소 덜 직접적인(예를 들면, 데이터베이스 드라이버 등록과 같은 클래스내 정적인 초기자가 트리거 될 필요가 있을때) bean들 사이의 의존성이 있는 비교적 드물게 발생하는 상황을 위해 depends-on 요소는 이 초기화된 요소를 사용하는 bean이전에 초기화되기 위한 하나 이상의 bean을 명시적으로 강제하기 위해 사용된다.

다음은 예제설정이다.

```
<bean id="beanOne" class="ExampleBean" depends-on="manager">
    <property name="manager"><ref local="manager"/></property>
</bean>

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

3.3.6. Autowiring 협력자

BeanFactory는 협력자 bean들 사이의 관계를 autowire 할수 있다. 이것은 BeanFactory의 내용을 조사함으로써 당신의 bean을 위해 Spring이 자동적으로 협력자(다른 bean)를 분석하는것이 가능하다는 것을 의미한다. autowiring 기능은 5개의 모드를 가진다. Autowiring은 다른 bean이 autowire되지 않는 동안 bean마다 명시되고 몇몇 bean을 위해 가능하게 될수 있다. autowiring을 사용하면 명백하게 많은 양의 타이핑을 줄이고 프라퍼티나 생성자의인자를 명시할 필요를 줄이거나 제거하는것이 가능하다. [1]XmlBeanFactory에서 bean정의를 위한 autowire모드는 bean요소의 autowire 속성을 사용하여 명시한다. 다음의 값이 허용된다.

Table 3.2. Autowiring 모드

모드	설명
no	autowiring이 전혀없다. bean참조는 ref 요소를 통해 정의되어야만 한다. 이 값은 디폴트이고 좀더 큰 제어와 명백함을 주는 협력자를 명시하기 때문에 좀더 큰 배치를 위해 이것이 억제되도록 변경한다. 몇몇 확장을 위해 이것은 시스템의 구조에 대한 문서의 형태이다.
byName	프라퍼티 이름에 의한 Autowiring. 이 옵션은 BeanFactory를 조사하고 autowire될 필요가 있는 프라퍼티와 같은 이름의 bean을 찾는다. 예를 들면 당신이 만약 이름에 의해 autowire하기 위해 셋팅하는 bean정의를 가지고 이것이 master 프라퍼티(이것은 setMaster(...) 메소드를 가진다.)를 포함한다면 Spring은 master라는 이름의 bean정의를 찾을것이고 프라퍼티를 셋팅하기 위해 이것을 사용한다.
byType	BeanFactory내 프라퍼티 타입의 bean이 정확하게 하나 있다면 프라퍼티를 autowire가 되도록 허용한다. 만약 하나 이상이 있다면 치명적인 예외가 던져지고

[1] Section 3.3.1, “bean프라퍼티와 협력자(collaborators) 셋팅하기” 를 보라.

모드	설명
	이것은 bean을 위한 byType autowiring을 사용하지 않는것을 나타낸다. 만약 대응되는 bean이 없다면 아무것도 발생하지 않는다(프라퍼티가 셋팅되지 않는다.). 만약 이 기능에 호감이 가지 않는다면 이 경우에 던져질 에러를 명시하는 dependency-check="objects" 속성값을 셋팅한다.
constructor	이것은 byType와 유사하지만 생성자의 인자에 적용한다. bean factory내 생성자의 인자타입의 bean이 정확하게 하나가 아닐경우 치명적인 에러가 발생한다.
autodetect	bean클래스의 내성을 통해 constructor 나 byType를 선택하라. 만약 디폴트 생성자가 발견된다면 byType는 적용된다.

property 와 constructor-arg내 명싱적인 의존성이 언제나 autowiring을 오버라이드한다는것에 주의하라. Autowire 행위는 모든 autowiring가 완성된후 수행될 의존성 체크와 조합될수 있다.

autowiring 주위의 pros와 cons를 이해하는것이 중요하다. 다음은 autowiring의 몇몇 장점이다.

☒ 이것은 요구되는 설정의 양을 명백하게 감소시킨다.(어쨌든 이 장 어디서든 언급되는 설정 "template"의 사용같은 기법은 여기서도 가치있다.)

☒ 당신의 객체가 발전하는것처럼 그것 자체를 최신식으로 유지하는 설정을 야기한다. 예를 들면 만약 당신이 클래스에 추가적으로 의존성을 추가할 필요가 있다면 그 의존성은 설정을 변경할 필요없이 자동적으로 만족될수 있다. 게다가 배치하는 동안 코드기초가 좀더 안정화가 될때 명시적으로 wiring하기 위한 교체의 옵션이 없이 autowiring을 위해 견고한 경우가 된다.

autowiring 의 몇몇 단점

☒ 명시적인 wiring보다는 좀더 마법적같다. 비록 위 테이블에서 언급된것처럼 Spring은 기대되지 않는 결과를 가지는 모호함과 같은 경우에 추측을 파하기 위해 주의한다. 당신의 Spring관리 객체들 간의 관계는 더 이상 명시적으로 문서화되지 않는다.

☒ wiring정보는 아마도 Spring애플리케이션 컨텍스트로부터 문서를 생성하는 툴을 위해 사용가능하지는 않을것이다.

☒ type에 의한 autowiring은 setter메소드나 생성자의 인자에 의해 명시되는 타입의 하나의 bean정의가 있을때만 작동할것이다. 당신은 어떠한 잠재적인 모호함이 있을경우 명시적인 wiring을 사용할 필요가 있다.

모든 경우에 "틀리다(wrong)" 나 "맞다(right)"가 답은 아니다. 우리는 프로젝트를 통한 일관성(consistency)의 정도(degree)를 추천한다. 예를 들면 autowiring이 대개 사용되지 않을때 이것은 개발자에게 하나또는 두개의 bean정의를 사용하는것에 혼동을 줄지도 모른다.

3.3.7. 의존성을 위한 체크

Spring은 BeanFactory로 배치되는 bean의 분석되지 않은 의존성의 존재를 체크하도록 시도하는 능력을 가진다. 그것들은 bean정의내 그것들을 위한 실제값을 셋팅하지 않거나 autowiring기능에 의해 자동적으로 제공되는 bean의 자바빈 프라퍼티이다.

이 기능은 모든 프라퍼티(또는 특정 타입의 모든 프라퍼티)가 bean에 셋팅되는지 확인하기를 원할때

때때로 유용하다. 물론 많은 경우에 bean클래스는 많은 프라퍼티 또는 모든 사용시나리오를 위해 적용하지 않는 몇몇 프라퍼티를 위한 디폴트 값을 가질것이다. 그래서 이 기능은 제한적으로 사용가능하다. 의존성체크는 autowiring기능처럼 bean단위로 사용가능하거나 사용불가능하다. 디폴트는 의존성을 체크하지 않는 것이다. 의존성체크는 다양한 모드로 다루어질수 있다. XmlBeanFactory에서 이것은 bean정의내 dependency-check속성을 통해 명시되고 다음의 값을 가진다.

Table 3.3. 의존성체크 모드

모드	설명
none	의존성 체크가 없다. 그것들을 위해 명시되는 값이 없는 bean의 프라퍼티가 간단하게 셋팅하지 않는다.
simple	원시타입과 collection(다른 beanㄷ처럼 협력자를 제외한 모든 것)을 위해 수행되는 의존성 체크.
object	협력자를 위해 수행되는 의존성 체크.
all	협력자, 원시타입 그리고 collection을 위해 수행되는 의존성 체크.

3.4. bean의 성질을 커스터마이징하기.

3.4.1. Lifecycle 인터페이스

Spring은 BeanFactory내 당신의 bean 행위를 변경하기 위한 다양한 표시자(marker)인터페이스를 제공한다. 그것들은 InitializingBean 과 DisposableBean를 포함한다. 이 인터페이스를 구현하는 것은 bean에게 초기화와 파괴화(destruction)의 작업을 수행하도록 허용하는 전자를 위해 afterPropertiesSet()을 후자를 위해 destroy()를 호출함으로써 BeanFactory내 결과를 생성한다.

내부적으로 Spring은 이것이 적당한 메소드를 찾고 호출할수 있는 어떠한 표시자(marker) 인터페이스를 처리하기 위해 BeanPostProcessors를 사용한다. 만약 Spring이 특별히 제공하지 않는 사용자 지정 기능이나 다른 생명주기 행위가 필요하다면 당신은 BeanPostProcessor를 구현할수 있다. 이것에 대한 좀더 상세한 정보는 Section 3.7, “BeanPostprocessors로 bean 커스터마이징하기” 에서 찾을수 있다.

모든 다른 종류의 생명주기 표시자(marker)인터페이스는 아래에서 언급된다. 추가물중 하나에서 당신은 Spring이 bean을 어떻게 관리하고 그러한 생명주기 기능들이 당신의 bean의 성질을 어떻게 변경하고 그들이 어떻게 관리되는지 보여주는 다이어그램을 찾을수 있다.

3.4.1.1. InitializingBean / init-method

org.springframework.beans.factory.InitializingBean을 구현하는것은 bean의 필요한 모든 프라퍼티가 BeanFactory에 의해 셋팅된 후 bean에게 초기화작업을 수행하는것을 허용한다. InitializingBean인터페이스는 정확하게 하나의 메소드만 명시한다.

```
 * 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;
```

메모 : 대개 InitializingBean 표시자(marker) 인터페이스의 사용은 제거될수 있다. (그리고 Spring에 코드를 불필요하게 결합한 후 억제된다.). bean정의는 명시되기 위한 일반적인 초기화 메소드를 위한 지원을 제공한다. XmlBeanFactory의 경우, 이것은 init-method 속성을 통해 수행된다. 예를 들면, 다음의 정의처럼.

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

public class ExampleBean {
    public void init() {
        // do some initialization work
    }
}
```

는 다음과 정확하게 같다.

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

하지만 Spring에 코드를 결합하지는 않는다.

3.4.1.2. DisposableBean / destroy-method

org.springframework.beans.factory.DisposableBean를 구현하는것은 BeanFactory가 파괴된(destroyed)것을 포함할때 bean에게 콜백을 얻는 것을 허용한다. DisposableBean인터페이스는 하나의 메소드를 명시한다.

```
    /**
    * 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;
```

메모 : DisposableBean 표시자(marker) 인터페이스의 사용은 제거될수 있다. (그리고 Spring에 코드를 불필요하게 결합한 후 억제된다.). bean정의는 명시되기 위한 일반적인 파괴(destroy) 메소드를 위한 지원을 제공한다. XmlBeanFactory의 경우에, 이것은 destroy-method 속성을 통해 수행된다. 예를 들면, 다음의 정의처럼.

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

public class ExampleBean {
    public void cleanup() {
        // do some destruction work (like closing connection)
    }
}
```

는 다음과 정확하게 같다.

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements DisposableBean {
    public void destroy() {
        // do some destruction work
    }
}
```

하지만 Spring에 코드를 결합하지는 않는다.

중요한 메모 : 프로토타입 모드로 bean을 배치할때, bean의 생명주기는 미세한게 변경된다. 정의에 의해 Spring은 이것이 생성된 이후 non-singleton/prototype bean의 완벽한 생명주기를 관리할수는 없다. 클라이언트와 컨테이너에 주어진 것은 더이상 추적하지 않는다. 당신은 'new' 연산자를 위한 대체물처럼 non-singleton/prototype bean에 대해 이야기 할때 Spring의 역활에 대해 생각할수 있다. 어떤 생명주기 형상은 클라이언트에 의해 다루어질수 있는 지점을 지난다. BeanFactory내 bean의 생명주기는 Section 3.4.1, "Lifecycle 인터페이스" 에서 좀더 상세하게 언급된다. .

3.4.2. 당신이 누구인지 알고 있다.(Knowing who you are)

3.4.2.1. BeanFactoryAware

org.springframework.beans.factory.BeanFactoryAware인터페이스를 구현하는 클래스는 BeanFactory에 의해 생성되었을때 이것을 생성하는 BeanFactory에 대한 참조를 제공한다.

```
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;
}
```

이것은 bean에게 org.springframework.beans.factory.BeanFactory인터페이스를 통하거나 추가적인 기능을 드러내는 이것의 알려진 하위클래스에 대한 참조를 형변환함으로써 프로그램마다 다르게 그것들을 생성한 BeanFactory를 변경하는것을 허용한다. 원래 이것은 다른 bean의 프로그램마다 다른 검색으로 구성된다. 이 기능이 유용할때 이것이 Spring에 코드를 결합하고 Inversion of Control스타일을 따르지 않는 이 후 프라퍼티처럼 bean에 제공되는 협력자가 위치한 곳에 이것이 대개 제거될수 있다.

3.4.2.2. BeanNameAware

만약 bean이 org.springframework.beans.factory.BeanNameAware인터페이스를 구현하고 BeanFactory내 배치된다면 BeanFactory는 이것이 배치된 id의 bean을 알리기 위한 인터페이스를 통해 bean을 호출할것이다. 콜백은 일반적인 bean프라퍼티의 활성화 이후지만 InitializingBean의 afterPropertiesSet 이나 사용자 지정 init-method같은 콜백을 초기화하기 전에 호출될것이다.

3.4.3. FactoryBean

org.springframework.beans.factory.FactoryBean인터페이스는 자체적으로 factory인 객체에 의해 구현되는

것이다. BeanFactory 인터페이스는 3개의 메소드를 제공한다.

☒ Object getObject(): 이 factory가 생성하는 객체의 인스턴스를 반환한다. 인스턴스는 공유될수(이 factory가 싱글톤이나 프로토타입을 반환하는지에 대한 여부에 의존하여) 있다.
☒ boolean isSingleton(): 만약 이 FactoryBean이 싱글톤을 반환한다면 true를 반환하고 다른경우라면 false를 반환한다.
☒ Class getObjectType(): getObject() 메소드에 의해 반환되는 객체 타입이나 타입이 미리 알려지지 않았다면 null을 반환한다.

3.5. 추상 그리고 자식 bean정의

bean정의는 잠재적으로 컨테이너 특정 정보(이를 테면, 초기화 메소드, 정적 factory 메소드명, 등등)와 생성자의 인자와 프라퍼티 값을 포함하는 많은 양의 설정정보를 포함한다. 자식 bean정의는 부모 정의로부터 설정정보를 상속하는 bean정의이다. 이것은 필요하다면 몇몇값을 오버라이드하거나 다른것을 추가할수 있다. 부모와 자식 bean정의를 사용하는것은 잠재적으로 많은 양의 타이핑을 줄일수 있다. 효과적으로 이것은 템플릿형태이다.

프로그램마다 다르게 BeanFactory로 작업을 수행할때 자식 bean정의는 ChildBeanDefinition 클래스에 의해 표현된다. 대부분의 사용자는 XmlBeanFactory와 같은 몇가지내에서 선언적인 설정 bean정의대신에 이 수준에서 그것들과 함께 작동하지는 않을것이다. XmlBeanFactory bean정의에서 자식 bean정의는 이 속성의 값처럼 부모 bean을 명시하는 parent 속성을 사용하여 간단하게 표시될수 있다.

```
<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>
```

자식 bean정의는 아무것도 명시가 되어 있지 않지만 이것을 오버라이드할수 있다면 부모 정의의 bean클래스를 사용할것이다. 후자의 경우 자식 bean클래스는 부모 bean클래스와 호환되어야만 한다. 이를 테면 부모의 프라퍼티 값을 받을수 있어야 한다.

자식 bean정의는 생성자의 인자값, 프라퍼티값과 부모로 부터 상속된 메소드를 새로운 값을 추가하는 선택사항과 함께 상속할것이다. 만약 메소드를 초기화한다면 destroy메소드와/또는 정적 factory메소드는 명시된다. 그것들은 관련된 부모 셋팅을 오버라이드할것이다.

남은 셋팅들은 언제나 자식 정의로부터 가져올것이다.: depends on, autowire mode, dependency check, singleton, lazy init.

위 예제에서 우리는 abstract 속성을 사용하여 추상적으로 부모 bean정의를 명시적으로 표시했다는 것을 알라. 이 경우 부모 정의는 클래스를 명시하지 않는다.

```
<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>
```

부모 bean은 불완전하고 또한 추상적이라고 생각된 이후에는 인스턴스화될수 없다. 정의가 이것처럼(명시적이거나 함축적인) 추상적이라고 생각될 때 이것은 자식 정의를 위한 부모 정의처럼 제공될 순수한 템플릿이나 추상 bean정의처럼 사용가능하다. 그것 자체(다른 bean의 ref프라퍼티를 참조하거나 부모 bean id를 가진 명시적인 getBean()호출을 하여)의 추상적인 부모 bean들을 사용하는것을 시도하면 에러를 보게될것이다. 유사하게도 컨테이너의 내부적인 preInstantiateSingletons 메소드는 추상적이라고 생각되는 bean정의를 완벽하게 무시할것이다.

중요한 메모: 애플리케이션 컨텍스트(간단한 bean factory가 아닌)는 디폴트에 의해 모든 싱글톤으로 미리 인스턴스화될것이다. 그러므로 이것은 만약 당신이 템플릿처럼만 오직 사용되는 경향이 있는 (부모) bean정의를 가지고 이 정의가 클래스를 명시한다면 당신은 abstract속성값을 true로 셋팅해야만 하는 반면에 애플리케이션 컨텍스트는 이것을 실질적으로 미리 인스턴스화할것이라는 것은 중요(적어도 싱글톤 bean을 위해서)하다.

3.6. BeanFactory와 상호작동하기

BeanFactory는 기본적으로 다른 bean과 그것들의 의존성의 등록을 유지하는 향상된 factory능력을 위한 인터페이스에 지나지 않는다. BeanFactory는 당신에게 bean factory를 사용하여 bean정의를 읽고 그것들에 접근하는 것을 가능하게 한다. BeanFactory를 사용할때 당신은 다음처럼 하나를 생성하고 XML형태의 몇몇 bean정의내에서 읽을것이다.

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

getBean(String)를 사용하여 당신은 당신의 bean인스턴스를 가져올수 있다. 당신은 이것을 싱글톤(디폴트)처럼 이것을 명시하여 같은 bean의 참조를 얻거나 singleton을 false로 셋팅한다면 매번 새로운 인스턴스를 얻을것이다. BeanFactory의 클라이언트측 시각은 놀라울정도로 간단한다. BeanFactory 인터페이스는 호출할 클라이언트를 위해 오직 5개의 메소드만을 가진다.

- ☒ boolean containsBean(String): BeanFactory가 bean정의나 주어진 이름에 대응되는 bean인스턴스를 포함한다면 true를 반환한다.
- ☒ Object getBean(String): 주어진 이름하에 등록된 bean의 인스턴스를 반환한다. bean이 어떻게 설정되는지는 BeanFactory설정에 의존한다. 싱글톤과 공유 인스턴스나 새롭게 생성되는 bean은 반환될것이다. BeansException은 bean을 찾을수 없을때(이 경우 이것은 NoSuchBeanDefinitionException이 될것이다.)나 bean을 인스턴스화하거나 준비하는 동안 예외가 발생할때 던져질것이다.
- ☒ Object getBean(String,Class): 주어진 이름하에 등록된 bean을 반환한다. 반환되는 bean은 주어진 클래스로 형변환될것이다. 만약 bean이 형변환될수 없다면 관련 예외(BeanNotOfRequiredTypeException)가 던져질것이다. 게다가 getBean(String) 메소드의 모든 규칙(위에서 본)을 적용한다.
- ☒ boolean isSingleton(String): 주어진 이름하에 등록된 bean정의나 bean인스턴스가 싱글톤이거나 프로토타입인지 아닌지 조사한다. 만약 주어진 이름에 관련된 bean이 발견되지 않는다면 예외(NoSuchBeanDefinitionException)가 던져질것이다.
- ☒ String[] getAliases(String): 만약 bean정의내 어떠한 것도 명시되어 있다면 주어진 bean이름을 위한 별칭을 반환한다.

3.6.1. BeanFactory의 생성물이 아닌 FactoryBean 얻기

때때로 BeanFactory가 생성하는 bean이 아닌 실질적인 FactoryBean인스턴스 자체를 위해 BeanFactory에 요청할 필요가 있다. 이것은 BeanFactory(ApplicationContext을 포함하는)의 getBean 메소드를 호출할때 &를 가진 bean id를 덧붙여서 수행된다. 그래서 id myBean를 가진 주어진 FactoryBean을 위해 BeanFactory의 getBean("myBean")를 호출하는것은 FactoryBean의 생성물을 반환할것이지만 getBean("&myBean")을 호출하는것은 FactoryBean인스턴스 자체를 반환할것이다.

3.7. BeanPostprocessors로 bean 커스터마이징하기

bean 후-처리자는 두개의 콜백메소드로 구성된 org.springframework.beans.factory.config.BeanPostProcessor인터페이스를 구현하는 자바클래스이다. 그러한 클래스가 BeanFactory에 의해 생성되는 각각의 bean인스턴스를 위해 BeanFactory와 함께 후-처리자처럼 등록되었을때 후-처리자는 어떠한 초기화 메소드(afterPropertiesSet와 어떤 선언된 초기화 메소드)가 호출되기 전과 나중에 BeanFactory로 부터 콜백을 얻을것이다. 후-처리자는 콜백을 완벽하게 무시하는것을 포함해서 bean으로 바라는것을 하는데 자유롭다. bean 후-처리자는 표시자(marker) 인터페이스를 체크하거나 프록시로 bean을 포장하는 것과 같은것을 수행한다. 몇몇 Spring 헬퍼(helper)클래스는 bean 후-처리자처럼 구현되었다.

BeanFactory가 ApplicationContext보다 미세할 정도로 다르게 bean 후-처리자를 처리하는것을 아는것은 중요하다. ApplicationContext는 BeanPostProcessor 인터페이스를 구현하고 bean생성하는 factory에 의해 적당히 호출되기 위한 후-처리자처럼 그것들을 등록하는 것으로 배치되는 어떠한 bean을 자동적으로 감지할것이다. 어떤 다른 bean으로 유사한 형태로 후-처리자를 배치 할 필요만 있다. 반면에 명백한 BeanFactory를 사용할때 bean 후-처리자는 다음같은 코드순으로 명시적으로 등록되어야 한다.

```
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
  ...
```

이 수작업(manual) 등록 단계는 편리하지 않고 ApplictionContexts는 BeanFactory의 기능적으로 수퍼셋이기 때문에 bean 후-처리자가 필요할 때 사용되는 ApplicationContext종류가 대개 추천된다.

3.8. BeanFactoryPostprocessors를 가진 bean factory커스터마이징하기

bean factory 후-처리자는 org.springframework.beans.factory.config.BeanFactoryPostProcessor 인터페이스를 구현하는 자바클래스이다. 이것은 생성된 후 전체 BeanFactory를 위한 몇몇 종류의 변경을 적용하기 위해 수동(BeanFactory의 경우)이나 자동(ApplicationContext의 경우)으로 수행된다. Spring은 밑에서 언급되는 PropertyResourceConfigurer 와 PropertyPlaceHolderConfigurer 그리고 이 문서 나중에 언급되는것처럼 다른 bean을 트랜잭션하게 포장하거나 다른 종류의 프록시와 매우 유용한 BeanNameAutoProxyCreator와 같은 많은 수의 미리존재하는 bean factory 후-처리자를 포함한다. BeanFactoryPostProcessor는 사용자지정 편집기(Section 3.9, “추가적인 사용자지정 PropertyEditors 등록하기” 에서 언급되는것처럼)를 추가하기 위해 사용될수 있다.

BeanFactory내에서 BeanFactoryPostProcessor 적용의 처리는 수동이고 이것과 유사할것이다.

```
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);
```

ApplicationContext는 BeanFactoryPostProcessor 인터페이스를 구현하는 것으로 배치되는 어떠한 bean을 감지할것이고 적당한 시점에 bean factory 후-처리자처럼 자동적으로 그것들을 사용한다. 다른 bean을 위한 유사한 형태로 이러한 후-처리자를 배치만하는것이다.

이 수작업(manual) 등록 단계는 편리하지 않고 ApplictionContexts는 BeanFactory의 기능적으로 수퍼셋이기 때문에 bean factory 후-처리자가 필요할 때 사용되는 ApplicationContext종류가 대개 추천된다.

3.8.1. PropertyPlaceholderConfigurer

bean factory 후-처리자처럼 구현된 PropertyPlaceholderConfigurer는 BeanFactory정의로부터 자바프라퍼티 형태의 다른 분리된 파일로 몇몇 프라퍼티값들을 구체화하기 위해 사용된다. 이것은 몇몇 key 프라퍼티(예를 들면 데이터베이스 URL, 사용자명, 비밀번호)를 복잡하거나 핵심이 되는 XML정의파일이나 BeanFactory을 위한 파일을 변경하는 위험없이 커스터마이징하기 위한 애플리케이션을 배치하는것을 사람에게 허용하는데 유용하다.

위치유지자(placeholder)값과 함께 데이터소스가 정의된 BeanFactory정의로부터의 일부를 검토하라.

아래의 예제에서 데이터소스가 명시되어 있고 우리는 외부 프라퍼티파일로부터 몇몇 프라퍼티를 설정할것이다. 실행시 우리는 데이터소스의 몇몇 프라퍼티를 대체할 BeanFactory로 PropertyPlaceholderConfigurer을 적용할것이다.

```
<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>
```

실질적인 값들은 프라퍼티형태로 다른 파일로부터 나온다.

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

BeanFactory와 같이 이것을 사용하기 위해 bean factory 후-처리자는 수동으로 수행된다.

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));
cfg.postProcessBeanFactory(factory);
```

ApplicationContexts는 BeanFactoryPostProcessor를 구현하는 그것들에 배치되는 bean을 자동적으로 인식하고 적용하는 것이 가능하다는 알라. 이것은 여기서 서술된것처럼 그것들을 의미한다. PropertyPlaceholderConfiguer를 적용하는 것은 ApplicationContext을 사용할때보다 좀더 편리하다. 이러한 이유로 이것이나 다른 bean factory 후-처리자를 사용하길 바라는 사용자는 BeanFactory대신에 ApplicationContext을 사용하는것이 추천된다.

PropertyPlaceHolderConfigurer는 당신이 명시한 프라퍼티파일내에서만 프라퍼티를 찾지는 않는다. 하지만 만약 당신이 사용하길 시도하는 프라퍼티를 찾을수 없다면 자바 시스템 프라퍼티에 대해 체크한다. 이 행위는 설정자의 systemPropertiesMode 프라퍼티를 셋팅함으로써 커스터마이징될수 있다. 이것은 3개의 값을 가진다. 언제나 오버라이드 하도록 설정하는 하나와 결코 오버라이드하지 않는 하나, 프라퍼티가 정의된 프라퍼티파일내 찾을수 없을때만 단지 오버라이드하는 것이 있다. 좀더 많은 정보를 위해서는 PropertiesPlaceholderConfigurer를 위한 JavaDoc를 보라.

3.8.2. PropertyOverrideConfigurer

PropertyOverrideConfigurer, 다른 bean factory 후-처리자는 PropertyPlaceholderConfigurer와 비슷하지만 후자와는 대조적으로 원래의 정의는 디폴트 값을 가지거나 bean프라퍼티를 위한 값을 가질수 없다. 만약 오버라이딩된 파일이 어떤 bean프라퍼티를 위한 항목을 가지지 않는다면 디폴트 컨텍스트 정의가 사용된다.

bean factory정의는 오버라이드된것을 인식하지 않는다. 그래서 이것은 사용될 설정자를 오버라이드한 XML정의 파일을 찾을 때 즉시 명확하지 않다는것에 주의하라. 다중 PropertyOverrideConfigurers가 같은 bean프라퍼티를 위해 다른 값을 정의하는 경우에 가장 마지막의 값이 사용될것이다.(오버라이드기법에 따라.)

프라퍼티 파일 설정 라인은 다음과 같은 형태로 될것이다.

```
beanName.property=value
```

예제 프라퍼티 파일은 다음처럼 보일것이다.

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

예제 파일은 driver 와 url 프라퍼티를 가진 dataSource라고 불리는 것안에서 bean을 포함하는 BeanFactory정의에 대해 사용가능할것이다.

3.9. 추가적인 사용자지정 PropertyEditors 등록하기

문자열 값처럼 bean프라퍼티를 셋팅할때 BeanFactory는 이 문자열을 프라퍼티의 복합(complex)타입으로 형변환하기 위한 표준적인 자바빈 PropertyEditors를 사용한다. Spring은 많은 수의 PropertyEditors를 미리 등록한다. (예를 들면, 문자열처럼 표현되는 클래스명을 실제 클래스객체로 변환하는). 추가적으로 자바의 표준적인 자바빈 PropertyEditor 룩업 기법은 적당하게 명명되기 위한 클래스를 위해 PropertyEditor를 허용하고 자동적으로 발견되기 위한 지원을 제공하는 클래스처럼 같은 패키지내 위치한다.

만약 사용자지정 PropertyEditors을 등록할 필요가 있다면 여기엔 사용가능한 다양한 기법이 있다.

대개 편리하지 않거나 추천되지 않는 대부분의 수동 접근법은 당신이 BeanFactory참조를 가진다고 가정하고 ConfigurableBeanFactory 인터페이스의 registerCustomEditor() 메소드를 간단히 사용한다.

좀더 편리한 기법은 CustomEditorConfigurer 라고 불리는 특별한 bean factory 후-처리자를 사용하는 것이다. 비록 bean factory 후-처리자가 BeanFactory와 함께 반 수동적으로 사용될수 있다 하더라도 이것은 여기에서 언급되는 것처럼 강력하게 추천되기 때문에 내포된 프라퍼티 셋업을 가진다. 이것은 다른 bean에 유사한 방법으로 배치되고 자동적으로 감지되며 적용되는 ApplicationContext과 함께 사용된다.

모든 bean factory와 애플리케이션 컨텍스트는 프라퍼티 전환을 다루기 위한 BeanWrapper라고 불리는 몇가지의 사용을 통해 자동적으로 내장된 프라퍼티 편집기를 사용한다. BeanWrapper이 등록하는 표준적인 프라퍼티 편집기는 다음 장에서 목록화된다 . 추가적으로 ApplicationContexts 또한 오버라이드하거나 애플리케이션 컨텍스트 타입을 명시하기 위한 선호하는 방법으로 자원 룩업을 다루기 위해 추가적인 3가지 편집기를 추가한다. 그것들은 InputStreamEditor, ResourceEditor 그리고 URLEditor이다.

3.10. 존재하는 bean을 위한 별칭을 추가하기 위한 별칭 요소 사용하기.

bean정의 자체에서 당신은 id속성을 통해 명시된 하나의 이름을 위한 조합을 사용하고 alias속성을 통해 많은 수의 다른 이름을 사용하는 방법을 통해 bean을 위한 한개 이상의 이름을 제공할것이다. 이 모든 이름은 같은 bean에 대해 동일한 별칭이 검토될수 있고 애플리케이션내 사용되는 각각의 컴포넌트가 컴포넌트 자체를 위해 명시하는 bean이름을 사용하는 공통적인 의존성을 참조하도록 허용하는것과 같은 몇가지 상황을 위해 유용하다.

bean이 실질적으로 정의될 때 모든 별칭을 명시하는것이 언제나 충분한것은 아니다. 이것은 때때로 어떤곳에 정의되는 bean을 위한 별칭을 소개하는것이 바람직하다. 이것은 단독으로 사용되는 alias 요소를 통해 수행될수 있다.

<alias name="fromName" alias="toName"/>

이 경우, 같은 컨텍스트내 fromName라는 이름의 bean은 toName처럼 참조될수 있는 별칭정의를 나중에 사용할수 있다.

견고한 예제처럼, A 컴포넌트가 XML일부내에서 componentA-dataSource라고 불리는 데이터소스 bean을 정의하는 경우를 검토하라. B 컴포넌트는 XML일부내에서 componentB-dataSource처럼 데이터소스를 참조할것이다. 그리고 주된 애플리케이션 MyApp는 자신만의 XML일부를 정의하고 모든 3개의 일부로부터 마지막 애플리케이션 컨텍스트를 조합한다. 그리고 myApp-dataSource처럼 데이터소스를 참조할것이다. 이 시나리오는 다음의 단독 별칭을 가진 MyApp XML일부를 추가함으로써 쉽게 다루어질수 있다.

<alias name="componentA-dataSource" alias="componentB-dataSource"/> <alias name="componentA-dataSource" alias="myApp-dataSource"/>

지금 각각의 컴포넌트와 주된 애플리케이션은 유일하고 다른 정의와 충돌하지 않도록 보증된 이름을 통해 데이터소스를 참조할수 있다. 게다가 그것들은 같은 bean을 참조한다.

3.11. ApplicationContext에 대한 소개

beans 패키지는 종종 프로그램마다 다른 방식으로 관리와 bean을 변경하기 위한 기초적인 기능을 제공하는

동안 context 패키지는 좀 더 프레임워크 기반 형태로 BeanFactory기능을 강화시키는 ApplicationContext [http://www.springframework.org/docs/api/org/springframework/context/ApplicationContext.html]를 추가한다. 많은 사용자는 이것을 수동으로 생성하지 않을뿐 아니라 J2EE 웹 애플리케이션의 일반적인 시작 프로세스의 일부처럼 자동적으로 ApplicationContext를 시작하기 위한 ContextLoader처럼 지원 클래스에 의존하는 대신에 완벽한 선언적인 형태로 ApplicationContext를 사용할것이다. 물론 이것은 ApplicationContext을 프로그램마다 다르게 생성하는것이 가능하다.

context 패키지는 위한 기초는 org.springframework.context 패키지에 위치한 ApplicationContext 인터페이스이다. BeanFactory 인터페이스에서의 파생물은 BeanFactory의 모든 기능을 제공한다. 좀더 프레임워크 기반의 형태로 작업하는것을 허용하기 위해 레이어와 구조적인 컨텍스트를 사용하라. context 패키지는 다음을 제공한다.

- ☒ MessageSource, i18n 스타일로 메시지에 대한 접근을 제공한다.
- ☒ 자원에 대한 접근, URL이나 파일과 같은 형태.
- ☒ 이벤트 전달(propagation) ApplicationListener인터페이스를 구현하는 bean을 위한
- ☒ 다중(구조적인) 컨텍스트의 로딩, 예를 들어 애플리케이션의 웹 레이어처럼, 각각을 하나의 특정 레이어에 집중될수 있도록 허용하는

ApplicationContext가 BeanFactory의 모든 기능을 포함하기 때문에 , 이것은 메소리 소비가 치명적이고 몇몇 추가적인 킬로바이트가 다른 아마도 애플릿과 같은 몇몇 제한된 상황을 위하는 것을 제외하고 BeanFactory에 우선하여 사용되는 것이 추천된다. 다음 부분은 ApplicationContext가 기본적인 BeanFactory기능에 추가한 기능을 언급한다.

3.12. ApplicationContext에 추가된 기능

이전 부분에서 벌써 밝힌것처럼 ApplicationContext는 BeanFactory로부터 이것을 구별하는 두어가지의 기능을 가진다. 우리는 그것들을 하나씩 먼저 알아보자.

3.12.1. MessageSource 사용하기

ApplicationContext 인터페이스는 MessageSource 라고 불리는 인터페이스를 확장해서 메시징(i18n또는 국제화)기능을 제공한다. NestingMessageSource와 함께 구조적인 메시지를 분석하는 능력을 가진다. 여기엔 Spring이 메시지 분석을 제공하는 기초적인 인터페이스가 있다. 여기에 정의된 메소드를 빨리 알아보자.

- ☒ String getMessage (String code, Object[] args, String default, Locale loc): MessageSource로 부터 메시지를 받기 위해 사용되는 기초적인 메소드. 특정 로케일을 위해 발견되는 메시지가 없을 때 디폴트 메시지가 사용된다. 전달된 인자는 표준적인 라이브러리에 의해 제공되는 MessageFormat 기능을 사용해서 대체값처럼 사용된다.
- ☒ String getMessage (String code, Object[] args, Locale loc): 이전 메소드와 기본적으로는 같다. 하지만 한가지가 다르다. 디폴트 메시지가 선언될수 없다. 만약 메시지가 발견될수 없다면, NoSuchMessageException가 던져진다.
- ☒ String getMessage(MessageSourceResolvable resolvable, Locale locale): 위 메소드에서 사용된 모든 프라퍼티는 이 메소드를 통해 사용할수 있는 MessageSourceResolvable 라는 이름의 클래스에 포장된다.

ApplicationContext가 로드될때 이것은 컨텍스트내 정의된 MessageSource bean을 위해 자동적으로 찾는다. bean은 messageSource을 가진다. 만약 그러한 bean이 발견된다면 위에서 언급된 메소드에 대한 모든 호출은 발견된 메시지소스에 위임될것이다. 만약 발견되는 메시지소스가 없다면 같은 이름을 가진 bean을 포함하는 부모를 가진다면 ApplicationContext가 보기를 시도한다. 만약 그렇다면 이것은 MessageSource처럼 그 bean을 사용한다. 만약 메시지를 위한 어떤 소스를 발견할수 없다면 빈

StaticMessageSource는 위에서 정의된 메소드에 호출을 받을수 있기 위해 인스턴스화될것이다.

Spring은 현재 두개의 MessageSource 구현물을 제공한다. 여기엔 ResourceBundleMessageSource 와 StaticMessageSource가 있다. 둘다 메시지를 내포하기 위해 NestingMessageSource을 구현한다. StaticMessageSource는 소스에 메시지를 추가하기 위한 프로그램마다 다른 방법을 제공하지만 거의 사용되지 않는다. ResourceBundleMessageSource는 좀더 흥미롭고 우리가 제공할 예제이다.

```
<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>
```

이것은 당신이 format, exceptions 과 windows라고 불리는 당신의 클래스패스내 정의된 3개의 자원번들을 가진다고 가정한다. ResourceBundles을 통한 메시지를 분석하는 JDK표준적인 방법을 사용하여 메시지를 분석하기 위한 요청이 다루어질것이다. TODO: SHOW AN EXAMPLE

3.12.2. events 전파하기

ApplicationContext내 이벤트 핸들링은 ApplicationEvent 클래스와 ApplicationListener인터페이스를 통해 제공된다. 만약 ApplicationListener 인터페이스를 구현하는 bean이 컨텍스트로 배치된다면 매번 ApplicationEvent는 통지될 bean인 ApplicationContext에 배포된다. 기본적으로 이것은 표준적인 Observer 디자인 패턴이다. Spring은 3가지 표준적인 이벤트를 제공한다.

Table 3.4. 내장된 이벤트

이벤트	설명
ContextRefreshedEvent	ApplicationContext가 초기화되거나 재생(refresh)될때 배포되는 이벤트. 여기서 초기화는 모든 bean이 로드되고 싱글톤은 미리 인스턴스화되며 ApplicationContext는 사용할 준비가 된다는 것을 의미한다.
ContextClosedEvent	ApplicationContext의 close()메소드를 사용하여 ApplicationContext가 닫힐때 배포되는 이벤트. 여기서 닫히는 것은 싱글톤이 없어지는(destroy)되는것을 의미한다.
RequestHandledEvent	웹 특정 이벤트는 HTTP요청이 서비스(이를 테면 요청이 종료될때 after가 배포될것이다.)되는 모든 bean을 말한다. 이 이벤트는 Spring의 DispatcherServlet을 사용하는 웹 애플리케이션에서만 적용가능하다.

사용자정의 이벤트를 구현하는것은 잘 작동될수 있다. ApplicationContext의 publishEvent() 메소드를 간단히 호출하는 것은 당신의 사용자정의 이벤트 클래스가 ApplicationEvent을 구현하는 파라미터를 명시하는 것이다. 이벤트 리스너(listener)는 이벤트를 동시에 받아들인다. 이것은 publishEvent() 메소드는 모든 리스너가 이벤트 처리를 종료할때 까지 블럭된다는것을 의미한다. 게다가 리스너가 이벤트를 받을때 이것은 배포자(publisher)의 만약 트랜잭션 컨텍스트가 사용가능하다면 트랜잭션

컨텍스트내 작동한다.

예제를 보자. 첫번째 ApplicationContext이다.

```
<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>
```

그리고 다음은 실질적인 bean이다.

```
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
        }
    }
}
```

물론 이 특별한 예제는 기본적인 이벤트 기법을 설명하기에는 충분하지만 좀더 나은 방법(아마도 AOP기능을 사용하여)으로 구현될수 있을것이다.

3.12.3. Spring내에서 자원(resources) 사용하기

많은 애플리케이션은 자원에 접근할 필요가 있다. 자원을 파일을 포함할수 있지만 웹페이지나 NNTP 뉴스피드와 같은 것들도 포함할수 있다. Spring은 프로토콜에 비의존적인 방법으로 자원에 접근하는 깔끔하고 투명한 방법을 제공한다. ApplicationContext 인터페이스는 이것을 다루는 메소드(getResource(String))를 포함한다.

Resource클래스는 모든 Resource구현물을 통해 공유되는 두어가지의 메소드를 정의한다.

Table 3.5. Resource 기능

메소드	설명
getInputStream()	자원에 대해 InputStream을 열고 이것을 반환한다.
exists()	자원이 존재하는지 체크하고 존재하지 않는다면 false를 반환한다.
isOpen()	true를 반환하면 다중 스트림(stream)이 이 자원을 위해 열릴수 없다. 이것은 몇몇 자원을 위해 false가 될것이다. 하지만 예를 들어 파일-기반의 자원은 동시에 여러번 읽을수 없다.
getDescription()	자원의 설명(description)을 반환한다. 종종 완전한 경로의 파일명이나 실질적인 URL이 반환된다.

두어가지의 Resource구현물이 Spring에 의해 제공된다. 그것들 모두 자원의 실질적인 위치를 표현하는 문자열값이 필요하다. 문자열값에 기반으로 하여 Spring은 당신을 위해 알맞은 Resource구현물을 자동적으로 선택할것이다. 처음으로 자원을 위해 ApplicationContext를 요청할 때 Spring의 모든것은 당신이 명시하고 어느 접두사로 찾는 자원의 위치를 조사할것이다. ApplicationContext의 구현물에 의존하여 하나 이상의 Resource구현물은 사용가능하다. Resource는 ResourceEditor를 사용하여 설절될때 가장 좋을수 있고 예로써 XmlBeanFactory가 있다.

3.13. ApplicationContext내에서 사용자정의 행위

BeanFactory는 이것(InitializingBean or DisposableBean처럼 표시자(marker) 인터페이스와 같은)에 배치된 bean들의 생명주기를 제어하는 많은 수의 기법을 이미 제공하고 있다. 그것들의 설정은 XmlBeanFactory 설정와 bean 후-처리자내 init-method 와 destroy-method속성과 같이 동등하다. ApplicationContext에서 그것들 모두 작동을 하지만 추가적인 기법은 bean과 컨테이너의 사용자정의 행위를 위해 추가된다.

3.13.1. ApplicationContextAware 표시자(marker) 인터페이스

BeanFactory와 함께 사용가능한 모든 표시자(marker) 인터페이스는 여전히 작동한다. ApplicationContext는 org.springframework.context.ApplicationContextAware를 구현하는 bean인 하나의 추가적인 표시자(marker) 인터페이스를 추가한다. 이 인터페이스를 구현하고 컨텍스트로 배치되는 bean은 인터페이스의 setApplicationContext()를 사용하여 bean의 생성을 콜백될것이다. 그리고 나중에 컨텍스트와 함께 상호작동을 위해 저장될 컨텍스트에 대한 참조를 제공한다.

3.13.2. BeanPostProcessor

org.springframework.beans.factory.config.BeanPostProcessor인터페이스를 구현하는 자바 클래스인 bean 후-처리자는 벌써 언급되었다. 이것은 여기서 언급할 가치가 있다. 후-처리자는 명확한 BeanFactory보다 ApplicationContexts내에서 사용하는것이 좀더 편리하다. ApplicationContexts내에서 위 표시자(marker) 인터페이스를 구현하는 어떤 배치된 bean은 factory내 각각의 bean을 위해 생성시각에 적당하게 호출되도록 bean 후-처리자처럼 자동적으로 감지하고 등록된다.

3.13.3. BeanFactoryPostProcessor

org.springframework.beans.factory.config.BeanFactoryPostProcessor 인터페이스를 구현하는 자바 클래스인 Bean factory 후-처리자는 벌써 언급되었다. 이것은 여기서 언급할 가치가 있다. 그 bean factory 후-처리자는 보통의 BeanFactory내에서 보다 ApplicationContexts내에서 사용되는 것이 좀더 편리하다. ApplicationContexts에서 위 표시자(marker)인터페이스를 구현하는 어느 배치된 bean은 적절한 시각에 호출되기 위해 bean factory 후-처리자처럼 자동적으로 감지된다.

3.13.4. PropertyPlaceholderConfigurer

PropertyPlaceholderConfigurer 는 BeanFactory와 함께 사용되므로 벌써 설명되었다. 이것은 여기서 언급되는것이 가치있다. 이것은 컨텍스트가 자동적으로 어느 bean factory 후-처리자를 인식하고 적용하기 때문에 그것들이 다른 bean처럼 ApplicationContext으로 간단하게 배치될때 ApplicationContext와 함께 사용하는것이 대개 좀더 편리하다. 이것을 수행하기 위한 수동모드의 단계는 필요가 없다.

```
<!-- property placeholder post-processor -->
<bean id="placeholderConfig"
       class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location"><value>jdbc.properties</value></property>
</bean>
```

3.14. 추가적인 사용자정의 PropertyEditors 등록하기

이미 언급된것처럼, 표준 자바빈 PropertyEditors는 프라퍼티를 프라퍼티의 실질적인 복합타입을 위한 문자열처럼 표시되는 값으로 변환하기 위해 사용된다. CustomEditorConfigurer, bean factory 후-처리자, ApplicationContext를 위해 추가적인 PropertyEditors를 위한 지원을 편리하게 추가하기 위해 사용된다.

사용자 클래스인 ExoticType와 프라퍼티처럼 ExoticType 셋을 필요로 하는 다른 클래스인 DependsOnExoticType를 검토해보자.

```
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;
    }
}
```

이것들이 적절히 셋업될 때, 우리는 PropertyEditor는 실제 ExoticType객체로 변환할 문자열처럼 타입

프라퍼티를 할당하는것을 가능하게 하도록 원한다.

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type"><value>aNameForExoticType</value></property>
</bean>
```

PropertyEditor 는 이것과 유사하게 보일수 있다.

```
// 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);
    }
}
```

마지막으로 우리는 필요할 때 처럼 이것을 사용할 ApplicationContext를 가진 새로운 PropertyEditor을 등록하기 위해 CustomEditorConfigurer을 사용한다.

```
<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>
```

3.15. 프라퍼티 표현에서 bean프라퍼티 또는 생성자의 인자를 셋팅하기.

PropertyPathFactoryBean은 주어진 대상 객체의 프라퍼티 경로를 평가하는 FactoryBean이다. 대상 객체는 직접적 또는 bean 이름을 통해 명시될수 있다. 이 값은 프라퍼티값이나 생성자의 인자처럼 다른 bean정의로 사용될수있다..

이것은 다른 bean에 대해 이름에 의해 사용되는 경로의 예제이다.

```
// 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>
```

이 예제에서 경로는 내부 bean에 대해 평가된다.

```
// 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>
```

bean이름이 프라퍼티 경로인 간략화된 형태 또한 있다.

```
// 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"/>
```

이 형태는 bean의 이름내 이것이 경로인 같은 id를 사용할 어떤 참조인 선택사항이 없다는 것을 의미한다. 물론 내부 bean처럼 사용된다면 전부를 참조할 필요는 없다.

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

결과 타입은 실질적인 정의내 명시적으로 셋팅될수 있다. 이것은 대부분의 사용 상황을 위해 필요하지는 않지만 몇몇 사항을 위해서 사용될수 있다. 이 기능을 위한 좀더 다양한 정보를 위해서 JavaDoc를 보라.

3.16. 필드값으로부터 bean프라퍼티 또는 생성자의 인자를 셋팅하기.

FieldRetrievingFactoryBean은 정적이거나 비-정적인 필드값을 가져오는 FactoryBean이다. 이것은 다른 bean을 위해 프라퍼티값이나 생성자의 인자를 셋팅하기 위해 사용될수 있는 public 형태의 정적인 final 상수를 가져오기 위해 전형적으로 사용된다.

staticField 프라퍼티를 사용해서 정적 필드가 나타나는 방법을 보여주는 예제이다.

```
<bean id="myField"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
  <property name="staticField"><value>java.sql.Connection.TRANSACTION_SERIALIZABLE</value></property>
</bean>
```

정적 필드가 bean이름처럼 정의되는 곳의 편리한 사용형태 또한 있다.

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

이것은 bean id내(그래서 다른 bean은 더 긴 이름을 사용하기 위해 참조한다.) 결코 어떠한 선택사항도 없지만 이 형태는 정의하기에 매우 간결하고 id가 bean참조를 위해 명시되지 않은 이후 내부 bean처럼 사용하기 위해 매우 편리하다.

```
<bean id="..." class="...">
  <proprty name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
  </property>
</bean>
```

이것은 또한 JavaDoc에서 언급되는 것처럼 다른 bean의 비-정적인 필드를 접근하는것이 가능하다.

3.17. 다른 메소드를 호출하고 선택적으로 반환값을 사용한다.

이것은 몇몇 다른 클래스가 사용되기 전 몇몇 순차적인 초기화를 수행하기 위해 하나의 클래스내 정적이거나 비-정적인 메소드를 호출하는것이 때때로 필요하다. 추가적으로 이것은 컨테이너내 다른 bean의 메소드 호출 결과나 어느 임의의 클래스의 정적 메소드호출 결과처럼 bean의 프라퍼티를 셋팅하는것이 때때로 필요하다. 이 두가지 목적을 위해 MethodInvokingFactoryBean를 호출하는 헬퍼 클래스는 사용될수 있다. 이것은 정적이나 인스턴스 메소드 호출의 결과인 값을 반환하는 FactoryBean 이다.

우리는 어쨌든 추천된다. 두번째 사용 상황을 위해 이전에 언급된 factory 메소드는 거의 대부분의 선택에 훨씬 좋다.

bean 정의의 예제(XML기반의 BeanFactory정의내)는 몇몇 순차적인 정적 초기화를 강제로 수행하기 위한 이 클래스를 사용한다.

```
<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>
```

bean1을 위한 정의는 첫번째 초기화 강제 초기화(force-init)를 유발할 강제 초기화(force-init) bean을 참조하기 위해 depends-on 속성을 사용하였다. 그리고 bean1이 첫번재 초기화될때 정적인 초기화 메소드를 호출한다.

정적인 factory메소드를 호출하기 위한 이 클래스를 사용하는 bean정의의 예제이다.

```
<bean id="myClass" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="staticMethod"><value>com.whatever.MyClassFactory.getInstance</value></property>
</bean>
```

자바 시스템 프라퍼티에서 얻기 위해 정적 메소드와 그 다음 인스턴스 메소드를 호출하는 예제이다. 어느정도 장황하지만 이것은 작동한다.

```
<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>
```

이것은 factory메소드에 접근하기 위해 대부분 사용되는것이 기대되는것을 알라. 디폴트에 의해 MethodInvokingFactoryBean는 싱글톤 형태로 작동한다. 객체를 생성하는 factory를 위해 컨테이너에 의한 첫번째 요청은 최근과 하위 요청을 위해 캐시되고 반환될 값을 반환하는 명시된 메소드 호출을 야기할것이다. factory의 내부 singleton 프라퍼티는 객체를 요청하는 매번 대상 메소드를 호출하는것을 야기하기 위해 false로 셋팅될수 있다.

정적인 대상 메소드는 정적 메소드가 정의되는 클래스를 명시하는 targetClass를 가지고 정적 메소드이름을 표시하는 문자열을 위해 targetMethod을 셋팅함으로써 명시될수 있다. 대안으로 대상 인스턴스 메소드는 대상 객체처럼 targetObject와 대상객체의 호출을 위한 메소드 이름처럼 targetMethod을 셋팅하여 명시될수 있다. 메소드 호출을 위한 인자는 args 프라퍼티를 셋팅하여 명시될수 있다.

3.18. 하나의 파일로부터 다른것으로 bean정의를 끌어오기

컨테이너 정의를 다중 XML파일로 분류하는것은 종종 유용하다. 그 다음 그러한 XML부분들로부터 설정된 애플리케이션 컨텍스트를 로드하는 하나의 방법은 다중 Resource위치를 가지는 애플리케이션 컨텍스트 생성자를 사용하는 것이다. bean factory를 사용하여 bean정의 리더(reader)는 각각의 파일로 부터 정의를 순서대로 읽기 위해 여러번 사용될수 있다.

대개 Spring팀은 이것이 컨테이너 설정 파일이 다른것과 조합된 사실을 인지할수 없도록 유지하기 때문에 위 접근법을 선호한다. 어쨌든 대안적인 접근법은 하나의 XML bean정의 파일로 부터이다. 하나 이상의 다른 파일로부터 정의를 로드하기 위해 하나 이상의 import 요소의 인스턴스를 사용한다. 어느 import요소는 끌어오기(import)를 수행하는 파일내에서 bean 요소 앞에 위치되어야만 한다. 샘플을 보자.

```
<beans>

  <import resource="services.xml"/>

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

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

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

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

이 예제에서 외부 bean정의는 3개의 파일인 services.xml, messageSource.xml, 과 themeSource.xml로 부터 로드되고 있다. 모든 위치 경로는 끌어오기(import)를 수행하는 정의 파일에 상대적으로 검토된다. 그래서 이 경우 messageSource.xml 과 themeSource.xml이 import파일의 위치 아래의 resources 위치에 있어야만 하는

동안 services.xml은 끌어오기(import)를 수행하는 파일처럼 같은 디렉토리나 클래스패스 위치에 있어야만 한다. 당신이 보는것처럼 앞쪽의 슬래쉬(slash)는 실질적으로 무시된다. 하지만 상대적인 경로를 검토할때 이것은 아마도 슬래쉬(slash)를 전혀 사용하지 않는것이 더 좋다.

끌어오기(import)를 수행중인 파일의 내용은 DTD를 통해 가장 상위레벨 beans요소를 포함하는 완전히 유요한 XML bean정의 파일이어야 한다.

3.19. 웹 애플리케이션으로부터 ApplicationContext생성하기.

프로그램마다 다르게 종종 생성될 BeanFactory에 적대되는 것처럼, ApplicationContexts는 예를 들어 ContextLoader을 사용하여 선언적으로 생성될수 있다. 물론 당신은 ApplicationContext 구현물중 하나를 사용하여 프로그램마다 다르게 ApplicationContext을 생성할수 있다. 첫번째 ContextLoader를 조사해보고 이것의 구현물들을 조사해보자.

ContextLoader는 ContextLoaderListener 와 ContextLoaderServlet의 두가지의 구현물을 가진다. 그것들 모두 같은 기능을 가지지만 리스너(listener)는 서블릿 2.2 호환 컨테이너내에서는 사용될수 없다는 것이 다르다. 서블릿 2.4 스펙이후로 리스너(listener)는 웹 애플리케이션의 시작 후 초기화를 요구한다. 많은 2.3 호환 컨테이너는 이미 이 기능을 구현한다. 이것은 당신이 사용하는 것에 따르지만 모든것은 당신이 아마도 ContextLoaderListener를 선호하는것과 동일하다. 호환성에 대한 좀더 상세한 정보를 위해서는 ContextLoaderServlet을 위한 JavaDoc를 보라.

당신은 다음처럼 ContextLoaderListener을 사용하여 ApplicationContext을 등록할수 있다.

```
<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>

<!-- OR USE THE CONTEXTLOADERSERVLET INSTEAD OF THE LISTENER
<servlet>
    <servlet-name>context</servlet-name>
    <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
-->
```

리스너(listener)는 contextConfigLocation 파라미터를 조사한다. 만약 이것이 존재하지 않는다면 이것은 디폴트로 /WEB-INF/applicationContext.xml을 사용할것이다. 이것이 존재할때, 이것은 미리 정의된 분리자(delimiter-콤마, 세미콜론 그리고 공백)를 사용하여 문자열을 분리하고 애플리케이션 컨텍스트가 검색되는 위치처럼 값을 사용할것이다. ContextLoaderServlet는 말하는것처럼 ContextLoaderListener대신에 사용될수 있다. 서블릿은 리스너(listener)가 하는것처럼 contextConfigLocation 파라미터를 사용할것이다.

3.20. Glue 코드와 좋지않은 싱글톤

애플리케이션내 코드의 대부분은 이것이 생성될때 컨테이너에 의해 자신만의 의존성이 제공되고 컨테이너를 완벽하게 자각하지 못하는 BeanFactory나 ApplicationContext 컨테이너에 코드가 제공되는 의존성 삽입(Dependency Injection-Inversion of Control)을 사용하여 쓰여지는게 가장 좋다. 어쨌든 다른 코드와 함께 묶기 위한 때때로 필요한 코드의 작은 glue레이어를 위해 BeanFactory나

ApplicationContext를 위해 접근하는 싱글톤(또는 준(quasi)-싱글톤 스타일을 위해 때때로 필요하다. 예를 들어 써드파티(third party) 코드가 BeanFactory의 객체를 얻기 위해 이것을 강제로 하게 할수 있는 능력없이 새로운 객체를 직접적으로 생성(Class.forName() 스타일)하도록 시도할수 있다. 만약 써드파티(third party)모드에 의해 생성된 객체가 작은 스텁(stub)나 프록시라면 위임하는 실제 객체를 얻기 위해 BeanFactory/ApplicationContext에 접근하는 싱글톤 스타일을 사용한다. inversion of control은 여전히 코드의 대부분을 위해 달성된다.(객체는 BeanFactory로 부터 나온다.). 게다가 대부분의 코드는 컨테이너나 이것이 접근되는 방법을 자각하지 않는다. 그리고 모든 이익을 가지는 다른 코드로 부터 커플링되지 않은체로 남게된다. EJB는 BeanFactory로 부터 나오는 명확한 자바 구현물 객체를 위해 위임하는 스텁/프록시 접근법을 사용할수 있다. BeanFactory는 이론상 싱글톤이 되지 않는 동안 이것은 비-싱글톤 BeanFactory를 사용하는 각각의 bean을 위해 메모리 사용이나 초기화 시점(Hibernate SessionFactory처럼 BeanFactory내 bean을 사용할 때)의 개념에서 비사실적일수 있다.

다른 예제처럼, 다중 레이어(이를테면, 다양한 JAR파일들, EJB들, 그리고 EAR처럼 패키지된 WAR파일)의 복잡한 J2EE애플리케이션에서 이것 자체의 ApplicationContext정의(구조를 효과적으로 형상화하는)를 가진 각각의 레이어와 함께 가장 상위 구조내 단 하나의 웹 애플리케이션(WAR)이 존재할때 각각의 레이어의 다중 XML정의 파일에서 하나의 복잡한 ApplicationContext를 간단하게 생성하기 위한 접근법이 선호된다. 모든 ApplicationContext종류는 이 형태로 다중 정의 파일로 부터 생성될수 있다. 어쨌든 구조의 가장 상위의 다중의 구성원(sibling) 웹 애플리케이션을 가진다면 이것은 아래쪽의 레이어로부터 대부분 일치하는 bean정의를 구성하는 각각의 웹 애플리케이션을 위한 ApplicationContext을 생성하는것이 메모리 사용을 증가시키거나 오랜 시간동안 초기화(이를테면, Hibernate SessionFactory)하는 그리고 부작용(side-effects)과 같은 여러개의 bean을 생성하는 문제의 소지가 있다. 대안으로 ContextSingletonBeanFactoryLocator [???] 나 SingletonBeanFactoryLocator [http://www.springframework.org/docs/api/org/springframework/beans/factory/access/SingletonBeanFactoryLocator.html] 와 같은 클래스는 웹 애플리케이션 ApplicationContexts의 부모처럼 사용될수 있는 효과적인 싱글톤형태로 다중 구조적 BeanFactory나 ApplicationContexts 로드를 요구하기 위해 사용될수 있다. 그 결과는 아래쪽의 레이어를 위한 bean정의가 필요할때만 오직 한번 로드되는것이다.

3.20.1. SingletonBeanFactoryLocator 와 ContextSingletonBeanFactoryLocator을 사용하기

당신은 각각의 JavaDoc를 봐서 SingletonBeanFactoryLocator [http://www.springframework.org/docs/api/org/springframework/beans/factory/access/SingletonBeanFactoryLocator.html] 와 ContextSingletonBeanFactoryLocator [???] 를 사용하는 상세화된 예제를 볼수 있을것이다.

EJB장에서 언급되는것처럼, EJB를 위한 Spring의 편리한 기본 클래스는 필요하다면 SingletonBeanFactoryLocator과 ContextSingletonBeanFactoryLocator 의 사용으로 쉽게 대체되는 비-싱글톤 BeanFactoryLocator을 대개 사용한다.

Chapter 4. PropertyEditors, data binding, validation and the BeanWrapper

4.1. 소개

유효성이 있는지 없는지에대한 중대한 질문은 비지니스 논리(business logic)에서 충분히 고려되어야 한다. 답변들 사이에는 찬반 양론이 있다 그리고 Spring은 그것들중의 어떤것도 배타적이지 않은 유효성(그리고 data binding)을 위한 디자인을 제공한다. 유효성은 명확하게 웹계층과 연관이 없어야하고, 어떤 장소에 배치하기(localize) 쉬워야한다 그리고 어떤 유효성을 가능하게하는 사람이 플러그인 할 수 있게 해야한다. 위에서 말한것을 고려하면, Spring은 어플리케인션 모든 계층내에 기본적이고 사용할 수 있는것 사이의 Validator 인터페이스를 제안하고 있다(has come up with).

Data binding은 어플리케이션 도메인 모델(또는 당신이 사용자 입력 진행을 위해 사용하는 객체들 무엇이든지)에 다이나믹하게 묶인것을 허락된 사용자가 입력하기에 유용하다. Spring은 Data binding을 정확하게 하기위해 소위 DataBinder을 제공한다. Validator와 DataBinder는 유효성 패키지를 구성하고, 처음에 사용되었던것 안에 구성되어있다. 그러나 MVC framework안에 제안되어있지는 않다.

BeanWrapper는 Spring Framework 내의 기본적인 개념이고, 많은곳에서 사용되어진다. 그러나, BeanWrapper를 직접 사용할 필요는 아마 결코 없을 것이다. 이것은 참고 문서이기 때문에, 우리는 적절한 몇몇 설명을 생각해 본다. 이번 장(chapter)에서는 BeanWrapper를 설명한다. 만약 여러분이 이것을 모두 사용한다면, 아마 BeanWrapper와 강하게 관련이 있는 객체인 bind data를 연습해 볼 수 있을것이다.

스프링은 모든 장소위에 PropertyEditors를 사용한다. PropertyEditor의 개념은 JavaBeans 명세(specification)의 일부분이다. BeanWrapper와 DataBinder가 밀접하게 연관된 이후로, BeanWrapper일때, 또한 이번 장(chapter)내에 PropertyEditors 사용을 잘 설명한다.

4.2. Binding data를 사용한DataBinder

DataBinder은 BeanWrapper 위에 만든다(builds).[2].

4.3. Bean 조작(manipulation)과 BeanWrapper

org.springframework.beans 패키지는 Sun에서 제공하는 JavaBeans 표준을 고수한다(adhere). JavaBean은 인수가 없는 디폴트 구성자로된 간단한 클래스이고, prop이란 이름의 속성(property)은 setter setProp(...)과 getter getProp()을 가진 네이밍 규칙을 따른다. JavaBeans과 명세서에 관한 더 많은 정보를 위해서, Sun의 웹사이트(java.sun.com/products/javabeans [http://java.sun.com/products/javabeans/])를 방문하기 바란다.

beans 패키지의 아주 중요한 한가지는 BeanWrapper 인터페이스이고 인터페이스에 대응하는 구현(BeanWrapperImpl)이다. JavaDoc의 주석과 같이 BeanWrapper는 기능적인 set과 get 속성값들(개별적인 하나하나 또는 대량)을 제공하고, 속성 서술자들(descriptors)을 얻고, 읽거나 쓸수있는지를 결정하는 속성들을 질의한다(query). 또한, BeanWrapper는 내포된 속성들을 위한 지원을

[2]더 많은 정보를 위해 the beans chapter를 보아라.

제공하고, 제한된 깊이의 하위-속성내의 속성을 설정 가능하게 해준다. 그 다음, BeanWrapper은 target 클래스안에 지원 코드의 필요 없이 PropertyChangeListeners와 VetoableChangeListeners 표준 JavaBeans를 더하는 능력을 지원한다. 마지막으로, BeanWrapper는 인덱스 속성들을 설정하기위한 지원을 제공한다. BeanWrapper은 보통 직접적으로 애플리케이션 코드에 사용되지 않는다. 하지만 DataBinder과 BeanFactory에는 사용된다.

BeanWrapper를 작업하는 방법은 부분적으로 이름에의해 지시되어진다: 설정속성들과 검색한 속성들과 같이 it wraps a bean은 bean안에서 활동이 수행된다. (it wraps a bean to perform actions on that bean, like setting and retrieving properties.)

4.3.1. Setting 과 getting 기본과 내포된 설정들

Setting과 getting 속성들은 setPropertyValue(s) 과 getPropertyValue(s) 메소드를 사용한다. (Setting and getting properties is done using the setPropertyValue(s) and getPropertyValue(s) methods that both come with a couple of overloaded variants.) Spring JavaDoc안에 더 자세한 모든것이 설명되어있다. 중요하게 알아야할것은 한객체가 지시하는 속성에 맞는 연결된(a couple of) 규약이다. 연결된 예들:

Table 4.1. Examples of properties

Expression	Explanation
name	name과 대응하는 속성은 getName() 또는 isName() 그리고 setName()를 나타낸다.
account.name	account 속성의 내포된 name과 대응되는 것은 속성은 예를드면 getAccount().setName() 또는 getAccount().getName() 메소드들을 나타낸다.
account[2]	account의 인덱스(Indexed) 속성, 세가지요소를 나타낸다. 인덱스 속성은 array의 형태, list 또는 있는 그대로의 순서화된 collection의 형태로 나타낼수 있다.
account[COMPANYNAME]	account Map 속성의 COMPANYNAME 키는 색인한 map entry의 값을 나타낸다.

get과 set 속성들을 BeanWrapper로 작업한 몇몇 예제들은 아래에 있다.

주의 : 이부분은 직접적으로 BeanWrapper를 작업할 계획이 없으면 중요하지 않다. 만약 DataBinder와 BeanFactory 그리고 아래 박스이외의(out-of-the-box) 구현을 사용한다면 PropertyEditors section으로 넘어가라.

다음 두 클래스들을 주시하라 :

```
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;
    }
}
```

다음 코드 조각들은 검색하는 방법과 속성들을 사례를 들어 증명하는 조작에 대한 예들이 있다: Companies 과 Employees

```
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");
```

4.3.2. 내장 PropertyEditors, 변환 타입들(Built-in PropertyEditors, converting types)

Spring은 PropertyEditors의 개념을 많이(heavily) 사용한다. 때때로 객체 자신보다 다른 방법으로 속성들을 알맞게 나타낼수 있을지도 모른다. 예를들어, 날짜는 사람이 읽을수 있는 방법으로 나타내야한다. 게다가 우리는 여전히 최초의 날짜를 거꾸로하여 사람이 읽을 수 있는 형태로 변환한다. (또는 더 나은것: Date objects의 뒤, 어떤 날짜를 사람이 읽을수 는 형태로 넣어 변환한다.) (or even better: convert any date entered in a human readable form, back to Date objects). 이 행위자는 java.beans.PropertyEditor 형태의 등록한 사용자 에디터들(registering custom editors)에의해 목적을 이룰수 있다. BeanWrapper내 또는 3장에서 언급한것 같이 특정한 Application Context 안에 등록한 사용자 에디터들은 속성들이 원하는 형태로 변환하는 방법이 주어진다. Sun사에서 제공하는 java.beans 패키지의 JavaDoc문서에있는 더많은 PropertyEditors 정보를 읽어라.

Spring에서 사용하되는 편집속성 예

☒ beans내의 설정 속성들은 PropertyEditors를 사용된다. XML 파일안에 선언한 몇몇 bean 속성값과 같이 java.lang.String을 언급할때, Spring은 (상응하는 setter가 Class-parameter를 가지고 있다면) Class object의 인수를 결정하기위해 ClassEditor를 사용한다.

☒ Spring MVC framework내의 HTTP request parameters 분석에는 CommandController의 모든 하위클래스로 수동으로 바인드 할 수 있는 PropertyEditors종류가 사용된다.

Spring은 생명주기(life)를 쉽게 만들기 위한 내장(built-in) PropertyEditors를 가진다. 각각은 아래에 목록화되어있고, org.springframework.beans.propertyeditors 패키지 안에 모두 위치해 있다. 대부분, (아래에 나타나 있는것과 같이) 모두 그런것은 아니고 BeanWrapperImpl 디폴트로 저장되어있다. 속성 에디터가 몇몇 형태로 구성할수 있고, 디폴트 형태를 오버라이드하여 자신의 형상으로 등록하는 과정을 할 수 있다.
:

Table 4.2. 내장 PropertyEditors(Built-in PropertyEditors)

Class	설명
ByteArrayPropertyEditor	byte 배열을 위한 Editor. Strings은 간단하게 byte 표현과 상응하여 변환될 것이다. BeanWrapperImpl에 의해 디폴트로 등록된다.
ClassEditor	Strings으로 표현된 클래스들을 실제 클래스와 다른 주위의 방법으로 분석하라. (Parses Strings representing classes to actual classes and the other way around.) 클래스를 찾을수 없을때, IllegalArgumentException을 던진다. BeanWrapperImpl에 의해 디폴트로 등록된다.
CustomBooleanEditor	Boolean속성들을 위해 커스터마이즈할수 있는 속성 에디터(editor). BeanWrapperImpl에 의해 디폴트로 등록된다.그러나, custom 에디터에의해 등록된 custom 인스턴스에를 오버라이드할 수 있다.
CustomCollectionEdit	Collection 형태의 목표가 주어졌을때 소스 Collection으로 전환하는, Collections을 위한 설정 editor.
CustomDateEditor	custom DateFormat을 지원한, java.util.Date을 위한 커스터마이즈할 수 있는 설정 editor. 디폴트에의해 등록할 수 없다. 사용자가 적절한 포맷을 소유함으로써 등록되어져야 한다.
CustomNumberEditor	Integer, Long, Float, Double과 같은 Number 서브클래스를 위한 커스터마이즈할 수 있는 설정 에디터. 그러나, 사용자(custom) 에디터로써 등록된 사용자(custom) 인스턴스를 오버라이드할 수 있다.
FileEditor	Strings에서 File-객체들을 결정 가능.(Capable of resolving Strings to File-objects.) BeanWrapperImpl에 의해 디폴트로 등록된다.
InputStreamEditor	단방향 설정 에디터는 텍스트 문자열을 가질 수 있고, InputStream을 생성할 수 있다 (ResourceEditor와 Resource의 조정자를 통해서). 그래서 InputStream 속성들은 Strings에서 File-객체들을 결정하는데 직접적으로 setCapable을 쓸 수도 있다. 디폴트 사용은 InputStream을 끝맺지(close) 않는다는 것을 주의하라!. BeanWrapperImpl에 의해 디폴트로 등록된다.
LocaleEditor	Strings에서 Locale-객체들을 결정 가능하고 반대로 Locale의 toString() 메소드도 같은 기능을 제공한다(String 포맷은

Class	설명
	[language]_[country]_[variant]이다) . BeanWrapperImpl에 의해 디폴트로 등록된다.
PropertiesEditor	Strings(Javadoc 안의 java.lang.Properties class에서 정의된 포맷되어 사용된 형태)을 Properties-객체들로 변환 가능하다. BeanWrapperImpl에 의해 디폴트로 등록된다.
StringArrayPropertyEditor	String을 String-배열로 콤마-범위로 목록화하여 결정할 수 있고, 반대로도 가능하다. BeanWrapperImpl에 의해 디폴트로 등록된다.
StringTrimmerEditor	Property 에디터는 Strings을 정리 정돈한다. 선택적으로 널(null) 값으로 되어있는 변형된 비어있는 문자열로 인정한다. 디폴트에의해 등록되어지지 않는다. 사용자는 필요할때에 등록해야한다.
URLEditor	URL의 String 표현을 실제 URL-객체로 결정할 수 있다. BeanWrapperImpl에 의해 디폴트로 등록된다.

Spring은 설정 에디터들이 필요할지도 모르는 것을 위하여 경로 찾기를 정하는 java.beans.PropertyEditorManager를 사용한다. 경로 찾기는 또한 글꼴, 색상, 모든 원시 형태들의 PropertyEditors를 포함하고 있는 sun.bean.editors를 나타낸다. 만약 다루는 클개스가 같은 패키지 안에 있고, 클래스가 같은 이름을 가지고 있고, 'Editor'가 추가되어있다면, 또한 표준 JavaBeans 기초구조는 (등록하는 절차 없이) 자동적으로 PropertyEditors를 발견한다는 것을 주의하라.

4.3.3. 언급할 가치가 있는 다른 기능들.

전체 섹션에서 그렇게 가치가 있지는 않지만, 그밖에 여러분이 흥미로와 했을지도 모를 연관된 기능들은 전 섹션에서 본 기능들이다.

☒ 가독성과 쓸수있는지 특성 결정: isReadable()과 isWritable() 메소드를 사용함은 속성이 가독성이 있는지 쓸수있는지를 결정할 수 있다.
☒ 검색하는 PropertyDescriptors: getPropertyDescriptor(String)과 getPropertyDescriptors()를 사용함은 java.beans.PropertyDescriptor형태의 객체들을 검색할 수 있고, 때때로 편리할지도 모른다.

Chapter 5. Spring AOP: Spring을 이용한 Aspect 지향적인 프로그래밍

5.1. 개념

Aspect-지향 프로그래밍 (AOP)는 프로그램 구조에 대한 다른 방식의 생각을 제공함으로써 OOP를 보완한다. OO가 애플리케이션을 객체구조로 분석하는 동안 AOP는 프로그램을 aspects 나 관심사(concerns)로 분석한다. 이것은 다중 객체로부터 공통적으로 잘라낼수 있는 트랜잭션 관리와 같은 관심사의 모듈화(이러한 관심사는 종종 crosscutting 관심사라는 용어로 사용된다.) 를 가능하게 한다.

Spring의 핵심이 되는 컴포넌트중 하나는 AOP 프레임워크이다. Spring IoC컨테이너(BeanFactory와 ApplicationContext) 가 AOP에 의존하지 않는 동안 당신이 원하지 않는다면 AOP를 사용할 필요가 없다는 것을 의미한다. AOP는 미들웨어 솔루션의 기능을 제공하기 위해 Spring IoC를 보완할것이다.

AOP는 Spring내에서 사용된다.

☒ 선언적인 기업용 서비스를 제공하기 위해 EJB 선언적 서비스를 위한 대체물처럼 사용될수 있다. 서비스처럼 가장 중요한 것은 Spring의 트랜잭션 추상화에서 빌드되는 선언적인 트랜잭션 관리이다.

☒ 사용자 정의 aspect를 구현하는 것을 사용자에게 허용하기 위해 AOP를 사용하여 OOP의 사용을 기능적으로 보완한다.

게다가 당신은 EJB없이 선언적인 트랜잭션 관리를 제공하는 것을 Spring에 허용하도록 하는 기술을 가능하게 하는것처럼 Spring AOP를 볼수 있다. 또는 사용자 지정 aspect를 구현하기 위한 Spring AOP프레임워크의 강력한 힘을 사용할수 있다.
만약 당신이 일반적인 선언적 서비스나 풀링과 같은 다른 미리 패키징된 선언적 미들웨어 서비스만 관심을 가진다면 당신은 Spring AOP를 사용하여 직접적으로 작업할 필요가 없다. 그리고 이 장의 대부분을 그냥 넘어갈수 있다.

5.1.1. AOP 개념

몇몇 중심적인 AOP개념을 명시함으로써 시작해보자. 이 개념들은 Spring에 종속적인 개념이 아니다. 운나쁘게도 AOP전문용어는 특히 직관적이지 않다. 어쨌든 Spring이 그 자신의 전문용어를 사용했다면 좀더 혼란스러울것이다.

☒ Aspect: 구현물이 다중 객체를 짤라내는 것을 위한 concern의 모듈화. 트랜잭션 관리는 J2EE애플리케이션의 crosscutting concern의 좋은 예제이다. aspect는 advisor이나 인터셉터처럼 Spring을 사용하여 구현된다.

☒ Joinpoint: 메소드 호출이나 특정 예외를 던지는 것과 같은 프로그램을 수행하는 지점. Spring AOP에서 joinpoint는 언제나 메소드 호출이다. Spring은 두드러지게 joinpoint개념을 사용하지는 않는다. joinpoint정보는 인터셉터로 인자를 전달하는 MethodInvocation의 메소드를 통해 접근가능하다. 그리고 org.springframework.aop.Pointcut인터페이스의 구현물에 의해 평가된다.

☒ Advice: 특정 joinpoint에서 AOP프레임워크에 의해 획득되는 액션. advice의 각각의 타입은 "around," "before" 과 "throws"를 포함한다. advice 타입은 밑에서 언급된다. Spring을 포함한

많은 AOP프레임워크는 인터셉터처럼 advice를 모델화하고 joinpoint "주위"로 인터셉터의 묶음(chain)을 유지한다.

☒ Pointcut: advice가 수행될때를 명시하는 joinpoint의 모음. AOP프레임워크는 개발자에게 예를 들면 정규식 표현을 사용하는 것과 같은 pointcut를 명시하도록 허용해야 한다.

☒ Introduction: advised 클래스에 메소드나 필드추가하기. Spring은 어떠한 advised클래스에 새로운 인터페이스를 소개(introduce)하는것을 허용한다. 예를 들면 당신은 간단한 캐싱을 위해 IsModified인터페이스를 구현한 어떠한 객체를 만들기 위해 introduction을 사용할수 있다.

☒ 대상 객체: 객체는 joinpoint를 포함한다. 또한 advised 나 proxied객체를 참조한다.

☒ AOP 프록시: AOP프레임워크에 의해 생성되는 advice를 포함한 객체. Spring에서 AOP프록시는 JDK동적 프록시나 CGLIB프록시가 될것이다.

☒ Weaving: advised객체를 생성하기 위한 aspect 조합. 이것은 컴파일(예를 들면 AspectJ컴파일러를 사용하는) 시각이나 수행시각에 수행될수 있다. 다른 순수한 자바 AOP프레임워크처럼 Spring은 수행시에 작성된다.

포함되는 다양한 타입의 advice

☒ Around advice: 메소드 호출과 같은 joinpoint주위(surround)의 advice. 이것은 가장 강력한 종류의 advice이다. Around advice는 메소드 호출 전후에 사용자 지정 행위를 수행한다. 그것들은 joinpoint를 처리하거나 자기 자신의 반환값을 반환함으로써 짧게 수행하거나 예외를 던지는 것인지에 대해 책임을 진다.

☒ Before advice: joinpoint전에 수행되는 advice. 하지만 joinpoint를 위한 수행 흐름 처리(execution flow proceeding)를 막기위한 능력(만약 예외를 던지지 않는다면)을 가지지는 않는다.

☒ Throws advice: 메소드가 예외를 던질다면 수행될 advice. Spring은 강력한 타입의 Throws advice를 제공한다. 그래서 당신은 Throwable 나 Exception으로 부터 형변환 할 필요가 없는 관심가는 예외(그리고 하위클래스)를 잡는 코드를 쓸수 있다.

☒ After returning advice: joinpoint이 일반적으로 예를 들어 메소드가 예외를 던지는것없이 반환된다면 완성된 후에 수행되는 advice.

Around advice는 가장 일반적인 종류의 advice이다. Nanning Aspects와 같은 대부분의 인터셉션-기반의 AOP프레임워크는 오직 around advice만을 제공한다.

AspectJ처럼 Spring이 advice타입의 모든 범위를 제공하기 때문에 우리는 요구되는 행위를 구현할수 있는 최소한의 강력한 advice타입을 사용하길 권한다. 예를 들어 당신이 메소드의 값을 반환하는 캐시만을 수정할 필요가 있다면 around advice가 같은것을 수행할수 있다고하더라도 around advice보다 advice를 반환한 후에 구현하는게 더 좋다. 대부분 특정 advice타입을 사용하는것은 잠재적으로 적은 에러를 가지는 간단한 프로그래밍 모델을 제공한다. 예를 들어 당신은 around advice를 위해 사용되는 MethodInvocation의 proceed()메소드를 호출할 필요가 없고 나아가 그것을 호출하는것을 실패할수도 있다.

pointcut 개념은 인터셉션(interception)을 제공하는 오래된 기술로 부터 AOP를 구별하는 수단이 되는 AOP의 핵심이다. pointcut는 OO구조가 비의존적으로 대상화되기 위한 advice를 가능하게 한다. 예를 들면 선언적인 트랜잭션을 제공하는 around advice는 다중 객체에 걸쳐있는 메소드 모음에 적용될수 있다. 게다가 pointcut는 AOP의 구조적인 요소를 제공한다.

5.1.2. Spring AOP의 기능과 대상

Spring AOP는 순수자바로 구현되었다. 특별한 편집 절차가 필요하지 않다. Spring AOP는 클래스로더 구조를 제어할 필요가 없고 J2EE웹 컨테이너나 애플리케이션 서버내 사용되는것이 적합하다.

Spring은 현재 메소드 호출의 인터셉션(interception)을 지원한다. 필드 인터셉션은 비록 필드인터셉션이 핵심 Spring AOP API에 영향없이 추가될수 있지만 구현되지 않았다.
필드 인터셉션은 어쩌면 OO캡슐화를 침범한다. 우리는 이것이 애플리케이션 개발에서 현명하다고 생각하지 않는다. 만약 당신이 필드 인터셉션을 요구한다면 AspectJ를 사용하는것을 생각해보라.

Spring은 pointcut과 다른 advice타입을 표현하기 위한 클래스를 제공한다. Spring은 aspect를 표현하는 객체를 위해 advisor라는 개념을 사용하고 joinpoint를 명시하기 위해 이것을 대상으로하는 advice와 pointcut모두를 포함한다.

다른 advice타입은 (AOP제휴(Alliance) 인터셉션 API로 부터)MethodInterceptor이고 advice인터페이스는 org.springframework.aop패키지내 정의된다. 모든 advice는 org.aopalliance.aop.Advice태그 인터페이스를 구현해야만 한다. advice는 MethodInterceptor; ThrowsAdvice; BeforeAdvice; 그리고 AfterReturningAdvice를 지원한다. 우리는 밑에서 advice타입에 대해서 상세하게 언급할것이다.

Spring은 AOP 제휴(Alliance)인터셉션 인터페이스(http://www.sourceforge.net/projects/aopalliance)를 구현한다. around advice는 AOP제휴 org.aopalliance.intercept.MethodInterceptor인터페이스를 구현해야만 한다. 이 인터페이스의 구현은 Spring내에서 뿐 아니라 다른 어떠한 AOP제휴 호환 구현물에서도 작동할수 있다. 현재 JAC는 AOP제휴 인터페이스를 구현하고 Nanning 과 Dynaop는 2004년 초 구현할 가능성이 있다.

AOP에 대한 Spring의 접근법은 대부분의 다른 AOP프레임워크와 다르다. 그 목적은 Spring AOP가 대부분 구현가능하더라도 완벽한 AOP구현물을 제공하기 않는것이다. 이것은 기업용 애플리케이션내 공통적인 문제점 해결을 돕기위해 AOP구현물과 Spring IoC사이의 닫힌(close) 통합을 제공하기 위해서이다.

게다가 예를 들어 Spring의 AOP기능은 Spring IoC컨테이너와 협력하여 작동된다. AOP advice는 비록 이것이 강력한 "autoproxying"기능을 허용한다고 할지라도 일반적인 빈 정의 문법을 사용해서 명시된다. advice와 pointcut는 Spring IoC에 의해 스스로 관리되는데 이것이 다른 AOP구현물과의 결정적인 차이점이다. Spring AOP를 사용하여 매우 잘 정리된 객체를 권하는(advise) 것처럼 당신이 쉽게 또는 효율적으로 할수 없는 것들이 있다. AspectJ는 아마도 이러한 경우에 최고의 선택이다. 어쨌든 우리의 경험은 Spring AOP가 AOP를 다루는 J2EE애플리케이션내 대부분의 문제를 위한 멋진 솔루션를 제공한다는 것이다.

Spring AOP는 복잡한 AOP솔루션을 제공하기 위해 AspectJ 나 AspectWerkz에 대해 침범하지 않을것이다. 우리는 Spring과 같은 프록시 기반 프레임워크와 AspectJ와 같은 성숙한 프레임워크 모두 가치있고 경쟁보다는 서로 보완할수 있다고 믿는다. 게다가 Spring 1.1의 가장 큰 우선사항은 일관적인 Spring기반의 애플리케이션 구조내 응답되는 위한 AOP의 모든 사용을 가능하게 하기 위해 Spring AOP와 AspectJ와의 IoC가 균일하게 통합되는 것이다. 이 통합은 Spring AOP API나 AOP제휴 API에 영향을 끼치지 않는다. Spring AOP는 호환적인 면만 남을것이다.

5.1.3. Spring 내 AOP 프록시

Spring은 AOP프록시를 위해 J2SE 동적 프록시(dynamic proxies)를 사용하는것이 디폴트이다. 이것은 프록시가 되기 위한 어떤 인터페이스나 인터페이스의 모음을 가능하게 한다.

Spring은 또한 CGLIB프록시도 사용가능하다. 이것은 인터페이스보다는 클래스를 프록시화 하기 위해 필요하다. CGLIB는 비지니스 객체가 인터페이스를 구현하지 않는다면 디폴트로 사용된다. 클래스보다는 인터페이스를 위한 프로그램을 위해 좋은 경험이기 때문에 비지니스 객체는 일반적으로 하나 이상의 비지니스 인터페이스를 구현할것이다.

이것은 강제로 CGLIB의 사용하도록 할수 있다. 우리는 이것을 밑에서 언급할것이다 당신이 왜 이렇게 하는것을 원하는지 설명할것이다.
Spring 1.0이후 Spring은 생성된 클래스들 전부를 포함하는 AOP프록시의 추가적인 타입을 제공한다. 이것은 프로그래밍 모델에 영향을 끼치지 않을것이다.

5.2. Spring내 Pointcuts

Spring이 중대한 pointcut개념을 어떻게 다루는지 보자.

5.2.1. 개념

Spring의 pointcut모델은 advice타입의 비의존성을 재사용한다. 이것은 같은 pointcut을 사용하여 다른 advice를 대상으로 하는것이 가능하다.

org.springframework.aop.Pointcut인터페이스는 특정 클래스와 메소드에 대해 대상 advice가 사용되는 중심적인 인터페이스이다. 완전한 인터페이스는 밑에서 보여준다.

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

Pointcut인터페이스를 두개의 부분으로 쪼개는 것은 부분들에 적합한 클래스와 메소드, 그리고 잘 조직된 기능(다른 메소드 적합자(matcher)와의 "union"를 수행하는것과 같은)의 재사용을 허용한다.

ClassFilter인터페이스는 주어진 대상 클래스의 모음을 위해 pointcut를 제한하기 위해 사용된다. 만일 matches()메소드가 언제나 true를 반환한다면 모든 대상 클래스는 적합할것이다.

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

MethodMatcher인터페이스는 일반적으로 좀더 중요하다. 완전한 인터페이스를 밑에서 보여준다.

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

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

matches(Method, Class) 메소드는 이 pointcut이 대상 클래스에서 주어진 메소드에 적합할지 테스트하기 위해 사용된다. 이 평가는 모든 메소드 호출에서 테스트를 위한 필요성을 제거하기 위해 AOP프록시가 생성되었을때 수행될수 있다. 만약 두개의 인자가 적합한 메소드가 주어진 메소드를 위해 true를 반환한다면 MethodMatcher를 위한 isRuntime()메소드는 true를 반환한다. 세개의 인자가 적합한 메소드는 모든 메소드 호출에서 호출될것이다. 이것은 대상 advice가 수행되기 전에 즉시 메소드 호출로 전달되는 인자를 보기 위한 pointcut을 가능하게 한다.

대부분의 MethodMatchers는 그들의 isRuntime()메소드가 false를 반환하는 의미에서 정적이다. 이 경우 세개의 인자가 적합한 메소드는 결코 호출되지 않을것이다.
가능하다면 pointcut을 AOP프록시가 생성될때 pointcut의 평가결과를 캐시하기 위한 AOP프레임워크를 허용하도록 정적으로 만들도록 시도하라.

5.2.2. pointcuts에서의 작업(operation)

Spring은 pointcut에서의 작업(union 과 intersection)을 지원한다.

Union은 어느한쪽의 pointcut가 들어맞는 방식을 의미한다.

Intersection은 pointcut둘다 들어맞는 방식을 의미한다.

Union이 언제나 좀더 유용하다.

Pointcuts는 org.springframework.aop.support.Pointcuts클래스내 정적 메소드를 사용하거나 같은 패키지내 ComposablePointcut클래스를 사용하여 구성될수 있다.

5.2.3. 편리한 pointcut 구현물

Spring은 다양하고 편리한 pointcut구현물을 제공한다. 몇몇은 특별히 사용될수 있다. 다른것은 애플리케이션에 종속적인 pointcut내에서 하위클래스화 되는 경향이 있다.

5.2.3.1. 정적 pointcuts

정적 pointcut는 메소드와 대상 클래스에 기반하고 메소드의 인자를 고려할수는 없다. 정적 pointcut는 대부분의 사용상황을 위해 충분하고 가장 좋다. Spring이 메소드가 처음 호출될때 정적 pointcut를 한번만 평가하는것은 가능하다. 그 후 각각의 메소드 호출시 pointcut를 다시 평가하는것은 필요하지 않다.

Spring에 포함된 몇몇 정적 pointcut구현물을 생각해보자.

5.2.3.1.1. 정규표현식 pointcuts

정적 pointcut를 명시하는 하나의 분명한 방법은 정규표현식이다. Spring외 다양한 AOP프레임워크는 이것이 가능하다. org.springframework.aop.support.RegexpMethodPointcut는 Perl 5 정규표현식 문법을 사용하는 일반적인 정규표현식 pointcut이다.

이 클래스를 사용하면 당신은 문자열 형태의 목록을 제공할수 있다. 만약 이러한 것들이 들어맞는다면 pointcut는 true로 평가할것이다. (그래서 이 결과는 이러한 pointcut의 표과적인 union이다.)

사용법은 아래에서 보여준다.

```
<bean id="settersAndAbsquatulatePointcut"
    class="org.springframework.aop.support.RegexpMethodPointcut">
```

```
    <property name="patterns">
        <list>
            <value>.*get.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

RegexpMethodPointcut, RegexpMethodPointcutAdvisor의 편리한 하위클래스는 우리에게 advice또한 참조하도록 허락한다.(advice는 advice나 advice를 던지는등 이전에 인터셉터될수 있다는 것을 기억하라.) 이것은 하나의 빈이 pointcut와 advisor모두를 제공하거나 아래의 사항처럼 배선(wiring)을 단순화한다.

```
<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>
```

RegexpMethodPointcutAdvisor는 어떠한 advice타입과도 사용될수 있다.
RegexpMethodPointcut클래스는 Jakarta ORO정규 표현식 패키지를 요구한다.

5.2.3.1.2. 속성지향(Attribute-driven) pointcuts

정적 pointcut의 중요한 타입은 metadata-driven pointcut이다. 이것은 메타데이타(전형적으로 소세레벨의 메타데이타) 속성들의 값을 사용한다.

5.2.3.2. 동적 pointcuts

동적 pointcut은 정적 pointcut보다 평가하는것이 손실이 크다(costlier). 그것들은 정적 정보만큼 메소드 arguments를 고려한다. 이것은 모든 메소드 호출에서 그들이 평가가 되어야 함을 의미한다. 결과는 인자가 다양하기 때문에 캐시될수 없다.

중요한 예제는 흐름 제어(control flow) pointcut 이다.

5.2.3.2.1. 흐름 제어(Control flow) pointcuts

Spring 흐름 제어 pointcut는 비록 덜 강력하지만 개념적으로 AspectJ cflow pointcut와 흡사하다. (현재 밑의 다른 pointcut를 수행하는 pointcut을 명시하는 방법은 없다.). 흐름 제어 pointcut는 최근의 호출 스택에 대응된다. 예를 들면 이것은 com.mycompany.web패키지내 메소드나 SomeCaller클래스에 의해 joinpoint가 호출된다면 실행한다. 흐름 제어 pointcut는 org.springframework.aop.support.ControlFlowPointcut클래스를 사용하여 몇시된다.

Note

흐름 제어 pointcut는 다른 동적 pointcut보다 수행시 평가되기 위해 명백하게 좀더 비싸다. Java1.4에서 다른 동적 pointcut의 5배이고 Java1.3에서는 10배이상이다.

5.2.4. Pointcut 수퍼클래스(superclasses)

Spring은 당신 자신의 pointcut를 구현하도록 당신을 돕기 위해 유용한 pointcut 수퍼클래스를 제공한다.

정적 pointcut이 가장 유용하기 때문에 당신은 밑에서 보여주는 것처럼 StaticMethodMatcherPointcut의 하위클래스를 만들것이다. 이것은 하나의 추상 메소드를 구현(비록 사용자 지정 행위를 위해 다른 메소드를 오버라이드하는것이 가능하더라도)하는것을 요구한다.

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }
}
```

동적 pointcut를 위해 수퍼클래스도 있다.

당신은 Spring 1.0 RC2그리고 그 이상에서 어떠한 advice타입으로도 사용자 지정 pointcut를 사용할수 있다.

5.2.5. 사용자지정 pointcuts

Spring내 pointcut가 언어적(AspectJ처럼)인 면보다 자바클래스이기 때문에 이것은 정적이든 동적이든 사용자지정 pointcut를 선언하는것이 가능하다. 어쨌든 AspectJ문법내 작성될수 있는 정교한 pointcut표현식을 위해 특별한 지원은 없다. 어쨌든 Spring내 사용자 지정 pointcut는 임의로 복잡하게 될수 있다.
Spring의 가장 최근버전은 아마 JAC에 의해 제공되는 것처럼 "의미적인(semantic) pointcut"를 위한 지원이 제공될지도 모른다. 예를 들면 "대상 객체내에서 인스턴스 변수를 변경하는 모든 메소드"

5.3. Spring 내 Advice타입들

Spring AOP가 advice를 어떻게 다루는지에 대해서 지금 보자.

5.3.1. Advice 생명주기

Spring advices는 모든 adviced객체를 통해 공유되거나 각각의 advised객체에 대해 유일할 수 있다. 이것은 per-class 나 per-instance advice에 일치한다.

Per-class advice는 매우 종종 사용된다. 이것은 트랜잭션 advisor처럼 일반적인 advice를 선호한다. 그것들은 프록시된 객체의 상태에 의존하지 않고 새로운 상태를 추가한다. 그것들은 단지 메소드와 인자에 영양을 준다.

Per-instance advice는 도입이나 mixin을 지원하기 위해 선호한다. 이 경우 advice는 프록시된 객체에 상태를 추가한다.

이것은 같은 AOP프록시내에서 공유되는것들의 mixin과 per-instance advice를 사용하는것이 가능하다.

5.3.2. Spring내 Advice 타입들

Spring은 다양하고 특별한 advice타입들을 제공한다. 그리고 독단적인 advice타입을 지원하기 위해 확장가능하다. 기본적인 개념과 표준적인 advice타입을 보자.

5.3.2.1. Interception around advice

Spring내 가장 기본적인 advice타입은 interception around advice이다.

Spring은 메소드 인터셉션을 사용해서 around advice를 위한 AOP제휴 인터페이스와 잘 작동한다. around advice를 구현하는 MethodInterceptors는 다음의 인터페이스를 구현해야만 한다.

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

invoke()메소드를 위한 MethodInvocation인자는 호출될 메소드(대상 joinpoint; AOP프록시; 그리고 메소드를 위한 인자)를 드러낸다. invoke()메소드는 호출 결과(joinpoint의 반환값)를 반환한다.

간단한 MethodInterceptor 구현은 다음과 같다.

```
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;
    }
}
```

MethodInvocation의 proceed()메소드를 호출하는것에 주의하라. 이것은 joinpoint쪽으로 인터셉터 연계작업(chain)을 처리한다. 대부분의 인터셉터는 이 메소드를 호출할것이고 그 자신의 반환값을 반환한다. 어쨌든 어떠한 around advice와 같은 MethodInterceptor는 처리 메소드를 호출하는 것보다 다른 값을 반환하거나 예외를 던질수 있다. 어쨌든 당신은 좋은 이유없이는 이것을 원하지 않을것이다. MethodInterceptors는 다른 AOP제휴 AOP구현물과 상호작용성을 제공한다. 다른 advice타입은 공통적인 AOP개념을 구현하는 이 부분의 나머지에서 Spring 특유의 방법으로 언급된다. 대부분의 특정 advice타입을 사용하는 방법으로 장점을 가지는 동안 만약 당신이 다른 AOP프레임워크내 aspect를 수행하길 원한다면 MethodInterceptor around advice에 충실하라. 현재 pointcut는 프레임워크 사이에 상호작용가능하지 않고 AOP제휴는 현재 pointcut인터페이스를 정의하지 않는다.

5.3.2.2. 전(Before) advice

좀더 간단한 advice타입은 전(before) advice이다. 이것은 오직 메소드에 들어가지전에 호출되기 때문에 MethodInvocation객체를 필요로 하지 않는다.

전(before) advice의 가장 중요한 장점은 proceed() 메소드를 호출할 필요가 없다는 것이다. 그리고 인터셉터 연계작업을 처리하는 것을 무심코 실패하는 가능성이 없다.

MethodBeforeAdvice 인터페이스는 아래에서 보여진다. (Spring API디자인은 비록 평상시 객체가 필드 인터셉션에 적용하고 Spring이 이것을 구현할 가능성은 없음에도 불구하고 advice앞의 필드를 허용한다.).

```
public interface MethodBeforeAdvice extends BeforeAdvice {

```

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

반환타입은 void라는것에 주의하라. 전(before) advice는 joinpoint가 수행되기 전에 사용자 지정 행위를 추가할수 있다. 하지만 반환값을 변경할수는 없다. 만약 전(before) advice가 예외를 던진다면 이것은 인터셉터 연계작업의 더이상의 수행을 취소할것이다. 예외는 인터셉터 연계작업을 뒤로 돌린다. 만약 이것이 체크되지 않았거나 호출된 메소드의 시그너처라면 이것은 클라이언트로 직접적으로 전달될것이다. 반면에 이것은 AOP프록시에 의한 체크되지 않은 예외를 포장할것이다.

Spring내 전(before) advice의 예제는 모든 메소드 호출을 센다.

```
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;
    }
}
```

전(before) advice 는 어떠한 pointcout와 사용될수 있다.

5.3.2.3. Throws advice

Throws advice는 joinpoint가 예외를 던진다면 joinpoint를 반환한후에 호출된다. Spring은 throws advice타입을 제공한다. 이것은 org.springframework.aop.ThrowsAdvice인터페이스가 어떠한 메소드도 포함하지 않는다는 것을 의미한다. 이것은 주어진 객체가 하나 이상의 throws advice타입의 메소드를 구현하는 것을 표시하는 태그 인터페이스이다. 그것들은 폼의 형태가 될것이다.

```
afterThrowing([Method], [args], [target], subclassOfThrowable)
```

오직 마지막의 인자만이 요구된다. 게다가 한개에서 네개까지의 인자는 advice메소드가 메소드와 인자에 연관되는지에 의존한다. 다음은 throws advice의 예제이다.

이 advice는 RemoteException과 하위 클래스가 던져진다면 호출될것이다.

```
public  class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

다음의 advice는 ServletException가 던져진다면 호출된다. 위 advice와는 다르게 이것은 네개의 인자를 명시한다. 그래서 이것은 메소드 인자와 대상 객체로 호출되는 메소드에 접근한다.

```
public static class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something will all arguments
    }
}
```

마지막 예제는 두개의 메소드가 RemoteException 과 ServletException 둘다 다루는 하나의 클래스내에서 어떻게 사용되는지 설명한다. 많은 수의 throws advice 메소드는 하나의 클래스내에서 조합될수 있다.

```
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 어떠한 pointcut와 사용될수 있다.

5.3.2.4. advice를 반환한 후(after returning advice)

Spring내에서 advice를 반환한 후(after returning advice)는 밑에서 보여지는 것처럼 org.springframework.aop.AfterReturningAdvice인터페이스를 구현한다.

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

after returing advice는 반환하는 값(변경할수 없는), 호출된 메소드, 메소드 인자와 대상에 접근한다.

다음의 after returning advice는 예외를 던지지 않는 모든 성공적인 메소드 호출의 갯수를 센다.

```
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;
    }
}
```

이 advice는 실행경로를 변경하지 않는다. 만약 이것이 예외를 던진다면 이것은 반환값 대신에 인터셉터 연계를 던질것이다.
After returning advice는 어떠한 pointcut도 함께 사용될수 있다.

5.3.2.5. Introduction advice

Spring은 특별한 종류의 interception advice처럼 introduction advice를 처리한다.

Introduction은 다음의 인터페이스를 구현하는 IntroductionAdvisor 와 IntroductionInterceptor를 요구한다.

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

AOP제휴 MethodInterceptor 인터페이스로 부터 상속된 invoke() 메소드는 introduction을 구현해야만 한다. 만약 호출된 메소드가 introduced인터페이스에 있다면 introduction인터셉터는 메소드 호출을 다룰 책임을 가지고 있으며 이것은 proceed()을 호출할수 없다.

Introduction advice는 메소드보다는 클래스에만 적용하는 것처럼 어떠한 pointcut와 함께 사용될수 없다. 당신은 다음의 메소드를 가지는 IntroductionAdvisor를 사용하여 introduction advice만을 사용할수 있다.

```
public interface IntroductionAdvisor extends IntroductionAdvisor {

    ClassFilter getClassFilter();

    IntroductionInterceptor getIntroductionInterceptor();

    Class[] getInterfaces();
}
```

introduction advice와 관련하여 MethodMatcher가 없고 나아가 Pointcut도 없다. 오직 클래스 필터링이 논리적이다.

getInterfaces() 메소드는 이 advisor에 의해 소개된(introduced) 인터페이스만을 반환한다.

Spring테스트 묶음으로 부터 간단한 예제를 보자. 하나 이상의 객체에 다음의 인터페이스를 소개(introduce)하길 원한다고 가정하자.

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

이것은 mixin을 묘사한다. 우리는 그것들의 타입이 무엇이든 lock와 unlock메소드를 호출하든 advised 객체가 Lockable로 형변환될수 있길 원한다. 우리가 lock()메소드를 호출한다면 우리는 LockedException를 던지기 위해 모든 setter메소드를 원한다. 게다가 우리는 이것에 대한 어떠한 지식을 가진 그것들 없이도 객체를 바꿀수 없는 상태로 만들기 위한 능력을 제공하는 aspect를 추가할수 있다. AOP의 좋은 예제이다.

첫째로 우리는 대량으로 발생하는 IntroductionInterceptor이 필요할 것이다. 이 경우 우리는 org.springframework.aop.support.DelegatingIntroductionInterceptor를 확장한다. 우리는 직접적으로 IntroductionInterceptor를 구현할수 있다. 하지만 DelegatingIntroductionInterceptor을 사용하는 것은 대부분의 경우에 최고의 선택이다.

DelegatingIntroductionInterceptor는 introduced인터페이스의 실질적인 구현을 위해 introduction을 위임하도록 디자인되었고 그렇게 하기 위한 인터셉션의 사용을 숨긴다. 그 위임은 생성자 인자를 사용해서 어떠한 객체를 위한 셋팅될수 있다. 디폴트(인자없는 생성자를 사용할 때) 위임이 이것이다. 게다가 이것은 아래의 예제에 있다. 위임은 DelegatingIntroductionInterceptor의 LockMixin 하위 클래스이다. 주어진 위임(디폴트에 의해)님 DelegatingIntroductionInterceptor인스턴스는 위임(IntroductionInterceptor이 아닌)에 의해 구현된 모든 인터페이스를 찾는다. 그리고 그것들에 대해 introduction을 지원할것이다. 인터페이스를 억제하기 위한 suppressInterflace(Class intf) 메소드를 호출하기 위해 LockMixin과 같은 하위클래스가 드러낼수(exposed) 없다. 어쨌든 많은 인터페이스 IntroductionInterceptor가 지원되기 위해 준비되는지는 문제되지 않는다. 사용되는 IntroductionAdvisor는 실질적으로 드러나는 인터페이스를 제어할것이다. introduced인터페이스는 대상에 의해 같은 인터페이스의 어떠한 구현물을 숨길것이다.

게다가 LockMixin은 DelegatingIntroductionInterceptor의 하위클래스이고 Lockable를 구현한다. 수퍼클래스는 Lockable가 introduction를 지원할수 있다는 것을 자동적으로 포착한다. 그래서 우리는 그것을 명시할 필요가 없다. 우리는 이 방법으로 많은 인터페이스를 소개(introduce)할수 있다.

locked인스턴스 변수 사용에 주의하라. 이것은 대상 객체를 유지하는 추가적인 상태를 효과적으로 추가한다.

```
public class LockMixin extends DelegatingIntroductionInterceptor
    implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0)
            throw new LockedException();
        return super.invoke(invocation);
    }

}
```

종종 이것은 invoke()메소드를 오버라이드(메소드가 소개(introduced)되었다면 위임메소드를 호출하는 DelegatingIntroductionInterceptor구현) 할 필요가 없다. 반면에 joinpoint를 향한 처리는 언제나 충분하다. 현재 상태에서 우리는 locked상태라면 호출될수 있는 setter메소드가 없다는 것을 체크할 필요가 있다.

요구되는 introduction advisor는 간단하다. 이것을 하기 위해 필요한 모든것은 구별되는 LockMixin인스턴스를 유지하고 Lockable의 경우에서 소개된(introduced) 인터페이스를 명시하는것이다. 좀더 복잡한 예제는 introduction인터셉터(프로토타입처럼 명시된)에 대한 참조를 가져온다. 이 경우 LockMixin에 관련된 설정은 없다. 그래서 new를 사용해서 이것을 간단하게 생성한다.

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

우리는 매우 간단하게 이 advisor를 적용할수 있다. 이것은 설정을 요구하지 않는다. (어쨌든 이것은 필요하다. IntroductionAdvisor없이 IntroductionInterceptor을 사용하는 것은 불가능하다.) 늘 그렇듯이 introduction으로 advisor은 이것이 상태유지(stateful)인 것처럼 인스턴스당 하나가 되어야 한다. 우리는 각각의 advised객체를 위해 LockMixinAdvisor의 다른 인스턴스와 LockMixin이 필요하다. advisor는 advised객체의 상태의 일부로 구성된다.

우리는 프로그램에 따라 Advised.addAdvisor() 를 사용하거나 XML설정(추천되는 방식)으로 다른

advisor처럼 이 advisor를 적용할수 있다. 밑에서 언급되는 "자동 프록시 생성자"를 포함한 모든 프록시 생성 선택은 introduction과 상태유지 mixin을 다룬다.

5.4. Spring내 Advisors

Spring내 advisor는 aspect의 모듈이다. advisor는 전형적으로 advice와 pointcut를 결합한다.

introduction의 특별한 경우 이외에 어떤 advisor는 어떠한 advice와 사용될수 있다. org.springframework.aop.support.DefaultPointcutAdvisor는 가장 공통적으로 사용되는 advisor클래스이다. 예를 들면 이것은 MethodInterceptor, BeforeAdvice 또는 ThrowsAdvice과 사용될수 있다.

같은 AOP프록시로 Spring내에서 advisor과 advice타입들을 혼합하는 것은 가능하다. 예를 들면 당신은 하나의 프록시 설정으로 interception around advice, throws advice 그리고 before advice를 사용할수 있다. Spring은 자동적으로 필요한 생성 인터셉터 연계작업(create interceptor chain)을 생성할 것이다.

5.5. AOP프록시를 생성하기 위한 ProxyFactoryBean사용하기

만약 당신이 당신의 비지니스 객체를 위해 Spring IoC컨테이너(ApplicationContext나 BeanFactory)를 사용한다면 그리고 사용해야만 한다면 당신은 Spring의 AOP FactoryBean중 하나를 사용하길 원할것이다. (factory빈은 다른 타입의 객체를 생성하는 것을 가능하게 하는 indirection의 레이어를 소개(introduce)한다는 것을 기억하라.)

Spring내 AOP프록시를 생성하는 기본적인 방법은 org.springframework.aop.framework.ProxyFactoryBean을 사용하는 것이다. 이것은 적용할 pointcut와 advice의 완벽한 제어능력를 부여한다. 어쨌든 당신이 이러한 제어능력을 필요로 하지 않는다면 선호될수 있는 유사한 옵션들이 있다.

5.5.1. 기본

다른 Sprign indirection과 같은 FactoryBean구현물인 ProxyFactoryBean은 indirection의 레벨을 소개(introduce)한다. 만약 당신이 foo이름으로 ProxyFactoryBean를 정의한다면 foo를 참조하는 객체는 ProxyFactoryBean인스턴스 자신이 아니다. 하지만 객체는 getObject() 메소드의 ProxyFactoryBean's구현물에 의해 생성된다. 이 메소드는 대상 객체를 포장하는 AOP프록시를 생성할것이다.

AOP프록시를 생성하기 위해 ProxyFactoryBean나 다른 IoC형태로 감지되는 클래스를 사용하는 가장 중요한 이득중 하나는 이것이 IoC에 의해 advice와 pointcut가 관리될수 있다는 것이다. 이것은 다른 AOP프레임워크로는 달성하기 힘든 어떠한 접근법을 가능하게 하는 강력한 기능이다. 예를 들면 애플리케이션 객체(어떤 AOP프레임워크내 사용가능할수 있는 대상을 제외하고)를 참조할수 있는 advice는 의존성 삽입(Dependency Injection)의 의해 제공되는 모든 플러그인 가능한 능력으로 부터 이익을 취할수 있다.

5.5.2. 자바빈 프라퍼티

Spring에서 제공되는 대부분의 FactoryBean구현물처럼 ProxyFactoryBean은 자체가 자바빈이다. 이것의 프라퍼티는 다음을 위해 사용된다.

☒ 당신이 프록시 되기를 원하는 대상을 명시

☒ CGLIB를 사용할지에 대한 명시

몇몇 핵심적인 프라퍼티는 모든 AOP프록시 factory를 위한 수퍼클래스인 org.springframework.aop.framework.ProxyConfig로부터 상속된다. 그것들은 다음을 포함한다.

☒ proxyTargetClass: 만약 우리가 인터페이스보다는 대상 클래스를 프록시화 한다면 true이다. 만약 이것이 true이면 우리는 CGLIB를 사용할 필요가 있다.

☒ optimize: 생성된 프록시를 위해 공격적인 최적화를 적용할지에 대한 값. 만약 당신이 관련 AOP프록시가 최적화를 다루는 방법을 이해하지 못한다면 이 셋팅을 사용하지 말라. 이것은 현재 CGLIB프록시를 위해서만 사용된다. 이것은 JDK동적 프록시에 대해 영향을 끼치지 않는다.(디폴트)

☒ frozen: advice변경이 프록시 factory가 설정되었을때 허용될지에 대한 값. 디폴트는 false이다.

☒ exposeProxy: 현재 프록시가 대상에 의해 접근될수 있는 ThreadLocal으로 활성화될지에 대한 값. (이것은 ThreadLocal을 위한 필요성 없이 MethodInvocation을 통해 사용가능하다.) 만약 대상이 프록시와 exposeProxy을 얻을 필요가 있다면 true이다. 대상은 AopContext.currentProxy()메소드를 사용할수 있다.

☒ aopProxyFactory: 사용하기 위한 AopProxyFactory의 구현물. 동적 프록시, CGLIB또는 다른 프록시 전략을 사용할지에 대한 커스터마이징의 방법을 제공한다. 디폴트 구현물은 동적 프록시나 CGLIB를 선호한다. 이 프라퍼티를 사용할 필요는 없다. 이것은 Spring 1.1내 새로운 프록시 타입의 추가를 허용하는 경향이 있다.

ProxyFactory빈을 위해 명시하는 다른 프라퍼티는 다음을 포함한다.

☒ proxyInterfaces: Spring인터페이스 이름의 배열. 만약 이것을 제공하지 않는다면 대상 클래스를 위한 CGLIB프록시는 사용될것이다.

☒ interceptorNames: 적용하기 위한 Advisor, 인터셉터 또는 다른 advice이름의 문자열 배열. 정렬은 중요하다. 처음들어와서 처음 제공하는 그것이다. 리스트내 첫번째 인터셉터는 만약 이것이 통상의 MethodInterceptor 나 BeforeAdvice에 관계한다면 첫번째 호출의 인터셉터를 가능하게 할것이다.

  이름들은 조상 factory로 부터 bean 이름을 포함한 현재 factory내 bean 이름들이다. 당신은 advice의 싱글톤 셋팅을 무시하는 ProxyFactoryBean으로 결과를 내기 때문에 여기에 bean참조를 언급할수 없다.

  당신은 별표(*) 를 인텃베터 이름에 덧붙일수 있다. 이것은 적용될 별표(*) 앞에 부분으로 시작하는 이름으로 모든 advisor 빈즈의 애플리케이션으로 결과를 생성한다. 이 특징을 사용하는 예제는 아래에서 찾을수 있다.

☒ 싱글톤 : factory가 하나의 객체를 반환하거나 하지 않거나 종종 getObject()메소드가 호출되는 것은 문제가 아니다. 다양한 FactoryBean구현물은 그러한 메소드를 제공한다. 디폴트 값은 true이다. 만약 당신이 상태유지(stateful) advice를 사용하길 원한다면 예를 들어 상태유지 mixin. false의 싱글톤 값을 가지고 프로토타입 advice를 사용하라.

5.5.3. 프록시 인터페이스

액션내 ProxyFactoryBean의 간단한 예제를 보자. 이 예제는 다음을 포함한다.

☒ 프록시 될 대상 bean. 이것은 밑의 예제내 "personTarget" bean 정의이다.

☒ advice를 제공하기 위해 사용되는 advisor와 인터셉터

☒ AOP프록시 bean정의는 적용할 advice에 따라 대상 객체(personTarget bean)와 프록시를 위한 인터페이스를 명시한다.

```
<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>
```

interceptorNames 프라퍼티는 문자열 타입의 리스트(현재 factory내 인터셉터나 advisor의 bean이름들)를 가진다. advisor, 인터셉터, before, after return 그리고 throw advice객체는 사용될수 있다.
advisor의 절렬은 중요하다.
당신은 왜 리스트가 bean참조를 유지하지 않는지 이상할것이다. 이유는 만약 ProxyFactoryBean의 싱글톤 프라퍼티가 false로 셋팅된다면 이것은 비의존적인 프록시 인스턴스를 반환하는 것이 가능해야만 한다는 것이다. 만약 어느 advisor 자체가 프로토타입이라면 비의존적인 인스턴스는 반환될 필요가 있을것이다. 그래서 이것은 factory로 부터 프로토타입의 인스턴스를 얻는 것이 가능하게 되는 것이 필요하다. 참조를 유지하는 것은 중요하지 않다.

위의 "person" bean정의는 다음처럼 Person구현물을 대체하여 사용될수 있다.

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

같은 IoC컨텍스트내 다른 빈즈는 보통의 자바객체를 사용하는 것처럼 강력한 의존성을 표시할수 있다.

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref local="person" /></property>
</bean>
```

이 예제내 PersonUser클래스는 Person타입의 프라퍼티를 드러낸다. 이것이 관여된만큼 AOP프록시는

"실제" person구현물을 대신해서 투명하게 사용될수 있다. 어쨌든 이 클래스는 동적 프록시 클래스가 될 것이다. 이것은 이것을 Advised인터페이스(밑에서 언급되는)로 형변환하는 것이 가능할 것이다.

다음처럼 익명의 내부 bean을 사용하는 대상과 프록시사이의 차이을 숨기는 것은 가능하다. 단지 ProxyFactoryBean정의만이 다르다. advice는 완성도를 위해서만 포함된다.

```
<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>

    <!-- Use inner bean, not local reference to target -->
    <property name="target">
        <bean class="com.mycompany.PersonImpl">
            <property name="name"><value>Tony</value></property>
            <property name="age"><value>51</value></property>
        </bean>
   </property>

    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

이것은 오직 Person타입의 객체만이 있다는 장점을 가진다. 우리가 애플리케이션 컨텍스트의 사용자가 un-advised객체에 대한 참조를 얻는것을 방지하길 원하거나 Spring IoC autowiring으로 어떤 모호함을 피할 필요가 있다면 유용하다. ProxyFactoryBean정의내 장점은 스스로 포함(self-contained)한다는 것이다. 어쨌든 factory로 부터 un-advised대상를 얻는것이 가능할때 실질적으로 장점이 될수 있다.

5.5.4. 프록시 클래스

만약 당신이 하나 이상의 인터페이스 보다 클래스를 프록시 할 필요가 있다면 무엇인가.?

위에 있는 우리의 예제를 생각해보라. Person인터페이스는 없다. 어떠한 비지니스 인터페이스도 구현하지 않는 Person을 호출하는 클래스를 advise할 필요가 있다. 이 경우 당신은 동적 프록시 보다 CGLIB프록시를 사용하기 위해 Spring을 설정할수 있다. 위 ProxyFactoryBean의 proxyTargetClass 프라퍼티를 간단하게 true로 셋팅하라. 클래스보다는 인터페이로 작업을 수행하는 것이 최상이지만 인터페이스를 구현하지 않는 클래스를 advise하는 기능은 기존 코드로 작업할때 유용할수 있다(일반적으로 Spring은 규범적이지 않다. 이것이 좋은 상황을 적용하는 것이 쉽다면 이것은 특정 접근법에 집중하는것을 피한다.)

만약 당신이 원한다면 당신이 인터페이스를 가지고 작업하는 경우조차도 CGLIB를 사용하는 것에 집중할수 있다.

CGLIB프록시는 수행시 대상 클래스의 하위 클래스를 생성함으로써 작동한다. Spring은 초기 대상에 메소드 호출을 위임하기 위한 생성된 하위클래스를 설정한다. 이 하위클래스는 advice내 짜여진 Decorator패턴을 구현한다.

CGLIB프록시는 사용자에게는 알기쉬워야한다. 어쨌든 여기에 생각해 볼 몇가지 사항들이 있다.

☒ Final 메소드는 오버라이드가 될수 없는것처럼 advised 될수 없다.

☒ 당신은 클래스패스내 CGLIB2 바이너리가 필요할것이다. 동적 프록시는 JDK와 함께 사용가능하다.

CGLIB프록시와 동적 프록시사이에는 작은 성능상의 차이가 있다. Spring 1.0에서 동적 프록시는 미세하게 좀더 빠르다. 어쨌든 이것은 차후에 변경될것이다. 성능은 이 경우에 명백하게 결정될수가 없다.

5.5.5. 'global' advisor 사용하기

인터셉터 이름에 별표(*) 를 덧붙임으로써 별표(*)앞의 부분과 대응되는 bean이름을 가지는 모든 advisor는 advisor연계에 추가될것이다. 이것은 당신이 표준적인 'global' advisor의 모음을 추가할 필요가 있다면 편리하다.

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target"><ref bean="service"/></property>
  <property name="interceptorNames">
    <list>
      <value>global*</value>
    </list>
  </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance"
  class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

5.6. 편리한 프록시 생성

종종 우리는 트랜잭션 관리와 같은 하나의 aspect에만 관심을 가지지 때문에 ProxyFactoryBean의 모든 기능이 필요하지 않다.

특정 aspect에 집중하기를 원할때 우리가 AOP프록시를 생성하기 위해 사용할수 있는 편리한 많은 factory가 있다. 다른 장에서 언급되었기 때문에 우리는 여기서 그것들 중 몇가지의 빠른 조사결과를 제공할것이다.

5.6.1. TransactionProxyFactoryBean

Spring에 포함된 jPetStore 샘플 애플리케이션은 TransactionProxyFactoryBean의 사용을 보여준다.

TransactionProxyFactoryBean은 ProxyConfig의 하위클래스이다. 그래서 기초적인 설정은 ProxyFactoryBean와 공유한다. (위의 ProxyConfig 프라퍼티의 목록을 보라.)

jPetStore으로부터 다음의 예제는 이것이 어떻게 작동을 하는지 보여준다. ProxyFactoryBean를 사용하는 상태에서 여기에 대상 bean정의가 있다. 의존성은 대상 POJO("petStoreTarget")보다 프록시화된 factory bean정의(여기엔 "petStore")에 표시되어야 한다.

TransactionProxyFactoryBean은 대상과 트랜잭션 적인것과 요구되는 전달(propagation), 그리고 다른 셋팅이 될 방법을 명시하는 "트랜잭션 속성"에 대한 정보를 요구한다.

```
<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>
```

ProxyFactoryBean를 사용하는 상태에서 우리는 가장 상위레벨의 대상 bean에 대한 참조대신에 target프라퍼티의 값을 셋팅하는 내부 bean을 사용하는것을 선택해야만 한다.

TransactionProxyFactoryBean은 트랜잭션 속성에 기초를 둔 pointcut을 포함하는 트랜잭션 advisor를 자동적으로 생성한다. 그래서 오직 트랜잭션적인 메소드만이 advised된다.

TransactionProxyFactoryBean은 선인터셉터(preInterceptors)와 후인터셉터(postInterceptors) 프라퍼티를 사용하는 "선" 과 "후" advice의 명시를 허용한다. 트랜잭션 인터셉터의 앞이나 뒤 인터셉션 연계작업에 두기 위한 인터셉터의 객체 배열, 다른 advice또는 advisor를 가진다. 다음처럼 XML bean정의내 <list> 요소를 사용하여 활성화될수 있다.

```
<property name="preInterceptors">
    <list>
        <ref local="authorizationInterceptor"/>
        <ref local="notificationBeforeAdvice"/>
    </list>
</property>
<property name="postInterceptors">
    <list>
        <ref local="myAdvisor"/>
    </list>
</property>
```

이 프라퍼티들은 위의 "petStore" bean정의에 추가될수 있다. 공통적인 사용법은 선언적인 보안과 함께 트랜잭션을 조합하는 것이다. EJB에 의해 제공되는 것과 유사한 접근법이다.

ProxyFactoryBean처럼 bean이름보다 실질적인 인스턴스 참조의 사용때문에 선,후 인터셉터는 오직 공유-인스턴스 advice만이 사용될수 있다. 게다가 그것들은 상태유지(stateful) advice를 위해서는 유용하지 않다. 이것은 TransactionProxyFactoryBean의 목적과 일치한다. 이것은 공통적인 트랜잭션 셋업의 간단한 방법을 제공한다. 만약 당신이 좀더 복잡하고, 사용자 정의된 AOP를 원한다면 일반적인 ProxyFactoryBean을 사용하거나 자동 프록시 생성자를 사용하는것을 고려하라.
특별히 우리가 많은 경우에 EJB의 대체물로 Spring AOP를 본다면 우리는 대부분의 advice가 꽤 일반적이고 공유-인스턴스 모델을 사용하는것을 알게된다. 선언적인 트랜잭션 관리와 보안 체크는 표준적인 예제이다.

TransactionProxyFactoryBean은 이것의 transactionManager 자바빈 프라퍼티를 통해

구현물에 의존한다. 이것은 JTA, JDBC또는 다른 전략에 기초를 둔 플러그인 가능한 트랜잭션 구현물을 허용한다. 이것은 AOP보다 Spring트랜잭션 추상화와 관련이 있다. 우리는 다음 장에서 트랜잭션 내부구조에 대해 언급할것이다.
만약 당신이 선언적인 트랜잭션 관리에만 관심을 가진다면 TransactionProxyFactoryBean은 ProxyFactoryBean보다 간단하고 좋은 해결방안이다.

5.6.2. EJB 프록시

다른 전용 프록시는 호출 코드에 의해 직접적으로 사용되기 위한 EJB "business methods" 인터페이스를 가능하도록 하는 EJB를 위한 프록시를 생성한다. 호출 코드는 JNDI룩업을 수행하거나 EJB생성 메소드를 사용하기 위해 필요하지 않다. 이것은 읽기가능한 성질과 구조적인 유연성에 명백한 향상을 가진다.

더 많은 정보를 위해서 이 문서의 Spring EJB서비스 관련 장을 보라.

5.7. 간결한 프록시 정의

특별히 트랜잭션적인 프록시를 정의할때 당신은 많은 유사한 프록시 정의로 끝낼지도 모른다. 내부 bean정의에 따라 부모및 자식 bean정의의 사용은 좀더 깔끔하고 좀더 갈결한 프록시 정의라는 결과를 만들수 있다.

첫번째 부모, 템플릿, bean 정의는 프록시를 위해 생성된다.

```
<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>
```

이것은 자체적으로 인스턴스화가 결코 될수 없다. 그래서 실제로 완벽하지 않을지도 모른다. 생성될 필요가 있는 각각의 프록시는 내부 bean정의처럼 프록시의 대상을 포장하는 자식 bean정의이다. 대상은 자기 자신의 것이 결코 사용되지 않을것이다.

```
<bean id="myService" parent="txProxyTemplate">
    <property name="target">
      <bean class="org.springframework.samples.MyServiceImpl">
      </bean>
    </property>
</bean>
```

트랜잭션 위임(propagation) 셋팅과 같은 경우처럼 부모 템플릿으로부터 프라퍼티를 오버라이드 하는것은 물론 가능하다.

```
<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>
```

위 예제에서 우리는 부모 bean정의를 앞서 서술된것처럼 abstract 속성을 사용해서 abstract처럼 명시적으로 표시한다. 그래서 이것은 실질적으로 인스턴스화 된 적이 없을것이다. 애플리케이션 컨텍스트(간단한 bean factory는 아닌)는 디폴트로 모든 싱글톤이 미리 인스턴스화될것이다. 그러므로 이것은 당신이 템플릿처럼 사용할 경향이 있는 (부모) bean정의를 가지고 이 정의가 클래스를 명시한다면 당신은 abstract속성을 true로 셋팅해야만 하고 반면에 애플리케이션 컨텍스트는 실질적으로 이것을 먼저 인스턴스화할 것이다.

5.8. ProxyFactory로 프로그램적으로 AOP프록시를 생성하기.

Spring을 사용해서 프로그램적으로 AOP프록시를 생성하는 것은 쉽다. 이것은 당신에게 Spring IoC에서 의존성없이 Spring AOP를 사용하는것을 가능하게 한다.

다음의 리스트는 하나의 인터셉터와 하나의 advisor로 대상 객체를 위한 프록시의 생성을 보여준다. 대상 객체에 의해 구현된 인터페이스는 자동적으로 프록시화될것이다.

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

첫번째 단계는 org.springframework.aop.framework.ProxyFactory 타입의 객체를 생성하는 것이다. 당신은 위 예제처럼 대상 객체와 함께 이것을 생성할수 있거나 대안이 되는 생성자로 프록시될 인터페이스를 명시할수 있다.

당신은 인터셉터나 advisor을 추가할수 있고 ProxyFactory의 생명을 위해 그것들을 조작할수 있다. 만약 당신이 IntroductionInterceptionAroundAdvisor를 추가한다면 당신은 추가적인 인터페이스를 위한 프록시를 야기할수 있다.

또한 ProxyFactory에는 당신에게 before advice와 throw advice와 같은 다른 advice타입을 추가하도록 허용하는 편리한 메소드(AdvisedSupport로부터 상속된)가 있다. AdvisedSupport는 ProxyFactory와 ProxyFactoryBean모두의 수퍼클래스이다.
IoC프레임워크를 사용해서 AOP프록시 생성을 통합하는 것은 대부분의 애플리케이션에서 최상의 선택이다. 우리는 당신이 일반적인 것처럼 AOP를 사용해서 자바코드로 부터 설정을 구체화하는것을 추천한다.

5.9. advised 객체 조작하기.

당신이 AOP프록시를 생성한다고 해도 당신은 org.springframework.aop.framework.Advised인터페이스를 사용해서 그것들을 조작할수 있다. 어떤 AOP프록시가 다른 어또한 인터페이스를 구현한다고 해도 이 인터페이스로 형변환될수 있다. 이 인터페이스는 다음의 메소드를 포함한다.

```
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();
```

getAdvisors() 메소드는 모든 advisor, 인터셉터 또는 factory에 추가될수 있는 다른 advice타입을 위해 advisor을 반환할것이다. 만약 당신이 advisor를 추가한다면 이 시점에 반환되는 advisor는 당신이 추가한 객체가 될것이다. 만약 당신이 인텃베터나 다른 advice타입을 추가한다면 Spring은 이것을 언제나 true를 반환하는 pointcut를 가지고 advisor로 포장할것이다. 게다가 당신이 MethodInterceptor을 추가한다면 이 시점을 위해 반환된 advisor는 당신의 MethodInterceptor과 모든 클래스와 메소드에 대응되는 pointcut을 반환하는 DefaultPointcutAdvisor가 될것이다.

addAdvisor()메소드는 어떤 advisor을 추가하기 위해 사용될수 있다. 언제나 pointcut와 advice를 유지하는 advisor는 어떤 advice나 pointcut(introduction은 아닌)와 사용될수 있는 일반적인 DefaultPointcutAdvisor이 될것이다.

디폴트에 의해 한번 프록시가 생성되었을때 advisor나 인터셉터를 추가하거나 제거하는것은 가능하다. 오직 제한은 factory로 부터 존재하는 프록시가 인터페이스 변경을 보여주지 않을것이라는 것처럼 introduction advisor을 추가하거나 제거하는것이 불가능하다.(당신은 이 문제를 피하기 위해 factory로 부터 새로운 프록시를 얻을수 있다.)

AOP프록시를 Advised 인터페이스로 형변환하고 이것의 advice를 시험하고 조작하는것의 간단한 예제이다.

```
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);
```

이것은 비록 정확한 사용법이 의심스럽지 않더라도 제품내 비지니스 객체의 advice를 변경하는 것이 현명한것인지(advisable) 아닌지 의심스러울 것이다. 어쨌든 이것은 개발에서 매우 유용할수 있다. 예를 들면, 테스트에서 내가 테스트하길 원하는 메소드 호출내에서 얻어지는 인터셉터나 다른 advice의 형태내 테스트코드를 추가하는것이 가능하게 되는것이 매우 유용하다는 것을 때때로 발견하곤 한다.(예를 들어,

advice는 그 메소드(예를 들면, 롤백을 위해 트랜잭션을 표시하기 전에 데이터베이스가 정확하게 수정되었는지 체크하기 위해 SQL수행하는것) 생성된 트랜잭션 내부에서 얻어질수 있다.

당신이 프록시를 생성하는 방법에 대해 의존하라. 당신은 Advised isFrozen()메소드가 true를 반환하고 추가나 삭제를 통해 advice를 변경하는 어떤 시도가 AopConfigException을 결과로 보내는 경우에 frozen 플래그를 셋팅할수 있다. advised 객체의 상태를 고정하기 위한 능력은 몇몇 경우에 유용하다. 예를 들면 보안 인터셉터를 제거하는 호출 코드를 제한하기 위해. 이것은 아마도 수행 advice 변경이 요구되지 않는다면 공격적인 최적화를 허용하기 위해 Spring 1.1내에서 사용된다.

5.10. "autoproxy" 기능 사용하기

지금까지 우리는 ProxyFactoryBean이나 유사한 factory bean을 사용해서 AOP프록시의 명시적인 생성을 설명했다.

Spring은 또한 자동적으로 선택된 bean정의를 프록시할수 있는 "autoproxy" bean정의를 사용하는 것을 허용한다. 이것은 Spring에 컨테이너 로드처럼 어떤 bean정의의 변경을 가능하게 하는 "bean 후 처리자" 구조로 내장되었다.

이 모델에서 당신은 자동 프록시 내부구조를 설정하는 XML bean정의파일내 몇몇 특별한 bean정의를 셋업한다. 이것은 당신에게 자동프록시를 위해 적당한 대상을 선언하도록 허용한다. 당신은 ProxyFactoryBean을 사용할 필요가 없다.

이것을 하기 위한 두가지 방법이 있다.

☒ 현재 컨텍스트내 bean을 명시하기 위해 참조하는 autoproxy 생성자 사용하기.

☒ 개별적으로 검토되기 위한 가치가 있는 autoproxy 생성의 특별한 경우. autoproxy 생성은 소스레벨 메타데이타 속성에 의해 이루어진다.

5.10.1. autoproxy bean정의

org.springframework.aop.framework.autoproxy패키지는 다음의 표준적인 autoproxy 생성자를 제공한다.

5.10.1.1. BeanNameAutoProxyCreator

BeanNameAutoProxyCreator는 이름과 문자값또는 와일드카드가 들어맞는 bean을 위해 자동적으로 AOP프록시를 생성한다.

```
<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>
```

ProxyFactoryBean를 사용하는것처럼 프로토타입 advisor을 위해 정확한 행위를 허용하는 인터셉터의 리스트보다 interceptorNames 프라퍼티가 있다. 명명된 "인터셉터"는 advisor나 다른 advice타입이 될수

있다.

일반적으로 자동 프록시를 사용하는 것처럼 BeanNameAutoProxyCreator를 사용하는 중요한 점은 다중객체에 일관적으로 같은 설정을 적용하고 최소한의 설정을 가진다. 이것은 다중 객체에 선언적인 트랜잭션을 적용하기 위해 널리 알려진 선택이다.

위 예제에서 "jdkMyBean" 과 "onlyJdk"처럼 이름이 대응되는 bean정의는 대상 클래스를 가지는 명백한 옛 bean정의이다. AOP프록시는 BeanNameAutoProxyCreator에 의해 자동적으로 생성될것이다. 같은 advice는 모든 대응되는 bean에 적용될것이다. 만약 advisor가 사용된다면(위 예제안의 인터셉터보다) pointcut은 다른 bean에 다르게 적용될것이다.

5.10.1.2. DefaultAdvisorAutoProxyCreator

좀더 일반적이고 굉장히 강력한 자동 프록시 생성자는 DefaultAdvisorAutoProxyCreator이다. 이것은 autoproxy advisor의 bean정의내 특정 bean이름을 포함할 필요없이 현재 컨텍스트에 적절한 advisor를 자동적으로 적용할것이다. 이것은 일관적인 설정의 장점과 BeanNameAutoProxyCreator처럼 중복의 회피를 제공한다.

이 기법을 사용하는 것은 다음을 포함한다.

☒ DefaultAdvisorAutoProxyCreator bean정의를 명시하기.

☒ 같거나 관련된 컨텍스트내 많은 수의 advisor명시하기. 인터셉터나 다른 advice가 아닌 advisor이 되어야만 한다. 이것은 평가하기 위한, 후보 bean정의를 위해 각각의 advice의 적절함을 체크하기 위한 pointcut가 되어야만 하기 때문에 필요하다.

DefaultAdvisorAutoProxyCreator는 각각의 비지니스 객체(예제내에서 "businessObject1" 과 "businessObject2" 와 같은) 를 적용해야하는 advice가 무엇인지 보기 위해 각각의 advisor내 포함된 pointcut를 자동적으로 평가할것이다.

이것은 많은 수의 advisor가 각각의 비지니스 객체에 자동적으로 적용될수 있다. 만약 어떠한 advisor내 pointcut이 비지니스 객체내 어떠한 메소드와도 대응되지 않는다면 객체는 프록시화 되지 않을것이다. bean정의가 새로운 비지니스 객체를 위해 추가된다면 그것들은 필요할때 자동적으로 프록시화될것이다.

일반적인 자동프록시는 un-advised객체를 얻기 위한 호출자나 의존적인 것을 불가능하게 만드는 장점을 가진다. 이 ApplicationContext의 getBean("businessObject1")을 호출하는 것은 대상 비지니스 객체가 아닌 AOP프록시를 반환할것이다. ("내부 bean" 표현형식은 좀더 빨리 보여지고 또한 이 이득을 제공한다.)

```
<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>
```

DefaultAdvisorAutoProxyCreator는 만약 당신이 많은 비지니스 객체에 일관적으로 같은 advice를 적용하길 원한다면 매우 유용하다. 내부구조 정의가 대체될때 당신은 특정 프록시 설정을 포함하는것 없이 새로운 비지니스 객체를 간단하게 추가할수 있다. 당신은 예를 들어 설정의 최소한의 변경으로 추적및 성능 모니터링 aspect처럼 추가적인 aspect를 매우 쉽게 감소시킬수있다.

DefaultAdvisorAutoProxyCreator는 필터링과 정렬을 위한 지원을 제공한다.(명명 규칙을 사용하는 것은 같은 factory내 AdvisorAutoProxyCreators를 오직 특정 advisor가 평가하고, 다중 사용을 허용하고, 다르게 설정된다.) advisor는 이것이 쟁점이라면 정확한 정렬을 보장하기 위한 org.springframework.core.Ordered 인터페이스를 구현할수 있다. 위 예제에서 사용된 TransactionAttributeSourceAdvisor는 설정가능한 정렬값을 가지지만 디폴트는 정렬되지 않는다.

5.10.1.3. AbstractAdvisorAutoProxyCreator

이것은 DefaultAdvisorAutoProxyCreator의 수퍼클래스이다. advisor정의가 프레임워크 DefaultAdvisorAutoProxyCreator의 행위를 위해 부족한 사용자지정을 제공하는 가능성이 희박한 경우에 당신은 이 클래스를 하위클래스화하여 당신 자신의 autoproxy생성자를 생성할수 있다.

5.10.2. 메터데이타-지향 자동 프록시 사용하기.

autoproxy의 특별히 중요한 타입은 메타데이타에 의해 다루어진다. 이것은 .NET ServicedComponents에 유사한 프로그래밍 모델을 생산한다. EJB처럼 XML배치 서술자를 사용하는 대신에 트랜잭션 관리와 다른 기업용 서비스를 위한 설정은 소스레벨 속성내 유지된다.

이 경우 당신은 메터데이타 속성을 이해하는 advisor와의 조합으로 DefaultAdvisorAutoProxyCreator을 사용한다. 이 메터데이타는 autoproxy생성 클래스 자체보다는 후보 advisor의 pointcut부분내 유지됨을 명시한다.

이것은 DefaultAdvisorAutoProxyCreator의 특별한 경우이다. 하지만 그것 자신의 보상을 할만하다. (메터데이타-인식 코드는 AOP프레임워크 자체가 아닌 advisor내 포함된 pointcut내에 있다.)

jPetStore샘플 애플리케이션 의 /attributes디렉토리는 속성-지향 자동프록시의 사용을 보여준다. 이 경우 TransactionProxyFactoryBean를 사용할 필요는 없다. 메터데이타-인식 pointcut의 사용이기 때문에 간단하게 비지니스 객체의 트랜잭션적인 속성을 정의하는 것은 충분하다. bean정의는 /WEB-INF/declarativeServices.xml내 다음의 코드를 포함한다. 이것은 일반적이고 jPetStore밖에서 사용될수 있다.

```
<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"
/>
```

이 경우 "autoProxyCreator"라고 불리는 DefaultAdvisorAutoProxyCreator bean정의는 이름이 중요하지 않지만(이것은 생략될수도 있다.) 현재의 애플리케이션 컨텍스트내 모든 적절한 pointcut를 가져올것이다. 이 경우 TransactionAttributeSourceAdvisor타입의 "transactionAdvisor" bean정의는 클래스나 트랜잭션 속성을 운반하는 메소드에 적용할것이다. TransactionAttributeSourceAdvisor는 생성자 의존성을 통해 TransactionInterceptor에 의존한다. 예제는 autowiring을 통해 이것을 해결한다. AttributesTransactionAttributeSource는 org.springframework.metadata.Attributes인터페이스의 구현물에 의존한다. 이 잔해에서 "attributes" bean은 속성정보를 얻기 위해 Jakarta Commons Attributes API 사용하는 것을 만족한다. (애플리케이션 코드는 Commons Attributes 집계작업을 사용하여 컴파일되어야만 한다.)

여기에 정의된 TransactionInterceptor는 이것은 애플리케이션의 트랜잭션 요구사항(전형적으로 예제에서 처럼 JTA, 또는 Hibernate, JDO, JDBC)에 명시될 것이기 때문에 이 일반적인 파일내 포함되지 않는 PlatformTransactionManager정의에 의존한다.

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

만약 당신이 선언적인 트랜잭션 관리만을 요구한다면 이러한 일반적인 XML정의를 사용하는 것은 Spring내에서 트랜잭션 속성을 가진 모든 클래스나 메소드를 자동적으로 프록싱하는 결과를 낳는다. 당신은 AOP로 직접적으로 작업을 하는것을 필요로 하지 않을것이고 프로그래밍 모델은 .NET ServicedComponents의 그갓과 유사하다.

이 기법은 확장가능하다. 사용자지정 속성에 기반하여 자동프록싱을 하는것은 가능하다. 당신이 다음 사항을 할 필요가 있다.

☒ 당신의 사용자 지정 속성 명시하기.

☒ 클래스나 메소드의 사용자지정 속성의 존재에 의해 처리되는 pointcut를 포함하는 필요한 advice를 가진 advisor명시하기. 당신은 사용자 지정 속성을 가져오는 정적 pointcut를 거의 구현하는 존재하는 advice를 사용하는것이 가능할지도 모른다.

각각의 advised클래스에 유일하기 위한 그러한 advisor(예를 들면 mixin)은 가능하다. 그것들은 싱글톤, bean정의보다는 프로토타입처럼 간단하게 정의될 필요가 있다. 예를 들어 위에서 보여진 Spring 테스트 슈트로부터의 LockMixin 소개(introduction) 인터셉터는 여기서 보여지는 것처럼 목표 mixin에 속성-지향 pointcut이 결합되어 사용되는 것이 가능하다. 우리는 자바빈 프라퍼티를 사용하여 설정된 포괄적인 DefaultPointcutAdvisor을 사용한다.

```
<bean id="lockMixin"
    class="org.springframework.aop.LockMixin"
    singleton="false"
```

```
/>

<bean id="lockableAdvisor"
    class="org.springframework.aop.support.DefaultPointcutAdvisor"
    singleton="false"
>
    <property name="pointcut">
        <ref local="myAttributeAwarePointcut"/>
    </property>
    <property name="advice">
        <ref local="lockMixin"/>
    </property>
</bean>

<bean id="anyBean" class="anyclass" ...
```

만약 속성 인식 pointcut이 anyBean이나 다른 bean정의내 어떠한 메소드와 대응된다면 mixin은 적용될것이다. lockMixin와 lockableAdvisor은 프로토타입이라는것에 주의하라. myAttributeAwarePointcut pointcut은 개별적인 advised객체를 위한 상태를 유지하지 않는 것처럼 싱글톤 정의가 될수 있다.

5.11. TargetSources 사용하기

Spring은 org.springframework.aop.TargetSource인터페이스내에서 표현되는 TargetSource의 개념을 제공한다. 이 인터페이스는 joinpoint를 구현하는 "대상 객체(target object)"를 반환하는 책임을 가진다. TargetSource구현은 AOP프록시가 메소드 호출을 다루는 시점마다 대상 인스턴스를 요청한다.

Spring AOP를 사용하는 개발자는 대개 TargetSources를 직접적으로 작업할 필요가 없다. 하지만 이것은 풀링, 핫 스왑 그리고 다른 정교한 대상을 지원하는 강력한 방법을 제공한다. 예를 들면 풀링 TargetSource는 인스턴스를 관리하기 위한 풀을 사용하여 각각의 호출을 위한 다른 대상 인스턴스를 반환할수 있다.

만약 당신이 TargetSource을 명시하지 않는다면 디폴트 구현물은 로컬 객체를 포장하는것이 사용된다. 같은 대상은 (당신이 기대하는것처럼) 각각의 호출을 위해 반환된다.

Spring에 의해 제공되는 표준적인 대상 소스를 보자. 그리고 당신이 그것들을 어떻게 사용할수 있는지도 보자.
사용자 지정 대상 소스를 사용할때 당신의 대상은 싱글톤 bean정의보다 프로토타입이 될 필요가 있을것이다. 이것은 요구될때 Spring이 새로운 대상 인스턴스를 생성하는것을 허용한다.

5.11.1. 핫 스왑가능한 대상 소스

org.springframework.aop.target.HotSwappableTargetSource는 이것에 대한 참조를 유지하기 위한 호출자를 허용하는 동안 교체되기 위한 AOP프록시의 대상을 허용하기 위해 존재한다.

대상 소스의 대상을 변경하는 것은 즉시 영향을 끼친다. HotSwappableTargetSource는 쓰레드에 안전하다(threadsafe).

당신은 다음처럼 HotSwappableTargetSource의 swap()메소드를 통해 대상을 변경할수 있다.

```
HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

요구되는 XML정의는 다음처럼 볼수 있다..

```
<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>
```

위의 swap() 호출은 스왑가능한 bean의 대상을 변경한다. 그 bean에 대한 참조를 유지하는 클라이언트는 변경을 인식하지 못할것이지만 새로운 대상에 즉시 도달할것이다.

비록 이 예제는 어떠한 advice를 추가하지 않고 TargetSource를 사용하기 위한 advice를 추가할 필요가 없다. 물론 어떤 TargetSource는 임의의 advice로 결합하여 사용될수 있다.

5.11.2. 풀링 대상 소스

풀링 대상 소스를 사용하는것은 일치하는 인스턴스의 풀이 메소드 호출로 풀내 객체가 자유롭게 되는 방식으로 유지되는 비상태유지(stateless) 세션 EJB와 유사한 프로그래밍 모델을 제공한다.

Spring풀링과 SLSB풀링 사이의 결정적인 차이점은 Spring풀링은 어떠한 POJO에도 적용될수 있다는 것이다. 대개 Spring을 사용하는 것은 이 서비스가 비-침략적인 방법으로 적용될수 있다.

Spring은 상당히 효과적인 풀링 구현물을 제공하는 Jakarta Commons Pool 1.1을 위한 특별한 지원을 제공한다. 당신은 이 기능을 사용하기 위해 애플리케이션 클래스패스내 commons-pool.jar파일이 필요할것이다. 이것은 다른 풀링 API를 지원하기 위해 org.springframework.aop.target.AbstractPoolingTargetSource의 하위클래스를 구현하는 것이 가능하다.

샘플 설정은 아래에서 보여진다.

```
<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>
```

예제내 대상 객체인 "businessObjectTarget"이 프로토타입이 되어야만 한다는 것에 주의하라. 이것은 PoolingTargetSource구현물이 필요할때 풀의 증가를 위한 대상의 새로운 인스턴스를 생성하는것을 허용한다. 이것의 프라퍼티에 대한 정보를 위해 사용하기 위한 AbstractPoolingTargetSource와 견고한 하위클래스를 위한 JavaDoc를 보라. maxSize는 가장 기본적이고 표현되기 위해 항상 보증된다.

이 경우 "myInterceptor"는 같은 IoC컨텍스트내 정의될 필요가 있는 인터셉터의 이름이다. 이것은 풀링을 사용하기 위해 인터셉터를 명시할 필요가 없다. 만약 당신이 오직 풀링만을 원하고 다른 advice는 원하지 않는다면 interceptorNames 프라퍼티를 전부 셋팅하지 말라.

소개(introduction)을 통해 풀의 설정과 현재 크기에 대한 정보를 드러내는 org.springframework.aop.target.PoolingConfig인터페이스를 위한 어떤 풀링된 객체를 형변환하는것을 가능하게 하는것처럼 Spring을 설정하는것은 가능하다. 당신은 이것처럼 advisor을 명시할 필요가 있을것이다.

```
<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>
```

이 advisor는 AbstractPoolingTargetSource클래스의 편리한 메소드를 호출하고 나아가 MethodInvokingFactoryBean의 사용하여 얻을수 있다. 이 advisor의 이름(여기 "poolConfigAdvisor")은 풀링된 객체를 드러내는 ProxyFactoryBean내 인터셉터 이름의 목록이 될수 있다.

형변환은 다음처럼 보일것이다.

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

풀링 비상태유지(stateless) 서비스 객체는 언제나 필요한것은 아니다. 우리는 이것이 대부분의 비상태유지(stateless) 객체가 근본적으로 쓰레드에 안전하고 인스턴스 풀링은 자원이 캐시된다면 골치거리가 되는것처럼 디폴트 선택이 될것이라고 믿지는 않는다.

좀더 간단한 풀링은 자동프록싱을 사용하여 사용가능하다. 어떤 autoproxy생성자에 의해 사용될수 있는 TargetSources 셋팅은 가능하다.

5.11.3. 프로토 타입 대상 소스

"프로토타입" 대상 소스를 셋업하는 것은 풀링 TargetSource와 유사하다. 이 경우 대상의 새로운 인스턴스는 모든 메소드호출에서 생성될것이다. 비록 새로운 객체를 생성하는 비용이 요즘 JVM내에서는 높지않더라도 새로운 객체(IoC의존성을 만족하는)를 묶는 비용은 좀더 비쌀것이다. 게다가 당신은 매우좋은 이유없이 이 접근법을 사용하지 않을것이다.

이것을 하기 위해 당신은 다음처럼 위에서 보여진 poolTargetSource정의를 변경할수 있다. (나는 명백하게 하기 위해 이름을 변경했다.)

```
<bean id="prototypeTargetSource"
    class="org.springframework.aop.target.PrototypeTargetSource">
    <property name="targetBeanName"><value>businessObjectTarget</value></property>
</bean>
```

여기엔 오직 하나의 프라퍼티(대상 빈의 이름)가 있다. 상속은 일관적인 명명을 확실히 하기 위한 TargetSource구현물내 사용되었다. 풀링 대상 소스를 사용하는 것처럼 대상 bean은 프로토타입 bean정의가 되어야만 한다.

5.11.4. ThreadLocal 대상 소스

ThreadLocal 대상 소스는 만약 당신이 들어오는 각각의 요청(쓰레드마다)을 위해 생성되기 위한 객체가 필요하다면 유용하다. ThreadLocal의 개념은 쓰레드와 함께 자원을 투명하게 저장하기 위한 JDK범위의 기능을 제공한다. ThreadLocalTargetSource를 셋업하는 것은 다른 대상 소스를 위해 설명되는 것과 거의 같다.

```
<bean id="threadlocalTargetSource"
  class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName"><value>businessObjectTarget</value></property>
</bean>
```

ThreadLocals은 멀티-쓰레드와 멀티-클래스로더 환경내 그것들을 정확하게 사용하지 않았을때 다양한 문제(잠재적으로 메모리 누수와 같은 결과)를 발생시킨다. 하나는 몇몇 다른 클래스로 threadlocal를 포장하는 것을 언제나 검토해야만 하고 ThreadLocal자체(물론 래퍼클래스를 제외하고)를 결코 직접적으로 사용하지 말라. 또한 하나는 쓰레드를 위한 로컬 자원을 정확하게 셋팅하고 셋팅하지 않는 것을(후자는 ThreadLocal.set(null)에 대한 호출을 간단하게 포함한다.) 언제나 기억해야만 한다. 셋팅하지 않는것은 문제가 되는 행위를 야기하는 셋팅을 하기 때문에 이 경우 수행될수 있다. Spring의 ThreadLocal지원은 당신을 위해 이것을 하고 다른 임의의 핸들링 코드없이 ThreadLocals를 사용하여 검토되어야만 한다.

5.12. 새로운 Advice 타입을 정의하기

Spring AOP는 확장가능하기 위해 디자인되었다. 인터셉션 구현물 전략이 내부적으로 사용되는 동안 특별히 지원되는 임의의 advice타입에 추가적으로 인터셉션 around advice, before, throw advice그리고 after returning advice를 지원하는 것이 가능하다.

org.springframework.aop.framework.adapter패키지는 핵심 프레임워크 변경없이 추가되기 위한 사용자 지정 advice타입을 위한 지원을 허용하는 SPI패키지이다. 사용자 지정 advice타입의 제한은 org.aopalliance.aop.Advice태그 인터페이스를 구현해야만 한다는 것이다.

더 많은 정보를 위해서 org.springframework.aop.framework.adapter패키지의 JavaDoc를 참조하라.

5.13. 추가적으로 읽을거리와 자원들

나는 AOP에 대한 소개를 위해 Ramnivas Laddad (Manning, 2003)에 의해 쓰여진 훌륭한 AspectJ in Action을 추천한다.

Spring AOP의 좀더 다양한 예제를 위해 Spring샘플 애플리케이션을 참조하라.

☒ JPetStore의 디폴트 설정은 선언적인 트랜잭션 관리를 위한 TransactionProxyFactoryBean의 사용을 설명한다.

☒ JPetStore의 /attributes 디렉토리는 속성-지향 선언적인 트랜잭션 관리의 사용을 설명한다.

만약 당신이 Spring AOP의 좀더 향상된 기능에 관심이 있다면 테스트 슈트를 살펴보라. 테스트 적용범위는 90%이상이다. 그리고 이것은 이 문서에서 언급되지 않은 향상된 기능을 설명한다..

5.14. 로드맵

Spring의 나머지와 같은 Spring AOP는 활발히 개발되고 있다. 핵심 API는 안정적이다. Spring의 나머지처럼 AOP프레임워크는 기초적인 디자인을 보존하는 동안 확장을 가능하게 하는 매우 모듈적이다. 다양한 향상은 이전버전과의 호환성을 보존하는 Spring 1.1에 계획되어 있다. 그것들은 다음을 포함한다.

☒ 성능향상: AOP프록시의 생성은 Strategy인터페이스를 통해 factory에 의해 다루어진다. 게다가 우리는 사용자 코드나 핵심 구현에 영향없이 추가적인 AopProxy타입을 지원할수 있다. CGLIB를 위한 명백한 성능 최적화는 1.0.3버전에 계획되어 있다. 이 경우 Spring 1.1의 최적화는 advice가 수행시 변경되지 않을것이다. 이것은 AOP프레임워크의 오버헤드를 명백하게 감소시킨다. 어쨌든 AOP프레임워크의 오버헤드는 일반적인 사용법에서는 문제가 되지 않는다.

☒ 좀더 의미있는 pointcut: 현재 Spring은 의미있는 Pointcut인터페이스를 제공한다. 하지만 우리는 좀더 pointcut구현물을 추가함으로써 값을 추가할수 있다. 우리는 Spring 설정파일내에서 사용되기 위한 AspectJ pointcut표현을 허용할 AspectJ와의 통합을 본다. 그리고 만약 당신이 유용한 pointcut에 기여하길 바란다면 그렇게 해달라.

대부분의 명백한 향상은 AspectJ커뮤니티와 상호작용할 AspectJ와의 관련 통합일것이다. 우리는 이것이 다음과 같은 영역에서 Spring과 AspectJ둘다를 위한 명백한 이득을 제공할 것이라는것을 믿는다.

☒ Spring IoC를 사용하여 설정되기 위한 AspectJ aspect허용하기. 이것은 선호하는 애플리케이션으로 AspectJ aspect를 Spring aspect가 애플리케이션 IoC컨텍스트와의 통합과 같은 방법으로 통합하는것의 가능성을 가진다.

☒ AspectJ pointcut표현을 대상 Spring advice를 위한 Spring설정으로 허용하기. 이것은 우리 자신의 pointcut표현언어를 디자인하는데 명백한 이득을 가진다. AspectJ는 둘다 잘되고 문서화가 잘되어 있다.

이러한 통합 둘다 Spring 1.1에서 반드시 사용가능하게 될것이다.

Chapter 6. AspectJ 통합

6.1. 개요

Spring의 프록시 기반 AOP프레임워크는 많은 일반적인 미들웨어와 애플리케이션 특유의 문제를 다루는데 매우 적합하다. 어쨌든 좀더 강력한 AOP솔루션이 필요한 때(예를 들면 우리가 클래스에 추가적인 필드를 추가해야 할 필요가 있다거나 Spring IoC컨테이너에 의해 생성되지 않은 잘 정제된 객체를 알리는(advice) 것과 같은)가 자주 있다.

우리는 이러한 경우 AspectJ의 사용을 권한다. 따라서 1.1버전에서 Spring은 AspectJ와의 강력한 통합을 제공한다.

6.2. Spring IoC를 사용하여 AspectJ 설정하기.

Spring/AspectJ 통합의 가장 중요한 부분은 DI를 사용하여 Spring이 AspectJ를 설정하도록 하는것이다. 이것은 객체를 위해 aspect을 사용하는 것과 유사한 이득을 가져온다.

☒ aspect를 위해 특별한 목적의 설정 기법을 사용할 필요가 없다. 그들은 같고, 일관적이고, 전체 애플리케이션을 위해 사용되는 접근법으로 설정될수 있다.

☒ aspect는 애플리케이션 객체의 의존할수 있다. 예를 들면 보안 aspect는 짧은 예제에서 볼수 있는 것처럼 보안 관리자에게 의존할수 있다.

☒ 이것은 관련 Spring컨텍스트를 통해 aspect에 대한 참조를 얻는것이 가능하다. 이것은 aspect의 동적 재설정을 위해 가능할수 있다.

AspectJ aspect는 setter삽입(Injection)을 위해 자바빈즈 프라퍼티를 드러낼수 있다. 그리고 BeanFactoryAware와 같은 Spring생명주기 인터페이스를 구현할수 있다.
AspectJ aspect는 생성자 삽입이나 메소드 삽입을 사용할수 없다는 것에 주의하라. 이 제한은 aspect가 객체의 생성자처럼 호출될수 있는 생성자를 가지지 않기 때문에 발생한다.

6.2.1. "싱글톤" aspects

대부분의 경우 AspectJ aspect는 클래스 로더당 하나의 인스턴스를 가지는 싱글톤이다. 이 하나의 인스턴스는 다중 객체 인스턴스를 알릴 책임을 가진다.

Spring IoC컨테이너는 aspect가 호출가능한 생성자를 가지지 않기 때문에 aspect를 인스턴스화 할수 없다. 하지만 이것은 AspectJ가 모든 aspect를 위해 명시하는 정적인 aspectOf()메소드를 사용하여 aspect에 대한 참조를 얻을수 있고 aspect로 의존성을 삽입할수 있다.

6.2.1.1. 예제

보안 관리자에 의존적인 보안 aspect를 고려해볼때 이 aspect는 Account클래스내 balance인스턴스 변수의 값의 모든 변경에 적용한다. (우리는 Spring AOP를 사용해서는 같은 방법으로 이것을 할수 없다.)

aspect를 위한 AspectJ코드(Spring/AspectJ 샘플중에 하나인)가 아래에서 보인다.
SecurityManager인터페이스의 의존성은 자바빈 프라퍼티내에서 표현된다.

```
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();
   }
}
```

우리는 보통의 클래스처럼 같은 방법으로 이 aspect를 설정한다. 우리가 프라퍼티 참조를 셋팅하는 방법은 동일하다는 것을 알라. 우리는 aspectOf() 정적 메소드를 사용해서 생성된 aspect를 원한다는 것을 명시하기 위해 factory-method속성을 사용해야만 한다. 사실 이것은 생성보다는 위치 선정(locating)이다. 하지만 Spring컨테이너는 관리하지 않는다.

```
<bean id="securityAspect"
   class="org.springframework.samples.aspectj.bank.BalanceChangeSecurityAspect"
   factory-method="aspectOf"
>
   <property name="securityManager">
      <ref local="securityManager"/>
   </property>
</bean>
```

우리는 이 aspect를 목표로 하기 위해 Spring설정내 어떤것도 할 필요가 없다. 이것은 적용할 곳에서 제어할 AspectJ코드내 포인트컷(pointcut)정보를 포함한다. 게다가 이것은 Spring IoC컨테이너에 의해 관리되지 않는 객체에도 적용가능하다.

6.2.1.2. 정렬 이슈

완성되기 위해

6.2.2. 싱글톤 형식이 아닌 aspect

** Complete material on pertarget etc.

6.2.3. Gotchas

완성되기 위해

- 싱글톤 이슈

6.3. 목표 Spring advice를 위한 AspectJ 포인트컷(pointcut) 사용하기

Spring의 차후 발표될 릴리즈에서 우리는 Spring XML이나 다른 빈 정의 파일내에서 목표 Spring advice를 위해 사용되기 위한 AspectJ 포인트컷 표현을 위한 기능을 제공할 계획중이다. 이것은 Spring의 프록시 기반의 AOP프레임워크에 적용되기 위한 AspectJ 포인트컷 모델의 몇가지 힘을 허락할것이다.이것은 순수한 자바에서도 작동하고 AspectJ컴파일러를 요구하지 않을것이다. 오직 AspectJ 포인트컷의 부분 집합이 사용가능한 메소드 수행에 관련된다.

이 기능은 Spring을 위해 포인트컷 표현 언어를 생성하는 이전의 계획을 대신한다.

6.4. AspectJ를 위한 Spring aspect

Spring의 차후 발표될 릴리즈에서 우리는 AspectJ aspect와 처럼 선언적인 트랜잭션 관리 서비스와 같은 몇몇 Spring서비스들을 패키징 할것이다. 이것은 Spring AOP 프레임워크의 의존성또는 어쩌면 Spring IoC컨테이너의 의존성없이 AspectJ사용자에 의해 사용될 그것들을 가능하게 한다.

이 기능은 아마도 Spring사용자 보다 AspectJ사용자에게 좀더 흥미로운 일일것이다.

Chapter 7. 트랜잭션 관리

7.1. Spring 트랜잭션 추상화

Spring은 트랜잭션 관리를 위한 일관된 추상화를 제공한다. 이 추상화는 Spring의 추상화들 중 가장 중요한 것 중 하나이며 다음과 같은 장점들을 가져다준다.

☒ JTA, JDBC, Hibernate, iBATIS 데이터베이스 계층과 JDO와 같은 서로 다른 트랜잭션 API를 포괄하는 일관된 프로그래밍 모델을 제공한다.

☒ 이러한 대부분의 트랜잭션 API들이 제공해주는 것보다 보다 간단하고 사용하기 쉬운 프로그래밍적인 트랜잭션 관리 API를 제공한다.

☒ Spring의 데이터 접근 추상화와 통합된다.

☒ Spring의 선언적 트랜잭션 관리를 지원한다.

전통적으로, J2EE 개발자들은 트랜잭션 관리에 있어 두 가지 선택사항들을 가지는데 글로벌 혹은 로컬 트랜잭션을 사용하는 것이다. 글로벌 트랜잭션은 JTA를 사용하여 어플리케이션 서버에 의해 관리된다. 로컬 트랜잭션은, 예를 들어 JDBC 커넥션과 연관된 트랜잭션처럼, 리소스 특성을 따른다. 이 선택은 심오한 의미를 가진다. 글로벌 트랜잭션은 다중 트랜잭션 리소스들을 가지고 동작할 수 있게 해준다. (이것은 대부분의 어플리케이션들이 하나의 트랜잭션 리소스를 사용하기 때문에 그다지 유용하지 않다.) 로컬 트랜잭션을 사용한다면, 어플리케이션 서버는 트랜잭션 관리에 관여하지 않으며, 다중 리소스에 걸쳐 정확함을 보증해주지 않는다.

글로벌 트랜잭션은 중요한 약점을 가진다. 사용하기에 번거로운(부분적으로 예외 모델의 탓인) API인 JTA를 사용해야만 한다는 것이다. 더군다나, JTA UserTransaction은 일반적으로 JNDI를 통해 얻어야만 한다. 이것은 우리가 JTA를 사용하기 위해서는 JNDI와 JTA 모두 사용해야만 한다는 것을 의미한다. 명백하게 글로벌 트랜잭션을 사용하는 것은 JTA가 일반적으로 오로지 어플리케이션 서버 환경에서만 가능하기 때문에, 어플리케이션 코드의 재사용성을 제약할 것이다.

글로벌 트랜잭션을 사용할 때 선호되는 방법은, 선언적 트랜잭션 관리 형태 (왜냐하면 이것은 프로그래밍적인 트랜잭션 관리와는 구별되기 때문이다.)인 EJBCMT(Container Managed Transaction)를 경유하는 방법이다. EJB CMT는 비록 EJB 자신이 JNDI의 사용을 필요로 함에도 불구하고, 트랜잭션과 연관된 JNDI 룩업의 필요성을 제거해준다. 이것은 또한 트랜잭션을 컨트롤하기 위한 자바 코드를 작성할 필요성 역시 대부분--전부는 아니지만-- 없애준다. 중요한 약점은 CMT는 (명백히) JTA와 어플리케이션 서버에 묶여 있다는 것이다; 그리고 이것은 우리가 비지니스 로직을 EJB 혹은 최소한 트랜잭션적인 EJB 퍼싸드의 뒤에서 구현하는 경우에만 가능하다. EJB를 둘러싼 부정적인 측면은 일반적으로 매우 크기 때문에 선언적 트랜잭션 관리에 대한 대안이 있을 때에는 그다지 매력적인 제안은 되지 못한다.

로컬 트랜잭션은 훨씬 더 사용하기 쉽지만, 중요한 단점 역시 가지고 있다. 이것은 다중 트랜잭션 리소스에 걸쳐서 사용할 수 없으며 프로그래밍 모델을 침범하는 경향이 있다. 예를 들어, JDBC 커넥션을 사용하여 트랜잭션을 관리하는 코드는 글로벌 JTA 트랜잭션 내에서는 동작하지 못한다.

Spring은 이러한 문제점들을 해결해준다. Spring은 어플리케이션 개발자들로 하여금 어떠한 환경에서라도 일관적인 프로그래밍 모델을 사용할 수 있게 해준다. 당신이 코드를 한 번 작성하면 그것은 다른 환경에서의 다른 트랜잭션 관리 전략에서도 (잘 작동함으로써) 이익을 가져다 줄 것이다. Spring은 선언적/프로그래밍적 트랜잭션 관리방법 모두를 지원한다. 선언적 트랜잭션 관리는 대부분의 사용자들에게

선호되며 대부분의 경우 추천되는 방법이다.

프로그래밍적인 트랜잭션 관리를 하는 개발자들은 어떠한 기반 트랜잭션 하부구조와도 동작할 수 있는 Spring 트랜잭션 추상화로 개발한다. 선호되는 선언적 모델 개발자들은 전형적으로 트랜잭션 관리와 관련된 코딩을 매우 적거나 혹은 아예 하지 않는다. 그리고 Spring 혹은 어떤 다른 트랜잭션 API에 의존하지 않는다.

7.2. 트랜잭션 전략

Spring 트랜잭션 추상화의 핵심은 transaction strategy에 대한 개념이다.

아래의 코드는 org.springframework.transaction.PlatformTransactionManager 인터페이스에서 캡쳐해온 것이다.

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

비록 이것이 프로그래밍적으로 사용될 수 있다고는 하지만, 근본적으로는 SPI 인터페이스이다. (역자주 : SPI 인터페이스는 하드웨어쪽 용어로 '두 개의 주변장치간에 직렬 통신으로 데이터를 교환할 수 있게 해주는 직렬 인터페이스(serial peripheral interface)'라는 의미인데, DB와 어플리케이션의 트랜잭션을 연결해주는 인터페이스라는 의미로 사용된 것으로 생각된다.) Spring의 철학처럼 이것은 인터페이스라는 데 주목하라. 따라서, 이것은 필요하다면 쉽게 mock되거나 stub될 수 있다.. 더군다나 이것은 JNDI처럼 룩업 전략에 묶이지도 않는다 : PlatformTransactionManager의 구현은 Spring IoC 컨테이너에서의 다른 객체들처럼 동일하게 정의된다. 이러한 장점은 심지어 JTA로 작업조차도 구현할 보람이 있는 추상화로 만들어 준다는 것이다 : 트랜잭션 코드는 직접 JTA를 사용하는 것보다 훨씬 더 쉽게 테스트될 수 있다.

Spring의 철학에서처럼, TransactionException unchecked 예외이다. 트랜잭션 하부구조의 실패는 대부분 늘 치명적이다. 어플리케이션 코드가 그것들로부터 복구될 수 있는 아주 드문 경우에는, 어플리케이션 개발자는 여전히 TransactionException을 캐치하고 핸들링하는 것을 선택할 수 있다.

getTransaction() 메써드는 TransactionDefinition 파라미터에 따라 TransactionStatus 객체를 반환한다. 반환된 TransactionStatus는 아마도 새롭게 생성되거나 (만약 현재의 call스택에 동일한 트랜잭션이 있다면) 존재하고 있는 트랜잭션을 의미할 것이다.

J2EE 트랜잭션 컨텍스트처럼 TransactionStatus는 실행 쓰레드와 연관되어 있다.

TransactionDefinition 인터페이스는 다음과 같이 명기하고 있다 :

☒ 트랜잭션 고립성: 이 트랜잭션의 고립성의 등급은 다른 트랜잭션들의 작업으로부터 가진다. 예를 들어, 이 트랜잭션이 다른 트랜잭션들로부터 커밋되지 않은 쓰기작업 내용을 볼 수 있는가?

☒ 트랜잭션 전달: 일반적으로 트랜잭션 영역 내에서 실행되는 모든 코드는 그 트랜잭션 내에서 실행될 것이다. 그러나, 만약 트랜잭션 컨텍스트가 이미 존재하는 상황에서 트랜잭션적인 메써드가 실행된다면 그 동작을 지정하는 몇가지 옵션들이 있는데, 예를 들어, (대부분의 경우) 현존하는 트랜잭션 내에서 단순히 실행되기 혹은 현존 트랜잭션을 중지하고 새로운 트랜잭션 생성하기 등이 그것이다. Spring은

EJB CMT로부터 익숙한 트랜잭션 전달 옵션을 제공한다.

☒ 트랜잭션 타임아웃: 이 트랜잭션이 타임아웃(자동적으로 기반 트랜잭션 하부구조에 의해 롤백되는)되기까지의 시간

☒ read-only 상태: read-only 트랜잭션은 어떠한 데이터도 수정하지 않는다. read-only 트랜잭션은 (Hibernate를 사용할 때와 같이) 몇몇 경우에서 유용한 최적화 방식이 될 수 있다.

이러한 세팅들은 기본적인 개념을 반영한다. 만약 필요하다면, 트랜잭션 고립성과 다른 핵심적인 트랜잭션 개념들에 대한 논의자료들을 참조하길 바란다. 그런 핵심 개념들을 이해하는 것은 Spring 혹은 다른 트랜잭션 관리 솔루션을 사용함에 있어서 필수적인 것이다.

TransactionStatus 인터페이스는 트랜잭션 실행과 쿼리 트랜잭션 상태를 제어하기 위한 트랜잭션 코드를 작성하기 위한 간단한 방법을 제공해준다. 모든 트랜잭션 API에 공통적인 것이기 때문에 기본적인 개념은 매우 친숙하게 느껴질 것이다.

```
public interface TransactionStatus {

    boolean isNewTransaction();

    void setRollbackOnly();

    boolean isRollbackOnly();
}
```

아무리 Spring 트랜잭션 관리를 사용한다고 해도 PlatformTransactionManager 인터페이스를 구현하는 것은 필수적이다. 훌륭한 Spring 형태에서는, 중요한 정의는 IoC를 사용하여 만들어진다.

PlatformTransactionManager의 구현은 일반적으로 작업환경이 JDBC인지, JTA인지, Hibernate인지 등에 대한 지식을 필요로 한다.

Spring jPetStore 샘플 어플리케이션의 dataAccessContext-local.xml에서 추출한 다음의 예제는 로컬 PlatformTransactionManager 구현이 정의되는 방법을 보여준다. 이것은 JDBC 환경에서 작동하는 것이다.

우리는 JDBC 데이터소스를 정의해야만 한다. 그리고나서 DataSource에 대한 참조를 넘겨줌으로써 Spring의 DataSourceTransactionManager를 사용할 것이다.

```
<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>
```

PlatformTransactionManager 정의는 다음과 같을 것이다 :

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource"><ref local="dataSource"/></property>
</bean>
```

동일한 샘플 어플리케이션에 있는 dataAccessContext-jta.xml 파일에서처럼, 만약 우리가 JTA를 사용한다면 JNDI를 경유해 얻어진 우리는 컨테이너 DataSource를 사용할 필요가 있으며, JtaTransactionManager를 구현해야 한다. JtaTransactionManager는 컨테이너의 글로벌 트랜잭션 관리를 사용할 것이기 때문에, DataSource 혹은 어떤 다른 리소스들에 대해 에 대해 알 필요가 없다.

```
<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"/>
```

우리는 Spring PetClinic 샘플 어플리케이션에서 가져온 다음의 예제에서처럼 Hibernate 로컬 트랜잭션을 쉽게 사용할 수 있다.

이 경우, 우리는 어플리케이션 코드가 Hibernate Session들을 가져오기 위해 사용할 Hibernate LocalSessionFactory를 정의할 필요가 있다.

DataSource 빈 정의는 위의 예제들과 비슷하지만 보여지지 않는다. (만약 이것이 컨테이너 DataSource라면, 이것은 Spring처럼 컨테이너보다 트랜잭션적이지 않을 것이다.)

이 경우 "transactionManager" 빈은 HibernateTransactionManager 클래스이다. DataSource에 대한 참조를 필요로 한 DataSourceTransactionManager에서와 비슷하게, HibernateTransactionManager는 세션 팩토리에 대한 참조를 필요로 한다.

```
<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>
```

Hibernate와 JTA 트랜잭션을 같이 사용하려면 우리는 JDBC 혹은 어떤 다른 리소스 전략들처럼 JtaTransactionManager를 그냥 사용하면 된다.

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

이것은 어떤 트랜잭션 리소스에 참여하는 글로벌 트랜잭션처럼, 어떤 리소스들에 대한 JTA 설정과 동일하다는 점을 명심하라.
이런 모든 경우에서, 어플리케이션 코드는 아무것도 변경될 필요가 없을 것이다. 우리는 로컬에서 글로벌 트랜잭션으로 혹은 그 반대의 경우 역시 단지 설정을 바꾸는 것만으로 트랜잭션이 관리되는 방법을 변경할 수 있다.

글로벌 트랜잭션을 사용하지 않을 때에는 하나의 특별한 코딩 규칙을 따를 필요가 있다. 운좋게도 이것은 매우 간단하다. 커넥션 사용을 끌어오고 필요에 따라 트랜잭션 관리를 적용하기 위해 적절한 PlatformTransactionManager 구현을 허용하는 특별한 방법으로 커넥션 혹은 세션 리소스를 획득할 필요가 있다.

예를 들어, JDBC를 사용할 경우, 당신은 DataSource의 getConnection() 메써드를 호출해서는 안되고, 다음의 예에서 보여지는 것처럼 Spring의 org.springframework.jdbc.datasource.DataSourceUtils 클래스를 사용해야만 한다.

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

이것은 어떤 SQLException도 Spring의 (Spring의 체크되지 않은 DataAccessExceptions 하위 클래스의 하나인) CannotGetJdbcConnectionException으로 싸여진다는 부가적인 이점을 가진다. 이러한 점은 당신이 SQLException으로부터 얻을 수 있는 것보다 더 많은 정보를 줄 것이며, 다른 데이터베이스들, 심지어 다른 영속전략들에 걸쳐 이식성을 보장해 줄 것이다.

이것은 Spring 트랜잭션 관리가 없이도 잘 동작할 것인데, 그래서 Spring을 쓰건 쓰지 않건 이 방법을 사용할 수 있다.

물론, 한 번 Spring의 JDBC 지원 혹은 Hibernate 지원을 사용한다면, 당신은 DataSourceUtils 혹은 다른 헬퍼 클래스들을 사용하고 싶어하지 않을 것이다. 왜냐하면 관련된 API를 가지고 직접 작업하는 것보다 Spring 추상화를 통해 작업하는 것이 훨씬 행복할 것이기 때문이다. 예를 들어, 만약 JDBC 사용을 간단하게 하기 위해 Spring의 JdbcTemplate 혹은 jdbc.object 패키지를 사용한다면, 정확한 커넥션 복구가 보이지 않는 곳에서 이루어질 것이고, 당신은 어떤 특별한 코드를 추가할 필요가 없을 것이다.

7.3. 프로그래밍적인 트랜잭션 관리

Spring 프로그래밍적인 트랜잭션 관리에 있어 두 가지 방법을 제시한다.

☒ TransactionTemplate의 사용

☒ 직접 PlatformTransactionManager 구현

우리는 일반적으로 전자의 접근방법을 추천한다.

두 번째 접근방법은 (비록 예외처리는 덜 성가시지만) JTA UserTransaction API의 사용과 비슷하다.

7.3.1. TransactionTemplate 사용하기

TransactionTemplate은 JdbcTemplate와 HibernateTemplate와 같은 다른 Spring templates와 동일한 접근 방식을 적용하고 있다. 이것은 콜백(callback) 접근방법을 사용하는데, 리소스 획득과 해제작업으로부터 어플리케이션 코드를 해방시켜준다.(더이상 try/catch/finally를 할 필요가 없다.) 다른 templates처럼, TransactionTemplate는 쓰레드 안전하다.

트랜잭션 컨텍스트 내에서 실행되어야 하는 어플리케이션 코드는 다음과 같을 것이다. TransactionCallback이 값을 반환하기 위해 사용되는 부분에 주목하라.

```
Object result = tt.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
```

```
        updateOperation1();
        return resultOfUpdateOperation2();
    }
});
```

만약 반환될 값이 없다면, 다음과 같이 TransactionCallbackWithoutResult를 사용하라.

```
tt.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

콜백 내의 코드는 TransactionStatus 객체의 setRollbackOnly() 메써드를 호출함으로써 트랜잭션을 롤백할 수 있다.

TransactionTemplate를 사용하려는 어플리케이션 클래스들은 반드시 PlatformTransactionManager를 통해야 한다. 항상 자바빈 프라퍼티나 생성자 인자처럼 노출된다.

mock 혹은 stub PlatformTransactionManager를 가진 그런 클래스들을 유닛 테스트 하기는 쉬운 일이다. 여기에는 JNDI 룩업 혹은 정적인 마법이 존재하지 않는다 : 이것은 단순한 인터페이스이다. 대개, 당신은 유닛 테스트를 간단하게 만들기 위해 Spring을 사용할 수 있다.

7.3.2. PlatformTransactionManager 사용하기

당신은 트랜잭션을 직접 관리하기 위해 org.springframework.transaction.PlatformTransactionManager도 역시 사용할 수 있다. 단순히 사용하고 있는 PlatformTransactionManager의 구현 클래스의 빈 참조를 당신의 빈에 넘기기만 하면 된다. 그리고나서, TransactionDefinition와 TransactionStatus 객체를 사용함으로써, 당신은 트랜잭션을 초기화하고, 롤백, 커밋할 수 있다.

```
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. 선언적 트랜잭션 관리

Spring은 또한 선언적 트랜잭션 관리를 제공한다. 이것은 Spring AOP에 의해 가능하다.
대부분의 Spring 사용자들은 선언적 트랜잭션 관리를 선택한다. 이것은 어플리케이션 코드에 대한 최소한의 충격을 주는 선택이다. 그리고, 이것은 (어플리케이션 코드에)침입하지 않는 경량 컨테이너의 이상에 일관된다.

EJB CMT를 검토하고 Spring 선언적 트랜잭션 관리와의 유사점과 차이점을 설명함으로써 시작에 도움을 줄 수 있을 것이다. 기본적인 접근방법은 비슷하다 : 개별적인 메쏘드들에 대한 트랜잭션의 행동을 지정함으로써 가능하다. 그리고 만약 필요하다면, 트랜잭션 컨텍스트 내의 setRollbackOnly() 호출을 만드는 것 역시 가능하다. 하지만 차이점은 다음과 같다:

☒ JTA에 묶여 있는 EJB CMT와 다르게, Spring의 선언적 트랜잭션 관리는 어떠한 환경에서도 동작한다. 이것은 JDBC, JDO, Hibernate 혹은 내부의 어떠한 다른 트랜잭션과도 동작할 수 있으며, 단지 설정의 변경만으로 가능하다.

☒ Spring은 EJB와 같은 특별한 클래스들 뿐만 아니라, 어떠한 POJO에 대해서도 선언적 트랜잭션 관리를 적용할 수 있게 해준다.

☒ Spring은 선언적 롤백 규칙을 제공한다 : 아래에서 논의할 EJB와 동등하지 않는 특징인데, 롤백은 단지 프로그래밍적인 방법만이 아니라 선언적으로도 제어가능해진다.

☒ Spring은 AOP를 사용해서 트랜잭션적인 행위들을 커스터마이징하기 위한 기회를 제공한다. 예를 들어, 만약 당신이 트랜잭션 롤백 상황에 임의의 행위를 삽입하고자 한다면, 할 수 있다. 또한 트랜잭션적인 통보와 함께 부가적인 통보를 추가할 수도 있다. EJB CMT에서는 당신은 setRollbackOnly() 이상으로 컨테이너의 트랜잭션 관리에 영향을 끼칠 수 있는 방법이 없다.

☒ Spring은 높은 성능의 어플리케이션 서버들처럼, 원격 호출에 걸쳐 트랜잭션 컨텍스트의 전달을 지원하지는 않는다. 만약 당신이 이러한 특성이 필요하다면, EJB를 사용하라고 권해주고 싶다. 그러나, 이러한 특성은 거의 사용되지 않는다. 일반적으로 우리는 원격 호출을 확장하기 위해 트랜잭션을 원하지는 않는다.

롤백 규칙의 개념은 중요하다: 이 규칙은 우리가 어떤 예외상황일 때 자동 롤백이 발생되어야 하는지를 지정할 수 있게 해준다. 우리는 이것을 자바 코드가 아니라 설정파일에 선언적으로 지정한다. 그래서 우리가 프로그래밍적으로 현재 트랜잭션을 롤백하기 위해 여전히 TransactionStatus 객체의 setRollbackOnly() 를 호출할 수 있을지라도, 대부분 MyApplicationException이 항상 롤백을 발생시키도록 규칙을 지정할 수 있다. 이것은 비지니스 오브젝트들은 트랜잭션 하부구조에 의지할 필요가 없다는 중요한 장점을 가진다. 예를 들어, 그 클래스들은 어떠한 Spring API, 트랜잭션 혹은 다른 무엇도 import할 필요가 없다.

EJB의 (트랜잭션 롤백에 대한) 디폴트 행위는 시스템 예외 (대개 런타임 예외)시에 EJB 컨테이너가 트랜잭션 롤백을 자동으로 수행하지만, EJB CMT는 어플리케이션 예외(java.rmi.RemoteException를 제외한 체크된 예외)시 자동으로 트랜잭션 롤백을 하지 않는다. 선언적 트랜잭션 관리를 위한 Spring의 기본적인 동작은 EJB 규칙(체크되지 않은 예외에 대해서만 자동 롤백)을 따르지만, 이것을 종종 커스터마이징하기에 유용하다.

우리의 벤치마크에 따르면, Spring 선언적 트랜잭션 관리의 성능은 EJB CMP의 것을 앞서고 있다.

Spring에서 트랜잭션적인 프록시를 세팅하는데 사용되는 일반적인 방법은 TransactionProxyFactoryBean를 통한 것이다. 우리는 트랜잭션 프록시로 감쌀 타겟 객체가 필요하다. 타겟 객체는 일반적으로 POJO 빈 정의를 따른다. TransactionProxyFactoryBean을 정의할 때, 관련된 PlatformTransactionManager에 대한 참조와 트랜잭션 속성을 넘겨주어야 한다. 트랜잭션 속성은 위에서 얘기한 트랜잭션 정의를 포함한다. 다음의 예시를 보자.

```
<!-- 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>
```

트랜잭션 프록시는 타겟의 인터페이스를 구현한다. 위 예시의 경우, petStoreTarget이라는 아이디를 가진 빈이다. (CGLIB를 사용하여 타겟 객체에 트랜잭션 프록시를 구현하는 것이 가능하다. 이것을 위해 proxyTargetClass 프라퍼티를 true로 세팅하라. 만약 타겟 객체가 어떠한 인터페이스도 구현하고 있지 않다면 자동으로 발생할 것이다. 물론, 일반적으로 우리는 클래스보다는 인터페이스를 프로그래밍할 것을 원한다.) proxyInterfaces 프라퍼티를 사용하여 단지 특정한 타겟 인터페이스만 프록시하도록 트랜잭션 프록시를 제한하는 것은 가능하다. (그리고 대개의 경우 훌륭한 생각이다.) 그리고 또한 org.springframework.aop.framework.ProxyConfig로부터 상속받은 몇가지 프라퍼티들을 통해 TransactionProxyFactoryBean의 행위를 커스터마이징하고 모든 AOP 프록시 팩토리들과 공유하는 것 역시 가능하다.

여기에서의 transactionAttributes는 org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource 클래스에 정의된 프라퍼티 포맷을 사용하여 세팅된다. 와일드카드를 포함한 메써드명의 매핑은 매우 직관적이다. insert* 매핑의 값이 롤백 규칙을 포함하고 있다는 것을 눈여겨 보아라. 여기에서의 -MyCheckedException는 만약 메써드가 MyCheckedException 혹은 그 하위 예외 클래스를 던진다면, 트랜잭션은 자동으로 롤백될 것이다. 다중 롤백 규칙 역시 콤마 구별자로 여기에 지정될 수 있다. - 접두사는 롤백을 수행하고, + 접두사는 커밋을 지정한다. (+ 옵션을 주는 것은 체크되지 않은 예외시에조차 커밋을 허용한다. 당신이 무엇을 하고 있는지 확실히 알아야만 한다!)

TransactionProxyFactoryBean은 추가적으로 끼여드는 행위를 위해, "preInterceptors", "postInterceptors" 프라퍼티를 사용하여 "pre" 혹은 "post" advice를 세팅하게 해준다. pre, post advice는 얼마든지 세팅될 수 있고, 그 타입은 Advisor (이 경우 그것은 pointcut을 포함할 수 있다.) 일 것이다. MethodInterceptor 혹은 어떤 advice 타입도 현재의 Spring 설정 (ThrowsAdvice, AfterReturningtAdvice 혹은 BeforeAdvice 등이 기본적으로 지원된다.)에 의해 지원된다. 이러한 advice들은 반드시 공유-인스턴스 모델을 지원해야 한다. 만약 당신이 상태유지 믹스인 (stateful mixins)처럼 발전된 AOP 특성들을 가진 트랜잭션적인 프록싱을 필요로 한다면, 일반적으로 편리한 프록시 생성자인 TransactionProxyFactoryBean보다는 포괄적인 org.springframework.aop.framework.ProxyFactoryBean를 사용하는 것이 최선일 것이다.

오토프록싱을 세팅하는 것 역시 가능한데, AOP 프레임워크를 세팅함으로써 클래스들은 자동으로 개별 프록시 정의 없이도 프록시될 수 있다.

더 많은 정보와 예제들은 AOP 챕터를 참조하길 바란다.

노트: TransactionProxyFactoryBean 정의를 위에서와 같은 형태로 사용하는 것은 많은 동일한 트랜잭션 프록시들이 생성될 필요가 있을 때 과도하게 장황해보일 수 있다. Section 5.7, “간결한 프록시 정의” 에서술된 바처럼, 당신은 트랜잭션 프록시 정의의 장황함을 상당부분 제거하기 위해, 내부 빈 정의의 장점을 가진 부모/자식 빈 정의의 이점을 가지기를 바랄 것이다.

Spring의 선언적 트랜잭션 관리를 효과적으로 사용하기 위해서, 당신은 AOP 전문가가 될 필요가 없다--혹은 최소한, AOP의 대부분에 대해 알 필요도 없다. 하지만, 당신이 정말로 Spring AOP의 "끝내주는 사용자"가 되고자 한다면, 당신은 강력한 AOP의 능력을 가진 선언적 트랜잭션 관리를 조합하는 것이 간단한 일임을 알 수 있을 것이다.

7.4.1. BeanNameAutoProxyCreator, 또 다른 선언적 접근방법

TransactionProxyFactoryBean은 매우 유용하고, 트랜잭션 프록시로 객체들을 감쌀 때 전체적인 제어를 할 수 있게 해준다. 부모/자식 빈 정의와 타겟을 가지고 있는 내부 빈들을 함께 사용하는 것은 일반적으로 트랜잭션적인 감싸기를 위한 최고의 방법이다. 당신이 완전히 동일한 형태로 많은 수의 빈들을 감쌀 필요가 있을 경우(예를 들어, 반복 사용 어구, '모든 메쏘드들을 트랜잭션하게 만들어라'), BeanNameAutoProxyCreator라고 불리는 BeanFactoryPostProcessor를 사용하는 것은 덜 장황하고 간단한 대안적인 접근방법이 될 것이다.

재감싸기를 위해서, ApplicationContext이 그것의 초기화 정보를 읽을 때, 그 안에 있는 BeanPostProcessor 인터페이스를 구현하는 모든 빈들을 초기화하고, 그 빈들에게 ApplicationContext 내의 모든 다른 빈들을 이후에 처리할(post-process) 기회를 제공한다. 때문에 이러한 메카니즘을 사용하면, 적절하게 설정된 BeanNameAutoProxyCreator는 ApplicationContext 내의 모든 다른 빈들을 (이름으로 그것들을 인식하여) 이후에 처리하기 위해 사용될 수 있다. 생성된 실제 트랜잭션 프록시는 TransactionProxyFactoryBean를 사용하여 생성되었다는 점에서 반드시 동일한데, 여기에 대해서는 이후에도 논의되지는 않을 것이다.

아래의 설정 예시를 보도록 하자.

```
<!-- 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>
```

ApplicationContext 내에 이미 TransactionManager 인스턴스를 가지고 있다고 가정한 상태로, 첫번째

우리가 해야 할 일은 사용할 TransactionInterceptor 인스턴스를 생성하는 일이다. TransactionInterceptor는 프라퍼티로 넘겨진 TransactionAttributeSource 구현 객체에 기반하여 어떤 메써드가 인터셉트되어야 하는지를 결정한다. 위의 경우, 우리는 모든 메써드를 일치시키는 매우 간단한 경우를 다루고자 한다. 이것은 물론 가장 효율적인 접근방법은 아니지만, 매우 빨리 세팅될 수 있다. 왜냐하면 우리는 특별히 이미 정의된, 모든 메써드를 단순히 일치시켜주는, MatchAlwaysTransactionAttributeSource를 사용할 수 있기 때문이다. 만약 우리가 좀 더 기술하려면, MethodMapTransactionAttributeSource, NameMatchTransactionAttributeSource 혹은 AttributesTransactionAttributeSource와 같이 다른 것을 사용할 수 있다.

이제 트랜잭션 인터셉터를 가지게 되었다면, 우리가 정의한 BeanNameAutoProxyCreator 인스턴스를 동일한 형태로 포장되길 바라는 ApplicationContext 안의 6개의 빈들의 이름과 함께, 인터셉터에 단순히 넘겨주기만 하면 된다. 당신이 알 수 있듯이, 최종결과는 6개의 빈들을 TransactionProxyFactoryBean으로 감싸는 것보다 매우 간소하다. 7번째의 빈을 감싸는 것 역시 설정에서 한 줄을 추가하기만 하면 된다.

당신은 우리가 다중 인터셉터를 적용할 수 있다는 점을 알아차렸을지 모른다. 이런 경우, 우리는 역시 우리가 이전에 (bean id=hibInterceptor)라고 정의했던 HibernateInterceptor 또한 적용할 수 있으며, 이것은 우리를 위해 Hibernate Sessions를 관리해줄 것이다.

TransactionProxyFactoryBean과 BeanNameAutoProxyCreator의 사용 사이를 왔다갔다 할 때, 빈 명명에 관해 당신이 기억해두어야 할 것이 하나 있다. 전자를 위해서는, 만약 타겟빈이 내부 빈으로 정의되지 않았다면, 당신은 일반적으로 당신이 감싸고자 하는 타겟빈을 myServiceTarget와 비슷한 형태의 id로 넘겨주고, 프락시 객체는 myService라는 id로 넘겨줄 것이다. 그리고 감싸여진 객체의 사용하는 모든 사람들은 단지 프락시, 다시 말해, myService만을 참조할 것이다. (이것은 단지 명명규칙의 예시이다. 중요한 점은 목표객체는 프락시와 다른 이름을 가진다는 것이고, 둘 다 ApplicationContext으로부터 사용가능하다는 점이다.) 그러나, BeanNameAutoProxyCreator를 사용할 경우, 당신은 타겟 객체에다 myService 비슷한 이름을 부여한다. 그러면, BeanNameAutoProxyCreator가 타겟 객체를 전처리하고 프락시를 생성할 때, 어플리케이션 컨텍스트의 원래의 빈 이름 아래에 그 프락시를 위치시킨다. 그 지점에서 단지 프락시(감싸여진 객체)만이 ApplicationContext로부터 사용가능하다. 내부 빈으로 정의된 타겟을 가진 TransactionProxyFactoryBean를 사용할 경우, 명명 이슈는 신경쓸 바가 아니다. 왜냐하면, 내부 빈에는 일반적으로 이름이 주어지지 않기 때문이다.

7.5. 프로그래밍적/선언적 트랜잭션 관리 중 선택하기

프로그래밍적인 트랜잭션 관리는 대개 당신이 매우 적은 수의 트랜잭션적인 동작들을 다룰 때에만 좋은 생각이다. 예를 들어, 만약 당신이 어떤 업데이트 동작들에만 트랜잭션이 필요한 웹 어플리케이션을 개발한다면, 당신은 Spring 혹은 다른 기술들을 사용해서 트랜잭션적인 프락시를 세업하는 것을 바라지 않을지도 모른다. 이 경우, TransactionTemplate을 사용하는 것은 좋은 접근방법이 될 것이다.

반면, 만약 당신의 어플리케이션이 매우 많은 트랜잭션적인 동작들을 가진다면, 선언적 트랜잭션 관리는 대개 매우 가치있는 판단일 것이다. 이것은 비지니스 로직의 외부에서 트랜잭션 관리를 유지해주며, Spring 에서 그 설정을 하는 것은 어려운 것이 아니기 때문이다. EJB CMT보다 Spring을 사용한다면, 선언적 트랜잭션 관리 설정의 비용은 매우 감소된다.

7.6. 트랜잭션 관리를 위한 어플리케이션 서버가 필요한가?

Spring의 트랜잭션 관리 능력들--그리고 특히 그것의 선언적 트랜잭션 관리--은 J2EE 어플리케이션이 어플리케이션 서버를 필요로할 때 트랜잭션적인 판단을 현저하게 변화시킨다.

특히, 당신이 단지 EJB를 통해 선언적 트랜잭션을 얻고자 한다면 어플리케이션 서버는 필요가 없다. 사실, 강력한 JTA의 능력을 가진 어플리케이션 서버를 가지고 있다고 할지라도, 당신은 EJB CMT보다 더욱 강력하고 더더욱 생산적인 프로그래밍 모델을 제공해주는 Spring 선언적 트랜잭션을 선택하는 것이 좋을지도 모른다.

오로지 당신이 다중 트랜잭션 리소스를 지원할 필요가 있을 때에만, 어플리케이션 서버의 JTA 능력이 필요하다. 많은 어플리케이션들은 이러한 요구에 직면하지 않는다. 예를 들어, 많은 고성능 어플리케이션들은 Oracle 9i RAC와 같이 크게 확장가능한 하나의 데이터베이스를 사용한다.

물론 당신은 JMS와 JCA와 같은 어플리케이션 서버의 또 다른 능력들을 필요로 할 지 모른다. 그러나, 당신이 만약 JTA만을 필요로 하는 것이라면, 당신은 JOTM과 같은 JTA가 첨부된 오픈 소스의 사용을 고려할 수 있다. (Spring은 JOTM과 외부에서 통합된다.) 그렇지만, 2004년 초반, 고성능 어플리케이션 서버들은 XA 트랜잭션들에 대한 보다 튼튼한 지원을 제공한다.

가장 중요한 점은 Spring을 사용하게 되면, 언제 당신의 어플리케이션을 최종적인 어플리케이션 서버로 확장시킬 것인가에 대한 시점을 선택할 수 있다. EJB CMT 혹은 JTA를 사용하는 것에 대한 대안이라고는 오로지 JDBC 커넥션과 같은 로컬 트랜잭션들을 사용해서 코딩했다가 그 코드가 글로벌 컨테이너 관리 트랜잭션에서 작동할 필요가 생겼을 때 엄청난 개정작업에 직면해야만 했던 시절은 갔다. Spring을 사용한다면 바꾸기 위해 필요한 것은 오로지 설정뿐이다. 당신의 코드는 바꿀 필요가 없다.

7.7. 공통적인 문제

개발자들은 그들의 요구사항들에 맞는 적절한 PlatformTransactionManager 구현을 사용하는데 주의를 기울여야 한다.

Spring 트랜잭션 추상화가 JTA 글로벌 트랜잭션과 동작하는 방식을 이해하는 것은 중요한 일이다. 적절하게 사용되었을 때, 여기엔 아무런 문제가 없다. Spring은 단지 간소화하고 이식가능한 추상화만을 제공한다.

만약 당신이 글로벌 트랜잭션을 사용한다면, 당신은 모든 트랜잭션적인 동작들에 대해 Spring의 org.springframework.transaction.jta.JtaTransactionManager을 반드시 사용해야만 한다. 만약 그렇지 않으면 Spring은 컨테이너 데이터소스들과 같은 리소스들에서 로컬 트랜잭션을 수행하고자 할 것이다. 그런 로컬트랜잭션들은 말이 안되며, 좋은 어플리케이션 서버라면 그것들을 에러로 간주할 것이다.

Chapter 8. 소스 레벨 메타데이타 지원

8.1. 소스-레벨 메타데이타

소스-레벨 메타데이타 프로그램 요소(대개 클래스 그리고/또는 메소드)에 속성(attributes) 이나 annotations을 추가한 것이다.

예를 들어, 우리는 다음처럼 클래스에 메타데이타를 추가할수 있다.

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.DefaultTransactionAttribute()
 */
public class PetStoreImpl implements PetStoreFacade, OrderService {
```

우리는 다음처럼 메소드에 데타데이타를 추가할수 있다.

```
/**
 * 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 {
    ....
}
```

이러한 예제들 모두 Jakarta Commons Attributes 문법을 사용한다.

소스-레벨 메타데이타는 트랜잭션, 풀링 그리고 다른 행위를 제어하기 위한 소스레벨 속성을 사용하는 Microsoft's .NET platform에 의해 릴리즈되었고 자바진영에서는 XDoclet에 의해 가담되어 소개되었다.

이 접근법내 값은 J2EE커뮤니티내에서 인식되고 있다. 예를 들어 EJB에 의해 사용되는 전통적인 XML배치 서술자보다는 다소 덜 장황하다. 프로그램 소스코드로부터 어떤것을 구체화하는것이 바람직할 동안 몇몇 중요한 기업용 셋팅들-명백한 트랜잭션 특성-어쩌면 프로그램 소스내 포함된다. EJB스펙의 가정에 대해 반대로 이것은 좀처럼 메소드의 트랜잭션적인 특성을 변경할려고 시도하지 않는다.(비록 트랜적션 타임아웃(timeout)과 같은 파라미터가 변경될지라도)

비록 메타데이타 속성들은 대개 요구하는 서비스 애플리케이션 클래스를 서술하는 프레임워크 구조에 의해 주로 사용된다. 이것은 수행시 쿼리되는 메타데이타 속성이 가능하다. 이것은 EJB가공물같은 코드를 생성하는 방법처럼 메타데이타를 보는 XDoclet같은 솔루션으로부터 키가 되는 차이이다.

여기엔 다음을 포함해서 많은 수의 솔루션이 있다.

☒ JSR-175: 자바 1.5에서 사용가능한 표준적인 자바 메타데이타 구현물, 하지만 우리는 자바 1.4그리고 1.3을 위해서도 이 솔류션이 필요하다.

☒ XDoclet: 잘 적립된 솔류션, 주로 코드 생성을 시도하려고 한다.

☒ 자바 1.3 그리고 1.4를 위한 다양한 오픈소스 속성 구현물들, Commons Attribute의 것은 대부분

약속된것처럼 보인다. 모든것들은 특별한 pre-(선) 또는 post-(후) 컴파일 단계를 요구한다.

8.2. Spring의 메타데이타 지원

중요한 개념을 넘어선 추상화 조항을 유지하여 Spring은 org.springframework.metadata.Attributes 인터페이스의 형태로 메타데이타 구현물을 위한 외관(facade)을 제공한다.

외관은 다양한 이유를 위해 값을 추가한다.

☒ 현재 표준적인 메타데이타 솔루션이 없다. 자바 1.5는 하나를 제공할것이지만 이것은 Spring 1.0의 것처럼 여전히 베타상태이다. 게다가 최소한 2년 동안은 1.3과 1.4애플리케이션내 메타데이타 지원이 필요할것이다. Spring은 지금 작동중인 솔류션을 제공하는것이 목적이다. 1.5를 기다리는것은 중요한 영역에서 선택사항이 아니다.

☒ Commons Attributes(Spring 1.0에 의해 사용되는)과 같은 최근의 메타데이타 API는 테스트하기가 힘들다. Spring은 모조품을 위해 좀더 쉬운 간단한 메타데이타 인터페이스를 제공한다.

☒ 자바 1.5가 언어레벨에서 메타데이타 지원을 제공할때 추상화와 같이 제공하는 값이 될것이다.

- ☒ JSR-175 메타데이타는 정적이다. 이것은 컴파일 시각에 클래스와 관련된다. 그리고 배치된 환경에서 변경이 될수 없다. 예를 들어, XML파일내에서 구조적인 메타데이타가 필요하고 배치내에서 어떤 속성값을 오버라이드하는 능력을 제공한다.
- ☒ JSR-175 메타데이타는 자바 reflection API를 통해 반환된다. 이것은 테스트 시간동안 모방이 불가능하다. Spring은 이것을 허용하기 위한 간단한 인터페이스를 제공한다.

Spring Attributes 인터페이스는 이것처럼 보인다.

```
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);
}
```

이것은 가장 낮은 공통적인 공통점(denominator) 인터페이스이다. JSR-175는 메소드 인자의 속성처럼 이것보다 좀더 많은 기능을 제공한다. Spring 1.0에서처럼 Spring은 자바 1.3이상에서 EJB나 .NET의 선언적인 기업용 서비스를 효과적으로 제공하기 위해 요구되는 메타데이타의 부분세트(subset)를 제공하는것이 목적이다. Spring 1.2에서 유사한 JSR-175 annotation은 Commons Attribute의 직접적인 대안처럼 JDK1.5에서 지원된다.

이 인터페이스는 .NET처럼 Object 속성을 제공한다. 이것은 오직 String 속성만 제공하는 Nanning Aspects 와 JBoss 4의 그것처럼 속성시스템으로 부터 이것과 구별된다. Object속성을 지원하는데는 명백한

장점을 가진다. 이것은 속성이 클래스 구조에 관계되는 것을 가능하게 하고 설정 파라미터로 속성이 현명하게 반응하는것을 가능하게 한다.

대부분의 속성 제공자(provider)에서, 속성 클래스는 생성자의 인자나 자바빈 프라퍼티를 통해 설정될것이다. Commons Attributes지원또한 설정된다.

모든 Spring 추상 API처럼 Attributes는 인터페이스이다. 이것은 단위 테스트를 위한 속성 구현물을 모방하는것을 쉽게 한다.

8.3. Jakarta Commons Attributes과 통합

현재 Spring은 비록 이것이 다른 메타데이타 제공자를 위한 org.springframework.metadata.Attributes의 구현물을 제공하는것이 쉽더라도 특별히 Jakarta Commons Attributes만을 지원한다.

Commons Attributes 2.1 (http://jakarta.apache.org/commons/attributes/) 은 필요한 능력을 가진 속성 솔루션이다. 속성 정의내 좀더 나은 문서화를 제공하는 이것은 생성자의 인자와 자바빈 프라퍼티를 통해 속성 설정을 지원한다.(자바빈 프라퍼티를 위한 지원은 Spring팀에 의해 요청되어 추가되었다.)

우리는 Commons Attributes 속성 정의의 두가지 예제를 이미 보였다. 대개 우리는 표현할 필요가 있을것이다.

☒ 속성 클래스의 이름. 이것은 위에서 보여준 것처럼 FQN이 될수 있다. 만약 관련 속성 클래스가 이미 import되었다면 FQN은 요구되지 않는다. 이것은 속성 컴파일러-설정내에서 "속성 패키지"를 명시하는것이 가능하다.

☒ 생성자의 인자나 자바빈 프라퍼티를 통해 필요한 파라미터로 나타내기

bean 프라퍼티는 다음처럼 보일것이다.

```
/**
 * @@MyAttribute(myBooleanJavaBeanProperty=true)
 */
```

(Spring IoC처럼) 생성자의 인자와 자바빈 프라퍼티를 조합하는것은 가능하다.

자바 1.5 속성과는 다르기 때문에 Commons Attributes는 자바언어와 통합되지 않는다. 이것은 빌드 처리의 일부처럼 특별한 속성 컴파일 단계를 수행할 필요가 있다.

빌드 처리의 일부처럼 Commons Attributes를 수행하기 위해 당신은 다음처럼 할 필요가 있다.

1. 필요한 라이브러리 jar파일을 $ANT_HOME/lib 로 복사하라. 4개의 jar파일이 요구되고 모두 Spring과 함께 배포된다.

☒ Commons Attributes 컴파일러 jar와 API jar

☒ XDoclet으로 부터의 xjavadoc.jar

☒ Jakarta Commons으로 부터의 commons-collections.jar

2. 다음처럼 Commons Attributes ant작업을 당신의 프로젝트 빌드 스크립트에 추가하라.

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

3. 소스내 속성을 "컴파일(compile)" 하기 위한 Commons Attributes 속성-컴파일 작업을 사용할 속성 컴파일 작업을 정의하라. 이 처리는 destdir속성에 의해 정의된 위치에 추가적인 소스의 생성한다. 우리는 임시 디렉토리의 사용한다.

```
<target name="compileAttributes" >

    <attribute-compiler
        destdir="${commons.attributes.tempdir}"
    >
        <fileset dir="${src.dir}" includes="**/*.java"/>
    </attribute-compiler>

</target>
```

소스에 Javac를 시행하는 컴파일 대상은 속성 컴파일 작업에 의존한다. 그리고 우리의 대상 임시 디렉토리에 결과를 만드는 생성된 소스를 컴파일해야만 한다. 만약 당신의 속성 정의에 문법적인 에러가 있다면 속성 컴파일러에 의해 잡힐것이다. 만약 속성 정의가 문법적으로 그럴듯하지만 유효하지 않은 타입이나 클래스명을 명시한다면 생성된 속성 클래스의 컴파일이 실패할것이다. 이 경우 당신은 문제를 야기하는 생성된 클래스를 찾을수 있다.
Commons Attributes 또한 Maven지원을 제공한다. 더 많은 정보를 위해서는 Commons Attributes 문서를 참조하라.

속성 컴파일 처리가 완벽해 보이는 동안 사실 이것은 한번만(one-off cost)에 이루어진다. 셋업할때 속성 컴파일은 증가한다. 그래서 이것은 언제나 눈에 띄게 빌드처리가 늦지는 않다. 컴파일 처리가 셋업된다면 당신은 이 장에서 언급되는것처럼 속성의 사용이 당신에게 다른 영역에서 많은 시간을 절약하게 한다는것을 알게 될것이다.

만약 당신이 속성 인덱스 지원(속성-대상이 된 웹 컨트롤러를 위해 Spring에 의해 요구되는)을 요구한다면 당신은 컴파일된 클래스의 jar파일에서 수행되어야만 하는 추가적인 단계가 필요할것이다. 이것은 선택적인 단계로 Commons Attributes는 수행시 효과적으로 찾기 위해 소스에 정의된 모든 속성의 인덱스를 생성할것이다. 이 단계는 다음처럼 보일것이다.

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

    <classpath refid="master-classpath"/>

</attribute-indexer>
```

빌드 처리의 예제인 Spring jPetStore예제 애플리케이션의 /attributes 디렉토리를 보라. 당신은 당신의 프로젝트를 위해 이것을 포함하거나 변경하는 빌드 스크립트를 가질수 있다.

만약 당신의 단위 테스트가 속성에 의존한다면 Commons Attributes보다는 Spring Attributes 추상화에 의존성을 표시하라. 이것은 좀더 이식가능하다. 예를 들어 당신의 테스트가 나중에 자바 1.5로 교체된다고 하더라고 여전히 작동할것이다. 이것은 테스트를 좀더 쉽게 만든다. Spring이 쉽게 모방할수있는 메타데이타 인터페이스를 제공하는 반면에 Commons Attributes는 정적 API이다.

8.4. 메타데이타와 Spring AOP 자동 프록시

메타데이타 속성의 가장 중요한 사용은 Spring AOP와 결합하는것이다. 이것은 선언적인 서비스가 메타데이타 속성을 선언하는 애플리케이션 객체에 자동적으로 제공되는 .NET같은 프로그래밍 모델을 제공한다. 메타데이타 속성은 선언적인 트랜잭션 관리나 사용자 지정 사항처럼 프레임워크에 의해 특별히 지원될수 있다.

여기엔 AOP와 메타데이타 속성간의 넓은 시너지 효과를 가져다 준다.

8.4.1. 기초

이것은 Spring AOP 자동프록시 기능위에서 빌드된다. 설정은 다음과 같을수 있다.

```
<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"
/>
```

여기의 기본적인 개념은 AOP장의 자동프록시에서 언급된것과 유사하다.

가장 중요한 bean정의는 autoproxy 와 transactionAdvisor 라는 이름으로 명명된다. 실질적인 bean이름은 중요하지 않다. 클래스가 중요하다.

org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator 클래스의 자동프록시 bean정의는 Advisor구현물에 대응되는 현재 factory내 모든 bean인스턴스에 자동적으로 advise("autoproxy")할것이다. 이 클래스는 속성에 대해 아무것도 모르지만 Advisor의 pointcut대응에 의존한다. pointcut는 속성에 대해서 안다.

게다가 우리는 속성에 기반한 선언적인 트랜잭션 관리를 제공할 AOP advisor이 필요하다.

임의로 사용자정의 Advisor구현물을 추가하는것은 가능하다. 그리고 그것들은 자동적으로 평가되고 적용될것이다. (당신은 필요하다면 같은 자동프록시 설정내 속성외에도 기준(criteria)에 대응되는 pointcut의 Advisor을 사용할수 있다.)

마지막으로 attributes bean은 Commons Attributes속성 구현물이다. 다른 소스로부터 소스 속성을 위한 org.springframework.metadata.Attributes의 구현물을 대체한다.

8.4.2. 선언적인 트랜잭션 관리

소스레벨 속성의 가장 공통적인 사용은 선언적인 트랜잭션 관리를 하는것이다. 위의 bean정의는 이를 대신한다. 당신은 선언적인 트랜잭션을 요구하는 많은 수의 애플리케이션 객체를 정의할수 있다. 트랜잭션 속성을 가진 클래스나 메소드는 트랜잭션 advice가 주어질것이다. 당신은 요구된 트랜잭션 속성을 정의하는것을 제외하고 아무것도 할 필요가 없다.

.NET 과는 다르게, 당신은 클래스나 메소드 레벨에서 트랜잭션 속성을 명시할수 있다. 모든 메소드에 의해 "상속"받는다면 클래스-레벨 속성, 클래스-레벨 속성을 전체적으로 오버라이드한다면 메소드 레벨 속성이다.

8.4.3. 풀링(Pooling)

다시 .NET처럼, 당신은 클래스-레벨 속성을 통해 풀링을 가능하게 할수 있다. Spring은 POJO에 이 행위를 적용할수 있다. 당신은 다음처럼 풀링되는 비지니스 객체네에서 풀링 속성을 정의할 필요가 있다.

```
 /**
 * @@org.springframework.aop.framework.autoproxy.target.PoolingAttribute (10)
 *
 * @author Rod Johnson
 */
public class MyClass {
```

당신은 대개 자동프록시 설정이 필요할 것이다. 당신은 다음처럼 풀링 TargetSourceCreator을 명시할 필요가 있다. 풀링은 대상의 생성에 영향을 끼치므로 우리는 정규(regular) advice를 사용할수 없다. 클래스에 적용가능한 advisor가 없고 클래스가 풀링 속성을 가진다면 풀링은 적용할것이다.

```
<bean id="poolingTargetSourceCreator"
    class="org.springframework.aop.framework.autoproxy.metadata.AttributesPoolingTargetSourceCreator"
    autowire="constructor" >
</bean>
```

관련 자동프록시 bean정의는 풀링 대상 소스 생성자를 포함하는 "사용자 정의 대상 소스 생성자"의 목록을 명시할 필요가 있다. 우리는 다음의 프라퍼티를 포함하기 위해 위의 예제를 변경할수 있다.

```
<bean id="autoproxy"
    class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
>
    <property name="customTargetSourceCreators">
        <list>
            <ref local="poolingTargetSourceCreator" />
        </list>
    </property>
</bean>
```

대개 Spring에서 메타데이타를 사용하는것처럼 이것은 한번만에 이루어진다. 셋업이 이루어진다면 추가적인 비지니스 객체를 위한 풀링을 사용하는것은 매우 쉽다.
풀링의 필요성이 드물다는것은 논쟁의 여지가 있다. 그래서 많은 수의 비지니스 객체를 위해 풀링을 적용하는것은 좀처럼 필요하지 않다. 이 기능은 종종 사용되지 않는다.

좀더 상세항 사항을 위해서는 org.springframework.aop.framework.autoproxy 패키지를 위한 JavaDoc를 보라. 이것은 최소한의 사용자정의 코딩을 가진 Commons Pool보다 다른 풀링 구현물을 사용하는것이 가능하다.

8.4.4. 사용자정의 메타데이타

우리는 자동프록시 구조를 참조하는 유연성때문에 .NET메타데이타 속성의 기능을 능가할수 있다.

우리는 선언적인 행위의 종류를 제공하기 위해 사용자정의 속성을 정의할수 있다. 이것을 위해서 당신은 다음처럼 해야할 필요가 있다.

☒ 사용자정의 속성 클래스 정의하기

☒ 사용자정의 속성의 존재에서 수행되는 pointcut를 가진 Spring AOP Advisor정의하기

☒ 일반적인 자동프록시 구조를 대체하는 애플리케이션 컨텍스트를 위한 bean정의처럼 Advisor를 추가하기

☒ POJO에 속성 추가하기.

당신이 사용자정의 선언적인 보안이나 캐싱과 같은 것을 하길 원하는 여러가지 잠재적인 영역이 있다. 이것은 몇몇 프로젝트에서 효과적으로 설정을 줄일수 있는 강력한 기법이다. 어쨌든 AOP에 의존한다는것을 기억하라. 좀더 많은 Advisor은 좀더 복잡한 수행 설정이 될것이다.
(만약 당신이 어느 객체에 적용되는 advice를 보길 원한다면 참조를 org.springframework.aop.framework.Advised로 형변환하라. 이것은 Advisor을 조사하는것을 가능하게 한다.)

8.5. MVC 웹티어 설정을 최소화하기 위한 속성 사용하기

1.0의 Spring 메타데이타의 다른 중요한 사용은 Spring MVC웹 설정을 단순화하기 위한 선택사항을 제공하는것이다.

Spring MVC는 들어온 요청을 컨트롤러(또는 다른 핸들러) 인스턴스로 맵핑하는 유연한 핸들어 맵핑을 제공한다. 대개 핸들러 맵핑은 관련 Spring DispatcherServlet을 위한 xxxx-servlet.xml파일내 설정된다.

DispatcherServlet 설정파일내 맵핑을 유지하는것은 좋은것이다. 이것은 최대한의 유연성을 제공한다. 특히

☒ XML bean정의를 통해 Spring IoC에 의해 명시적으로 관리되는 컨트롤러 인스턴스

☒ 맵핑은 컨트롤러를 위한 형식이다. 그래서 같은 컨트롤러 인스턴스는 같은 DispatcherServlet 컨텍스트내에서 다중 맵핑이 주어질수 있거나 다른 설정에서 재사용될수 있다.

☒ Spring MVC는 대부분의 다른 프레임워크에서 사용가능한 요청 URL-대-컨트롤러(URL-to-controller) 맵핑보다 어느 기준(criteria)에 기반하는 맵핑을 지원하는것이 가능하다.

어쨌든 이것은 각각의 컨트롤러를 위해 우리는 대개 핸들러 맵핑(핸들러 맵핑 XML bean정의내에서)과 컨트롤러 자체를 위한 XML맵핑이 필요하다는것을 의미한다.

Spring은 좀더 간단한 시나리오에서 매력적인 선택사항인 소스-레벨 속성에 기반하는 좀더 간단한 접근법을 제공한다.
이 장에서 언급된 접근법은 비교적 간단한 MVC시나리오에 가장 적합하다. 이것은 다른 맵핑를 가진 같은 컨트롤러를 사용하기 위한 능력과 요청 URL보다 어떤것에 맵핑에 기초로 두는 능력과 같은 Spring MVC의 몇몇 강력함을 희생한다.

이 접근법에서 컨트롤러는 맵핑될 하나의 URL을 명시하는 하나 이상의 클래스-레벨 메타데이타 속성과 함께 표시된다.

다음의 예제는 접근법을 보여준다. 이 경우, 우리는 Cruncher타입의 비지니스 객체에 의존하는 컨트롤러를 가진다. 대개 이 의존성은 의존성 삽입(Dependency Injection)에 의해 해석될것이다. Cruncher는 관련 DispatcherServlet XML 파일이나 부모 컨텍스트내 bean정의를 통해 사용가능해야만 한다.

우리는 이것을 맵핑하는 URL을 명시하는 컨트롤러 클래스로 속성을 첨부한다. 우리는 자바빈 프라퍼티나 생성자의 인자를 통해 의존성을 표시할수 있다. 이 의존성은 autowiring에 의해 해석될수 있어야만 한다. 컨텍스트내 사용가능한 Cruncher타입의 비지니스 객체가 되어야만 한다.

```
/**
 * 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");
    }

}
```

작동하기 위한 자동-맵핑을 위해, 우리는 속성 핸들러 맵핑을 명시하는 관련 xxxx-servlet.xml파일에 다음을 추가할 필요가 있다. 이 특별한 핸들러 맵핑은 위의 속성을 가진 많은수의 컨트롤러를 다룰수 있다. bean id("commonsAttributesHandlerMapping")는 중요하지 않다. 타입이 중요하다.

```
<bean id="commonsAttributesHandlerMapping"
    class="org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping"
/>
```

우리는 위 예제처럼 Attributes bean정의가 현재 필요하지 않다. 이 클래스가 Commons Attributes API와 직접적으로 자동하기 때문에 Spring 메타데이타 추상화를 통하지 않는다.

우리는 각각의 컨트롤러를 위한 XML설정이 필요하지 않다. 컨트롤러는 명시된 URL에 자동적으로 맵핑된다. 컨트롤러는 Spring의 autowiring능력을 사용하여 IoC로 부터 이득을 가진다. 예를 들어 위의 간단한 컨트롤러의 "cruncher" bean프라퍼티내 표현되는 의존성은 현재 웹 애플리케이션 컨텍스트내에서 해석된다. setter와 생성자 의존성 삽입(Constructor Dependency Injection) 모두 각각 설정을 가지지 않고 사용가능하다.

다중 URL경로를 보여주는 생성자 삽입의 예제이다.

```
/**
* 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");
    }

}
```

이 접근법은 다음의 이익을 가진다.

☒ 명백하게 제거된 설정양. 매번 우리는 XML설정을 추가할 필요가 없는 컨트롤러를 추가한다. 속성-기반 트랜잭션 관리처럼 기초적인 구조가 대체한다. 좀더 많은 애플리케이션 클래스를 추가하는것이 매우 쉽다.

☒ 우리는 컨트롤러를 설정하기 위한 Spring IoC의 강력함을 유지한다.

이 접근법은 다음의 제한을 가진다.

☒ 좀더 복잡한 빌드 처리에서 한번만의 처리(One-off cost). 우리는 속성 컴파일 단계와 속성 인덱스 단계가 필요하다.어쨌든 한번의 대체로 이것은 문제가 되지는 않을것이다.

☒ 비록 나중에 추가될 다른 속성 제공자(provider)를 위한 지원이 있더라도 현재 Commons Attributes만 지원한다,

☒ "타입에 의한 autowiring" 의존성 삽입은 컨트롤러를 위해 지원된다. 어쨌든 이것은 Struts Action(프레임워크로 부터 IoC지원이 없는)과 WebWork Action(기본적인 IoC지원만 하는)의 장점으로 그것들을 남긴다.

☒ 자동적인 마법같은 IoC해석의 신뢰는 혼동된다.

타입에 의한 autowiring은 명시된 타입의 의존성이 되어야만 하는것을 의미한다. 우리는 AOP를 사용한다면 주의할 필요가 있다. TransactionProxyFactoryBean을 사용하는 공통적인 경우에 예를 들어 우리는 Cruncher처럼 비지니스 인터페이스의 두가지 구현물(원래의 POJO정의와 트랜잭션적인 AOP프록시)이 된다. 이것은 애플리케이션 컨텍스트가 타입 의존성을 분명하게 해석하지 못하는것처럼 작동하지 않을것이다. 해결법은 자동프록시 구조를 셋업하는 AOP 자동프록시를 사용하는것이다. 그래서 정의된 Cruncher의 하나의 구현물만이 있고 구현물은 자동적으로 advised된다. 게다가 이 접근법은 위에서 언급된것처럼 속성-대상화된 선언적인 서비스와 잘 작동한다. 속성 컴파일 처리가 웹 컨트롤러 대상화(targeting)를 다루기 위해 대체되어야만 하는것처럼 이것은 셋업하기 쉽다.

다른 메타데이타 기능과는 달리, 사용가능한 Commons Attributes구현물(org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping)만이 있다. 이 제한은 속성 컴파일이 필요하고 속성 인덱싱(PathMap속성을 가진 모든 클래스를 위해 속성 API에 요청하는 능력)이 필요하다는 사실이다. 인덱싱은 org.springframework.metadata.Attributes에서 현재 제공되지 않지만 나중에 지원될것이다.(만약 당신이 인덱싱을 지원하는 다른 속성 구현물을 위한 지원을 추가하길 원한다면 당신은 당신이 선호하는 속성 API를 사용하는 두개의 protected성격의 추상 메소드를 구현하는 CommonsPathMapHandlerMapping의 수퍼클래스인 AbstractPathMapHandlerMapping을 쉽게 확장할수 있다.)

게다가 우리는 빌드 처리내에서 두가지의 추가적인 단계(속성 컴파일과 속성 인덱싱)가 필요하다. 속성 인덱서(indexer) 작업의 사용은 위에서 보여준다. 현재 Commons Attribute는 인덱싱을 위한 입력처럼 jar파일을 요구한다.
만약 당신이 핸들러 메타데이타 맵핑 접근법으로 시작한다면 이것은 고전적인 Spring XML맵핑 접근법에 어느 지점(point)을 교체하는것이 가능하다. 그래서 당신은 이 선택사항을 닫지 않는다. 이러한 이유로 나는 내가 메타데이타 맵핑을 사용하여 웹 애플리케이션을 종종 시작하는것을 알았다.

8.6. 메타데이타 속성의 다른 사용

메타데이타 속성의 다른 사용은 인기가 증가되면 나타난다. Spring 1.2처럼 JMX노출(exposure)을 위해 Commons Attributes(JDK1.3이상)와 JSR-175 annotations(JDK 1.5) 모두를 통해 메타데이타 속성이 지원된다.

8.7. 추가적인 메타데이타 API를 위한 지원 추가하기

다른 메타데이타 API를 위한 지원을 제공하길 원한다면 이것은 그렇게 하기 쉽다.

당신의 메타데이타 API를 위한 외형처럼 org.springframework.metadata.Attributes 인터페이스를 간단히 구현한다. 당신은 위에서 보여진것처럼 당신의 bean정의내 이 객체를 포함할수 있다.

AOP 메타데이타-기반 자동프록시처럼 메타데이타를 사용하는 모든 프레임워크 서비스는 당신의 새로운 메타데이타 제공자(provider)를 사용하는것이 자동적으로 가능하게 될것이다.

Chapter 9. DAO support

9.1. 소개

Spring에서 DAO(데이터 접근 객체)지원은 JDBC, Hibernate또는 표준화된 방법으로의 JDO와 같은 데이터 접근 기술을 가지고 작업하는것을 쉽게 하자는데 가장 큰 목적이 있다. 이것은 당신에게 그것들 사이에 교체를 쉽게 하도록 하고 각각의 기술로 명시한 캐치하는 예외에 대한 걱정없이 코딩하도록 허락한다.

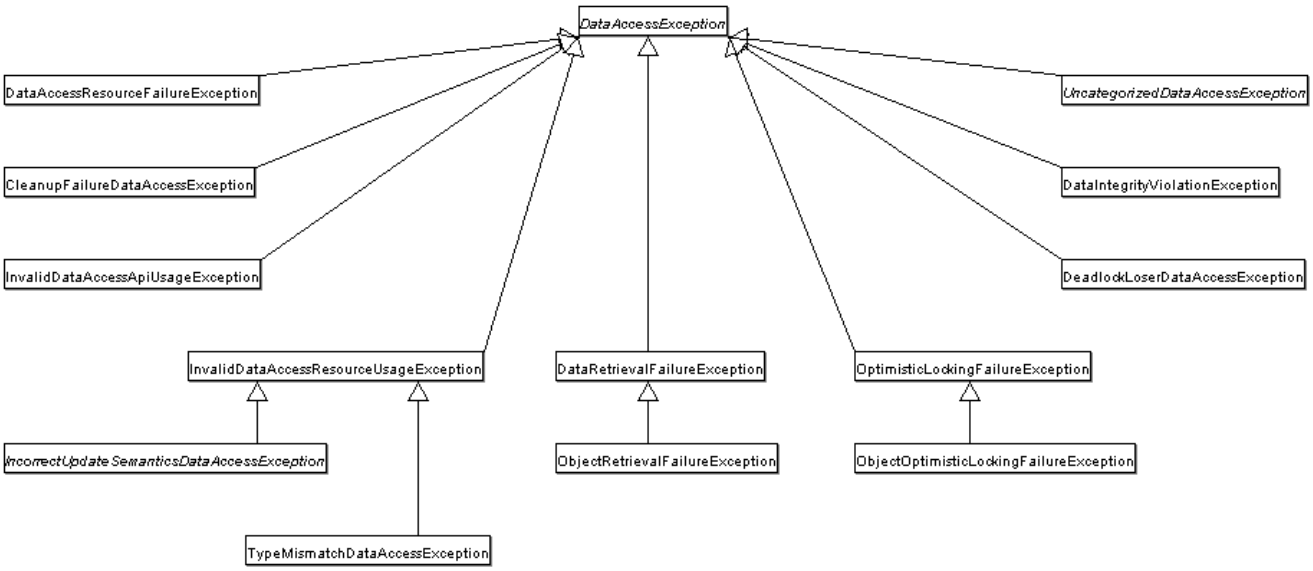
9.2. 일관된 예외 구조

Spring은 가장 상위 예외처럼 DataAccessException 과 함께 자기자신만의 예외구조를 위해 SQLException 처럼 예외를 서술하는 기술로부터 편리한 변환을 제공한다. 잘못된것처럼 어떤 정보를 손실하는 위험이 결코 없도록 이런 예외는 원래의 예외를 포장한다.

JDBC예외에 추가적으로 Spring은 Hibernate예외를 포장할수 있고 소유자, 체크되지 않은 예외로 부터 변환하고 추상화된 런타임예외의 설정할수 있다. 이것은 JDO예외에서도 같다. 이것은 괴로운 반복적 catches/throws구문과 예외선언 없이 적당한 레이어에서만 회복될수 없는 영속성 예외를 다루도록 허락한다. 당신은 여전히 당신이 필요한 어느곳에서든 예외를 잡고 다룰수 있다. 위에서 언급한 것처럼 JDBC예외(DB 정의 dialects)는 같은 구조로 변환한다. 변함없는 프로그래밍 모델내에서 JDBC와 함께 몇몇 작업을 수행할수 있다는 것을 의미한다.

위에서 ORM접근 프레임워크의 Template버전을 위해서 참이다. 만약 당신이 클래스에 기초한 Interceptor를 사용한다면 애플리케이션은 스스로 HibernateExceptions와 JDOExceptions을 다루어야만 한다. SessionFactoryUtils 의 convertHibernateAccessException 또는 convertJdoAccessException 메 소드로 각각 위임하는것을 선호한다. 이 메소드들은 org.springframework.dao 예외 구조와 호환이 되는 것으로 변환한다. JDOExceptions이 체크되지 않은것 처럼 그들은 간단히 던져질수 있다. 예외의 개념에서 일반적인 DAO추상화를 희생한다.

Spring이 사용하는 예외 구조는 다음 그래프내에서 윤곽이 그려진다.



9.3. DAO지원을 위한 일관된 추상클래스

JDBC, JDO그리고 Hibernate같은 일관적인 방법으로 데이터접근하는 기술의 다양함을 사용해서 쉽게 작업을 수행하기 위해서 Spring은 당신이 확장할수 있는 추상화된 DAO클래스들을 제공한다. 이런 추상화된 클래스들은 데이터소스를 셋팅하고 당신이 현재 사용중인 기술을 명시하는 다른 설정상의 셋팅을 하기 위한 메소드를 가지고 있다.

DAO지원 클래스:

- JdbcDaoSupport - JDBC데이터 접근 객체를 위한 슈퍼클래스(super class), 하위 클래스에 기초를 두는 JdbcTemplate을 제공하고 셋팅되기 위한 DataSource를 요구한다.
- HibernateDaoSupport - Hibernate데이터 접근 객체를 위한 슈퍼클래스(super class), 하위 클래스에 기초를 두는 HibernateTemplate을 제공하고 셋팅되기 위한 SessionFactory를 요구한다. SessionFactory, flush mode, 예외 번역 등등 처럼 나중에 셋팅을 재사용하기 위해 HibernateTemplate를 통해 대안으로 직접 초기화될수 있다.
- JdoDaoSupport - JDO데이터 접근 객체를 위한 슈퍼클래스(super class), 하위 클래스에 기초를 두는 JdoTemplate을 제공하고 셋팅되기 위한 PersistenceManagerFactory를 요구한다.

Chapter 10. JDBC를 사용한 데이터 접근

10.1. 소개

JDBC추상 프레임워크는 Spring에 의해 제공되는 4개(core , datasource , object , 그리고 support)의 패키지로 구성된다.

org.springframework.jdbc.core 패키지는 JdbcTemplate를 포함하고 이것의 다양한 callback인터페이스, 거기다가 다양한 관련 클래스를 포함한다.

org.springframework.jdbc.datasource 패키지는 쉬운 데이터소스 접근을 위한 유틸리티 클래스를 포함하고 J2EE컨테이너밖에서 변경이 되지 않은 JDBC코드를 테스트하고 실행하기 위해 사용될수 있는 여러가지 간단한 DataSource구현을 포함한다. 유틸리티클래스는 필요하다면 JNDI로 부터 Connection을 얻고 Connection을 닫는 정적 메소드를 제공한다. 이것은 DataSourceTransactionManager를 사용하는 것처럼 쓰레드범위의 연결을 지원한다.

그 다음 org.springframework.jdbc.object 패키지는 쓰레드에 안전하고 재사용가능한 객체처럼 RDBMS 쿼리, update 그리고 저장 프로시저를 표현하는 클래스를 포함한다. 이 접근법은 JDO에 의해 영상화 되었다. 쿼리에 의해 반환된 객체는 데이터베이스로 부터 “disconnected” 된다. JDBC추상화의 높은 레벨은 org.springframework.jdbc.core 패키지내에서 하위 레벨에 의존한다.

마지막으로 org.springframework.jdbc.support 패키지는 SQLException 번역 기능과 몇개의 유틸리티 클래스를 찾을수 있는 곳이다.

JDBC처리중에 던져진 예외는 org.springframework.dao 패키지내에서 정의된 예외로 번역이 된다. 이것은 Spring JDBC추상 레이어를 사용하는 코드가 JDBC또는 RDBMS특성 에러 처리를 구현할 필요가 없다는 것을 의미한다. 모든 번역된 예외는 호출자에게 전파되기 위한 다른 예외를 허락하는 동안 당신이 복구할수 있는 예외를 잡는 옵션을 제공하고 체크되지 않는다.

10.2. 기본적인 JDBC처리와 에러 처리를 위한 JDBC Core클래스 사용하기

10.2.1. JdbcTemplate

이것은 JDBC Core패키지에서 핵심 클래스이다. 이것은 자원을 생성하고 해재함으로써 JDBC의 사용을 단순화시킨다. 이것은 연결을 닫는것을 잊어버리는것처럼 공통적으로 발생할수 있는 에러를 피하도록 도와준다. 이것은 statement생성및 수행, SQL을 생성하고 결과물을 반환하고 애플리케이션 코드를 벗어나는 핵심적인 JDBC절차를 수행한다. 이 클래스는 SQL쿼리, update문 또는 저장 프로시저 호출, ResultSets를 넘어서 순환을 모방하고 반환된 인자값을 보여주는 작업을 수행한다. 이것은 또한 JDBC예외를 잡고 일반적인 것으로 그것들을 번역하고 좀더 다양한 정보를 제공하도록 하고 org.springframework.dao 패키지내에 정의된 예외 구조제공한다.

이 클래스를 사용하는 코드는 단지 명백하게 정의된 규칙을 제공하는 callback인터페이스만 구현할 필요가 있다. PreparedStatementCreator callback인터페이스는 SQL과 필요한 인자를 제공하는 클래스에 의해 제공되는 Connection으로 prepared statement를 생성한다. 호출 가능한 statement를 생성하는 것은 CallableStatementCreateor 인터페이스이다. RowCallbackHandler 인터페이스는 ResultSet으로 부터 각각의

row에서 값을 뽑아낸다.

이 클래스는 데이터소스 참조또는 애플리케이션 컨텍스트내에서 준비되고 빈(bean)참조처럼 서비스하기 위해 직접적인 초기화를 통해 서비스구현내에서 사용될수 있다. 주의: 데이터소스는 애플리케이션 컨텍스트내에서 언제나 빈처럼 설정되어야 한다. 이 클래스는 callback인터페이스와 SQLExceptionTranslator인터페이스에 의해 인자화 되기 때문에 이것을 하위클래스화 할 필요가 없다. 이 클래스에 의해 발생된 모든 SQL은 로그화된다.

10.2.2. DataSource

데이터베이스로부터 데이터작업을 수행하기 위해서 우리는 데이터베이스로 부터 Connection을 얻을 필요가 있다. Spring은 DataSource 을 통해서 이것을 수행한다. DataSource 는 JDBC스펙의 일부이고 생성된 connection공장처럼 볼수 있다. 이것은 컨테이너또는 프레임워크에게 높은 성능의 Connection pooling와 애플리케이션 코드로 부터 트랜잭션 관리 부분을 숨길수 있도록 한다. 개발자의 입장에서 당신은 데이터베이스에 연결하는 상세내역을 알 필요가 없다. 이것은 데이터베이스를 셋팅하는 관리자의 책임이다. 당신은 당신의 코드를 개발하고 테스트하는 동안 두가지 책임을 모두 수행해야 할지도 모르지만 어떻게 데이터소스가 설정이 되는지에 대해서 알필요는 없다.

Spring의 JDBC레이어를 사용할때 당신은 JNDI로 부터 데이터소스를 얻거나 Spring배포내에서 제공되어 있는 구현물로 자신만의 설정을 할수도 있다. 후자의 경우 웹 컨테이너밖에서 단위테스팅을 능숙하게 할수 있게 한다. 우리는 나중에 다루어질 여러개의 추가적인 구현물이 있지만 이 섹션에서 DriverManagerDataSource 을 사용할것이다. DriverManagerDataSource 는 당신이 JDBC Connection을 얻었을때 작업하기 위해서 사용되어진 것과 같은 방식으로 작동한다. 당신은 DriverManager 가 드라이버클래스를 로드할수 있도록 JDBC드라이버의 패키지를 포함한 전체이름을 명시해야 한다. 그 다음 당신은 JDBC드라이버사이에 변경이 되는 url을 제공해야만 한다. 여기서 정확한 값을 위해서 당신 드라이버의 문서를 찾아보아야 한다. 마지막으로 당신은 데이터베이스 연결에 사용되는 사용자명과 비밀번호를 제공해야만 한다. 이것은 DriverManagerDataSource :을 설정하기 위한 방법을 보여주는 예제이다.

```
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 은 SQLExceptions과 우리의 데이터접근 전략에 얽매이지 않는 org.springframework.dao.DataAccessException 사이에 해석할수 있는 클래스에 의해 구현될수 있는 인터페이스이다.

구현은 좀더 정확성을 위해서 일반적(예를 들면, JDBC를 위해 SQLState코드를 사용하는)이거나 소유(예를 들면, Oracle에러코드를 사용하는)될수있다.

SQLErrorCodeSQLExceptionTranslator 는 초기설정에 의해서 사용이 되는 SQLExceptionTranslator의 구현이다. 이 구현은 업체코드를 명시하는데 사용한다. SQLState 구현보다 좀더 정확하지만 업체에 종속적이다. 에러코드해석은 SQLErrorCodes 이라고 불리는 자바빈 타입의 코드에 기초를 둔다. 이 클래스는 "sql-error-codes.xml"라는 이름의 설정파일의 내용에 기초를 두는 SQLErrorCodes 를 생성하기 위한 공장같은 이름의 SQLErrorCodesFactory 에 의해서 생성되고 활성화된다. 이 파일은 업체코드에 의해 활성화되고 DatabaseMetaData로 부터 얻어진 DatabaseProductName에 기초를 둔다.

SQLErrorCodeSQLExceptionTranslator 는 다음의 일치규칙(matching rules)을 적용한다.

☒ 어느 하위 클래스에 의해서 구현되는 사용자지정해석(custom translation). 이 클래스는 이 규칙을 적용하지 않는 경우에 견고해지고 스스로 사용되는 것에 주의하라.
☒ 에러코드일치를 적용하라. 에러코드는 초기설정에 의해서 SQLErrorCodesFactory으로 부터 얻어진다. 이것은 클래스패스로부터 에러코드를 찾고 데이터베이스 메타데이터로부터 데이터베이스 이름으로 부터 키를 입력한다.
☒ fallback해석자를 사용하라. SQLStateSQLExceptionTranslator는 초기설정의 fallback해석자이다.

SQLErrorCodeSQLExceptionTranslator 는 다음과 같은 방법으로 확장할수 있다.

```
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;
    }
}
```

이 예제에서 에러코드 '-12345'는 해석되었거나 초기설정 해석자구현에 의해서 해석되기 위해서 남겨진 다른 에러코드이다. 이 사용자지정 해석자(custom translator)를 사용하기 위해서 setExceptionTranslator 메소드를 사용하고 이 해석자가 필요한 데이터 접근 처리를 위해 JdbcTemplate 를 사용하기 위한 JdbcTemplate 으로 값을 넘길필요가 있다. 여기에 사용자지정 해석자가 어떻게 사용되는지에 대한 예제가 있다.

```
// 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();
```

이 사용자지정 해석자는 sql-error-codes.xml 내의 에러코드를 찾기위해 디폴트 해석자를 원하기 때문에 데이터소스를 전달했다.

10.2.4. Statements 실행하기

SQL문을 실행하기 위해 필요한 작은 코드가 있다. 당신이 필요한 모든것은 DataSource 와 JdbcTemplate 이다. 당신이 그것을 가졌을때 당신은 JdbcTemplate 과 함께 제공되는 많은 편리한 메소드를 사용할수 있다. 여기에 작지만 새로운 테이블을 생성하는 모든 기능적인 클래스를 위해 필요한 짧은 예제를 보여준다.

```
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. 쿼리문 실행하기

메소드를 수행하는 것에 추가적으로 여기엔 많은 수의 쿼리 메소드가 있다. 그 메소드의 몇몇은 하나의 값을 반환하는 쿼리를 위해 사용되는 경향이 있다. 아마도 당신은 하나의 레코드로 부터 카운트나 특정값을 가져오길 원할지도 모른다. 만약 그 경우라면 당신은 queryForInt , queryForLong 또는 queryForObject 를 사용할수 있다. 후자는 반환된 JDBC타입을 인자처럼 전달된 자바 클래스로 변환할 것이다. 만약 타입변환이 유효하지 않다면 InvalidDataAccessApiUsageException 를 던질것이다. 여기에 int 를 위한것과 String 을 위한 두개의 쿼리 메소드를 포함하는 예제가 있다.

```
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;
    }
}
```

하나의 결과물을 위한 쿼리 메소드에 추가적으로 쿼리가 반환하는 각각의 레코드를 가지는 List를 반환하는 다양한 메소드가 있다. 가장 일반적인 하나는 각각의 레코드를 위한 칼럼값을 표현하는 Map 형태의 List 를 반환하는 queryForList 이다. 만약 우리가 모든 레코드의 리스트를 가져오는 메소드를 추가한다면 다음과 같을것이다.

```
    public List getList() {
        jt = new JdbcTemplate(dataSource);
        List rows = jt.queryForList("select * from mytable");
        return rows;
    }
```

반환되는 리스트는 이것저것 보일것이다. [{name=Bob, id=1}, {name=Mary, id=2}].

10.2.6. 데이터베이스 수정하기

당신이 사용할수 있는 많은 update메소드가 있다. 나는 어떠한 기본키를 위한 칼럼을 수정하는 예제를 보여줄것이다. 이 예제에서 나는 레코드 파라미터를 위한 위치자(place holders)를 가진 SQL문을 사용한다. 대부분의 쿼리및 update메소드는 이 기능을 가진다. 파라미터 값은 객체의 배열내에 전달된다.

```
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. 데이터베이스에 연결하는 방법을 제어하기

10.3.1. DataSourceUtils

헬퍼 클래스는 필요하다면 JNDI로 부터 connection을 얻거나 connection을 닫기 위한 정적 메소드를 제공하고 쓰레드 범위의 connection을 위한 지원 예를 들면 DataSourceTransactionManager를 사용한다.

주의 : getDataSourceFromJndi메소드는 BeanFactory를 사용하지 않는 애플리케이션을 목표로 한다. 이것은 factory내 당신의 빈즈나 JdbcTemplate 인스턴스를 먼저 설정하는것을 선호한다. JndiObjectFactoryBean 는 JNDI로 부터 DataSource 를 꺼내고 다른 빈즈에게 DataSource 빈참조를 주는데 사용될수 있다. 다른 DataSource 로의 교체는 설정의 문제이다. 당신은 non-JNDI의 DataSource 를 가진 FactoryBean 의 정의를 교체할수 있다.

10.3.2. SmartDataSource

클래스에 의해 구현되기 위한 인터페이스는 관계 데이터베이스로의 connection을 제공할수 있다. javax.sql.DataSource 인터페이스를 확장하는것은 클래스가 주어진 작업후에 닫혀야만하는 connection이거나 그렇지 않더라도 쿼리하기 위해 사용하도록 허락한다. 이것은 때때로 우리가 connection을 재사용하길 원한다는 것을 안다면 효율을 위해 유용할수 있다.

10.3.3. AbstractDataSource

Spring의 DataSource 구현을 위한 추상 기본 클래스는 "시시함(uninteresting)"을 처리한다. 이것은 당신이 자신의 DataSource 구현을 쓴다면 확장할 클래스이다.

10.3.4. SingleConnectionDataSource

하나의 connection을 포장한 SmartDataSource 의 구현은 사용후에 닫히질 않는다. 분명히 이것은 다중 쓰레드 성능은 아니다.

만약 퍼시스턴스 툴들을 사용할때 suppressClose 을 true로 설정한 것처럼 클라이언트코드가 풀링된 connection의 소비내 close를 호출할 것이다. 이것은 물리적 connection대신에 close-억제 프록시를 반환할것이다. 당신은 이것을 고유의 Oracle connection이나 다른 어떠한 것처럼 형변환할수 없다는것을 알라.

이것은 기본적으로 테스트 클래스이다. 예를 들면 이것은 간단한 JNDI환경과 함께 연결되어 애플리케이션 서버밖의 코드를 쉽게 테스팅 가능하도록 한다. DriverManagerDataSource 와는 대조적으로 이것은 언제나 물리적 connection생성 초과를 피하고 같은 connection을 재사용한다.

10.3.5. DriverManagerDataSource

SmartDataSource 의 구현은 빈 프라퍼티를 통해 일반적인 예전 JDBC드라이버를 설정하고 모든 시점에 새로운 connection을 반환한다.

각각의 ApplicationContext내 DataSource 빈 이나 간단한 JNDI환경과의 연결내에서 어느쪽이건 테스트나 J2EE컨테이너 밖의 단독 환경을 위해 유용하다. 풀 성격의 Connection.close() 호출은 간단하게 connection을 닫을 것이다. 그래서 어떠한 DataSource-인식 퍼시스턴스코드도 작동할것이다.

10.3.6. DataSourceTransactionManager

하나의 JDBC데이터 소스를 위한 PlatformTransactionManager구현은 특정 데이터 소스로 부터 쓰레드에 JDBC connection을 바인드한다. 잠재적으로 데이터 소스당 하나의 쓰레드 connection을 허락한다.

애플리케이션 코드는 J2EE의 표준적인 DataSource.getConnection 대신에 DataSourceUtils.getConnection(DataSource) 을 통해 JDBC connection을 가져와야만 한다. 이것은 어쨌든 추천된다. 그리고 이것은 체크된 SQLException 대신에 체크되지 않은 org.springframework.dao 예외를 던진다. JdbcTemplate 와 같은 모든 프레임워크 클래스는 절대적으로 이 전략을 사용한다. 만약 이 트랜잭션 관리자를 사용하지 않는다면 룩업 전략은 공통된 것과 같이 정확하게 작동한다.

사용자 지정 격리 레벨을 지원하고 선호하는 JDBC statement쿼리 중단처럼 적용된 것을 중단한다. 후자를 지원하기 위해 애플리케이션 코드는 JdbcTemplate 나 각각의 생성된 statement를 위한 DataSourceUtils.applyTransactionTimeout 메소드 호출을 사용해야만 한다.

이 구현은 하나의 자원일 경우 JTA를 지원하기 위해 컨테이너를 요구하지 않는것처럼 JtaTransactionManager 대신에 사용될수 있다. 두가지 사항 사이의 전환은 설정상의 문제이다. 만약 당신이 요구되는 connection룩업 패턴을 고집한다면 JTA는 사용자 지정 격리 레벨을 지원하지 않는다는 것에 주의하라.!

10.4. 자바 객체처럼 JDBC작업을 모델링 하기.

org.springframework.jdbc.object 패키지는 좀더 객체 지향적인 방법으로 데이터베이스에 접근하는 것을 허락하는 클래스를 포함한다. 당신은 쿼리를 수행하고 관계적인 칼럼 데이터를 비지니스 객체의 프라퍼티로 맵핑하는 비지니스 객체를 포함하는 리스트처럼 결과를 얻을수 있다. 당신은 또한 저장 프로시저와 update, delete그리고 insert 구문을 실행할수 있다.

10.4.1. SqlQuery

SQL쿼리를 표현하기 위한 쓰레드에 안전한 객체를 재사용가능하다. 하위 클래스는 ResultSet를 반복하는 동안 결과를 저장할수 있는 객체를 제공하기 위해 newResultReader()메소드를 구현해야만 한다. 이 클래스는 드물게 직접적으로 사용된다. 이 클래스를 확장하는 MappingSqlQuery 가 레코드를 자바 플래스로 맵핑하기 위한 좀더 편리한 구현을 제공한다. SqlQuery 을 확장하는 다른 구현은 MappingSqlQueryWithParameters 와 UpdatableSqlQuery 이다.

10.4.2. MappingSqlQuery

MappingSqlQuery 는 명확한 하위 클래스가 JDBC ResultSet 의 각각의 레코드를 객체로 변환하기 위한 추상메소드인 mapRow(ResultSet, int) 를 구현함으로써 재사용가능한 쿼리이다.

모든 SqlQuery 구현으로 이것은 매우 종종 사용되고 이것은 사용하기 가장 쉬운 것중 하나이다.

여기에 사용자 지정 테이블로부터 데이터를 Customer라고 불리는 자바 객체로 맵핑하는 사용자 지정 쿼리의 예제를 간단히 설명하는것이 있다.

```
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;
  }
}
```

우리는 오직 파라미터로 DataSource 를 가지는 사용자지정 쿼리를 위한 생성자를 제공한다. 이 생성자에서 우리는 DataSource 와 이 쿼리를 위해 레코드를 가져오기 위해 수행되어야 하는 SQL을 가진 수퍼클래스의 생성자를 호출한다. 이 SQL은 수행되는 동안 전달될 어떠한 파라미터를 위한 위치자를 포함한 PreparedStatement 를 생성하기 위해 사용될 것이다. 각각의 파라미터는 SqlParameter 내에 전달될 declareParameter 메소드를 사용해서 선언되어야 한다. SqlParameter 는 이름과 java.sql.Types 내에 정의될 JDBC타입을 가진다. 모든 파라미터가 compile 메소드를 호출해서 정의된 후에 statement는 준비되고 나중에 수행된다.

이 사용자 정의 쿼리가 초기화되고 수행되는 코드를 보자.

```
  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;
  }
```

예제내 메소드는 오직 파마리터로써 전달되는 id와 함께 customer를 가져온다. CustomerMappingQuery 클래스의 인스턴스를 생성한 후에 우리는 전달될 모든 파라미터를 포함할 객체의 배열을 생성한다. 이 경우에 오직 한개의 파라미터가 있고 이것은 Integer 로 전달된다. 지금 우리는 파라미터의 배열을 사용해서

쿼리를 수행할 준비가 되었고 우리의 쿼리를 통해 반환되는 각각의 레코드를 위한 Customer 객체를 포함하는 List 를 얻게된다. 이 경우에 적합한 경우 하나의 항목이 될것이다.

10.4.3. SqlUpdate

RdbmsOperation 하위 클래스는 SQL update를 상징한다. 쿼리처럼 update객체는 재사용가능하다. 모든 RdbmsOperation객체처럼 update는 파라미터를 가지고 SQL내 정의된다.

이 클래스는 쿼리 객체의 execute()메소드를 위한 많고 유사한 update()메소드를 제공한다.

이 클래스는 명확하다. 비록 이것이 하위 클래스(예를 들면 사용자 정의 update메소드를 추가하는)가 될수 있지만 이것은 SQL을 셋팅하고 파라미터를 선언함으로써 쉽게 파라미터화 될수 있다.

```
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

RDBMS 저장 프로시저의 객체 추상화를 위한 수퍼클래스이다. 이 클래스는 추상적이고 execute메소드들은 protected상태이다. 좀더 단단하게 타이핑된 하위 클래스를 통해 다른것보다 좀더 사용이 제한적이다.

상속된 sql프라퍼티는 RDBMS에 저장된 저장프로시저의 이름이다. JDBC 3.0은 명명된 파라미터를 소개한다. 비록 이 클래스에 의해 제공되는 다른 특징이지만 여전히 JDBC 3.0에서 필요하다.

이것은 Oracle데이터베이스에서 사용되는 sysdate()함수를 호출하는 프로그램 예제이다. 저장 프로시저 기능을 사용하기 위해 당신은 StoredProcedure 를 확장한 클래스를 생성해야만 한다. 여기엔 입력 파라미터가 없지만 SqlOutParameter 클래스를 사용한 date처럼 선언된 출력 파라미터는 있다. execute() 메소드는 key로써 파라미터이름을 사용한 각각의 선언된 출력 파라미터를 위한 항목을 가지는 map을 반환한다.

```
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

결과의 하나의 레코드를 반환하는 쿼리를 위한 SQL "함수" 랩퍼. 디폴트 행위는 int를 반환하는 것이지만 추가적인 반환 타입 파라미터를 가진 메소드를 사용해서 오버라이드 할수 있다. 이것은 JdbcTemplate 의 queryForXxx 메소드를 사용하는것이 유사하다. SqlFunction 이 가진 장점은 JdbcTemplate 을 생성할 필요가

없다는 것이다. 이것은 상태(scenes)뒤에서 행해진다.

이 클래스는"select user()" 나 "select sysdate from dual" 와 같은 쿼리를 사용해서 하나의 결과를 반환하는 SQL함수들을 호출하는 것을 사용하는 경향이 있다. 이것은 좀더 복잡한 저장 프로시저를 호출하거나 저장 프로시저나 저장 함수를 호출하기 위한 CallableStatement 를 사용하는 경향은 아니다. 이러한 타입의 처리를 위해 StoredProcedure 나 SqlCall 을 사용하라.

이것은 하위 클래스에 알반적으로 필요하지 않은 명확한 클래스이다. 이 패키지를 사용하는 코드는 이 타입의 객체를 생성하고 SQL과 파라미터를 선언하고 함수를 수행하기 위해 반복적으로 선호하는 run메소드를 호출 할수 있다. 이것은 테이블로 부터 레코드의 카운트를 가져오는 예제이다.

```
public int countRows() {
    SqlFunction sf = new SqlFunction(dataSource, "select count(*) from mytable");
    sf.compile();
    return sf.run();
}
```

Chapter 11. 객체-관계 연결자(O/R Mappers)를 이용한 데이터 접근

11.1. 소개

Spring은 자원관리측면에서 Hibernate, JDO, iBATIS SQL Maps, DAO구현 지원, 트랜잭션 전략등의 통합을 제공한다. Hibernate를 위해서 많은 IoC편리 기능과 많은 전형적인 Hibernate통합 이슈를 할당하는 첫번째 클래스가 있다. 여기의 모든것은 Spring의 일반적인 트랜잭션와 DAO exception구조를 따른다.

Spring은 데이터접근 애플리케이션을 생성하기 위해 당신이 선택한 O/R mapping레이어를 사용할때 중요한 지원을 추가한다. 그중에 첫번째는 당신은 O/R맵핑을 위해서 Spring에서 제공하는것을 사용해서 시작해야 한다는것을 알아야만 한다. 당신은 모든것을 해야 할 필요는 없다. 확장하는것과는 상관없이 당신은 다시보기 위해서 초대되었고 Spring접근에 영향을 끼친다. 유사한 하부조직을 만드는 노력과 위험을 가지는데 대해 결정을 먼저해야 한다. 대부분의 O/R맵핑지원은 라이브러리 스타일내에서 사용되는 기술과는 상관없이 모든것은 재사용가능한 자바빈처럼 디자인 되었다. ApplicationContext나 BeanFactory내부에서의 사용은 설정과 배치의 쉬움이라는 면에서 추가적인 이득을 제공한다. 이 섹션의 예제들은 Applicationcontext내에서 설정이 됨을 보여준다.

다음은 O/R맵핑 애플리케이션을 생성하기 위해 Spring을 사용함으로써 발생하는 몇몇 장점들이다.

☒ 업체에 종속적인 락(lock-in)방식을 피할수 있고 mix-and-match구현방식을 허락한다. Hibernate는 강력하고 확장이 용이하며 오픈소스이고 free하다. 이것은 여전히 자신만의 API를 사용한다. 더군다나 누구는 iBATIS가 좀더 가볍다는 것에 대해서 논쟁을 벌일수도 있다. 복잡한 O/R맵핑 전략을 요구하지 않는 애플리케이션내에서의 사용은 매우 환상적이다. 주어진 선택에서 이것은 기능과 성능 그리고 다른 어떠한 이유로 다른 구현으로 교체해야 할 경우에 언제나 표준적이고 추상적인 API들을 사용해서 주요한 애플리케이션 기능을 구현하도록 바랄것이다. 예를 들면 Spring의 Hibernate트랜잭션과 exception의 추상화는 데이터접근 기능을 구현하고 있는 맵퍼/DAO객체내에서 쉽게 교환할수 있도록 하는 IoC접근으로 Hibernate의 어떠한 성능적 손실없이 당신의 애플리케이션내에서 모든 Hibernate관련 코드를 쉽게 분리하도록 한다. DAO와 함께 처리되는 높은 단계의 서비스 코드는 그 구현에 대해서 어떤것도 알필요가 없다. 이 접근은 mix-and-match접근으로 방해가 되지 않은 방식내에서 의도적으로 데이터접근을 쉽게 구현하도록 만든다는 추가적인 이익을 가져다 준다. 잠재적으로 기존코드를 계속적으로 사용하게 하는것과 각각의 기술의 강력함을 그대로 유지시킨다는 큰 이익을 제공하기도 한다.

☒ 테스트의 용이함(ease) Spring의 IoC접근은 Hibernate세션 팩토리(session factories)의 구현과 위치, DataSource, 트랜잭션 관리자 그리고 맵퍼객체 구현을 쉽게 교체할수 있게 만든다. 이것은 격리내에서 각각의 영속성과 관련된 코드를 쉽게 격리시키고 테스트 할수 있게 만든다.

☒ 일반적인 자원 관리 Spring애플리케이션 컨텍스트는 Hibernate SessionFactories, JDBC datasource, iBATIS SQLMaps설정 객체, 그리고 다른 관련 자원의 위치와 설정을 다룰수 있다. 이것은 그런 값들을 쉽게 관리하고 변경할수 있게 만든다. Spring은 효율적이고 쉽고 안전하게 Hibernate Sessions을 다룰수 있는 기능을 제공한다. Hibernate를 사용하는 관련코드는 효율적이고 적합한 트랜잭션 관리를 위해 Hibernate Session객체를 사용할 필요가 있다. Spring은 각각의 선언과 AOP메소드 접근, 명시, 자바코드단계에서 탬플릿 래퍼 클래스를 사용해서 현재 쓰레드에 투명하게 session을 생성하고 바인딩하는것을 쉽게 만든다. 게다가 Spring은 Hibernate포럼에 반복적으로

올라오는 많은 사용상의 이슈를 쉽게 풀어놓았다.

☒ Exception wrapping Spring은 선택한 O/R맵핑 툴로부터 exception을 제어할수 있다. 선호하고 체크된 exception으로 부터 추상화된 런타임exception으로 변형할수 있다. 이것은 당신에게 대부분의 복수할수 없고 선호하는 단계에서만 발생하는 짜증나는 반복적인 catches/throws구문과 exception선언 없이 영속성관련 exception을 다룰수 있도록 한다. 당신은 당신이 필요한 어느곳에서든지 exception을 잡아서 처리할수 있다. JDBC exception들 또한 당신이 일관적인 프로그래밍 모델내에서 JDBC를 가지고 몇몇 기능을 수행할수 있다는것을 의미하는 같은 구조로 변환이 가능하다는것을 기억하라.

☒ 통합된 트랜잭션 관리 Spring은 선언, AOP스타일의 메소드 인터셉터, 또는 자바코드 단계에서 명백한 'template'래퍼 클래스를 가지고 O/R맵핑 코드를 만들수 있다. 이런 경우에 트랜잭션의미는 당신을 위해서 다루어지고 exception이 관리되는 경우에 명백하게 트랜잭션이 다루어진다. 밑에 논의되는 것처럼 당신은 Hibernate관련 코드에 영향이 없이 다양한 트랜잭션 관리자를 사용하거나 교체할수 있게 되는 장점또한 가지게 된다. 그리고 추가되는 장점은 JDBC관련 코드가 O/R맵핑을 사용하는 코드와 완벽하게 트랜잭션적으로 통합이 될수 있다. 이것은 예를 들면 Hibernate나 iBATIS내에서 구현되지 않은 기능을 다룰 경우에 유용하다.

11.2. Hibernate

11.2.1. 자원 관리

전형적인 비지니스 애플리케이션은 종종 반복적인 자원 관리 코드가 소스를 뒤죽박죽 만든다. 많은 프로젝트를 이런일을 위해서 자신만의 솔루션을 만들기를 시도한다. 때때로 프로그래밍의 편의성을 위한 명백한 실패의 제어를 희생하기도 한다. Spring은 template을 통한 IoC 즉 callback인터페이스와 함께 하부구조 클래스, 또는 AOP인터셉터 적용등으로 명백한 자원 관리를 위한 간단한 해결법을 제공한다. 하부구조는 명백한 자원 핸들링을 하고 체크되지 않은 하부구조 exception구조를 특정 API exception의 적합하게 변환한다. Spring은 어떠한 데이터 접근 전략에도 적용가능한 DAO exception구조를 소개한다. JDBC를 사용하기 위해서는 JdbcTemplate클래스가 connection핸들링을 위해 이전 섹션에서 언급되었고 SQLException이 데이터베이스에 종속적인 SQL에러코드를 의미있는 exception클래스로 해석하는것을 포함하는 DataAccessException구조로 변환된다. 이것은 각각의 Spring트랜잭션 관리자를 통해서 JTA와 JDBC트랜잭션을 지원한다. Spring은 JdbcTemplate와 유사한 HibernateTemplate/JdoTemplate, HibernateInterceptor/JdoInterceptor 그리고 Hibernate/JDO트랜잭션 관리자로 구성된 Hibernate와 JDO지원을 제공한다. 이것의 커다란 목표는 어떠한 데이터 접근와 트랜잭션 기술을 가지고 깔끔한 애플리케이션 계층화가 애플리케이션 객체의 느슨한 커플링된 상태에서 가능하도록 하는것이다. 데이터 접근과 트랜잭션 전략에서 더이상 비지니스 객체의 의존성 문제가 없고 더 이상 하드코딩형 자원탐색이 없으며, 더 이상 hard-to-replace싱글톤과 고객 서비스 등록자가 없는것이다. 애플리케이션 객체를 묶는 간단하고 일관적인 접근은 가능한 한 컨테이너 의존성으로 부터 재사용가능하고 free하게 유지시켜준다. 모든 개별적인 데이터 접근 기능은 그들 자신에게는 재사용가능하지만 Spring을 알 필요가 없는 XML기반의 설정과 상호간에 참조되는 자바빈 인스턴스를 제공하는 Spring의 애플리케이션 컨텍스트 개념과 함께 잘 통합된다. 전형적인 Spring애플리케이션에서 많은 중요한 객체(데이터 접근 탬플릿, 탬플릿을 사용하는 데이터 접근 객체, 트랜잭션 관리자, 데이터 접근객체와 트랜잭션 관리자를 사용하는 비지니스 객체, 웹의 화면 해설자(resolvers), 비지니스 객체를 사용하는 웹 컨트롤러 등등)는 자바빈이다.

11.2.2. 애플리케이션 컨텍스트내에서 자원 정의

하드코딩형 자원 탐색을 위한 애플리케이션 생성을 피하기 위해서 Spring은 애플리케이션 컨텍스트내에

빈즈처럼 JDBC DataSource나 Hibernate SessionFactory처럼 자원 정의를 하게 한다. 애플리케이션 객체는 빈참조를 통해 미리 선언된 인스턴스에 참조를 받은 자원에 접근하기 위해 필요한 것이다. 다음의 XML애플리케이션 컨텍스트 선언으로 부터 발췌는 JDBC DataSource와 Hibernate SessionFactory를 설정하는 방법을 보여준다.

```
<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>
```

JNDI를 사용하는 DataSource에서 Jakarta Commons DBCP BasicDataSource처럼 로컬에 정의된 것으로 바꾸는것은 설정의 문제라는것을 기억하라.

```
<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>
```

당신은 또한 JNDI를 사용하는 SessionFactory를 사용할수도 있지만 전형적으로 EJB컨텍스트 외부에서는 필요하지 않다.

11.2.3. Inversion of Control: Template and Callback

탬플릿팅을 위한 기본적인 프로그래밍 모델은 어떤 데이터접근 객체나 비지니스 객체의 부분이 될수 있는 메소드를 위해 다음처럼 볼수 있다. 펼쳐진 모든 객체의 구현에서 제한은 없다. 이것은 Hibernate의

SessionFactory을 제공할 필요가 있다. 이것은 어디서든 나중에 얻을수 있지만 간단한 setSessionFactory 빈 속성 setter을 통해 Spring애플리케이션 컨텍스트로 부터 빈처럼 참조할것이다. 다음의 작은 조각(snippets)은 Spring애플리케이션 컨텍스트내에서 DAO선언을 보여준다. 위에서 선언된 SessionFactory를 참조하고 있고 DAO메소드 구현을 위한 에제이다.

```
<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;
                }
            }
        );
    }
}
```

callback구현은 어떤 Hibernate데이터 접근을 위해 사용되는데 영향을 끼칠수 있다. HibernateTemplate은 Session들이 명백하게 열고 닫고 트랜잭션내에서 자동적으로 함께하는것을 확실시한다. 이 탬플릿 인스턴스는 쓰레드에 안전하고(thread-safe) 재사용가능하다. 그들은 주위 클래스의 인스턴스 변수처럼 유지될수 있다. 하나의 검색, 로드, saveOrUpdate또는 삭제 호출처럼 간단한 한단계의 작업은 HibernateTemplate이 대안적으로 편리한 한라인 callback구현처럼 대체될수 있는 메소드를 제공한다. 게다가 Spring은 SessionFactory를 받기위한 setSessionFactory메소드를 제공하는 편리한 HibernateDaoSupport base클래스를 제공한다. 그리고 하위클래스에 의해 사용되기 위한 getSessionFactory과 getHibernateTemplate를 제공한다. 이것은 전형적인 요구사항을 위해 매우 간단한 DAO구현을 허락한다.

```
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. 탬플릿 대신에 AOP인터셉터 적용하기.

HibernateTemplate 을 사용하는것에 대한 대안으로는 위임하는 try/catch블럭내에서 Hibernate코드와 함께 callback구현과 애플리케이션 컨텍스트내의 각각의 인터셉터 설정을 대체하는 Spring의 AOP HibernateInterceptor를 사용하는 것이다. 다음의 조각(snippets)은 각각의 DAO, 인터셉터 그리고 Spring애플리케이션 컨텍스트내의 프록시 선언을 보여주고 DAO메소드 구현의 예를 보여준다.

```
<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);
        }
    }
}
```

이 메소드는 먼저 쓰레드 범위의 세션을 열고 메소드 호출 후에는 이것을 닫는 것을 위해 단지 HibernateInterceptor와 작업을 수행한다. getSession의 false플래그 Session이 벌써 존재한다는 것을

확신하는것이다. 반면에 SessionFactoryUtils는 아무것도 발견되지 않는다면 새로운 Session을 생성할 것이다. 만약에 쓰레드에 바운드하는 SessionHolder이 존재한다면 예를 들면 HibernateTransactionManager트랜잭션에 의해 SessionFactoryUtils이 자동적으로 어떠한 경우에든 일부를 가져온다. HibernateTemplate은 내부적으로 SessionFactoryUtils를 사용한다. 이것은 모두 같은 하부구조이다. HibernateInterceptor의 가장 큰 장점은 HibernateTemplate이 callback내에서 체크되지 않은 exception에 제한되는 동안 데이터접근 코드내에서 던져지기 위한 어떤 체크된 애플리케이션 exception을 허락한다는 것이다. 어떤것은 종종 개별적인 체크와 callback후에 애플리케이션의 exception의 발생을 미룰수도 있다. 인터셉터의 중요한 결점은 컨텍스트에서 특별한 셋팅이 필요하다는 것이다. HibernateTemplate의 편리한 메소드는 많은 경우를 위해 좀더 간단한 의미를 제공한다.

11.2.5. 프로그램의 트랜잭션 구분(Demarcation)

그런 하위 레벨의 데이터 접근 서비스의 가장 상위에서 트랜잭션은 애플리케이션의 더 높은 레벨내에서 구분될수 있다. 여기서는 또한 비지니스 객체의 구현에서 어떠한 제한도 없다. 이것은 단지 Spring의 PlatformTransactionManager만을 필요로 한다. 나중에 어디서부터든지 올수(호출할수?) 있지만 마치 productDAO이 setProductDao메소드를 통해서 생성이 되듯이 setTransactionManager메소드를 통해 빈 참조처럼 될수도 있다. 다음의 조각(snippets)은 트랜잭션 관리자와 Spring 애플리케이션 컨텍스트내에서 비지니스 객체 선언을 보여준다. 그리고 비지니스 메소드의 구현을 위한 예제를 보여준다.

```
<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. 선언적인 트랜잭션 구분

대신에 누구는 애플리케이션 컨텍스트내에서 인터셉터 설정과 함께 트랜잭션 구분 코드를 대신할수 있는 Spring의 AOP TransactionInterceptor를 사용할수도 있다. 이것은 당신에게 비지니스 객체를 각각의 비지니스 메소드내에 반복적인 트랜잭션 구분코드의 free상태로 유지시키도록 허락한다. 게다가 전달행위와 격리레벨같은 트랜잭션 구문은 설정파일내에서 변경될수도 있다. 그리고 비지니스 객체 구현에는 어떠한 영향도 끼치지 않는다.

```
<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);
        ...
    }

    ...

}
```

HibernateInterceptor와 함께 TransactionInterceptor은 TransactionTemplate이 callback내에서 체크되지 않은 exception에 제한적인 동안 callback코드내에 던져진 체크된 애플리케이션 exception을 허락한다. TransactionTemplate는 체크되지 않은 애플리케이션 exception의 경우이거나 애플리케이션에 의해 rollback-only일 경우에 롤백처리를 한다. TransactionTemplate은 체크되지 않은 애플리케이션 예외일 경우 롤백을 유발하거나 트랜잭션이 애플리케이션(TransactionStatus을 통해)에 의해 롤백만을 수행하도록 되어 있다면 TransactionInterceptor는 디폴트에 의해 같은 방식으로 작동하지만 메소드별로 설정가능한 롤백 정책을 허락한다. 선언적인 트랜잭션 셋팅방법의 편리한 대안은 TransactionProxyFactoryBean이다. 특히 다른 어떠한 AOP인터셉터가 포함되지 않았다면 더욱 그렇다. TransactionProxyFactoryBean은 특정 목표 빈을 위한 트랜잭션 설정과 함께 자신의 프록시 정의를 조합한다. 이것은 하나의 목표 빈에 하나의 프록시 빈을 더하는 설정상의 수고를 제거한다. 게다가 당신은 전통적인 메소드가 정의되는 인터페이스나 클래스를 정의 할 필요가 없다.

```
<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. 트랜잭션 관리 전략

TransactionTemplate과 TransactionInterceptor는 Hibernate애플리케이션을 위한 HibernateTransactionManager(ThreadLocal Session을 사용하는 하나의 Hibernate SessionFactory를 위한)나 JtaTransactionManager(컨테이너의 JTA하위 시스템으로 위임하는) 가 될수 있는 PlatformTransactionManager인스턴스로 실질적인 트랜잭션 핸들링을 위임한다. 당신은 사용자 정의 PlatformTransactionManager구현을 사용할수도 있다. 그래서 근본적인 Hibernate트랜잭션관리로 부터 JTA로의 전환(예를 들면 당신의 애플리케이션의 어떠한 배치작업을 위한 분산된 트랜잭션 요구사항에 직면했을때)은 설장상의 문제가 된다. Spring의 JTA트랜잭션 구현으로 Hibernate 트랜잭션 관리자를 간단하게 대신한다. 트랜잭션 구분과 데이터 접근 코드는 변경없이 작동할 것이다. 그리고 그들은 일반적인 트랜잭션 관리 API들을 사용한다. 다중 Hibernate session factories를 통한 분산된 트랜잭션을 위해 다중 LocalSessionFactoryBean정의와 함께 트랜잭션 전략처럼 JtaTransactionManager을 간단하게 조합한다. 각각의 DAO들은 그것의 개별적인 빈 프라퍼티로 전달된 하나의 특정 SessionFactory참조를 얻게된다. 모든 근본적인 JDBC데이터 소스는 트랜잭션적인 컨테이너이다. 비지니스 객체는 전략으로 JtaTransactionManager을 사용하는 한 많은 DAO와 특정 고려없는 많은 session factory를 통해 트랜잭션의 경계를 지정할수 있다.

```
<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>
```

HibernateTransactionManager 와 JtaTransactionManager는 트랜잭션 관리자 룩업이나 JCA연결자(트랜잭션을 초기화하기 위해 EJB를 사용하지 않는 한)를 정의하는 컨테이너 없이 Hibernate를 사용하여 적당한 JVM레벨의 캐시 핸들링을 허락한다. 추가적으로 HibernateTransactionManager는 평범한 JDBC접근 코드를 위해 Hibernate의해 사용되는 JDBC connection을 추출할수 있다. 이것은 하나의 데이터베이스에 접근하는 한 JTA없이 완벽한 Hibernate/JDBC혼합 접근으로 높은 레벨의 트랜잭션 구분을 허락한다.

선언적으로 트랜잭션 경계를 구분하기 위한 TransactionProxyFactoryBean을 사용하기 위해서 사용하는 접근법의 대안을 위해서 Section 7.4.1, “BeanNameAutoProxyCreator, 또 다른 선언적 접근방법” 를 보라.

11.2.8. 컨테이너 자원 대 로컬 자원

Spring의 자원 관리는 애플리케이션 코드의 한줄의 변경도 없이 JNDI SessionFactory와 JNDI DataSource와 같은 로컬 SessionFactory사이의 간단한 전환를 허락한다. 컨테이너내 자원 정의를 유지하거나 애플리케이션 내 로컬 상태로 유지하더라도 사용되는 트랜잭션 전략의 주요한 문제이다. Spring정의 로컬 SessionFactory에 비교하여 수동으로 등록된 JNDI SessionFactory는 어떠한 이득도 제공하지 않는다. Hibernate의 JCA연결자를 통해서 등록되었다면 특히 EJB내에서 JTA트랜잭션 내 투명하게 참여한 추가된 값이 있다. Spring의 트랜잭션 지원의 중요한 이득은 컨테이너에 전혀 바운드 되지 않는 것이다. JTA가 아닌 다른 전략에 설정하는 것은 단독으로 작동하거나 테스트 환경에서도 잘 작동할것이다. 하나의 데이터베이스 트랜잭션의 전형적인 경우를 위해 특별히 이것은 가볍고 JTA에 강력한 대안이다. 트랜잭션을 다루기 위해 로컬 EJB 비상태 유지 세션빈을 사용할때 당신은 비록 하나의 데이터베이스만을 사용하고 CMT를 통해 선언적인 트랜잭션을 위해 SLSB를 사용하더라도 EJB컨테이너와 JTA에 모두 의존한다. 프로그램적으로 JTA를 사용하는것의 대안도 J2EE환경을 요구한다. JTA는 JTA와 JNDI DataSource의 개념에서 컨테이너 의존성을 포함하지 않는다. Spring을 사용하지 않는 JTA에 의도한 Hibernate트랜잭션을 위해 당신은 적당한 JVM레벨의 캐시를 위해 Hibernate JCA연결자를 사용하거나 JTATransaction이 설정된 추가적인 Hibernate트랜잭션 코드를 사용해야 한다. Spring에 의도한 트랜잭션은 만약 하나의 데이터베이스에 접근한다면 로컬 JDBC DataSource처럼 로컬에 정의된 Hibernate SessionFactory와 잘 작동할수 있다. 그러므로 당신은 분산 트랜잭션 요구사항에 실질적으로 직면했을때 Spring의 JTA트랜잭션 전략으로 물러나야 한다. JCA연결자는 컨테이너 특유의 배치단계를 필요로 하고 명백하게 첫번째 단계에서 JCA지원을 필요로 한다. 이것은 로컬 자원 정의와 Spring이 의도한 트랜잭션과 함께 간단한 웹 애플리케이션을 배치하는것보다 더 괴롭다. 그리고 당신은 종종 컨테이너의 기업용 버전(예를 들면 웹로직 익스프레스 버전은 JCA를 제공하지 않는다.)을 필요로 한다. 로컬 자원과 하나의 데이터베이스를 확장하는 트랜잭션을 가진 Spring애플리케이션은 Tomcat, Resin, 또는 Jetty와 같은 어떠한 J2EE 웹 컨테이너(JTA, JCA, 또는 EJB 없이)내에서도 작동한다. 추가적으로 미들티어같은 것은 데스크탑 애플리케이션이나 테스트 슈트를 쉽게 재사용할수 있다. 모든것을 고려해서 당신이 EJB를 사용하지 않는다면 로컬 SessionFactory 셋팅과 Spring의 HibernateTransactionManager 나 JtaTransactionManager에 충실하라. 당신은 어떠한 컨테이너 배치의 귀찮음 없이 적당한 트랜잭션적인 JVM레벨의 캐싱과 분산 트랜잭션을 포함한 모든 이득을 가질것이다. JCA연결자를 통한 Hibernate SessionFactory의 JNDI등록은 EJB를 사용하기 위해 단지 값만 추가한다.

11.2.9. 샘플들

Spring배포판의 Petclinic샘플은 DAO구현물과 Hibernate, JDBC, 그리고 아파치 OJB를 위한 애플리케이션 컨텍스트 설정의 대안을 제공한다. Petclinic은 Spring 웹 애플리케이션내 Hibernate의 사용을 묘사하는 샘플 애플리케이션처럼 제공한다. 이것은 다른 트랜잭션 전략으로 선언적인 트랜잭션 구분에 영향을 끼친다.

11.3. JDO

ToDo

11.4. iBATIS

org.springframework.orm.ibatis패키지를 통해 Spring은 iBATIS SqlMaps 1.3.x 과 2.0.x을 지원한다. iBATIS지원은 Hibernate처럼 템플릿 스타일 프로그래밍을 지원하는 면에서 Hibernate지원과 많은 공통점을 가진다. iBATIS지원은 Spring의 예외구조와 함께 작동하고 당신은 Spring이 가지는 모든 IoC특징을 즐기자.

11.4.1. 1.3.x and 2.0 사이의 개요와 차이점

Spring은 iBATIS SqlMaps 1.3 과 2.0 모두를 지원한다. 첫번째 둘 사이의 차이점을 보자.

xml설정 파일이 노드와 속성명에서 조금 변경되었다. 또한 당신이 확장할 필요가 있는 Spring클래스는 몇몇 메소드 명처럼 다르다.

Table 11.1. 1.3 과 2.0을 위한 iBATIS SqlMaps 지원 클래스

특징	1.3.x	2.0
SqlMap의 생성	SqlMapFactoryBean	SqlMapClientFactoryBean
템플릿 스타일의 헬퍼 클래스	SqlMapTemplate	SqlMapClientTemplate
MappedStatement를 사용하기 콜백	SqlMapCallback	SqlMapClientCallback
DAO를 위한 수퍼 클래스	SqlMapDaoSupport	SqlMapClientDaoSupport

11.4.2. iBATIS 1.3.x

11.4.2.1. SqlMap을 셋업하기

iBATIS SQLMaps를 사용하는 것은 statement와 result map들을 포함하는 SQLMaps설정파일을 생성하는것을 포함한다. Spring은 SqlMapFactoryBean을 사용하여 이것들을 로드하는것을 처리한다.

```
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;
        }
}
```

우리가 이 클래스를 맵핑하길 원한다고 가정하자. 우리는 다음의 SQLMaps를 생성한다. 쿼리를 사용하여 우리는 그들의 이메일 주소를 통해 나중에 사용자를 가져올수 있다. 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>
```

Sql Map을 정의한 후에 우리는 iBATIS를 위한 설정파일(sqlmap-config.xml)을 생성해야만 한다.

```
<sql-map-config>

        <sql-map resource="example/Account.xml"/>

</sql-map-config>
```

iBATIS는 클래스패스로 부터 자원을 로드한다. 그래서 클래스패스 어딘가에 Account.xml파일을 추가하라.

Spring을 사용할때 우리는 SqlMapFactoryBean을 사용해서 SQLMaps를 매우 쉽게 셋업할수 있다.

```
<bean id="sqlMap" class="org.springframework.orm.ibatis.SqlMapFactoryBean">
        <property name="configLocation"><value>WEB-INF/sqlmap-config.xml</value></property>
</bean>
```

11.4.2.2. SqlMapDaoSupport 사용하기

SqlMapDaoSupport클래스는 HibernateDaoSupport과 JdbcDaoSupport타입과 유사한 지원 클래스를 제공한다. DAO를 구현하자.

```
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);
        }
}
```

당신이 볼수 있는것처럼 우리는 쿼리를 수행하기 위해 SqlMapTemplate을 사용한다. Spring은 SqlMapFactoryBean을 사용하기 위해 SQLMap을 초기화하고 다음처럼 SqlMapAccountDao를 셋업할때 당신은 다음처럼 셋팅한다.

```
<!-- 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>
```

11.4.2.3. 트랜잭션 관리

iBATIS를 사용하는 애플리케이션을 위해 선언적인 트랜잭션 관리를 추가하는 것은 쉽다. 당신이 해야할 필요가 있는 한가지는 애플리케이션 컨텍스트에 트랜잭션 관리자를 추가하고 TransactionProxyFactoryBean예제를 위해 선언적으로 당신의 트랜잭션 경계를 셋팅하는 것이다. 이것에 대한 좀더 다양한 정보는 Chapter 7, 트랜잭션 관리에서 찾을수 있다.

TODO elaborate!

11.4.3. iBATIS 2

11.4.3.1. SqlMap 셋업하기

우리는 iBATIS 2를 사용해서 앞의 Account를 맵핑하기를 원한다면 우리는 다음의 SQLMaps Account.xml을 생성할 필요가 있다.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC
  "-//iBATIS.com//DTD SQL Map 2.0//EN"
  "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="Account">

        <resultMap id="result" class="examples.Account">
                <result property="name" column="NAME" columnIndex="1"/>
                <result property="email" column="EMAIL" columnIndex="2"/>
        </resultMap>

        <select id="getAccountByEmail" resultMap="result">
                select
                        ACCOUNT.NAME,
                        ACCOUNT.EMAIL
                from ACCOUNT
                where ACCOUNT.EMAIL = #value#
        </select>

        <insert id="insertAccount">
                insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
        </insert>

</sqlMap>
```

iBATIS 2를 위한 설정파일(sqlmap-config.xml)은 극히 적은 부분만 바꾼다.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig PUBLIC
  "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
  "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
```

```
<sqlMapConfig>

        <sqlMap resource="example/Account.xml"/>

</sqlMapConfig>
```

iBATIS는 클래스패스로부터 자원을 로드한다는것을 기억하라. 그래서 클래스패스 어딘가에 Account.xml파일을 추가하는것을 확인하라.

We can use the SqlMapClientFactoryBean in the Spring application context :

```
<bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocation"><value>WEB-INF/sqlmap-config.xml</value></property>
</bean>
```

11.4.3.2. SqlMapClientDaoSupport 사용하기

SqlMapClientDaoSupport클래스는 SqlMapDaoSupport와 유사한 지원 클래스를 제공한다. . 우리는 우리의 DAO를 구현하기 위해 이것을 확장한다.

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {

        public Account getAccount(String email) throws DataAccessException {
                Account acc = new Account();
                acc.setEmail();
                return (Account)getSqlMapClientTemplate().queryForObject("getAccountByEmail", email);
        }

        public void insertAccount(Account account) throws DataAccessException {
                getSqlMapClientTemplate().update("insertAccount", account);
        }
}
```

DAO에서 우리는 애플리케이션 컨텍스트내에서 SqlMapAccountDao를 셋업한후에 쿼리를 수행하기 위해 SqlMapClientTemplate를 사용한다.

```
<!-- 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="sqlMapClient"><ref local="sqlMapClient"/></property>
</bean>
```

Chapter 12. 웹 MVC framework

12.1. 웹 MVC framework 소개

Spring의 웹 MVC framework는 설정가능한 핸들러 맵핑들, view분석, 로케일과 파일 업로드를 위한 지원같은 테마분석과 함께 핸들러에 요청을 할당하는 DispatcherServlet을 기반으로 디자인되었다. 디폴트 핸들러는 ModelAndView handleRequest(request,response)메소드를 제공하는 매우 간단한 컨트롤러 인터페이스이다. 이것은 이미 애플리케이션 컨트롤러를 위해 사용될수 있지만 AbstractController, AbstractCommandController그리고 SimpleFormController같은 것들을 포함한 구현 구조를 포함한것을 더 선호할것이다. 애플리케이션 컨트롤러들은 전형적으로 이것들의 하위 클래스가 된다. 당신이 선호하는 빈 클래스를 선택하도록 하라. 만약에 당신이 form을 가지지 않는다면 당신은 FormController가 필요없다. 이것이 Struts와 가장 큰 차이점이다.

당신은 커맨드나 form객체와 같은 어떤 객체를 사용할 수 있다. 여기엔 인터페이스를 구현하거나 base클래스로부터 가져올 필요가 없다. Spring의 데이터 바인딩은 매우 유연성이 좋다. 예를 들면 이것은 시스템 에러가 아닌 애플리케이션에 의해서 평가될수 있는 유효성 에러와 같은 타입 불일치를 처리할수 있다. 그래서 당신은 당신의 폼객체내에서 유효하지 않은 서브밋을 다룰수 있게 되거나 문자열 프라퍼티로 형변환하기 위해 문자열같은 비지니스 객체의 프라퍼티를 반복시킬 필요가 없다. 대신 이것은 종종 당신의 비지니스 객체에 직접적으로 바인드 하는것이 더 바람직하다. 이것은 액션의 모든 타입을 위한 Action 과 ActionForm처럼 요구되는 기본 클래스가 도처에 빌드되는 Struts와의 가장 큰 차이점이다.

WebWork와 비교해서 Spring은 좀더 차별적인 객체 역할을 가진다. 이것은 컨트롤러, 옵션적인 커맨드나 폼 객체, 그리고 view에 전달되는 모델의 개념을 지원한다. 모델은 커맨드나 폼객체를 일반적으로 포함하지만 또한 임의의 참조 데이터도 포함한다. 대신에 WebWork 액션은 그러한 모든 역할을 하나의 객체로 조합한다. WebWork는 당신에게 당신의 폼 일부처럼 존재하는 비지니스 객체를 사용하도록 하지만 그것들을 만듬으로써 개별적인 Action클래스의 빈 프라퍼티가 된다. 마지막으로 요청을 다루는 같음 Action인스턴스는 view안의 평가와 폼 활성화를 위해서 사용된다. 게다가 참조 데이터는 Action의 빈 프라퍼티처럼 모델화 될 필요가 있다. 하나의 객체를 위해 어쩌면 너무 많은 역할이 있다.

Spring의 view분석은 매우 유연하다. 컨트롤러 구현물은 ModelAndView처럼 null을 반환하는 응답에 직접적으로 view를 작성할수 있다. 일반적인 경우에 ModelAndView인스턴스는 view이름과 빈 이름과 관련 객체(참조 데이터를 포함하는 커맨드또는 폼 같은)를 포함하는 모델 Map으루 구성된다. View이름 분석은 빈 이름, 프라퍼티 파일, 당신 자신의 ViewResolver구현물을 통해 쉽게 설정가능한 사항이다. 추상 모델 Map은 어떠한 귀찮은 작업 없이 view기술의 완벽한 추상화를 허락한다. 어떠한 표현자(renderer)는 JSP, Velocity 또는 다른 어떠한 표현 기술이든지 직접적으로 통합될수 있다. 모델 Map은 간단하게 JSP요청 속성또는 Velocity템플릿 모델과 같은 선호하는 형태로 변형된다.

12.1.1. 다른 MVC구현물의 플러그인 가능성

몇몇 프로젝트가 다른 MVC구현물을 사용하는것을 선호하는데는 여러가지 이유가 있다. 많은 팀은 업무적인 능력과 도구로 그들의 존재하는 투자물에 영향을 끼치도록 기대한다. 추가적으로 Struts프레임워크를 위해 유효한 지식과 경험의 많은 부분이 있다. 게다가 당신이 Struts의 구조적인 돌풍과 함께 한다면 이것은 웹 레이어를 위한 가치있는 선택이 될수 있다. 같은 것은 WebWork와 다른 웹 MVC프레임워크에 적용한다.

만약 당신이 Spring의 웹 MVC를 사용하길 원하지 않지만 Spring이 제공하는 다른 솔루션에 영향을 끼치는 경향이라면 당신은 당신이 선택한 웹 MVC프레임워크와 Spring을 쉽게 통합할수 있다. 이것의

ContextLoaderListener을 통해 Spring 루트 애플리케이션 컨텍스트를 간단하게 시작하고 Struts또는 WebWork액션내로 부터 이것의 ServletContext속성을 통해 접근한다. 어떠한 "plugins"도 포함되지 않았고 전용 통합이 필요하지 않다는 것에 주의하라. 웹 레이어의 관점에서 당신은 라이브러리처럼 간단하게 항목점(entry point)처럼 루트 애플리케이션 컨텍스트 인스턴스와 함께 Spring을 사용할 것이다.

당신의 등록된 모든 빈즈와 Spring의 모든 서비스는 Spring의 웹 MVC이 없더라도 쉽게 될수 있다. Spring은 이러한 시나리오로 Struts나 WebWork와 경쟁하지 않는다. 이것은 빈 설정으로 부터 데이터 접근과 트랜잭션 관리에서 순수한 웹 MVC프레임워크가 하지 않는 많은 면을 담당한다. 그래서 만약 당신이 (예를 들어 JDBC나 Hibernate로 트랜잭션 추상화) 사용하길 원한다면 Spring미들티어 그리고/또는 데이터 접근 티어를 사용해서 당신의 애플리케이션을 가치있게 할수 있다.

12.1.2. Spring MVC의 특징

Spring의 웹 모듈은 다음을 포함하는 유일한 웹 지원의 가치를 제공한다.

- ☒ 역할의 분명한 분리 - 컨트롤러, 유효성 체커, 커맨드 객체, 폼 객체, 모델 객체, DispatcherServlet, 핸들러 맵핑, view분석자, 등등. 각각의 역할은 객체를 특수화된 객체에 의해 충족될수 있다.
- ☒ 프레임워크와 자바빈즈같은 애플리케이션 클래스의 강력하고 일관적인 설정, 웹 컨트롤러에서 비지니스 객체와 유효성 체커와 같은 컨텍스트를 통한 쉬운 참조를 포함한다.
- ☒ 적합성, 침범적이지 않은, 모든것을 위해 하나의 컨트롤러로 부터 유도되는 대신에 주어진 시나리오를 위해 필요한(plain, 커맨드, 폼, 마법사, 다중 액션, 또는 사용자지정의 것) 컨트롤러 하위 클래스가 무엇이든지 사용하라.
- ☒ 재사용가능한 비지니스 코드 - 중복을 위해 필요한 것이 없다. 당신은 특정 프레임워크 기본 클래스를 확장하기 위해 그것들을 반영하는 대신에 커맨드 폼객체처럼 존재하는 비지니스 객체를 사용할수 있다.
- ☒ 사용자 지정 가능한 바인딩과 유효성 체크 - 수동 파싱과 비지니스 객체로 변환하는 오직 문자열만을 사용하는 폼 객체 대신에 잘못된 값, 지역화된 날짜 그리고 숫자 바인딩, 등등 을 유지하는 애플리케이션 레벨의 유효성 에러처럼 타입 부적합
- ☒ 사용자 지정가능한 핸들러 맵핑과 view분석 - 핸들러 맵핑과 view분석 전략은 간단한 URL기반의 설정에서 정교한, 목적이 내포된 분석 전략까지 범위에 둔다. 이것은 특정 기술을 위임하는 몇몇 웹 MVC프레임워크 보다 좀더 유연하다.
- ☒ 유연한 모델 이전 - 모델은 어떠한 view기술과 함께 쉬운 통합을 지원하는 이름/값 Map을 통해 이전한다.
- ☒ 사용자 지정 가능한 로케일과 테마 분석, Spring 태그 라이브러리를 사용하거나 사용하지 않은 JSP를 위한 지원, JSTL을 위한 지원, 추가적인 연결자를 위한 필요성없이 Velocity를 위한 지원
- ☒ 어떠한 가격적인 면에서 HTML생성을 피하는 간단하지만 강력한 태그 라이브러리, 마크업의 개념에서 최대한의 유연성을 허락한다.

12.2. DispatcherServlet

Spring의 웹 MVC프레임워크는 많은 다른 웹 MVC프레임워크와 비슷하고 요청 기반의 웹 MVC프레임워크이고 컨트롤러에 요청을 할당하는 서블릿에 의해 디자인되었고 웹 애플리케이션의 배치를 용이하게 하는 다른 기능을 제공한다. Spring의 DispatcherServlet은 어쨌든 그것보다 더 많은 기능을 수행한다. 이것은 Spring ApplicationContext와 완벽하게 통합되고 당신이 Spring이 가지는 다른 기능을 사용하도록 허락한다.

흔한 서블릿처럼 DispatcherServlet은 당신 웹 애플리케이션의 web.xml내에 선언된다. 당신이 다루는 DispatcherServlet을 원하는 요청은 web.xml파일내 URL맵핑을 사용해서 맵핑될것이다.

```
<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>
```

위 예제에서 .form으로 끝나는 모든 요청은 DispatcherServlet에 의해 다루어질것이다. DispatcherServlet은 지금 설정될 필요가 있다. Section 3.11, “ApplicationContext에 대한 소개” 에서 설명되는 것처럼, Spring내 ApplicationContexts는 범위화 될수 있다. 웹 MVC프레임워크내에서 각각의 DispatcherServlet은 DispatcherServlet설정 빈즈를 포함하는 자신만의 WebApplicationContext를 가진다. DispatcherServlet에 의해 사용되는 디폴트 BeanFactory는 XmlBeanFactory이고 DispatcherServlet은 당신의 웹 애플리케이션 WEB-INF디렉토리내 [servlet-name]-servlet.xml라고 불리는 파일을 룩업하여초기화 될것이다. DispatcherServlet에 의해 사용되는 디폴트 값은 서블릿 초기화 파라미터를 사용함으로써 변경될수 있다. (좀더 많은 정보를 위해서 아래를 보라.)

WebApplicationContext는 웹 애플리케이션을 위해 필요한 몇몇 추가적인 특징을 가지는 보통의 ApplicationContext이다. 이것은 테마(Section 12.7, “테마(themes) 사용하기” 를 보라)를 분석하는 기능면에서 대개의 ApplicationContext와 다르고 관련된 서블릿이 무엇인지 (ServletContext로의 링크를 가짐으로써) 안다. WebApplicationContext은 ServletContext내에 바운드 되고 당신이 언제나 WebApplicationContext을 룩업할수 있는 RequestContextUtils을 사용하는것이 이 경우에 필요하다.

Spring DispatcherServlet은 요청을 처리하고 선호하는 view들을 표현할수 있도록 하기 위해 이것을 사용하는 두개의 특별한 빈즈를 가진다. Spring프레임워크에 포함되고 WebApplicationContext내에 설정될수 있는 그러한 빈즈는 꼭 설정될 다른 빈즈들과 같다. 그러한 각각의 빈즈들은 밑에서 좀더 상세하게 설명된다. 지금 우리는 그것들을 설명할것이다. 그것들이 존재하고 DispatcherServlet에 대해서 계속적으로 얘기할수 있도록 하자. 대부분의 빈즈를 위해 디폴트는 당신이 그것들을 설정할 걱정을 하지 않도록 제공된다.

Table 12.1. WebApplicationContext내 특별한 빈즈들

표현	설명
핸들러 맵핑	(Section 12.4, “Handler mappings”)은 그것들이 어떠한 기준(예를 들면 컨트롤러와 함께 명시된 URL에 적합할때)에 적합할때 수행되어야 할 선처리자, 후처리자 그리고 컨트롤러의 리스트이다.
컨트롤러	(Section 12.3, “컨트롤러”)는 MVC 3가지 요소의 부분처럼 실질적인 기능(또는 적어도 기능에 접근하는)을 제공하는 빈즈이다.
view 분석자	(Section 12.5, “view와 view결정하기”)는 DispatcherServlet에 의해 사용되는 view를 위한 view이름을 분석하는 능력
로케일 분석자	(Section 12.6, “로케일 사용하기.”)는 국제화된 view를 제공할수 있도록 하기 위해 클라이언트가 사용하는 로케일을 분석하는 능력
테마 분석자	(Section 12.7, “테마(themes) 사용하기”)는 당신의 웹 애플리케이션이 사용할수 있는 테마(예를 들면 개인화된 레이아웃을 제공하기 위한)를 분석하는

표현	설명
	능력.
멀티파트 분석자	(Section 12.8, “Spring의 multipart (파일업로드) 지원”)는 HTML폼으로 부터 파일 업로드 처리를 위한 기능을 제공한다.
핸들러 예외 분석자	(Section 12.9, “예외 다루기”)는 view에 예외를 맵핑하거나 좀더 복잡한 예외 핸들링 코드를 구현하는 기능을 제공한다.

DispatcherServlet이 사용하기 위해 셋업되고 특정 DispatcherServlet을 위해 요청이 접수되면 이것을 처리하도록 시작된다. 밑에서 서술된 리스트는 DispatcherServlet에 의해 핸들링 된다면 요청을 완벽하게 처리한다.

1. WebApplicationContext는 사용하기 위한 프로세스에서 컨트롤러와 다른 요소를 위해 순서대로 속성처럼 요청을 찾고 바운드 한다. 이것은 DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE키 하위 디폴트에 의해 바운드된다.
2. 로케일 분석자는 요청을 처리할때(view를 표현하고, 데이터를 준비하는 등등) 사용하기 위한 로케일을 프로세스가 분석하도록 요청에 바운드된다. 만약 당신이 분석자를 사용하지 않는다면 이것은 어떠한 영향도 끼치지 않는다. 그래서 당신이 로케일 분석이 필요하지 않다면 그것을 사용하지 않아도 된다.
3. 테마 분석자는 view가 사용하기 위한 테마가 무엇인지 결정하는 요청에 바운드된다. 테마 분석자는 사용하지 않는다면 어떠한 영향도 끼치지 않는다. 그래서 당신이 테마를 사용할 필요가 없다면 당신은 그것을 무시할수 있다.
4. 만약 멀티파트 분석자가 명시되었다면 요청은 멀티파트를 위해서 그리고 그들이 발견된다면 조사될것이다. 이것은 프로세스내에서 다른 요소에 의해 좀더 처리되기 위해 MultipartHttpServletRequest내에 포장된다. (멀티파트 핸들링에 대해서 더 많은 정보를 위해서 Section 12.8.2, “MultipartResolver 사용하기” 를 보라.).
5. 선호하는 핸들러는 검색되기 위한것이다. 만약 핸들러가 찾아졌다면 핸들러(선처리자, 후처리자, 컨트롤러)가 속한 수행 묶음이 모델을 준비하기 위해 수행될것이다.
6. 모델이 반환된다면 WebApplicationContext에서 설정된 view분석자를 사용해서 view는 표현된다. 만약 어떠한 모델도 반환되지 않는다면(선 또는 후처리자가 요청을 가로챌수 있는, 예를 들면 보안적인 이유로) 이미 처리된 요청이후 표현되는 view는 없다.

요청 처리도중 던져질수 있는 예외는 WebApplicationContext내 선언된 어떠한 핸들러예외 분석자에 의해 처리된다. 이러한 예외 분석자를 사용함으로써 당신은 이러한 예외가 던져지는 경우 사용자 정의 행위를 정의할수 있다.

Spring DispatcherServlet은 역시 서블릿 API에 의해 특성화된 last-modification-date를 반환하기 위한 지원을 가진다. 특정 요청을 위해 가장 마지막에 변경된 날짜를 알아내는 프로세스는 매우 간단하다. DispatcherServlet은 먼저 선호하는 핸들러 맵핑을 룩업할것이고 핸들러가 LastModified 인터페이스를 구현한것을 찾을지 테스트할것이다. 만약 찾게된다면 long getLastModified(request)이 클라이언트에 반환된다.

당신은 web.xml파일이나 서블릿 초기화 파라미터에 컨텍스트 파라미터를 추가함으로써 Spring의 DispatcherServlet을 사용자 정의화 할수 있다. 가능한 사항을 밑에서 보여준다.

Table 12.2. DispatcherServlet 초기화 파라미터

파라미터	설명
contextClass	서블릿에 의해 사용되는 컨텍스트를 초기화하기 위해 사용될 WebApplicationContext을 구현하는 클래스. 만약 이 파라미터가 명시되지 않는다면 XmlWebApplicationContext이 사용될 것이다.
contextConfigLocation	발견될수 있는 컨텍스트의 위치를 표시하기 위한 컨텍스트 인스턴스(contextClass에 의해 정의되는)로 전달되는 문자열. 이 문자열은 잠재적으로 다중(다중 컨텍스트의 경우 두개로 명시된 빈의 경우 후자가 상위 서열을 가진다.) 컨텍스트를 지원하기 위해 다중(콤마를 분리자로 사용하여) 문자열로 나누어진다.
namespace	WebApplicationContext의 명명공간(namespace). 디폴트 값은 [server-name]-servlet이다.

12.3. 컨트롤러

컨트롤러의 개념은 MVC디자인 패턴의 일부이다. 컨트롤러는 애플리케이션 행위를 명시하거나 적어도 애플리케이션 행위에 대한 접근을 제공한다. 컨트롤러는 사용자 입력을 해석하고 사용자 입력을 view에 의해 사용자에게 표현될 실용적인 모델로 변형된다. Spring은 생성될 굉장히 넓은 범위의 다양한 종류의 컨트롤러를 가능하게 하는 추상화된 방법으로 컨트롤러의 개념을 구현했다. Spring은 폼 컨트롤러, 커맨드 컨트롤러, 마법사 스타일의 로직을 수행하는 컨트롤러 그리고 더 다양한 방법을 포함한다.

Spring의 컨트롤러 구조의 기본은 밑에 있는 org.springframework.web.servlet.mvc.Controller인터페이스이다.

```
public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the DispatcherServlet
     * will render.
     */
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception;
}
```

당신이 볼수 있는것처럼 Controller인터페이스는 요청을 처리하고 선호하는 모델및 view로 반환하는 능력을 가지는 하나의 메소드를 요구한다. 이러한 3개의 개념은 Spring MVC구현물(-ModelAndView 와 Controller)을 위한 기본이다. Controller인터페이스가 추상적인 동안 Spring은 당신이 필요한 많은 기능을 포함한 많은 컨트롤러를 제공한다. Controller인터페이스는 모든 컨트롤러의 요구되는 가장 공통적인 기능(-요청을 처리하고 모델및 view를 반환하는)을 명시한다.

12.3.1. AbstractController 와 WebContentGenerator

물론 컨트롤러 인터페이스로는 충분하지 않다. 기본적인 구조를 제공하기 위해서 Spring의 모든 컨트롤러는 캐시지원과 예를 들면 mimetype셋팅을 제공하는 클래스인 AbstractController로부터 상속되었다.

Table 12.3. AbstractController에 의해 제공되는 특징

특징	설명
supportedMethods	이 컨트롤러가 받아들이는 방식이 무엇인지 표시. 언제나 이것은 GET 과 POST로 셋팅되지만 당신은 지원하길 원하는 방식를 반영하기 위해 변경할수 있다. 만약 요청이 컨트롤러에 의해 지원되지 않는 방식으로 들어왔다면 클라이언트는 이것을 정보화(ServletException)을 이용해서) 할것이다.
requiresSession	컨트롤러가 작업을 하기위해 세션을 필요로 하는지 하지 않는지를 표시. 이 기능은 모든 컨트롤러에 의해 제공된다. 만일 컨트롤러가 요청을 받을때 세션이 존재하지 않는다면 사용자는 ServletException을 사용해서 정보화한다.
synchronizeSession	당신이 사용자의 세션에서 동기화되기 위해 이 컨트롤러에 의해 핸들링하길 원한다면 이것을 사용하라. 좀더 상세화되기 위해 컨트롤러를 확장하는 것은 당신이 이 변수를 정의한다면 동기화되기 위해 handleRequestInternal메소드를 오버라이드 할것이다.
cacheSeconds	당신이 HTTP응답에서 캐시를 직업적으로 생성할 컨트롤러를 원한다면 여기에 양수값을 명시하라. 디폴트는 직접적으로 캐시를 포함하지 않을것이기 때문에 -1로 셋팅된다.
useExpiresHeader	HTTP 1.0호환 "Expires"헤더를 명시하기 위해 당신의 컨트롤러를 부분적으로 개량하라. 디폴트는 true라는 값으로 셋팅함으로써 당신은 이것을 변경하지 않을것이다.
useCacheHeader	HTTP 1.1호환 "Cache-Control"헤더를 명시하기 위해 당신의 컨트롤러를 부분적으로 개량하라. 디폴트는 true라는 값으로 당신은 이것을 변경하지 않을것이다.

마지막 두개의 프라퍼티는 AbstractController의 수퍼클래스이지만 완성도를 위해 여기에 포함된 WebContentGenerator의 실질적인 부분이다.

당신의 컨트롤러(당신을 위해 작업을 이미 수행하는 많은 다른 컨트롤러때문에 추천되지는 않는)를 위한 기본 클래스처럼 AbstractController를 사용할때 당신은 단지 당신의 로직을 구현하고 ModelAndView객체를 반환하는 handleRequestInternal(HttpServletRequest, HttpServletResponse)메소드만 오버라이드 해야 한다. 이것은 웹 애플리케이션 컨텍스트내 클래스와 선언을 구성하는 짧은 예제이다.

```
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>
```

위 클래스와 웹 애플리케이션 컨텍스트내 선언은 매우 간단히 작동중인 컨트롤러를 얻기 위해 당신이 핸들러 맵핑(Section 12.4, “Handler mappings” 을 보라)을 셋업할 필요가 있는 모든것이다. 이 컨트롤러는 다시 체크하기 전에 2분동안 캐시하는것을 클라이언트에게 알리는 직접적인 캐시를

생성할것이다. 이 컨트롤러는 index(view에 대한 좀더 상세한 정보를 위해서는 Section 12.5, “view와 view결정하기” 을 보라.)라는 이름의 직접 작성된(흠.. 별로 좋지 않은) 코드의 view를 반환할것이다.

12.3.2. 간단한 다른 컨트롤러

비록 당신이 AbstractController을 확장할수 있다고 하더라도 Spring은 간단한 MVC애플리케이션내 공통적으로 사용될 기능을 제공하는 명확한 많은 구현물을 제공한다. ParameterizableViewController는 웹 애플리케이션 컨텍스트내 반환될 view이름을 명시할수 있다(하드코딩된 view이름은 필요없다.)는 사실을 제외하면 위 예제와 기본적으로 같다.

FileNameViewController는 URL을 조사하고 요청으로 부터 파일이름(http://www.springframework.org/index.html의 파일이름은 index이다.)을 가져온다. 그리고 view이름으로 그것을 사용한다. 그것을 위한 더이상의 것은 없다.

12.3.3. MultiActionController

Spring은 모두 기능적으로 그룹화되는 한개의 컨트롤러로 다중 액션을 합치도록 하는 다중액션(multi-action) 컨트롤러를 제공한다. 다중액션 컨트롤러는 별도의 패키지인 org.springframework.web.servlet.mvc.multiaction로 되어 있다. 그리고 메소드이름으로 요청을 맵핑하는 능력을 가지고 올바른 메소드 이름을 호출한다. 다중액션 컨트롤러를 사용하는 것은 하나의 컨트롤러로 많은 공통적인 기능을 가질때 특별히 편리하다. 하지만 컨트롤러에 대해 다중 엔트리 포인트를 가지길 원한다. 예를 들면 행위의 일부를 고치기 위해

Table 12.4. MultiActionController에 의해 제공되는 특징

특징	설명
delegate	MultiActionController을 위한 두가지 사용 시나리오가 있다. 당신이 MultiActionController를 세분화하고 하위 클래스에서 MethodNameResolver에 의해 분석될 메소드를 정의하거나 분석자(Resolver)에 의해 분석될 메소드가 호출되는 위임(delegate)객체를 명시한다. 만약 당신이 이 시나리오를 선택한다면 당신은 협력자(collaborator)처럼 이 설정 파라미터를 사용해서 위임을 정의할 것이다.
methodNameResolver	아무튼 MultiActionController는 들어온 요청에 기반한 호출할 메소드를 분석할 필요가 있을것이다. 당신은 설정 파라미터를 사용하는 능력이 있는 분석자(resolver)를 명시할수 있다.

다중액션 컨트롤러를 위해 명시한 메소드는 다음의 시그너처에 따를 필요가 있다.

```
// actionName can be replaced by any methodname
ModelAndView actionName(HttpServletRequest, HttpServletResponse);
```

메소드 오버로딩은 MultiActionController을 혼동시키기 때문에 허락되지 않는다. 게다가 당신은 명시한 메소드에 의해 던져질 예외 핸들링 능력을 가지는 exception handlers를 명시할수 있다. 예외 핸들러 메소드는 ModelAndView객체를 반환할 필요가 있다. 다른 액션 메소드도 다음의 시그너처에 따를 필요가 있다.

```
// anyMeaningfulName can be replaced by any methodname
```

```
ModelAndView anyMeaningfulName(HttpServletRequest, HttpServletResponse, ExceptionClass);
```

ExceptionClass는 java.lang.Exception 이나 java.lang.RuntimeException의 하위 클래스만큼 어떠한 예외도 될수 있다.

MethodNameResolver는 들어온 요청에 기반하여 메소드 이름을 분석하기 위해 추측된다. 당신의 배치(disposal)에 3개의 분석자가 있지만 물론 당신은 원한다면 당신 스스로 그것들보다 좀더 다양하게 구현할수 있을것이다.

☒ ParameterMethodNameResolver - 요청 파라미터를 분석하고 메소드이름(http://www.sf.net/index.view?testParam=testIt는 호출될 testIt(HttpServletRequest, HttpServletResponse) 메소드로 결과를 낼것이다.)처럼 그것을 사용하는 능력. paramName 설정 파라미터는 조사된 파라미터를 명시한다.

☒ InternalPathMethodNameResolver - 경로로 부터 파일이름을 가져오고 메소드 이름처럼 그것을 사용한다. (http://www.sf.net/testing.view는 호출될 testing(HttpServletRequest, HttpServletResponse)메소드내 결과를 낼것이다.)

☒ PropertiesMethodNameResolver - 요청 URL을 메소드이름에 맵핑되는 사용자 정의 프라퍼티 객체를 사용한다. 프라퍼티가 /index/welcome.html=doIt을 포함하고 /index/welcome.html로 요청이 들어올때 doIt(HttpServletRequest, HttpServletResponse)메소드는 호출된다. 이 메소드 이름 분석자는 PathMatcher과 함께 작동한다. (???를 보라.) 프라퍼티가 /**/welcom?.html을 포함한다면 이것또한 작동할것이다.

여기에 두개의 예제가 있다. 첫번째 예제는 ParameterMethodNameResolver를 보여주고 포함된 파라미터 메소드를 가지는 url로 요청을 받을 프라퍼티를 위임한다. 그리고 retrieveIndex에 셋팅한다.

```
<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"/></property>
</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()));
    }
}
```

위에서 보여진 위임을 사용할때 우리는 명시한 메소드에 두개의 URL로 매치시키기 위한 PropertiesMethodNameResolver을 사용할수 있다.

```
<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"/></property>
</bean>
```

12.3.4. CommandControllers

Spring의 CommandControllers는 Spring MVC패키지의 필수적인 부분이다. 커맨드 컨트롤러는 데이터 객체와 상호작동하기 위한 방법을 제공한다. HttpServletRequest로 부터 명시된 데이터 객체를 파라미터를 동적으로 바인드한다. 그것들은 Struts의 ActionForm과 유사한 역할을 수행하지만 Spring내에서 당신의 데이터 객체는 프레임워크 특정적인 인터페이스를 구현할 필요는 없다. 당신이 그것들로 무엇을 할수 있는지 개략적으로 살펴보기 위해 사용가능한 커맨드 컨트롤러가 어떤것이 있는지 조사해보자.

☒ AbstractCommandController - 당신 자신의 컨트롤러를 생성하기 위해 당신이 사용할수 있는 커맨드 컨트롤러는 당신이 명시하는 데이터 객체로 요청 파라미터를 바인드하는 능력을 가진다. 이 클래스는 폼 기능을 제공하지 않는다. 이것은 어쨌든 유효성체크 기능을 제공하고 요청으로 부터 파라미터를 채우는 커맨드 객체가 무엇인지 컨트롤러내 명시하자.

☒ AbstractFormController - 추상 컨트롤러는 폼 서브밋 지원을 제공한다. 이 컨트롤러를 사용하면 당신은 폼을 만들고 당신이 컨트롤러내에서 가져온 커맨드 객체를 사용하여 그것들을 형상화 할수 있다. 사용자가 폼을 채우고 난 뒤 AbstractFormController는 필드를 바인드하고 유효성 체크를 하며 선호하는 액션을 가지기 위해 컨트롤러에 반환된 객체를 다룬다. 지원되는 기능 : 타당치 않은 폼 서브밋, 유효성체크, 그리고 일반적인 폼 워크플로우. 당신은 view가 폼 표현이나 성공을 위해 사용되는지 알아내기 위한 메소드를 구현한다. 만약 당신이 폼이 필요하다면 이 컨트롤러를 사용하라. 하지만 애플리케이션 컨텍스트내 사용자를 보여주기 위해 당신이 사용하는 view를 명시하길 원하지 않는다.

☒ SimpleFormController - 유사한 커맨드 객체를 가진 폼을 생성할때 명백한 FormController는 좀더 많은 지원을 제공한다. SimpleFormController는 당신이 커맨드 객체, 폼을 위한 view이름, 폼 서브밋이 성공했을때 당신이 사용자에게 보여주기를 원하는 페이지를 위한 view이름을 명시하도록 한다.

☒ AbstractWizardFormController - 클래스 이름이 제시하는것처럼 이것은 당신의 WizardController가 이것을 확장해야만 하는 추상 클래스이다. 이것은 당신이 validatePage(), processFinish 그리고 processCancel를 구현해야만 한다는 것을 의미한다.

당신은 아마 적어도 setPages() 와 setCommandName()를 호출하는 계약자(contractor)를 쓰길 원할것이다. 이 형태자(former)는 인자를 문자열 타입 배열현태로 가진다. 이 배열은 당신의 마법사를 이루는 view의 목록이다. 나중에 인자를 당신의 view내로부터 커맨드 객체를 참조하기 위해 사용될 문자열처럼 가진다.

AbstractFormController의 어떠한 인스턴스처럼 당신은 당신의 폼으로 부터 데이터를 활성화할 자바빈인 커맨드 객체를 사용하기 위해 요구된다. 당신은 커맨드 객체의 클래스를 가진 생성자로부터 setCommandClass()을 호출하거나 formBackingObject()메소드를 구현하는 두가지 방법중에 하나로 이것을 할수 있다.

AbstractWizardFormController는 오버라이드할 많은 수의 메소드를 가진다. 물론 당신이 가장 유용하다고 볼수 있는 것은 당신이 Map의 폼내 당신의 view로 모델 데이터를 전달하기 위해 사용할수 있는 referenceData; 만약 당신의 마법사가 순서대로 페이지를 변경하거나 동적으로페이지를 생략할 필요가 있다면 getTargetPage; 만약 당신이 내장된 바인딩과 유효성 체크 워크 플로우를 오버라이드하기를 원한다면 onBindAndValidate이다.

마지막으로 이것은 현재 페이지에 유효성 체크가 실패한다면 마법사내에서 뒤로나 앞으로 움직이는

것을 사용자에게 허락하는 getTargetPage로 부터 호출할수 있는 setAllowDirtyBack 과 setAllowDirtyForward를 가리킬 가치가 있다.

메소드의 모든 목록을 위해 AbstractWizardFormController를 위한 JavaDoc를 보라. Spring배포판내 포함된 jPetStore내 마법사의 예제가 구현(org.springframework.samples.jpetstore.web.spring.OrderFormController)되어 있다.

12.4. Handler mappings

핸들러 맵핑을 사용하면 당신은 선호하는 핸들러로 들어온 웹 요청을 맵핑할수 있다. 여기에 당신이 특별히 사용할수 있는 몇몇 핸들러 맵핑이 있다. 예를 들면 SimpleUrlHandlerMapping 나 BeanNameUrlHandlerMapping이다. 하지만 먼저 HandlerMapping의 일반적인 개념을 알아보자.

기능적으로 기본 HandlerMapping은 들어온 요청에 매치하는 핸들러를 포함하고 요청에 적용되는 핸들러 입터셉터의 목록을 포함할수도 있는 HandlerExecutionChain의 전달을 제공한다. 요청이 들어왔을때 DispatcherServlet는 요청을 조사하도록 하는 핸들러 맵핑을 위해 이것을 다룰것이고 선호하는 HandlerExecutionChain을 생성할것이다. 그 다음 DispatcherServlet은 핸들러와 체인내 인터셉터를 수행할것이다.

임의로 인터셉터(실질적인 핸들러가 수행되기 전이나 후에 수행되는)를 포함할수 있는 설정가능한 핸들러 맵핑의 개념은 극히 강력하다. 많은 지원 기능은 사용자 정의 HandlerMapping에 내장될수 있다. 핸들러를 선택한 사용자 정의 핸들러 맵핑의 생각은 들어오는 요청의 URL에 기반할뿐 아니라 요청과 함께 속하는 세션의 특정 상태에도 기반을 둔다.

이 섹션은 Spring의 가장 공통적으로 사용되는 핸들러 맵핑 두가지를 서술한다. 그 둘은 AbstractHandlerMapping을 확장하고 다음의 프라퍼티 값을 공유한다.

- ☒ interceptors: 사용하기 위한 인터셉터의 목록. HandlerInterceptor는 Section 12.4.3, “HandlerInterceptors 추가하기” 에서 논의된다.
- ☒ defaultHandler: 핸들러 맵핑이 적합한 핸들러를 매치시키는 결과를 내지 못할때 사용하기 위한 디폴트 핸들러.
- ☒ order: order프라퍼티(org.springframework.core.Ordered인터페이스를 보라)의 값에 기반한다. Spring은 컨텍스트내에서 사용가능한 모든 핸들러 맵핑을 분류하고 첫번째 매치되는 핸들러를 적용한다.
- ☒ alwaysUseFullPath: 이 프라퍼티가 true로 셋팅된다면 Spring은 적당한 핸들러를 찾기 위해 현재 서블릿 컨텍스트내 완전한 경로(full path)를 사용할것이다. 만약 이 프라퍼티가 false(이 값이 디폴트 값이다.)로 셋팅된다면 현재 서블릿 맵핑내 경로가 사용될것이다. 예를 들면 서블릿이 /testing/*을 사용해서 맵핑되어 있고 alwaysUseFullPath프라퍼티가 true로 셋팅되어 있다면 /testing/viewPage.html이 사용될것이다. 만약 프라퍼티가 false라면 /viewPage.html가 사용될것이다.
- ☒ urlPathHelper: 이 프라퍼티를 사용하면 당신은 URL을 조사할때 사용되는 UrlPathHelper를 부분적으로 수정할수 있다. 일반적으로 당신은 디폴트 값을 변경하지 말아야 할것이다.
- ☒ urlDecode: 이 프라퍼티를 위한 디폴트 값은 false이다. HttpServletRequest는 번역(decode)되지 않은 요청 URL과 URI를 반환한다. 만약 HandlerMapping이 적당한 핸들러를 찾기 위해 그것들을 사용하기 전에 그것들이 번역되길 원한다면 이것을 true로 셋팅해야 한다.(이것은 JDK1.4를 요구한다는것에 주의하라.). 번역 방식은 요청에 의해 명시된 인코딩이나 디폴트인 ISO-8859-1 인코딩 스키마를 사용한다.
- ☒ lazyInitHandlers: singleton핸들러(프로토타입 핸들러는 언제나 늦은 초기화를 수행한다.)의 늦은(lazy)초기화를 허락한다. 디폴트 값은 false이다.

(주의: 마지막의 4개의 프라퍼티는 org.springframework.web.servlet.handler.AbstractUrlHandlerMapping의 하위클래스에서만 사용가능하다.).

12.4.1. BeanNameUrlHandlerMapping

매우 간단하지만 굉장히 강력한 핸들러 맵핑은 들어오는 HTTP요청을 웹 애플리케이션 컨텍스트내 명시된 빈즈의 이름으로 맵핑하는 BeanNameUrlHandlerMapping이다. 우리는 사용자가 계정을 추가하는것이 가능하길 원하고 우리는 벌써 적당한 FormController(커맨드와 FormControllers의 좀더 상세한 정보를 위해서 Section 12.3.4, “CommandControllers” 을 보라)과 폼을 표현하는 JSP view(또는 Velocity템플릿)가 제공된다고 얘기해보자. BeanNameUrlHandlerMapping을 사용할때 우리는 다음처럼 적당한 FormController를 위해 URL http://samples.com/editaccount.form을 HTTP요청에 맵핑할수 있다.

```
<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>
```

URL /editaccount.form을 위한 들어온 모든 요청은 위 소스 목록의 FormController에 의해 다루어질것이다. 물론 우리는 .form으로 끝나는 모든 요청을 통하기 위해 web.xml에 servlet-mapping을 정확하게 명시해야 한다.

```
<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>
```

주의: 만약 당신이 BeanNameUrlHandlerMapping을 사용하길 원한다면 당신은 웹 애플리케이션 컨텍스트내 이것을 반드시 명시할 필요가 없다(위에서 표시된것처럼). 디폴트에 의해 어떠한 핸들러 맵핑도 컨텍스트내에서 발견되지 않는다면 DispatcherServlet은 당신을 위해 BeanNameUrlHandlerMapping을 생성한다.

12.4.2. SimpleUrlHandlerMapping

한층 나아가서 그리고 좀더 강력한 핸들러 맵핑은 SimpleUrlHandlerMapping이다. 이 맵핑은 애플리케이션 컨텍스트내에서 설정가능하고 Ant스타일의 경로 매치 능력을 가진다.
(org.springframework.util.PathMatcher를 위한 JavaDoc를 보라.). 여기에 예제가 있다.

```
<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>
```

같은 디스패처(dispatcher) 서블릿에 의해 다루어지기 위해 .html과 .form으로 끝나는 모든 요청을 할당한다.

```
<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>
```

이 핸들러 맵핑은 어떤 디렉토리내 help.html을 UrlFilenameViewController(Section 12.3, “컨트롤러” 에서 찾을수 있는 컨트롤러에 대해서 추가적인)인 helpController로 요청을 보낸다. ex디렉토리내에서 view로 시작하고 .html로 끝나는 자원을 위한 요청은 someViewController로 경로가 지정될것이다. 둘 이상의 맵핑은 editAccountFormController을 위해 명시된다.

12.4.3. HandlerInterceptors 추가하기

Spring은 핸들러 맵핑 기법은 예를 들어 구매자를 체크하는 어떠한 요청을 위한 특정 기능을 적용하길 원할때 매우 유용할수있는 핸들러 인터셉터의 개념을 가진다.

핸들러 맵핑내 위치하는 인터셉터는 org.springframework.web.servlet패키지로부터 HandlerInterceptor를 구현해야만 한다. 이 인터페이스는 3개의 메소드를 선언한다. 하나는 실질적인 핸들러가 수행되기 전에 호출될것이고 하나는 핸들러가 수행된 후에 호출될것이다. 나머지 하나는 완전한 요청이 종료된후에 호출된다. 이 3가지 메소드는 선처리와 후처리의 모든 종류를 처리하기 위한 충분한 유연성을 제공할것이다.

preHandle 메소드는 bolean값을 반환한다. 당신은 작업을 중간에 종료하거나(break) 수행묶음(chain)의 처리를 계속하기 위해 이 메소드를 사용할수 있다. 이 메소드가 true를 반환할때 핸들러 수행묶음은 계속될것이다. false를 반환할때는 DispatcherServlet이 요청을 처리할 인터셉터(그리고 예를 들면 적당한 view를 표현하는)를 추정하고 수행묶음내 다른 인터셉터와 실질적인 핸들러를 계속적으로 수행하지 않는다.

아래의 예제는 모든 요청을 가로채고 오전 9시와 오후 6시가 아니라면 사용자를 특정 페이지로 경로를 변경시키는 인터셉터를 제공한다.

```
<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;
        }
    }
}
```

들어오는 어떠한 요청은 TimeBasedAccessInterceptor에 의해 차단당할것이고 현재 시각이 사무시간(office

hours)밖이라면 사용자는 정적 html파일로 전환될것이다. 다시 말하면 예를 들어 그는 사무시간(office hours)내에서만 웹사이트에 접근할수 있다.

당신이 볼수 있는 것처럼 Spring은 당신이 HandlerInterceptor를 확장하기 쉽도록 만드는 어댑터(adapter)를 가진다.

12.5. view와 view결정하기

웹 애플리케이션을 위한 모든 MVC프레임워크는 view를 결정하는 방법을 제공한다. Spring은 특정 view기술을 위해 당신이 기록할 필요없이 브라우저내에서 모델을 표현할수 있도록 만들어주는 view결정자(resolvers)를 제공한다. 특별히 Spring은 JSP, Velocity 템플릿, XSLT view를 사용가능하도록 한다. 예를 들면 Chapter 13, 통합 뷰 기술들는 다양한 view기술과의 통합에 대한 상세사항을 가진다.

Spring이 view를 다루는 방법을 위한 중요한 두개의 인터페이스는 ViewResolver 와 View이다. ViewResolver는 view이름과 실제 view사이의 맵핑을 제공한다. View인터페이스는 준비된 요청을 할당하고 요청을 view기술중 하나에게 처리하도록 넘겨버린다.

12.5.1. ViewResolvers

Section 12.3, “컨트롤러” 에서 논의된 것처럼 Spring 웹 MVC프레임워크내 모든 컨트롤러는 ModelAndView인스턴스를 반환한다. Spring내 view는 view이름에 의해 할당되고 view결정자에 의해 결정된다. Spring은 다수의 view결정자를 가진다. 우리는 그것들의 대부분의 목록을 볼것이며 두개의 예제를 제공한다.

Table 12.5. View 결정자(Resolvers)

ViewResolver	설명
AbstractCachingViewResolver	캐싱 view를 다루는 추상 view결정자. 종종 view를 사용되기 전에 준비작업이 필요하다. 이 view결정자를 확장하는 것은 view의 캐싱을 제공한다.
XmlViewResolver	Spring의 bean팩토리 처럼 같은 DTD를 가진 XML내 쓰여진 설정파일을 가져오는 ViewResolver의 구현물. 디폴트 설정 파일은 /WEB-INF/views.xml이다.
ResourceBundleViewResolver	번들 basename에 의해 명시된 ResourceBundle내 bean정의를 사용하는 ViewResolver의 구현물. 번들은 대개 클래스패스내 위치한 프라퍼티파일내 명시된다. 디폴트 파일명은 views.properties이다.
UrlBasedViewResolver	추가적인 맵핑 정의 없이 URL로 상징적인 view이름의 직접적인 결정을 허락하는 ViewResolver의 간단한 구현물. 이것은 당신의 상징적인 이름이 임의의 맵핑의 필요성이 없는 직접적인 방법으로 당신의 view자원의 이름을 대응한다면 적당하다.
InternalResourceViewResolver	InternalResourceView(예를 들면 서블릿과 JSP)와 JstlView, TilesView와 같은 하위 클래스를 지원하는 UrlBasedViewResolver의 편리한 하위 클래스. 이 결정자에 의해 생성되는 모든 view를 위한 view클래스는 setViewClass를 통해 정의될수 있다. 상세사항을 위해 UrlBasedViewResolver´s의 JavaDoc를 보라.

ViewResolver	설명
VelocityViewResolver / FreeMarkerViewResolver	직접적인 VelocityView(예를 들면 Velocity템플릿) 또는 FreeMarkerView와 그것들의 사용자 지정 하위 클래스를 지원하는 UrlBasedViewResolver의 편리한 하위 클래스.

예제처럼 view기술로 JSP를 사용할때 당신은 UrlBasedViewResolver을 사용할수 있다. 이 view결정자는 view이름을 URL로 번역하고 view를 표현하기 위해 요청을 RequestDispatcher로 보낸다.

```
<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>
```

view이름처럼 test를 반환할때 이 view결정자는 /WEB-INF/jsp/test.jsp로 요청을 보낼 RequestDispatcher로 요청을 보낼것이다.

웹 애플리케이션내 서로 다른 view기술을 혼합해서 사용할때 당신은 ResourceBundleViewResolver을 사용할수 있다.

```
<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>
```

ResourceBundleViewResolver는 basename에 의해 구별되는 ResourceBundle을 추정하고 각각의 view를 위해 이것은 결정하기 위한 제안된다. 이것은 view 클래스처럼 프라퍼티 [viewname].class의 값과 view url처럼 프라퍼티 [viewname].url의 값을 사용한다. 당신이 볼수 있는 것처럼 확장순서대로 프라퍼티 파일내 모든 view로부터 당신은 부모(parent) view를 구별할수 있다. 예들 들면 이 방법으로 당신은 디폴트 view클래스를 정의할수 있다.

캐싱쪽 노트(note) - 결정될수 있는 AbstractCachingViewResolver 캐시 view인스턴스의 하위클래스. 이것은 어떤 view기술을 사용할때 성능을 향상시킨다. cache프라퍼티를 false로 셋팅함으로써 캐시를 끌수도 있다. 게다가 당신이 수행시 어떤 view를 재생할수 있는 요구사항(예를 들면 Velocity템플릿이 변경될 때)을 가진다면 당신은 removeFromCache(String viewName, Locale loc)메소드를 사용할수 있다.

12.5.2. ViewResolvers 묶기(Chaining)

Spring은 하나의 view결정자보다 많은 수의 결정자를 지원한다. 이것은 당신에게 결정자를 묶는것을 가능하게 한다. 예를 들면 어떤 상황에서 특정 view를 오버라이드한다. view결정자를 묶는것은 당신의 애플리케이션 컨텍스트에 하나 이상의 결정자를 추가하는것처럼 상당히 일관적이다. 만약 필요하다면 order를 정의하기 위해 order프라퍼티를 셋팅하라. 더 큰 order프라퍼티는 나중에 view결정자가 묶음내 위치시킬것이다.

다음의 예제에서 view결정자의 묶음은 두개의 결정자인 InternalResourceViewResolver(두번째에 위치하고 묶음내 마지막 결정자)와 엑셀 view(InternalResourceViewResolver에 의해 지원되지 않는)를 정의하기 위한 XmlViewResolver로 이루어진다.

```
<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>
```

만약 특정 view결정자가 view결과를 내지 않을때 Spring은 다른 view결정자가 설정되었다면 보기 위해 컨텍스트를 조사할것이다. 만약 추가적인 view결정자가 있다면 이것은 그것들을 조사하기 위해 지속될것이다. 만약 그렇지 않다면 예외를 던질것이다.

당신은 염두해 두고 있는것을 유지해야한다. view결정자의 규칙은 view결정자가 찾을수 없는 view를 표시하기 위해 null을 반환할수 있다는 것을 말한다. 어쨋든 모든 view결정자기 이것을 하지는 않는다. 이것은 결정자가 view가 존재하는지 존재하지 않는지 검사할수 없을때와 같은 몇몇 경우에 야기된다. 예를 들면 InternalResourceViewResolver는 내부적으로 RequestDispatcher를 사용하고 디스패치(dispatching)는 JSP가 존재할때 설정하는 유일한 방법이다. 이것은 한번에 한하여 수행될수 있다. VelocityViewResolver와 몇몇 다른것들을 위해 같은것이 묶인다. 만약 당신이 존재하지 않는 view를 보고하지 않은 view결정자를 처리한다면 view결정자를 위한 JavaDoc를 체크하라. 이 결과처럼 마지막이 아닌 다른 어느위치내 묶음안에 InternalResourceViewResolver을 두는것은 InternalResourceViewResolver이 언제나 view를 반환하기 때문에 완전하게 조사되지 않는 묶음이라는 결과를 만들어낼것이다.

12.6. 로케일 사용하기.

Spring구조의 대부분은 Spring 웹 MVC프레임워크가 하는것처럼 국제화를 지원한다. DispatcherServlet은 당신에게 클라이언트 로케일을 사용하여 메시지를 자동적으로 결정하도록 한다. 이것은 LocaleResolver객체를 사용해서 수행한다.

요청이 들어올때 DispatcherServlet는 로케일 결정자를 찾고 로케일 결정자가 찾아진다면 로케일을 셋팅하기 위해 그 결정자를 사용한다. RequestContext.getLocale()메소드를 사용하여 당신은 로케일 결정자에 의해 결정된 로케일을 언제나 가져올수 있다.

자동적인 로케일 전환외에도 당신은 핸들러 맵핑에 인터셉터(핸들러 맵핑 인터셉터의 좀더 다양한 정보를 위해 Section 12.4.3, “HandlerInterceptors 추가하기” 를 보라)를 첨부할수 있다. 특정 상황하에 로케일을 변경하는 것은 요청의 파라미터에 기반한다.

로케일 결정자와 인터셉터는 org.springframework.web.servlet.i18n패키지내 모두 명시되고 일반적인 방법으로 당신의 애플리케이션 컨텍스트내 설정된다. 여기에 Spring에 포함된 로케일 결정자의 선택된 일부가 있다.

12.6.1. AcceptHeaderLocaleResolver

이 로케일 결정자는 클라이언트의 브라우저에 의해 보내어진 요청내 accept-language헤더를 조사한다. 언제나 이 헤더 필드는 클라이언트의 OS시스템의 로케일을 포함한다.

12.6.2. CookieLocaleResolver

이 로케일 결정자는 로케일이 정의되었다면 보기 위해 클라이언트내 존재하는 쿠키를 조사한다. 만약 쿠키가 존재한다면 그 특정 로케일을 사용한다. 로케일 결정자의 프라퍼티를 사용하여 당신은 최대생명(maximum age)만큼 쿠키의 이름을 명시할수 있다.

```
<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>
```

이것은 CookieLocaleResolver를 명시하는것의 예제이다.

Table 12.6. WebApplicationContext내 특별한 빈즈

프라퍼티	디폴트 값	설명
cookieName	classname + LOCALE	쿠키의 이름
cookieMaxAge	Integer.MAX_INT	쿠키가 클라이언트에 일관적으로 머무를 최대시간. 만약 -1이 정의된다면 쿠키는 저장되지 않는다. 이것은 단지 클라이언트가 브라우저는 닫을때 까지만 사용가능하다.
cookiePath	/	이 파라미터를 사용하여 당신은 당신 사이트의 특정부분을 위해 쿠키의 가시성(visibility)에 제한을 둘수 있다. cookiePath가 정의되었을때 쿠키는 오직 그 경로와 그 하위경로에서만 볼수 있을것이다.

12.6.3. SessionLocaleResolver

SessionLocaleResolver는 당신에게 사용자의 요청이 속한 세션으로 부터 로케일을 가져오도록 허락한다.

12.6.4. LocaleChangeInterceptor

당신은 LocaleChangeInterceptor을 사용해서 로케일을 변경할수 있다. 이 인터셉터는 하나의 핸들러 맵핑(Section 12.4, “Handler mappings” 을 보라.)에 추가될 필요가 있다. 이것은 요청내 파라미터를 찾아내고 로케일(이것은 컨텍스트내 존재하는 LocaleResolver의 setLocale()을 호출한다.)을 변경한다.

```
<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>
```

siteLanguage라는 이름의 파라미터를 포함하는 모든 *.view형태의 자원 호출은 지금 로케일을 변경할것이다. 그래서 http://www.sf.net/home.view?siteLanguage=nl의 호출은 사이트 언어를 네델란드어로 변경할것이다.

12.7. 테마(themes) 사용하기

견본절

12.8. Spring의 multipart (파일업로드) 지원

12.8.1. 소개

Spring은 웹 애플리케이션내 파일업로드를 다루기 위한 멀티파트(multipart)지원을 내장한다. 멀티파트 지원을 위한 디자인은 org.springframework.web.multipart패키지내 명시된 플러그인 가능한 MultipartResovler객체로 할수 있다. 특별히 Spring은 Commons FileUpload(http://jakarta.apache.org/commons/fileupload)와 COS FileUpload (http://www.servlets.com/cos)을 사용하기 위해 MultipartResolver를 제공한다. 파일을 업로드하는 지원되는 방법은 이 장의 끝에 서술될것이다.

디폴트에 의해 멀티파트 핸들링은 몇몇 개발자들이 그들 스스로 멀티파트를 다루길 원하는 것처럼 Spring에 의해 수행되지 않을것이다. 당신은 웹 애플리케이션 컨텍스트에 멀티파트 결정자를 추가함으로써 당신 스스로 이것을 가능하게 할수 있다. 당신이 그렇게 한 후에 각각의 요청은 그것이 멀티파트를 포함하는지 보기 위해 조사할것이다. 만약 멀티파트가 찾아지지 않는다면 요청은 기대되는 것처럼 계속될것이다. 어쨌든 멀티파트가 요청내에 발견된다면 당신의 컨텍스트내 명시된 MultipartResolver가 사용될것이다. 그리고 나서 요청내 멀티파트 속성은 다른 속성처럼 처리될것이다.

12.8.2. MultipartResolver 사용하기

다음의 예제는 CommonsMultipartResolver를 사용하는 방법을 보여준다.

```
<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>
```

이것은 CosMultipartResolver을 사용하는 예제이다.

```
<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>
```

물론 당신은 작업을 수행하는 멀티파트 결정자를 위해 클래스패스내 적당한 jar파일을 복사해 넣을 필요가 있다. CommonsMultipartResolver의 경우 당신은 commons-fileupload.jar을 사용할 필요가 있다. CosMultipartResolver의 경우 cos.jar를 사용한다.

지금 당신은 멀티파트 요청을 다루기 위해 Spring을 셋업하는 방법을 보았다. 이것을 실제로 사용하기 위해 어떻게 해야 하는지에 대해 얘기해보자. Spring DispatcherServlet가 멀티파트 요청을 탐지했을때 이것은 당신의 컨텍스트내 선언된 결정자를 활성화시키고 요청을 처리한다. 기본적으로 하는 것은 멀티파트를 위한 지원을 가진 MultipartHttpServletRequest로 최근의 HttpServletRequest을 포장해 넣는것이다. MultipartHttpServletRequest을 사용하면 당신은 이 요청에 의해 포함된 멀티파트에 대한 정보를 얻을수 있고 당신 컨트롤러내 스스로 멀티파트를 얻을수 있다.

12.8.3. 폼에서 파일업로드를 다루기.

MultipartResolver가 그 작업을 마친후 요청은 다른것처럼 처리될것이다. 이것을 사용하기 위해 당신은 파일 업로드 기능을 가진 폼을 생성하고 Spring이 폼의 필드에 바인드 하도록 하자. 다른 프라퍼티처럼 그것은 자동적으로 당신이 ServletRequestDatabinder로 사용자 지정 편집기(editor)를 등록하기 위해 당신 빈즈에 바이너리 데이터를 두기 위한 문자열이나 원시타입으로 형변화되지 않는다. 파일을 다루고 빈에 결과를 셋팅하기 위해 사용가능한 두개의 편집기가 있다. StringMultipartEditor는 파일을 문자열로 형변환(사용자 정의 문자셋을 사용하여)하는 능력을 가진다. 그리고 ByteArrayMultipartEditor는 파일을 바이트 배열로 형변환한다. 그것들은 CustomDateEditor가 하는것처럼 작동한다.

그리고 웹사이트내 폼을 사용하여 파일을 업로드하기 위해 결정자를 선언하라. 컨트롤러를 위한 url맵핑은 빈과 컨트롤러 자신을 처리할것이다.

```
<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>
```

그후 컨트롤러와 파일 프라퍼티를 가지는 실질적인 빈을 생성하라.

```
// 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;
    }
}
```

당신이 볼수 있는 것처럼 FileUploadBean은 파일 데이터를 가지는 byte[]타입의 프라퍼티를 가진다. 컨트롤러는 Spring이 빈에 의해 명시된 프라퍼티를 위한 찾을수 있는 멀티파트 객체 결정자가 변환하는 방식을 알도록 사용자 지정 편집기를 등록한다. 이 예제에서 빈의 byte[] 프라퍼티로 하는것은 아무것도 없다. 하지만 실제로 당신은 당신이 무엇을 원하든지(데이터베이스에 저장을 하거나 누군가에게 메일을 보내더라도) 할수 있다.

하지만 우리는 아직 끝나지 않았다. 실질적으로 사용자가 무언가를 업로드 하기 위해서 우리는 폼을 생성해야만 한다.

```
<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>
```

당신이 볼수 있는 것처럼 우리는 빈의 프라퍼티가 byte[]의 데이터를 가진후 명명된 필드를 생성할것이다. 게다가 우리는 멀티파트 필드를 인코딩하는 방법을 아는 브라우저를 위해 필요한 인코딩 속성을 추가한다(이것을 절대 잊지 마라.). 지금 우리는 해야할 모든것을 했다.

12.9. 예외 다루기

Spring은 당신의 요청이 적합한 컨트롤러에 의해 다루어지는 동안 발생하는 기대되지 않는 예외의 고통을 쉽게 하기 위해 HandlerExceptionResolvers제공한다. HandlerExceptionResolvers는 웹 애플리케이션 서술자인 web.xml내 당신이 명시할수 있는 예외 맵핑과 다소 비슷하다. 어쨌든 그들은 예외를 다루기 위한 좀더 유연한 방법을 제공한다. 그들은 예외가 던져질때 어떠한 핸들러가 수행되는지에 대한 정보를 제공한다. 게디가 예외를 다루는 프로그램적인 방법은 당신에게 요청이 다른 URL로 포워딩되기 전에 적절하게 응답하기 위한 방법을 위해 많은 옵션을 제공한다.(서블릿이 예외 맵핑을 명시하는것을 사용할때처럼 같은 결과를 낸다.)

더욱이 HandlerExceptionResolver을 구현하는 것은 오직 resolveException(Exception, Handler)메소드를 구현하고 ModelAndView를 반환하는 문제이다. 당신은 아마도 SimpleMappingExceptionResolver을 사용할지도 모른다. 이 결정자는 당신에게 던져지고 view이름에 그것을 맵핑하는 어떠한 예외의 클래스명을 가져오도록 할것이다. 이것은 서블릿 API로 부터 예외를 맵핑하는 기능과 기능적으로 유사하다. 하지만 이것은 또한 다른 핸들러로부터 잘 정제된 예외의 맵핑을 구현하는것이 가능하다.

Chapter 13. 통합 뷰 기술들

13.1. 소개

Spring의 탁월한 영역중의 하나는 MVC framework의 나머지 부분으로 부터 뷰 기술들을 분리하는 것이다. 예를 들어, JSP가 존재하는 곳에서 Velocity 또는 XSLT 사용을 결심하는 것은 근본적으로 구성(configuration)의 문제이다. 이 장에서는 Spring이 작업하는 주요한 뷰 기술들은 다루고, 새로운 기술들을 추가하는 간단한 방법을 알아본다. 이 장은 MVC framework과 연관된 일반적인 뷰의 기본적인 방법을 이미 다룬 Section 12.5, “view와 view결정하기” 와 유사할 것이다.

13.2. JSP & JSTL

Spring은 JSP와 JSTL뷰를 위해 특별히 연관된 해결책을 제공한다. JSP 또는 JSTL 사용은 WebApplicationContext에서 정의된 일반적인 뷰해결자(viewresolver)를 사용한다. 더욱이, JSP들을 쓸 필요가 있을때 실제로 뷰에서 표현할 것이다(render). 이 부분은 JSP 개발을 쉽게 할 수 있게 제공되는 몇몇 추가적인 기능들에서 설명한다.

13.2.1. 뷰 해결자(View resolvers)

Spring에 통합된 다른 뷰 기술처럼 뷰 해결자가 필요로하는 JSP들을 위해 여러분들의 목적(views)을 해결할 것이다. 대부분 보통 JSP들은 InternalResourceViewResolver과 ResourceBundleViewResolver를 개발할 때 뷰 해결자를 사용했다. 이 둘은 WebApplicationContext에 선언되어있다:

```
# 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
```

보는바와 같이, ResourceBundleViewResolver은 1)클래스와 2)URL을 대응시키기위해 뷰이름들을 정의하는 설정파일이 필요하다. ResourceBundleViewResolver와 함께 오직 해결자가 사용하는 뷰들의 다른 형태를 혼합할 수 있다.

```
<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>
```

InternalResourceBundleViewResolver는 위에서 설명한것처럼 JSP를 사용하기 위해 구성할 수 있다. 가장 좋은 실행은, WEB-INF 디렉토리 아래의 디렉토리안에 JSP 파일들을 위치시키는것을 강력하게 권장한다. 그래서 클라이언트들에 의해 직접적으로 접근할수 없게 한다.

13.2.2. 'Plain-old' JSPs 대(versus) JSTL

자바 표준 테그 라이브러리를 사용할때, 특별한 뷰 클래스, JstlView을 사용해야하고, JSTL는 i18N기능들을 작업하기전에 몇몇의 표현이 필요하다.

13.2.3. 추가적인 태그들을 쉽게 쓸수 있는 개발

스트링은 이전 챕터들에서 설명한것처럼 객체들을 명령하기위한 request 파라미터들의 데이터 바인딩을 제공한다. 데이터 바인딩 기능들의 조합에서 JSP 페이지들 개발을 쉽게 할 수 있게 하기 위해서, Spring은 더 쉽게 만들어진 몇몇 태그들을 제공한다. 모든 Spring 태그들은 html에서 벗어나는(escaping) 기능들을 가질 수 있거나 문자열들의 벗어나는 기능들을 가지지 않을 수 도 있다.

태그 라이브러리 서술자(TLD)는 자기 자신의 배치(distribution)안에 spring.jar와 같이 포함한다. 개별적인 태그에관한 더많은 정보은 온라인에서 찾을 수 있다:
http://www.springframework.org/docs/taglib/index.html.

13.3. Tiles

Spring을 사용하는 웹 애플리케이션 안에서 -다른 뷰 기술들과 같이- Tiles는 통합 가능하다. 다음은 대체로 타일을 사용하는 방법을 설명한다.

13.3.1. 의존물들(Dependencies)

Tiles을 사용할 수 있게 하기 위해서 여러분의 프로젝트 안에 포함된 연관된 추가적인 의존물들을 가진다. 다음은 여러분들이 필요로하는 의존물들의 목록이다.

- ☒ struts version 1.1
- ☒ commons-beanutils
- ☒ commons-digester
- ☒ commons-logging
- ☒ commons-lang

의존물들은 Spring 구분(distribution)안에 모두 이용할 수 있다.

13.3.2. Tiles를 통합하는 방법

Tiles를 사용 할 수 있게 하기 위해서, 정의들이 포함된 파일을 사용해서 구성해야한다 (정의들에 대한 기본적인 정보와 다른 Tiles 개념들, http://jakarta.apache.org/struts에서 볼 수 있다). Spring 안에서 TilesConfigurer가 사용되어진다. 다음에 보는 것은 ApplicationContext 구성 예의 일부분이다:

```
<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>
```

여러분이 볼수 있는 바와 같이, WEB-INF/defs 디렉토리에 위치한, 정의들을 포함한 5개의 파일들이 있다. WebApplicationContext의 초기와에서 파일들은 로드(loaded)될 것이고 factoryClass-설정 에의해 정의된 definitionsfactory은 초기화된다. 이를 마친 후, 정의 파일들에 포함된 tiles은 Spring 웹 애플리케이션 내에 뷰로써 사용되어질 수 있다. 뷰들을 사용가능하게 하기 위해서 Spring과 함께 사용되어지는 다른 기술처럼 ViewResolver를 가진다. 아래의 InternalResourceViewResolver과 ResourceBundleViewResolver의 두 가능성을 발견할 수 있다.

13.3.2.1. InternalResourceViewResolver

InternalResourceViewResolver는 각 뷰가 가지고있는 것을 분석하기위해(resolve) viewClass에서 주어진 것을 증명한다. 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

ResourceBundleViewResolver는 해결자(resolver)가 사용할 수 있는 뷰이름들과 뷰클래스들을 포함한 설정 파일들을 제공한다 :

```
<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
    ...
```

여러분이 볼수 있는 바와 같이, ResourceBundleViewResolver을 사용할때는, 다른 뷰 기술들을 사용하여 뷰를 혼합시킬 수 있다.

13.4. Velocity & FreeMarker

Velocity [http://jakarta.apache.org/velocity] 과 FreeMarker [http://www.freemarker.org]는 두 templating 언어이다. 이 둘은 Spring MVC 애플리케이션 내의 뷰 기술들과 같이 사용되어질수 있다.

언어들은 아주 유사하고 유사한 필요에 도움이 된다. 그래서 이번 섹션에서 함께 생각해 본다. 두 언어들 사이에 의미론상으로 구문론상의 차이점은 FreeMarker [http://www.freemarker.org] 웹사이트에서 보아라.

13.4.1. 의존물들(Dependencies)

여러분의 웹 애플리케이션은 Velocity 또는 FreeMarker 각각 작업을 하기 위해서 velocity-1.x.x.jar 또는 freemarker-2.x.jar를 포함시켜야 할 것이다. 그리고 commons-collections.jar 또한 Velocity를 이용하는데 필요할 것이다. 전형적으로 J2EE 서버에 의해 발견한 보증된 곳인 WEB-INF/lib 폴더에 포함되고 애플리케이션의 클래스 패스에 추가되어진다. 또한 WEB-INF/lib 폴더 안에 spring.jar가 이미 있다고 가정한다. 최신의 안정된 velocity, freemarker 그리고 commons collections jars는 Spring 프레임워크 안에 제공되어있고 관련된 /lib/ 하위-디렉토리들로부터 복사할 수 있다. 만약 Spring의 Velocity 뷰 안의 dateToolAttribute또는 numberToolAttribute를 사용하여 만든다면, 또한 velocity-tools-generic-1.x.jar를 포함시켜야 할 것이다.

13.4.2. 컨텍스트 구성(Context configuration)

알맞은 구성은 아래에 보는바와 같이 관련된 구성자가 *-servlet.xml를 정의를 추가하여 초기화하는 것이다.

```
<!--
  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: 애플리케이션 컨텍스트 정의 파일에 VelocityConfigurationFactoryBean 또는 FreeMarkerConfigurationFactoryBean을 web-apps에 추가시키지 말아라.

13.4.3. 생성 템플릿들(Creating templates)

템플릿들은 Section 13.4.2, “컨텍스트 구성(Context configuration)” 에서 보여준 *Configurer에 의해 명세화한 디렉토리안에 저장이 필요하다. 이문서는 두언어를 위해 생성 템플릿의 세부사항을 포함시키지 않는다 - 관련된 웹사이트에서 정보를 볼 수 있다. 만약 중요부분의 뷰 해결자들(resolvers)을 사용한다면, 논리적 뷰 이름을 JSP에대해 InternalResourceViewResolver 비슷한 형태인 템플릿 파일 이름과 관련시켜서 설명한다. 그래서 만약 제어자가 "welcome"이라는 뷰이름을 포함한 ModelAndView 객체를 되돌린다면 해결자는 /WEB-INF/freemarker/welcome.ftl 또는 /WEB-INF/velocity/welcome.vm 의 적합한템플릿을 찾을 것이다.

13.4.4. 진보한 구성(Advanced configuration)

위의 중요한 기본 구성들은 대부분 애플리케이션 요구사항에 적합할 것이다. 그러나 추가적인 구성선택들은 색다르거나 진보한 요구사항을 지시할때 이용할 수 있다.

13.4.4.1. velocity.properties

이 파일은 완전히 선택적이다. 그러나 명세화한다면, velocity 자체 구성을 하기 위해서 Velocity 런타임을 통과한 값을 포함한다. 단지 진보한 구성을 요구하는것, 만약 이 파일이 필요하다면 VelocityConfigurer 위치에서 위의 정의와 같이 명세화하라.

```
<bean
  id="velocityConfig"
  class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="configLocation">
    <value>/WEB-INF/velocity.properties</value>
  </property>
</bean>
```

대신에, 다음 inline properties와 "configLocation" 설정을 교체함에따라 Velocity 구성 bean을 위해 bean 정의에 직접적으로 velocity properties를 명세화할 수 있다.

```
<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>
```

Velocity의 Spring 설정을 위해 API 문서 [http://www.springframework.org/docs/api/org/springframework/ui/velocity/VelocityEngineFactory.html]을 참조하거나, velocity.properties 파일 자체의 예들과 정의들을 위한 Velocity 문서를 참조하라.

13.4.4.2. FreeMarker

FreeMarker 'Settings' 과 'SharedVariables' 는 FreeMarkerConfigurer bean의 적당한 bean프라퍼티를 셋팅하여 Spring에 의해 관리되는 FreeMarker Configuration 객체로 직접적으로 전달될수 있다. freemarkerSettings 프라퍼티는 java.util.Properties객체를 요구하고 freemarkerVariables 프라퍼티는 java.util.Map를 요구한다.

```
<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"/>
```

Configuration객체에 적용할 셋팅과 변수의 상세사항을 위해서 FreeMarker문서를 보라.

13.4.5. 바인드(Bind) 지원과 폼(form) 핸들링

Spring은 JSP에서 사용하기 위해서 <spring:bind>를 포함한 태그라이브러리를 제공한다. 이 태그는 주로 폼지원 객체로 부터 값을 표시하거나 웹티어나 비지니스티어내 Validator로 부터 실패한 유효성체크의 결과를 보여주기 위해서 폼을 가능하게 한다. 1.1버전에서 부터, Spring은 자체적인 폼 input 요소를 생성하기 위한 추가적인 편리한 매크로를 가지고 Velocity 와 FreeMarker 모두에 대해 같은 기능을 지원한다.

13.4.5.1. 바인드(bind) 매크로

매크로의 표준 세트는 두가지 언어(Velocity 와 FreeMarker)를 위한 spring.jar 파일내 유지된다. 그래서 그것들은 적합하게 설정된 애플리케이션을 위해 언제나 사용가능하다. 어쨌든 그것들은 당신의 view가 bean프라퍼티인 exposeSpringMacroHelpers를 true로 셋팅될때만 사용될수 있다. 당신의 view가 값을 상속할 모든 경우에 같은 프라퍼티는 VelocityViewResolver 나 FreeMarkerViewResolver에서 셋팅될수 있다. 이 프라퍼티는 Spring 매크로의 장점을 가지기를 원하는 곳을 제외하고HTML 폼 핸들링의 어떠한 양상을 위해 요구되지 않는다. 아래는 이러한 view의 정확한 설정을 보여주는 view.properties파일의 예제이다.

```
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
```

Spring 라이브러리내 정의된 몇몇 매크로는 내부적(개인적)으로 검토되지만 매크로 정의내 존재하는

범위는 모든 매크로를 호출 코드와 사용자 템플릿을 위해 볼수 있도록 만드는 것은 없다. 다음 부분은 당신에게 템플릿에서 직접적으로 호출될 필요가 있는 매크로에만 집중한다. 만약 당신이 매크로 코드를 직접적으로 보길 원한다면 파일은 spring.vm / spring.ftl 라고 불리고 org.springframework.web.servlet.view.velocity 나 org.springframework.web.servlet.view.freemarker 패키지내 존재한다.

13.4.5.2. 간단한 바인딩

Spring 폼 컨트롤러를 위한 'formView' 처럼 작동하는 html폼(vm / ftl 템플릿)내에서, 당신은 필드값으로 바인드하는 것과 유사한 코드를 사용하고 JSP와 유사한 형태로 각각의 input필드를 위한 에러메시지를 표시할수 있다. command객체의 이름은 디폴트에 의해 "command" 이지만 폼 컨트롤러의 'commandName' bean프라퍼티를 셋팅하여 당신의 MVC설정에 오버라이드될수 있다는 것을 알라. 예제 코드는 먼저 설정된 personFormV 과 personFormF view를 위해 밑에서 보여진다.

```
<!-- 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> 는 문장(period)과 당신이 바인드하고자 하는 command객체 필드의 이름에 의해 당신의 command객체(FormController프라퍼티를 변경하지 않는다면 이것은 'command'가 될것이다.)의 이름을 구성하는 'path' 인자를 요구한다. 내포된 필드는 "command.address.street" 처럼 사용될수 있다. bind 매크로는 web.xml내 ServletContext 파라미터인 defaultHtmlEscape에 의해 명시된 디폴트 HTML 회피(escaping) 행위를 가정한다.

#springBindEscaped / <@spring.bindEscaped> 라고 불리는 매크로의 선택적인 폼은 2개의 인자를 가지고

HTML회피가 상태 에러 메시지나 값에서 사용될지에 대해 명시한다. 요구되는 값은 true나 false이다. 추가적인 폼 핸들링 매크로는 HTML회피와 사용이 가능한 매크로의 사용을 단순화한다. 그것들은 다음 부분에서 설명된다.

13.4.5.3. 폼 input 생성 매크로

두가지 언어를 위한 추가적인 편리한 매크로는 바인딩과 폼 생성(유효성 체크 에러 표시를 포함하여)을 단순화한다. 폼 input필드를 생성하기 위한 매크로를 사용하는 것은 결코 필요하지 않다. 그것들은 간단한 HTML과 혼합되거나 대응될수 있거나 먼저 강조된 Spring 바인드 매크로에 직접적으로 호출한다.

사용가능한 매크로의 다음 테이블은 VTL과 FTL정의와 파라미터 목록을 보여준다.

Table 13.1. 매크로 정의 테이블

매크로	VTL 정의	FTL 정의
message (코드 파라미터에 기반한 자원 번들로부터 문자열 출력)	#springMessage($code)	<@spring.message code/>
messageText (코드 파라미터에 기반한 자원 번들로부터 문자열 출력, 디폴트 파라미터의 값으로 되돌아 감)	#springMessageText($code $default)	<@spring.messageText code, default/>
url (애플리케이션 컨텍스트 root를 가진 상대적인 URL을 접두사로 붙이는)	#springUrl($relativeUrl)	<@spring.url relativeUrl/>
formInput (사용자 입력을 모으기 위한 표준적인 input필드)	#springFormInput($path $attributes)	<@spring.formInput path, attributes, fieldType/>
formHiddenInput * (비-사용자 입력을 서브릿하기 위한 hidden input필드)	#springFormHiddenInput($path $attributes)	<@spring.formHiddenInput path, attributes/>
formPasswordInput * (비밀번호를 모으기 위한 표준적인 input 필드. 이 타입의 필드로 활성화될 값은 없다는것을 알라.)	#springFormPasswordInput($path $attributes)	<@spring.formPasswordInput path, attributes/>
formTextarea (long값을 모으기 위한 큰 텍스트 필드, freeform형태의 텍스트 input)	#springFormTextarea($path $attributes)	<@spring.formTextarea path, attributes/>
formSingleSelect (선택되기 위한 하나의 필수값을 허용하는 선택사항의 drop down 박스)	#springFormSingleSelect($path $options $attributes)	<@spring.formSingleSelect path, options, attributes/>
formMultiSelect (하나 이상의 값을 선택하기 위한 사용자를 허용하는 선택사항의 리스트 박스)	#springFormMultiSelect($path $options $attributes)	<@spring.formMultiSelect path, options, attributes/>

매크로	VTL 정의	FTL 정의
formRadioButtons (사용가능한 선택사항으로 부터 만들수 있는 하나의 selection을 허용하는 radio버튼의 세트)	#springFormRadioButtons($path $options $separator $attributes)	<@spring.formRadioButtons path, options separator, attributes/>
formCheckboxes (선택되기 위한 하나 이상의 값을 허용하는 checkbox의 세트)	#springFormCheckboxes($path $options $separator $attributes)	<@spring.formCheckboxes path, options, separator, attributes/>
showErrors (연결된 필드를 위한 유효성 체크 에러의 간단한 표시)	#springShowErrors($separator $classOrStyle)	<@spring.showErrors separator, classOrStyle/>

* FTL (FreeMarker) 에서, 두개의 매크로는 당신이 'hidden' 이나 fieldType 파라미터를 위한 값처럼 'password' 을 명시하는 일반적인 formInput 매크로를 사용할수 있는것처럼 실질적으로 필수가 아니다.

위 매크로중 어느것을 위한 파라미터는 일관적인 수단을 가진다.

☒ path: 바인드 하기 위한 필드의 이름(이를테면 "command.name")

☒ options: 모든 사용가능한 값의 map은 input필드내 선택될수 있다. 값을 표시하기 위한 map의 key는 form으로 부터 게시될것이고 command객체로 연결된다. key에 대응되어 저장되는 map객체는 사용자를 위한 폼에 표시되는 라벨이고 form에 의해 게시되는 관련값들과는 다르다. map은 언제나 컨트롤러에 의한 참조데이터처럼 제공된다. map구현물은 필수행위에 의존되어 사용될수 있다. 엄격하게 정렬된 map을 위해 적당한 비교자를 가진 TreeMap과 같은 SortedMap은 사용될수 있고 입력순으로 값을 반환해야만 하는 임의의 map을 위해 commons-collections의 LinkedHashMap 이나 LinkedMap를 사용하자.

☒ separator: 다중 옵션이 신중한(discreet) 요소(radio버튼이나 checkbox)처럼 사용가능한 곳. 순차적인 문자는 목록에서 각각 분리되기 위해 사용된다. (이를테면 "
").

☒ attributes: HTML 태그 자체에 포함되는 임의의 태그나 텍스트의 추가적인 문자열. 이 문자열은 매크로에 의해 문자그대로 울린다. 예를 들어, textarea필드에서 당신은 'rows="5" cols="60"' 처럼 속성을 제공하거나 'style="border:1px solid silver"' 처럼 스타일 정보를 전달할수 있다.

☒ classOrStyle: showErrors 매크로를 위해, 태그를 확장하는 CSS클래스의 이름은 사용할 각각의 에러를 포장한다. 아무런 정보도 없다면(또는 값이 공백이라면) 에러는 태그내 포장될것이다.

매크로의 예제는 몇몇 FTL과 VTL는 아래에서 간단하게 설명된다. 두가지 언어사이의 사용상의 차이점은 이 노트에서 설명된다.

13.4.5.3.1. input 필드

```
<!-- the Name field example from above using form macros in VTL -->
...
    Name:
    #springFormInput("command.name" "")<br>
    #springShowErrors("<br>" "")<br>
```

formInput 매크로는 path파라미터(command.name)와 위 예제에서 빈 추가적인 attribute속성을

가져온다. 다른 폼 생성 매크로와 함께 매크로는 path파라미터에 함축적으로 Spring 바인드를 수행한다. 바인딩은 새로운 바인드가 발생해서 showErrors 매크로가 다시는 path파라미터를 전달할 필요가 없을때까지 유효하다.

showErrors 매크로는 separator(분리자 - 문자들은 주어진 필드에 다중 에러를 분리하기 위해 사용될것이다.) 파라미터를 가지고 두번째 파라미터를 받아들인다. 이번은 클래스명과 style속성이다. FreeMarker는 Velocity와는 달리 attribute속성을 위한 디폴트값을 명시하는것이 가능하다. 그리고 위 두개의 매크로 호출은 다음의 FTL처럼 표시될수 있다.

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

출력은 name필드를 생성하는 form일부를 밑에서 보여준다. 그리고 form이 필드내 어떤값도 가지지 않고 서브밋된 후에 유효성체크 에러를 표시한다. 유효성체크는 Spring의 Validation프레임워크를 통해 발생한다.

생성된 HTML은 다음처럼 보일것이다.

```
Name:
  <input type="text" name="name" value=""
>
<br>
  <b>required</b>
<br>
<br>
```

formTextarea매크로는 formInput매크로와 같은 방법으로 작동하고 같은 파라미터 목록을 받아들인다. 공통적으로 두번째 파라미터(속성)는 스타일정보를 전달하거나 textarea를 위한 rows와 cols를 전달하기 위해 사용될것이다.

13.4.5.3.2. selection 필드

4개의 selection 필드 매크로는 HTML form내 공통 UI값 selection input를 생성하기 위해 사용될수 있다.

☒ formSingleSelect

☒ formMultiSelect

☒ formRadioButtons

☒ formCheckboxes

4가지 매크로 각각은 form필드를 위한 값과 그 값에 관련된 라벨을 포함하는 옵션의 map을 받아들인다. 값과 라벨은 같을수 있다.

FTL내 radio버튼의 예제는 밑에 있다. form지원 객체는 이 필드를 위한 'London'의 디폴트 값을 명시하고 유효성체크가 필요하지 않다. form이 표현될때 선택하는 city의 전체 목록이 'cityMap'이라는 이름하의 모델내 참조 데이터처럼 제공된다.

```
...
  Town:
  <@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>
```

이것은 분리자 ""를 사용하여 cityMap내 각각의 값중 하나인 radio버튼을 표현한다. 추가적인 속성은 제공되지 않는다(매크로를 위한 마지막 파라미터는 없다). cityMap은 map내 각각의 키(key)-값(value)쌍을 위한 같은 문자열을 사용한다. map의 키는 form이 실질적으로 전송된 요청 파라미터처럼 서브밋하는 것이다. map의 값은 사용자가 보는 라벨이다. 위 예제에서 form지원 객체내 주어진 3개의 잘 알려진 city이 목록과 디폴트 값이다.

```
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
```

만약 당신의 애플리케이션이 내부 코드에 의해 city를 다루는것을 기대한다면 예를 들어, 코드의 map은 아래의 예제처럼 적합한 key를 가지고 생성될것이다.

```
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;
}
```

코드는 radio값이 적절한 코드지만 사용자가 좀더 사용자에게 친숙한 city이름을 볼수 있는 출력을 생성할것이다.

```
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. HTML회피를 오버라이드하고 XHTML호환 태그를 만든다.

위 form매크로의 디폴트 사용은 HTML 4.01호환 HTML태그의 결과를 보일것이고 Spring의 바인트 지원이 사용하는것처럼 web.xml내 정의된 HTML회피를 위한 디폴트 값을 사용한다. XHTML호환 태그를 만들거나 디폴트 HTML회피 값을 오버라이드하기 위해 당신은 템플릿(또는 당신의 템플릿을 볼수 있는 모델내)에 두개의 변수를 명시할수 있다. 템플릿내 그것들을 명시하는 장점은 form내 다른 필드를 위해

다른 행위를 제공하기 위한 템플릿 처리로 그것들이 나중에 다른 값으로 변경될수 있다는 것이다.

당신의 태그를 위한 XHTML호환으로 변경하기 위해 xhtmlCompliant라는 이름의 모델/컨텍스트 변수를 'true'의 값을 명시하라.

```
## for Velocity..
#set($springXhtmlCompliant = true)

<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

Spring매크로에 의해 생성되는 태그는 직접적으로 처리된 후 XHTML호환이 될것이다.

유사한 방법으로 HTML회피는 필드마다 명시될수 있다.

```
<#-- 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

XSLT는 XML을 위한 변형언어이고 웹 애플리케이션내 view기술처럼 인기있다. 당신의 애플리케이션이 당연히 XML을 다루거나 당신의 모델이 XML로 쉽게 변활될수 있다면 XSLT는 view기술로 좋은 선택이 될수 있다. 다음 부분은 모델 데이터처럼 XML문서를 생성하는 방법을 보여주고 Spring애플리케이션내 XSLT로 변형한다.

13.5.1. 나의 첫번째 단어

이 예제는 Controller내 단어 목록을 생성하고 그것들을 모델 map으로 추가하는 사소한 Spring애플리케이션이다. map은 XSLT view의 view이름과 함께 반환된다. Spring Controller들의 상세사항을 위해 Section 12.3, "컨트롤러" 을 보라. XSLT view는 단어의 목록에서 변형될 준비가 된 간단한 XML문서로 바뀔것이다.

13.5.1.1. Bean 정의

설정은 간단한 Spring애플리케이션을 위한 표준이다. dispatcher 서블릿 설정파일은 URL맵핑과 하나의 컨트롤러 bean을 가지는 ViewResolver를 위한 참조를 포함한다.

```
<bean id="homeController"class="xslt.HomeController"/>
```

그것은 우리의 단어 생성 'logic'을 구현한다.

13.5.1.2. 표준적인 MVC 컨트롤러 코드

컨트롤러 로직은 정의된 핸들러 메소드와 함께 AbstractController의 하위클래스로 캡슐화된다.

```
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);
}
```

지금까지 우리는 XSLT가 명시하는것을 아무것도 하지 않았다. 모델 데이터는 당신이 다른 Spring MVC애플리케이션을 하는것처럼 같은 방법으로 생성된다. 지금 애플리케이션의 설정에 의존하여 단어의 목록은 JSP/JSTL에 의해 요청 속성을 추가하여 표시될수 있거나 VelocityContext에 객체를 구가하여 Velocity에 의해 다루어질수 있다. XSLT가 그것들을 표현하기 위해, 그것들은 XML문서로 변환된다. 자동적으로 'domify' 객체 그래프가 될 사용가능한 소프트웨어 패키지가 있다. 당신은 선택한 방법으로 모델에서 DOM을 생성하는 완벽한 유연성을 가진다. 이것은 domification처리를 관리하는 툴을 사용할때 위험한 모델 데이터의 구조에서 너무 큰 부분으로 작동하는 XML의 변형을 방지한다.

13.5.1.3. 모델 데이터를 XML로 변환하기

우리의 단어 목록이나 다른 모델 데이터로부터 DOM문서를 생성하기 위해 우리는 `org.springframework.web.servlet.view.xslt.AbstractXsltView`의 하위클래스를 만든다. 우리는 추상 메소드인 `createDomNode()`을 구현해야만 한다. 이 메소드에 전달되는 첫번째 파라미터는 모델 map이다. 우리의 단어 애플리케이션내 HomePage클래스의 완벽한 목록이다. 이것은 W3C Node를 요구하는 것으로 변환하기 전에 XML문서를 빌드하기 위한 JDOM을 사용한다. 하지만 이것은 W3C API보다 다루기 쉬운 JDOM(그리고 Dom4J) API를 알기 때문에 간단한다.

```
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. stylesheet 파라미터 추가하기

일련의 파라미터 이름/값 쌍은 선택적으로 변형객체로 추가될 하위클래스에 의해 정의될수 있다. 파라미터 이름은 파라미터를 명시하기 위한 <xsl:param name="myParam">defaultValue</xsl:param>로 선언되는 XSLT템플릿으로 정의되는 것들에 매치되어야만 한다. AbstractXsltView의 getParameters() 메소드를 오버라이드하고 이름/값 쌍의 Map을 반환한다. 만약 파라미터가 현재 요청으로부터 정보를 가져올 필요가 있다면 당신은 getParameters(HttpServletRequest request) 메소드를 대신 오버라이드(1.1버전부터)할수 있다.

13.5.1.3.2. 날짜와 화폐단위 포맷팅

JSTL 과 Velocity와는 달리, XSLT는 로케일에 기반한 화폐단위와 날짜 포멧팅을 위한 지원이 상대적으로 빈약하다. 그 사실을 인정해서 Spring은 그부분에 대한 지원을 위해 createDomNode() 메소드로 부터 사용할수 있는 헬퍼 클래스를 제공한다. org.springframework.web.servlet.view.xslt.FormatHelper를 위해 JavaDoc를 보라.

13.5.1.4. view프라퍼티 정의하기

views.properties 파일(또는 위 Velocity예제에서 처럼 XML기반한 view해설자(resolver)를 사용할때 동등한 xml정의)은 'My First Words'인 하나의 view를 가진 애플리케이션을 위한 것처럼 보인다.

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words
```

여기서, 당신은 view가 첫번째 프라퍼티인 '.class'내 모델 domification을 다루는 것에 의해 쓰여진 HomePage클래스와 묶이는 방법을 볼수 있다. stylesheetLocation 프라퍼티는 HTML파일로의 변형이 되는 XML을 다루는 XSLT파일을 가리키고 마지막 프라퍼티인 '.root'는 XML문서의 root처럼 사용될 이름이다. 이것은 createDomNode 메소드를 위한 두번째 파라미터로 위 HomePage 클래스에 전달된다.

13.5.1.5. 문서 변형

마지막으로, 우리는 위 문서를 변형하기 위해 사용되는 XSLT코드를 가진다. views.properties 파일에서 강조된것처럼 이것은 home.xslt로 불리고 WEB-INF/xsl하위의 war파일내 있다.

```
<?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. 요약

언급된 파일의 요약과 WAR파일내 그것들의 위치는 아래의 단순화된 WAR구조에서 보여진다.

```
ProjectRoot
  |
  +- WebContent
      |
      +- WEB-INF
          |
          +- classes
          |     |
          |     +- xslt
          |     |   |
          |     |   +- HomePageController.class
          |     |   +- HomePage.class
          |     |
          |     +- views.properties
          |
          +- lib
          |   |
          |   +- spring.jar
          |
          +- xsl
          |   |
          |   +- home.xslt
          |
          +- frontcontroller-servlet.xml
```

당신은 XML파서와 XSLT엔진이 클래스패스에서 사용가능한지를 확인할 필요가 있다. JDK 1.4는 디폴트로 그것들을 제공한다. 그리고 대부분의 J2EE컨테이너는 디폴트에 의해 그것들을 사용가능하게 만들것이다. 하지만 이것은 인식되는 에러의 가능한 원인이다.

13.6. 문서 views (PDF/Excel)

13.6.1. 소개

HTML페이지를 반환하는 것은 사용자에게 모델 출력을 보여주기 위해 언제나 가장 좋은 방법은 아니다. 그리고 Spring은 모델 데이터로부터 동적으로 PDF문서나 Excel 스프레드시트를 생성하는것을 쉽게 만든다. 문서는 view이고 서버로부터 응답시 클라이언트 PC가 스프레드시트나 PDF뷰어 애플리케이션을 실행할수 있도록 하는 올바른 컨텐츠타입을 가지고 나올것이다.

Excel 뷰를 사용하기 위해, 당신은 클래스패스내 'poi' 라이브러리를 추가할 필요가 있다. 그리고 PDF생성을 위해 iText.jar를 추가할 필요가 있다. 둘다 Spring 배포물에 포함되어 있다.

13.6.2. 설정 그리고 셋업

문서 기반 view는 XSLT view와 대부분 동일한 형태로 다루어진다. 그리고 다음의 부분은 XSLT예제에서 사용된 같은 컨트롤러가 PDF문서나 Excel 스프레드시트(Open Office에서 볼수 있거나 변경이 가능한)처럼 같은 모델을 표시하기 위해 호출되는 방법을 보여주어서 이전의 것을 빌드한다.

13.6.2.1. 문서 view정의

첫번째, views.properties파일(또는 xml파일형태의 프라퍼티 파일)을 수정하자. 그리고 두가지 문서

타입을 위해 간단한 view정의를 추가하자. 전체 파일은 이전에 XSLT view에서 보여진 것과 비슷하게 보일것이다.

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.class=excel.HomePage

pdf.class=pdf.HomePage
```

만약 당신의 모델 데이터를 추가하기 위해 템플릿 스프레드시트로 시작하길 원한다면 view정의내 'url' 프라퍼티로 위치를 명시하라.

13.6.2.2. 컨트롤러 코드

컨트롤러 코드에서 우리는 사용하기 위한 view의 이름을 변경하는것보다 이전의 XSLT예제로부터 같은것을 사용할것이다. 물론, 당신은 능숙할수 있고 URL파라미터나 몇몇 다른 로직에 기반하여 이것을 선택할수 있다. 이것은 Spring이 컨트롤러로부터 view를 디커플링하는데 매우 좋다는것을 증명한다.

13.6.2.3. Excel view를 위한 하위클래스 만들기

XSLT에제를 위해 했던것처럼, 우리는 출력문서를 생성하는 사용자정의 행위를 구현하기 위한 적합한 추상 클래스의 하위클래스를 만들것이다. Excel을 위해, 이것은 org.springframework.web.servlet.view.document.AbstractExcelView의 하위클래스를 생성하고 buildExcelDocument를 구현한다.

새로운 스프레드시트의 첫번째 칼럼의 연속적인 row내 모델 map으로 부터 단어목록을 보여주는 Excel view을 위한 완벽한 목록이다.

```
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));

        }
    }
}
```

만약 당신이 지금 view(새로운 ModelAndView("xl", map);를 반환하는)의 이름처럼 xl을 반환하는 컨트롤러를 수정하고 다시 애플리케이션을 실행한다면, 이전처럼 같은 페이지를 요청할때 자동적으로 Excel 스프레드시트가 생성되거나 다운로드되는것을 알게된다.

13.6.2.4. PDF view를 위한 하위클래스 만들기

단어 목록의 PDF버전은 좀더 간단하다. 이 시점에, 클래스는 org.springframework.web.servlet.view.document.AbstractPdfView를 확장하고 다음처럼 buildPdfDocument() 메소드를 구현한다.

```
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)));

    }
}
```

새로운 ModelAndView("pdf", map);을 반환하여 pdf view를 반환하기 위한 컨트롤러를 수정하고 애플리케이션내 URL을 다시 로드하라. 이 시점에 PDF문서는 모델 map에서 각각의 단어를 목록화 하는것을 나타낼것이다.

13.7. JasperReports

JasperReports (http://jasperreports.sourceforge.net)는 강력하고, 쉽게 이해되는 XML파일 포맷을 사용하여 디자인된 리포트의 생성을 지원하는 오픈소스 리포팅 엔진이다. JasperReports는 4가지 다른 포맷(CSV, Excel, HTML 그리고 PDF)으로 리포트 출력을 표시할수 있다.

13.7.1. 의존성

애플리케이션은 JasperReports의 최근 릴리즈를 포함할 필요가 있을것이다. 최근 릴리즈는 이 시점에 0.6.1이다. JasperReports자체는 다음의 제품에 의존성을 가진다.

☒ BeanShell

☒ Commons BeanUtils

☒ Commons Collections

☒ Commons Digester

☒ Commons Logging

☒ iText

☒ POI

JasperReports는 또한 JAXP호환 XML파서를 요구한다.

13.7.2. 설정

ApplicationContext에서 JasperReports view를 설정하기 위해 당신의 리포트가 표시되길 원하는 포맷에 의존하는 적당한 view클래스를 위한 view이름을 맵핑하는 ViewResolver를 정의해야만 한다.

13.7.2.1. ViewResolver 설정하기

전형적으로, 당신은 view클래스와 프라퍼티 파일내 파일을 위한 view이름을 맵핑하기 위한 ResourceBundleViewResolver을 사용할것이다.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename">
        <value>views</value>
    </property>
</bean>
```

우리는 기본이름인 views을 가진 자원번들내 view맵핑을 찾을 ResourceBundleViewResolver의 인스턴스를 설정한다. 이 파일의 정확한 내용은 다음 부분에서 언급된다.

13.7.2.2. View 설정하기

Spring은 JasperReports에 의해 지원되는 4가지의 출력 포맷중 하나에 관련된 것중 4가지의 JasperReports를 위한 5가지의 다른 View구현물을 포함하고 실행시 결정되는 포맷을 허용한다.

Table 13.2. JasperReports View 클래스

클래스명	표시(Render) 형태
JasperReportsCsvView	CSV
JasperReportsHtmlView	HTML
JasperReportsPdfView	PDF
JasperReportsXlsView	Microsoft Excel
JasperReportsMultiFormatView	실행시 결정됨(Section 13.7.2.4, “JasperReportsMultiFormatView 사용하기” 를 보라.)

view이름을 위한 클래스중 하나와 리포트 파일을 맵핑하는것은 여기서 보여진것처럼 이전 부분에서 설정된 자원번들로 적당한 항목을 추가하는 간단한 사항이다.

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
```

당신은 simpleReport라는 이름을 가진 view가 JasperReportsPdfView클래스에 맵핑되는것을 볼수 있다. 이것은 PDF포맷으로 표시되는 리포트의 출력을 만들것이다. view의 url 프라퍼티는 참조하는 리포트 파일의 위치를 셋팅한다.

13.7.2.3. 리포트 파일에 대해

JasperReports는 리포트 파일의 두가지 타입(.jrxml 확장자를 가지는 디자인 파일, .jasper 확장자를 가지는 컴파일된 리포트 파일)을 가진다. 전형적으로, 당신은 이것을 당신의 애플리케이션으로 배치하기전에 .jrxml 디자인 파일을 .jasper파일로 컴파일하기 위해 JasperReports Ant작업을 사용한다. Spring으로 당신은 리포트 파일을 위해 이러한 파일들을 맵핑할수 있다. 그리고 Spring은 당신을 위해 구동되는 .jrxml파일을 컴파일할것이다. 당신은 .jrxml파일이 Spring에 의해 컴파일된 후 그것을 주의해야 한다. 컴파일된 리포트는 애플리케이션 생명을 위해 캐시된다. 파일을 변경하기 위해 당신은 애플리케이션을 다시 시작할 필요가 있을것이다.

13.7.2.4. JasperReportsMultiFormatView 사용하기

JasperReportsMultiFormatView는 수행시 명시되는 리포트 포맷을 허용한다. 리포트의 실질적인 표시는 다른 JasperReports view클래스중 하나로 위임된다. JasperReportsMultiFormatView 클래스는 실행시 명시되는 구현물을 허용하는 레퍼(wrapper) 레이어를 간단하게 추가한다.

JasperReportsMultiFormatView 클래스는 포맷(format) 키와 식별자(discriminator) 키라는 두가지 개념을 소개한다. JasperReportsMultiFormatView 클래스는 실질적인 view구현물 클래스를 검사하기 위해 맵핑키를 사용하고 맵핑키를 검사하기 위해 포맷키를 사용한다. 코딩에서 당신은 키와 값으로의 맵핑키처럼 포맷키를 가진 모델을 위한 항목을 추가한다.

```
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);
}
```

이 예제에서, 맵핑키는 요청 URI의 확장으로부터 결정되고 디폴트 포맷키(format)의 모델에 추가된다. 만약 당신이 다른 포맷키를 사용하길 바란다면 JasperReportsMultiFormatView클래스의 formatKey 프라퍼티를 사용해서 설정할수 있다.

디폴트에 의해 다음의 맵핑키의 맵핑은 JasperReportsMultiFormatView내 설정된다.

Table 13.3. JasperReportsMultiFormatView 디폴트 맵핑키 맵핑

맵핑키	View 클래스
csv	JasperReportsCsvView
html	JasperReportsHtmlView
pdf	JasperReportsPdfView
xls	JasperReportsXlsView

위 예제에서 URI /foo/myReport.pdf에 대한 요청은 JasperReportsPdfView클래스에 맵핑될것이다. 당신은 JasperReportsMultiFormatView의 formatMappings프라퍼티를 사용하여 view클래스 맵핑에 대한 맵핑키를 오버라이드할수 있다.

13.7.3. ModelAndView 활성화하기

당신이 선택한 포맷으로 정확하게 리포트를 표시하기 위해서, 당신은 리포트를 활성화하기 위해 필요한 모든 데이터를 Spring에 제공해야만 한다. JasperReports를 위한 이 방법에서 당신은 리포트 데이터소스를 가지고 모든 리포트 파라미터를 전달해야만 한다. 리포트 파라미터는 간단한 이름/값 쌍이고 이름/값 쌍을 추가해야할 모델을 위한 Map에 추가되어야만 한다.

모델에 데이터소스를 추가할때 당신은 선택할 두가지 접근법을 가진다. 첫번째 접근법은 어떠한 임의의 키의 모델 Map을 위한 JRDataSource 나 Collection의 인스턴스를 추가하는것이다. Spring은 모델내 이 객체를 위치시키고 리포트 데이터소스처럼 이것을 처리한다. 예를 들어, 당신은 다음처럼 모델을 활성화할것이다.

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  model.put("myBeanData", beanData);
  return model;
}
```

두번째 접근법은 특정 키의 JRDataSource 나 Collection의 인스턴스를 추가하고 view클래스의 reportDataKey 프라퍼티를 사용하여 키를 설정한다. 두 경우 다 Spring은 JRBeanCollectionDataSource인스턴스내 Collection의 인스턴스일것이다. 예를 들어,

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  Collection someData = getSomeData();
  model.put("myBeanData", beanData);
  model.put("someData", someData);
  return model;
}
```

당신은 두개의 Collection인스턴스가 모델에 추가되는것을 볼수 있다. 사용되는것이 정확한지 확인하기 위해, 우리는 view설정을 간단하게 변경한다.

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
simpleReport.reportDataKey=myBeanData
```

첫번째 접근법을 사용할때 Spring은 JRDataSource 나 Collection의 인스턴스를 사용할 것이다. 만약 당신이 JRDataSource 나 Collection의 다중 인스턴스를 모델에 둘 필요가 있다면 당신은 두번째 접근법을 사용할

필요가 있다.

13.7.4. 하위-리포트로 작동하기

JasperReports는 당신의 주(master) 리포트 파일내 내장 하위-리포트를 위한 지원을 제공한다. 여기엔 당신의 리포트 파일내 하위-리포트를 포함하기 위한 다양한 기법이 있다. 가장 쉬운 방법은 리포트 경로와 디자인 파일의 하위 리포트를 위한 SQL쿼리를 하드코딩하는 것이다. 이 접근법의 결점은 분명하다. 당신의 리포트 파일에 들어가는 하드코딩된 값은 재사용성을 줄이고 리포트 디자인을 변경하거나 수정하는것을 힘들게 한다. 이것을 극복하기 위해 당신은 선언적인 하위-리포트를 설정할수 있다. 그리고 당신은 컨트롤러로 부터 직접적으로 하위-리포트를 위한 추가적인 데이터를 포함할수 있다.

13.7.4.1. 하위-리포트 파일 설정하기

Spring을 사용하여 주 리포트내 하위-리포트 파일이 포함되는것을 제어하기 위해, 당신의 리포트 파일은 외부 소스로부터 하위-리포트를 받아들이도록 설정이 되어야만 한다. 이것을 하기 위해 당신은 다음처럼 리포트 파일내 파라미터를 선언한다.

```
<parameter name="ProductsSubReport" class="net.sf.jasperreports.engine.JasperReport"/>
```

그 다음, 당신은 하위-리포트 파라미터를 사용하는 하위-리포트를 정의한다.

```
<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>
```

이것은 ProductsSubReport 파라미터하의 net.sf.jasperreports.engine.JasperReports의 인스턴스처럼 전달되는 하위-리포트를 기대하는 주 리포트 파일을 정의한다. 당신의 Jasper view클래스가 설정될때, 당신은 리포트 파일을 로드하도록 Spring에 지시할수 있고 subReportUrls 프라퍼티를 사용하여 하위-리포트같은 JasperReports엔진으로 전달할수 있다.

```
<property name="subReportUrls">
    <map>
        <entry key="ProductsSubReport">
            <value>/WEB-INF/reports/subReportChild.jrxml</value>
        </entry>
    </map>
</property>
```

여기서 Map의 키는 리포트 디자인 파일내 하위-리포트 파라미터의 이름과 관련된다. 그리고 항목은 리포트 파일의 URL이다. Spring은 리포트 파일을 로드할것이고 필요하다면 컴파일하고 주어진 키로 JasperReports엔진에 전달할것이다.

13.7.4.2. 하위-리포트 데이터소스 설정하기

이 단계는 Spring을 사용하여 하위-리포트를 설정할때 완전히 선택사항이다. 당신이 원한다면, 정적 쿼리를 사용하여 하위-리포트를 위한 데이터소스를 설정할수 있다. 어쨌든, 당신은 Spring이 ModelAndView내 반환되는 데이터를 JRDataSource의 인스턴스로 변환하기를 원한다면 Spring이 변환할 ModelAndView내 파라미터를 명시할 필요가 있다. 이 설정을 하기 위해 파라미터 이름의 목록은 선택된

view클래스의 subReportDataKeys 프라퍼티를 사용한다.

```
<property name="subReportDataKeys">
    <value>SubReportData</value>
</property>
```

여기서 당신이 제공하는 키는 ModelAndView내 사용되는 키와 리포트 디자인 파일내 사용되는 키 모두와 일치해야만 한다.

13.7.5. 전파자(Exporter) 파라미터 설정하기

만약 당신이 전파자(exporter) 설정을 위한 특별한 요구사항을 가진다면, 이를 테면 PDF리포트를 위한 페이지 크기를 명시하기를 원한다면 당신은 view클래스의 exporterParameters 프라퍼티를 사용하여 Spring설정 파일내 선언적으로 전파자(exporter) 파라미터를 설정할수 있다. exporterParameters 프라퍼티는 Map같은 타입이 되고 설정내 항목의 키는 전파자(exporter) 파라미터 정의와 당신이 파라미터에 할당하기를 원하는 값이 될 항목의 값을 포함하는 정적 필드의 전체경로의 이름이 되어야한다. 이것의 예제는 아래와 같다.

```
<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>
```

여기서 당신은 JasperReportsHtmlView가 결과 HTML내 footer를 출력할 net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER를 위한 전파자(exporter) 파라미터를 가지고 설정되는것을 볼수 있다.

Chapter 14. 다른 웹 프레임워크들과의 통합

14.1. 소개

Spring은 어떠한 자바 기반의 웹 프레임워크와도 쉽게 통합된다. 당신은 ContextLoaderListener [http://www.springframework.org/docs/api/org/springframework/web/context/ContextLoaderListener.htm를 당신의 web.xml에 선언하고, 어떤 컨텍스트 파일을 로드할 것인지를 세팅하기 위해 contextConfigLocation <context-param>를 사용하기만 하면 된다.

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>
```

노트: Listeners는 Servlet API 2.3 버전에 추가되었다. 만약 당신이 Servlet 2.2 컨테이너를 사용한다면, 당신은 동일한 기능을 얻기 위해 ContextLoaderServlet [http://www.springframework.org/docs/api/org/springframework/web/context/ContextLoaderServlet.html]를 사용할 수 있다.

만약 당신이 contextConfigLocation 컨텍스트 파라미터를 명시하지 않는다면, ContextLoaderListener는 로드할 /WEB-INF/applicationContext.xml 파일을 찾을 것이다. 일단 컨텍스트 파일이 로드되면, Spring은 빈 정의에 기반하여 WebApplicationContext [http://www.springframework.org/docs/api/org/springframework/web/context/WebApplicationContext.htr객체를 생성하고 이것을 ServletContext에 담아둔다.

모든 자바 웹 프레임워크들은 Servlet API에 기반하여 만들어졌기 때문에, 당신은 Spring이 생성한 ApplicationContext를 얻기 위해 다음의 코드를 사용할 수 있다.

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

WebApplicationContextUtils [http://www.springframework.org/docs/api/org/springframework/web/context/support/WebApplicationCo클래스는 편의를 위해 만들어진 것인데, 때문에 당신은 ServletContext 속성의 이름을 기억할 필요가 없다. 그것의 getWebApplicationContext() 메써드는 만약 (ApplicationContext) 객체가 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE 키로 존재하지 않는다면 null을 반환할 것이다. 어플리케이션에서 NullPointerExceptions를 받는 위험을 감수하는 것보다는 getRequiredWebApplicationContext() 메써드를 사용하는 것이 낫다. 이 메써드는 ApplicationContext를 찾지 못할 경우, 예외를 던진다.

일단 당신이 WebApplicationContext를 참조하게 되면, 당신은 이름 혹은 타입으로 빈들을 가져올 수 있다. 대부분의 개발자들은 빈들을 이름으로 가져와서 그것의 구현된 인터페이스들 중 하나로 캐스팅한다.

운 좋게도 이번 장에서 다루는 대부분의 프레임워크들은 빈들을 룩업하는 방식이 매우 간단하다. 빈들을 BeanFactory로부터 가져오는 것이 쉬울 뿐만 아니라, 컨트롤러에 의존성 삽입(dependency injection)을 사용할 수 있도록 해준다. 각각의 프레임워크 섹션에서 그것의 특화된 통합 전략들에 기반하여 보다 세부적인 사항들을 설명할 것이다.

14.2. JavaServer Faces

JavaServer Faces (JSF) 는 컴포넌트 기반, 이벤트 드리븐 웹 프레임워크이다. Sun Microsystem의 JSF Overview [http://java.sun.com/j2ee/javaserverfaces/overview.html] 에 따르면, JSF 기술은 다음을 포함하고 있다.

☒ UI 컴포넌트들을 표현하고 그것들의 상태를 관리하며, 이벤트들을 핸들링하고, 밸리데이션을 삽입하고, 페이지 네비게이션을 정의하고, 국제화와 접근성을 지원하는 API들의 세트

☒ JSP 페이지 내에서 JavaServer Faces 인터페이스를 표현하기 위한 JavaServer Pages (JSP) 커스텀 태크 라이브러리

14.2.1. DelegatingVariableResolver

당신의 Spring 미들티어를 JSF 웹 레이어와 통합하는 가장 쉬운 방법은 DelegatingVariableResolver [http://www.springframework.org/docs/api/org/springframework/web/jsf/DelegatingVariableResolver.html] 클래스를 사용하는 것이다. 이 변수 처리자(variable resolver)를 당신의 어플리케이션에 설정하려면, faces-context.xml를 수정해야 한다. <faces-config> 요소를 연 이후에, <application> 요소를 추가하고 그 안에 <variable-resolver> 요소를 추가하면 된다. 변수 처리자의 값은 Spring의 DelegatingVariableResolver를 참조해야만 한다.

```
<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>
```

Spring의 변수 처리자를 명시함으로써, 당신은 Spring 빈들을 당신이 관리하는 빈들의 프라퍼티들로 설정할 수 있다. DelegatingVariableResolver는 처음엔 값을 룩업하는 것을 기반하는 JSF 구현의 디폴트 처리자에 위임한다. 그리고나서 Spring의 루트 WebApplicationContext에 위임한다. 이것은 당신이 JSF에 의해 관리되는 빈들에 의존성을 쉽게 주입할 수 있도록 해준다.

관리되는 빈들은 faces-config.xml 파일에 정의된다. 아래는 Spring의 BeanFactory로부터 가져온 #{userManager} 빈을 어디에 정의하느냐를 보여준다.

```
<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>
```

DelegatingVariableResolver는 JSF와 Spring을 통합할 때 적극 추천되는 전략이다. 만약 당신이 보다 튼튼한 통합 예시를 찾는다면, JSF-Spring [http://jsf-spring.sourceforge.net/] 프로젝트를 볼 수 있다.

14.2.2. FacesContextUtils

커스텀 VariableResolver는 당신의 프라퍼티들을 faces-config.xml 내의 빈들과 매핑할 때 매우 잘 동작한다. 그러나, 종종 당신은 빈을 명시적으로 가로챌 필요가 있을 것이다. FacesContextUtils [http://www.springframework.org/docs/api/org/springframework/web/jsf/FacesContextUtils.html] 클래스는 그것을 쉽게 해준다. 이 클래스는 WebApplicationContextUtils과 비슷한데, ServletContext 파라미터가 아니라 FacesContext 파라미터를 가진다는 점만 다르다.

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

14.3. Struts

Struts [http://struts.apache.org] 는 자바 어플리케이션을 위한 사실상의 웹 프레임워크 그 자체이다. 이것은 주되게 Struts이 가장 먼저 릴리즈된(2001년 6월) 것 중 하나라는 사실에 기인한다. Craig McClanahan에 의해 창안된 Struts은 아파치 소프트웨어 재단에 의해 후원되는 오픈 소스 프로젝트이다. 처음부터, Struts는 JSP/Servlet 프로그래밍 패러다임을 획기적으로 간소화했고 개인 프레임워크들을 사용하던 많은 개발자들을 끌어들였다. 이것은 프로그래밍 모델을 간단하게 했으며 오픈소스였다. 그리고 이것은 이 프로젝트가 성장하고 자바 웹 개발자들 사이에 대중화될 수 있도록 하는 거대한 커뮤니티를 가졌다.

Struts 어플리케이션을 Spring과 통합하는 데는 두 가지 방법이 있다.

☒ ContextLoaderPlugin를 사용하여 Spring이 Action들을 빈들로 관리하도록 설정하고 Action들의 의존성을 Spring 컨텍스트 파일에 세팅하는 방법

☒ Spring의 ActionSupport 클래스를 상속해서 getWebApplicationContext() 메써드를 사용하여 Spring 관리되는 빈들을 명시적으로 가로채는 방법

14.3.1. ContextLoaderPlugin

ContextLoaderPlugin
[http://www.springframework.org/docs/api/org/springframework/web/struts/ContextLoaderPlugIn.html] 은 Struts ActionServlet을 위해 Spring 컨텍스트 파일을 로드하는 Struts 1.1 이상 버전의 플러그인이다. 이 컨텍스트는 ContextLoaderListener에 의해 로드된 WebApplicationContext를 그것의 부모 클래스로 참조한다. 컨텍스트 파일의 디폴트 이름은 매핑된 서블릿의 이름에 -servlet.xml을 더한 것이다. 만약 web.xml에서 ActionServlet이 <servlet-name>action</servlet-name>라고 정의되었다면, 디폴트는 /WEB-INF/action-servlet.xml이 될 것이다.

이 플러그인을 설정하기 위해서는 다음의 XML을 struts-config.xml 파일의 아래쪽의 plug-ins 섹션에 추가해야 한다.

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

컨텍스트 설정 파일의 위치는 "contextConfigLocation" 프라퍼티를 사용하여 임의대로 정할 수 있다.

```
<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>
```

모든 컨텍스트 파일들을 로드하기 위해 이 플러그인을 사용할 수 있는데, 이것은 StrutsTestCase와 같은 테스팅 툴들을 사용할 때 유용할 것이다. StrutsTestCase의 MockStrutsTestCase는 Listeners를 초기화하지 않을 것이기 때문에, 당신의 컨텍스트 파일들을 플러그인에 담는 것이 필요하다. 이 이슈에 대해서는 버그 정리
[http://sourceforge.net/tracker/index.php?func=detail&aid=1088866&group_id=39190&atid=424562]
를 참조하도록 하라.

이 플러그인을 struts-config.xml에 설정한 이후에야 Action을 Spring에 의해 관리되도록 설정할 수 있다. Spring 1.1.3은 이를 위해 두 가지 방법을 제공한다.

☒ Struts의 디폴트 RequestProcessor를 Spring의 DelegatingRequestProcessor로 오버라이드한다.

☒ <action-mapping>의 타입 속성에 DelegatingActionProxy 클래스를 사용한다.

이 메써드들 모두 당신이 Action들과 action-context.xml 파일에 있는 그것들의 의존성들을 관리할 수 있게 해준다. struts-config.xml과 action-servlet.xml 내의 Action들은 action-mapping의 "path"와 빈의 "name"으로 연결된다. 당신이 struts-config.xml 파일에 다음과 같은 설정을 가진다고 가정하자.

```
<action path="/users" .../>
```

그러면, 당신은 Action 빈을 "/users"라는 이름으로 action-servlet.xml 내에 정의해야만 한다.

```
<bean name="/users" .../>
```

14.3.1.1. DelegatingRequestProcessor

DelegatingRequestProcessor
[http://www.springframework.org/docs/api/org/springframework/web/struts/DelegatingRequestProcessor.
를 struts-config.xml 파일에 설정하기 위해서는, <controller> 요소 내의 "processorClass"를 오버라이드해야 한다. 이 줄들은 <action-mapping> 요소에 뒤따른다.

```
<controller>
        <set-property property="processorClass"
            value="org.springframework.web.struts.DelegatingRequestProcessor"/>
    </controller>
```

이 세팅을 추가한 후에, 당신의 Action은 그것이 무슨 타입이건 간에 자동적으로 Spring의 컨텍스트 파일에서 룩업될 것이다. 사실, 당신은 타입을 명시할 필요조차 없다. 다음의 조각코드들은 둘 다 모두 잘 동작할 것이다.

```
<action path="/user" type="com.whatever.struts.UserAction"/>
```

```
        <action path="/user"/>
```

만약 당신이 Struts의 modules 특징을 사용한다면, 당신의 빈 이름들은 반드시 모듈 접두어를 포함해야만 한다. 예를 들어, 모듈 접두어 "admin"을 가진 <action path="/user"/>로 정의된 action은 <bean name="/admin/user"/>라는 빈 이름을 가져야 한다.

노트: 만약 당신이 Struts 어플리케이션에서 Tiles를 사용한다면, 당신은 DelegatingTilesRequestProcessor [http://www.springframework.org/docs/api/org/springframework/web/struts/DelegatingTilesRequestProces 로 <controller>를 설정해야만 한다.

14.3.1.2. DelegatingActionProxy

만약 직접 작성한 RequestProcessor를 가지고 있어서 DelegatingTilesRequestProcessor를 사용할 수 없는 상황이라면, DelegatingActionProxy [http://www.springframework.org/docs/api/org/springframework/web/struts/DelegatingActionProxy.html] 를 action-mapping에서 사용할 수 있다.

```
<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>
```

action-servlet.xml에서의 빈 정의는 직접 작성한 RequestProcessor를 쓰건, DelegatingActionProxy를 쓰건 동일하다.

만약 당신이 컨텍스트 파일에 Action을 정의한다면, 그 Action에 대해 Spring 빈 컨테이너의 모든 특징들을 사용할 수 있을 것이다. 각각의 request에 대한 새로운 Action 인스턴스를 초기화하기 위한 옵션으로 의존성 주입을 사용하는 것 등. 후자를 사용하려면 당신의 Action 빈 정의에 singleton="false"를 추가해주어야 한다.

```
<bean name="/user" singleton="false" autowire="byName"
            class="org.example.web.UserAction"/>
```

14.3.2. ActionSupport 클래스들

앞에서 언급한 것처럼, WebApplicationContextUtils 클래스를 사용해서 ServletContext로부터 WebApplicationContext를 가져올 수 있다. 더 쉬운 방법은 Struts를 위한 Spring의 Action 클래스들을 상속받는 것이다. 예를 들어, Struts의 Action 클래스를 상속하는 것 대신에 Spring의 ActionSupport [http://www.springframework.org/docs/api/org/springframework/web/struts/ActionSupport.html] 클래스를 상속할 수 있다.

ActionSupport 클래스는 getWebApplicationContext()처럼 부가적인 편의 메써드들을 제공해준다. 아래의 예제는 Action에서 어떻게 이것을 사용할 수 있는지를 보여준다.

```
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");
    }
}
```

Spring은 모든 표준 Struts Action의 하위 클래스들을 포함한다. - Spring 버전은 단지 이름에 Support만을 붙혀놓았을 뿐이다.

☒ ActionSupport
[http://www.springframework.org/docs/api/org/springframework/web/struts/ActionSupport.html],
☒ DispatchActionSupport
[http://www.springframework.org/docs/api/org/springframework/web/struts/DispatchActionSupport.ht
☒ LookupDispatchActionSupport
[http://www.springframework.org/docs/api/org/springframework/web/struts/LookupDispatchActionSup
and
☒ MappingDispatchActionSupport
[http://www.springframework.org/docs/api/org/springframework/web/struts/MappingDispatchActionSu

추천하는 전략은 당신의 프로젝트에 가장 잘 맞는 접근방법을 사용하는 것이다. 하위클래스 방법은 당신의 코드를 보다 읽기 쉽게 만들어주며 어떻게 의존성들을 해결할 것인지 명확하게 알게 해준다. 반면, ContextLoaderPlugin을 사용하는 것은 컨텍스트 XML 파일에 새로운 의존성을 추가하는 것을 쉽게 해준다. 어느 쪽이던지, Spring은 두 개의 프레임워크들을 통합하기 위해 몇몇 멋진 옵션들을 제공한다.

14.4. Tapestry

Tapestry는 아파치의 자카르타 프로젝트로부터 나온 강력한 컴포넌트 지향 웹 어플리케이션 프레임워크이다.(http://jakarta.apache.org/tapestry) Spring이 그 자체의 강력한 ui 계층을 가지고 있지만, 웹 ui를 위해 Tapestry와의 조합을 사용하여 J2EE 어플리케이션을 개발하는 것은 독특한 몇가지 장점들이 있다. 이 문서는 이 두 개의 프레임워크들을 조합하기 위한 몇 가지 최선의 실습예제들을 상세시 설명할 것이다. 당신이 Tapestry와 Spring 프레임워크 기본 둘 다에 대해 어느정도 친숙하다고 전제하에서 그것들에 대해서 여기서 설명하지는 않을 것이다. Tapestry와 Spring에 대한 일반적인 소개글은 각각의 웹 사이트들에서 접할 수 있다.

14.4.1. 아키텍쳐

Tapestry와 Spring으로 개발된 전형적인 계층적 J2EE 어플리케이션은 최상위 UI 계층은 Tapestry로, 많은 하위 계층들은 하나 혹은 그 이상의 Spring 어플리케이션 컨텍스트에 의해 관리되는 형태로 구성될 것이다.

☒ 사용자 인터페이스 계층

- 사용자 인터페이스와 관련있다.

- 몇몇 어플리케이션 로직을 포함한다.

- Tapestry에 의해 제공된다.

- Tapestry를 경유하여 UI를 제공하는 것은 별개로 하고, 이 계층에서의 코드는 Service 계층의 인터페이스들을 구현한 객체들을 경유하여 작동한다. 이러한 서비스 계층 인터페이스들을 구현한 실질적인 객체들은 Spring 어플리케이션 컨텍스트로부터 가져온다.

☒ 서비스 계층

- 어플리케이션 특화된 '서비스' 코드

- 도메인 객체들과 작동하며 그 도메인 객체들을 몇몇가지 저장소(database)에 넣고 빼기 위해 Mapper API를 사용한다.

- 하나 혹은 그 이상의 Spring 컨텍스트에 의해 관리된다.

- 이 계층에서의 코드는 도메인 모델 내에서 어플리케이션 특화의 형태로 객체들을 조작한다. 그것은 이 계층 내의 다른 코드와 Mapper API를 통해 작동한다. 이 계층의 객체는 그것이 작동할 때 필요한 특정한 매퍼 구현체들을 Spring 컨텍스트를 통해 받는다.

- 이 계층의 코드는 Spring 컨텍스트에서 관리되기 때문에, 그 자신의 트랜잭션들을 관리하는 것 대신에, Spring 컨텍스트에 의해 트랜잭션화될 것이다.

☒ 도메인 모델

- 이 도메인에 특화된 데이터와 로직을 다루는 도메인 특화된 객체 계층구조이다.

- 비록 도메인 객체 계층이 어딘가 저장되고 이것에 대한 몇가지 일반적인 특권(예를 들어, 양방향적인 관계들) 가진다는 개념에 기반하여 만들어진 것이지만, 이것은 일반적으로 다른 계층들에 대해 전혀 알 필요가 없다. 그렇기 때문에, 이것은 독자적으로 테스트될 수 있고, 생산과 테스팅을 위해 서로 다른 매핑 구현체들과 함께 사용될 수 있다.

- 이 객체들은 독립형이거나 Spring 어플리케이션과 결합되어 사용된다. 이것은 고립성, IoC, 다른 전략적 구현체 등등의 컨텍스트의 몇가지 이점들을 가지게 한다.

☒ 데이터 소스 계층

- (데이터 접근 객체 -DAO 라고도 불리는) Mapper API : 도메인 모델을 몇 가지 (일반적으로 DB이지만 파일시스템, 메모리 등도 될 수 있는) 저장소에 저장할 때 사용되는 API

- Mapper API 구현체들 : 하나 혹은 그 이상의 Mapper API의 특화된 구현체들, 예를 들어, Hibernate 특화 매퍼, JDO 특화 매퍼, JDBC 특화 매퍼 또는 메모리 메퍼 등

- 매퍼 구현체들은 하나 혹은 그 이상의 Spring 어플리케이션 컨텍스트에 존재한다. 서비스 계층 객체는 컨텍스트를 통해 작동시 필요한 매퍼 객체들을 받는다.

☒ 데이터베이스, 파일시스템 혹은 다른 저장소들

- 도메인 모델 내의 객체들은 하나 혹은 그 이상의 매퍼 구현체들에 의해 하나 이상의 저장소에 저장된다.

- 저장소는 파일시스템처럼 매우 간단할 수도 있고, DB에서의 스키마처럼 도메인 모델로부터 자신만의

데이터 표현을 가질 수도 있다. 그러나, 다른 계층들에 대해서는 알지 못한다.

14.4.2. 구현체

유일한 실질적인 질문(이 문서에 의해 설명될 필요가 있는)은 어떻게 Tapestry pages가 Spring 어플리케이션 컨텍스트의 인스턴스로 정의된 빈들인 서비스 구현체들에 접근할 수 있느냐이다.

14.4.2.1. 샘플 어플리케이션 컨텍스트

우리가 xml 형태로 다음과 같은 간단한 어플리케이션 컨텍스트 정의를 가지고 있다고 가정하자.

```
<?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>
```

Tapestry 어플리케이션 내에서, 우리는 이 어플리케이션 컨텍스트를 로드할 필요가 있고, Tapestry pages로 하여금 AuthenticationService와 UserService 인터페이스를 각각 구현하고 있는 authenticationService와 userService 빈들을 가지게끔 해야 한다.

14.4.2.2. Tapestry pages에서 빈들을 얻어오기

이 지점에서, Spring의 정적 유틸리티 함수인 WebApplicationContextUtils.getApplicationContext(servletContext)를 호출함으로써 웹 어플리케이션은 어플리케이션 컨텍스트를 사용할 수 있다. 여기에서의 servletContext는 J2EE 명세의 표준 ServletContext를 의미한다. 그렇기 때문에, 페이지가 예를 들어 UserService 인스턴스를 얻기 위한 한가지 간단한 방법은 다음과 같은 코드를 가지는 것이다.

```
    WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
        getRequestCycle().getRequestContext().getServlet().getServletContext());
    UserService userService = (UserService) appContext.getBean("userService");
    ... some code which uses UserService
```

이러면 된다. 이런 방식은 page 혹은 component를 위한 베이스 클래스의 메써드에 있는 대부분의 기능들을 캡슐화함으로써 난잡함을 최소화할 수 있다. 그러나, 몇 가지 관점에서 이 방식은 Spring이 장려하고, 이 어플리케이션의 다른 계층에서 사용되고 있는 IoC 접근방식에 반대된다. IoC 접근방식 내에서는 이상적으로는 page가 컨텍스트에 이름으로 특정한 빈을 요청하지 않아야 하며, 사실 이상적으로는 page는 컨텍스트에 대해 전혀 몰라야 한다.

운 좋게도, 이것을 가능하게 하는 방식이 있다. 우리는 Tapestry가 이미 page에 프라퍼티를 선언적으로 추가하는 메커니즘을 가지고 있다는 사실에 의존한다. 그리고 이러한 선언적 형태로 page에 대한 모든 프라퍼티들을 관리하는 앞서 언급한 접근방식이 사실상 사용되기 때문에, Tapestry는 그것들의 생명주기를 page와 component 생명주기의 일부로 잘 관리할 수 있다.

14.4.2.3. 어플리케이션 컨텍스트를 Tapestry에 드러내기

먼저 Tapestry page 혹은 Component가 ServletContext를 가지지 않은 채로 ApplicationContext를 사용할 수 있도록 만들 필요가 있다. 이것은 page 혹은 component의 생명주기 내에서 ApplicationContext에 접근할 필요가 생기는 단계에서, ServletContext는 page에서 쉽게 사용하기 힘들기 때문에 우리는 WebApplicationContextUtils.getApplicationContext(servletContext) 를 직접 사용할 수 없다. 하나의 방법은 이것을

드러내주는 Tapestry IEngine의 커스텀 버전을 정의하는 것이다.

```
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);
        }
    }
}
```

이 엔진 클래스는 Spring 어플리케이션 컨텍스트를 Tapestry 어플리케이션의 '글로벌' 객체에 "appContext"라는 속성으로 위치시킬 것이다. 이 특별한 IEngine 인스턴스가 Tapestry 어플리케이션을 위해 사용된되었다는 사실을 Tapestry 어플리케이션 정의 파일에 시작지점으로 등록하는 것을 확실히 해야 한다. 예를 들어보자.

```
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 정의 파일들

이제 ApplicationContext로부터 우리가 필요한 빈들을 가져오기 위해 page 혹은 component 정의 파일(*.page 혹은 *.jwc)에서 우리는 단순히 property-specification 요소들을 추가하고 그것들을 위한 page 혹은 component 프라퍼티들을 생성하면 된다. 예시를 보자.

```
    <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>
```

property-specification 내의 OGNL(Original?) 표현은 프라퍼티를 위한 초기값을 컨텍스트로부터 얻어온 빈으로 나타낸다. 전체 page 정의는 아마 다음과 같을 것이다.

```
<?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. abstract accessors 추가하기

이제 page 혹은 component 자체를 위한 Java 클래스 정의를 위해, 우리가 정의한 프라퍼티들에 접근하기 위한 추상 getter 메써드를 추가해야 한다. page 또는 component가 실제로 Tapestry에 의해 로드될 때, 그것은 정의된 프라퍼티들을 추가하기 위해 클래스파일에 런타임 코드 방식을 실행하고, 새롭게 생성된 필드들에 대한 추상 getter 메써드들을 연결한다. 다음의 예를 보자.

```
    // 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. 개요

이 예제에서, 우리는 Spring ApplicationContext에 정의된 서비스 빈들을 선언적 형태로 page에 제공되도록 처리해왔다. page 클래스는 서비스 구현체들이 어디에서 왔는지 알지 못하며, 때문에 예를 들어,

테스트동안 다른 구현체로 살짝 빠져나가기 쉽다. 이러한 IoC는 Spring 프레임워크의 주요한 목적과 이점의 하나이며, 우리는 이 Tapestry 어플리케이션에서의 J2EE 스택을 향한 모든 지점에서 이것을 확장하도록 있도록 처리해왔다.

14.5. WebWork

WebWork [http://www.opensymphony.com/webwork] 는 간단함을 염두에 두고 설계된 웹 프레임워크이다. 이것은 일반적인 command 프레임워크인 XWork [http://www.opensymphony.com/xwork] 를 기반으로 개발되었다. XWork 역시 IoC 컨테이너를 가지고 있지만, 이것은 Spring만큼 총체적인 특징을 가지고 있지 못하며, 이 부분에서 그것을 다루지는 않을 것이다. WebWork 컨트롤러들은 Actions로 불리는데, 그것은 Action [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/Action.html] 인터페이스를 구현해야만 하기 때문일 것이다. ActionSupport [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/ActionSupport.html] 클래스는 이 인터페이스를 구현하는데, WebWork action들을 위한 일반적인 부모 클래스이다.

WebWork는 xwork-optional [https://xwork-optional.dev.java.net/] 프로젝트의 java.net에 위치한 자체 Spring 통합 프로젝트를 유지한다. 현재적으로, WebWork와 Spring을 통합하는데는 세 가지의 선택이 가능하다.

- SpringObjectFactory: XWork의 디폴트 ObjectFactory [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/ObjectFactory.html] 를 오버라이드 해서 XWork는 Spring 빈들을 루트 WebApplicationContext에서 찾을 것이다.
- ActionAutowiringInterceptor: Action의 의존성들을 그것들이 생성될 때 인터셉터를 사용하여 자동으로 묶는다.
- SpringExternalReferenceResolver: <action> 요소의 <external-ref> 요소에서 정의된 이름에 기반하여 Spring 빈들을 룩업한다.

이 모든 전략들은 WebWork's Documentation [http://wiki.opensymphony.com/display/WW/WebWork+2+Spring+Integration] 에서 보다 상세하게 설명되어 있다.

Chapter 15. JMS

15.1. 소개

Spring은 JMS API의 사용을 단순화하고 JMS 1.0.2와 1.1 API사이의 차이점으로부터 사용자를 감싸는 JMS 추상 프레임워크를 제공한다.

JMS는 기능적으로 대략 두개의 영역(메시지 생산자(production)와 소비자(consumption))으로 분리될수 있다. J2EE환경내에서 메시지를 비동기적으로 소비하기 위한 능력은 이것이 MessageListeners 나 ConnectionConsumers의 생성으로 인해 제공되는 독립형(standalone) 애플리케이션내에서 메시지-지향(driven) bean에 의해 제공된다. JmsTemplate내의 기능은 생산(producting) 메시지에 촛점을 맞춘다. Spring의 차후 릴리즈는 독립형(standalone) 환경에서 비동기적 메시지 소비자를 할당할것이다.

패키지 org.springframework.jms.core 는 JMS를 사용하기 위한 핵심적인 기능을 제공한다. 이것은 JDBC를 위한 JdbcTemplate처럼 자원의 생성과 릴리즈를 다루어서 JMS의 사용을 단순화하는 JMS 템플릿 클래스를 포함한다. Spring템플릿 클래스를 위한 디자인의 주요한 공통점은 사용자 구현 콜백 인터페이스를 위한 작업 처리의 기초를 위임하는 공통적인 작업을 수행하고 좀더 정교한 사용을 위한 헬퍼(helper) 메소드를 제공하는것이다. JMS 템플릿은 같은 디자인을 따른다. 클래스는 메시지 전달, 비동기적인 메시지 소비, 그리고 JMS세션과 사용자를 위한 메시지 생산자(producer)를 드러내기 위한 다양하고 편리한 메소드를 제공한다.

패키지 org.springframework.jms.support 는 JMSException 번역 기능을 제공한다. 번역은 체크된 JMSException구조를 체크되지 않은 예외의 반영된 구조로 형변환한다. 만약 어느 제공자(provider)가 체크된 javax.jms.JMSException의 하위클래스를 명시한다면 이 예외는 체크되지 않은 UncategorizedJmsException으로 포장된다. 패키지 org.springframework.jms.support.converter 는 자바객체와 JMS메시지 사이의 변환을 위한 MessageConverter 추상화를 제공한다. 패키지 org.springframework.jms.support.destination 는 JNDI내 저장된 목적지(destination)를 위한 서비스 위치자(locator)를 제공하는것처럼 JMS목적지(destination)관리를 위한 다양한 전략을 제공한다.

마지막으로 패키지 org.springframework.jms.connection 는 독립형 애플리케이션내에서 사용하기 위해 적합한 ConnectionFactory의 구현물을 제공한다. 이것은 또한 JMS를 위한 Spring의 PlatformTransactionManager의 구현물을 포함한다. 이것은 트랜잭션적인 자원에서 Spring의 트랜잭션 관리 기법까지처럼 JMS의 통합을 허용한다.

15.2. 도메인 단일화(unification)

JMS스펙에는 두가지 커다란 릴리즈(1.0.2와 1.1)가 있다. JMS 1.0.2에서는 point-to-point (Queues)와 publish/subscribe (Topics)의 메시지 도메인의 두가지 타입이 정의된다. 1.0.2 API는 각각의 도메인을 위한 병행하는(parallel) 클래스 구조를 제공하여 두개의 메시지 도메인을 반영한다. 결과적으로 클라이언트 애플리케이션은 JMS API의 사용으로 명시하는 도메인이다. JMS 1.1은 두개의 도메인사이 기능적인 차이점과 클라이언트 API의 차이점을 모두 단순화하는 도메인 단일화의 개념을 소개한다. 제거되는 기능적인 차이점의 예제는 만약 당신이 JMS 1.1 제공자를 사용한다면 당신은 하나의 도메인으로부터 메세지를 트랜잭션적으로 소비하고 같은 Session을 사용하여 다른것의 메시지를 생산할수 있다.

JMS 1.1스펙은 2002년 4월에 릴리즈되었다. 그리고 2003년 11월에 J2EE 1.4의 일부처럼 편입되었다. 결과처럼 대부분의 애플리케이션은 JMS 1.0.2를 지원하기 위해서만 요구되는 것을 사용한다.

15.3. JmsTemplate

JmsTemplate의 두가지 구현물이 제공된다. 클래스 JmsTemplate 은 JMS 1.1 API를 사용하고 하위클래스 JmsTemplate102 은 JMS 1.0.2 API를 사용한다.

콜백 인터페이스를 구현할 필요가 있는 JmsTemplate을 사용하는 코드는 그것들에게 명백하게 정의된 규칙을 부여한다. MessageCreator 콜백 인터페이스는 JmsTemplate으로 코드를 호출하여 제공된 세션을 부여하는 메시지를 생성한다. JMS API의 좀더 복잡한 사용법을 허용하기 위해 콜백 SessionCallback 은 JMS 세션을 가진 사용자를 제공하고 콜백 ProducerCallback 은 Session과 MessageProducer를 드러낸다.

JMS API는 send메소드의 두가지 타입을 드러낸다. 하나는 배달(delivery)모드, 우선순위(priority), 그리고 서비스의 품질(QOS) 파라미터와 같은 살아있는 시간(time-to-live)를 가져오고 다른 하나는 디폴트 값을 사용하는 QOS파라미터를 가지지 않는다. JmsTemplate내에서 많은 send메소드를 가진 이후 QOS파라미터의 셋팅은 많은 수의 send메소드로 중복을 피하기 위한 bean프라퍼티처럼 드러낸다. 유사하게도 동기적인 receive호출을 위한 timeout값은 프라퍼티 setReceiveTimeout를 사용하여 셋팅한다.

몇몇 JMS 제공자(provider)는 ConnectionFactory의 설정을 통해 관리자적인 디폴트 QOA값의 셋팅을 허용한다. 이것은 MessageProducer's의 send메소드 send(Destination destination, Message message) 를 호출이 JMS스펙에 명시되는 것보다 QOS의 다른 디폴트 값을 사용할것에 영향을 끼친다. 그러므로 QOS값의 일관적인 관리를 제공하기 위해서 JmsTemplate은 boolean프라퍼티인 isExplicitQosEnabled 를 true로 셋팅하여 이것 자체의 QOS값을 사용하는것이 구체적으로 가능해야만 한다.

15.3.1. ConnectionFactory

JmsTemplate은 ConnectionFactory에 대한 참조를 요구한다. ConnectionFactory는 JMS 스펙의 일부이고 JMS를 사용하여 작동하기 위한 항목점(entry point)처럼 제공한다. 이것은 JMS 제공자로 connection을 생성하고 SSL설정 옵션처럼 업체가 명시하는 다양한 설정 파라미터를 분리하기(encapsulates) 위한 factory처럼 클라이언트 애플리케이션에 의해 사용된다.

EJB내부에서 JMS를 사용할때 업체는 선언적인 트랜잭션에 참여하고 connection과 세션의 풀링을 수행할수 있도록 JMS인터페이스를 구현물을 제공한다. 이 구현물을 사용하기 위해 J2EE컨테이너는 전형적으로 당신이 EJB나 서블릿 배치 서술자 내부 resource-ref처럼 JMS connection factory를 선언하는것을 요구한다. EJB내부에서 JmsTemplate로 이러한 기능을 사용하는것을 확인하기 위해 클라이언트 애플리케이션은 ConnectionFactory의 관리 구현물을 참조하는것을 확인해야만 한다.

Spring은 ConnectionFactory 인터페이스인 SingleConnectionFactory 의 구현물을 제공한다. 그것은 모든 createConnection 호출에 같은 connection을 반환하고 close을 호출하여 무시한다. 이것은 테스트와 같은 connection 많은 수의 트랜잭션에 걸쳐있는 다중 JmsTemplate호출을 위해 사용될수 있기 때문에 독립형 환경에 유용하다. SingleConnectionFactory는 JNDI에서 전형적으로 나오는 표준적인 ConnectionFactory에 대한 참조를 가져온다.

15.3.2. 트랜잭션 관리

Spring은 하나의 JMS ConnectionFactory를 위한 트랜잭션을 관리하는 JmsTransactionManager 을 제공한다. 이것은 JMS 애플리케이션이 7장에서 언급된것처럼 Spring의 관리 트랜잭션 기능에 영향을 끼치는 것을 허용한다. JmsTransactionManager 는 쓰레드를 위한 명시된 ConnectionFactory로 부터 Connection/Session 짝을 묶는다. 어쨌든 J2EE환경내 ConnectionFactory는 connection과 세션을 풀링할것이다. 그래서 인스턴스는 풀링 행위에 의존하는 쓰레드를 묶는다. 독립형 환경에서 Spring의 SingleConnectionFactory을 사용하는것은 하나의 JMS connection과 자체의 세션을 가지는 각각의 트랜잭션을

사용하는 결과를 낳는다. JmsTemplate 은 JtaTransactionManager 와 분산 트랜잭션을 수행하기 위한 XA-성능의 JMS ConnectionFactory 을 함께 사용될수 있다.

관리되고 관리되지 않는 트랜잭션적인 환경에서 코드를 재사용하는것은 connection으로 부터 Session을 생성하기 위한 JMS API를 사용할때 구별되지 못할수도 있다. 이것은 JMS API가 세션을 생성하기 위한 오직 하나의 factory메소드를 가지고 트랜잭션과 승인(acknowledgement)모드를 위한 값을 요구하기 때문이다. 관리되는 환경에서 환경의 트랜잭션적인 구조의 책임내 이러한 값들을 셋팅한다. 그래서 이러한 값들은 업체가 JMS connection을 포장하여 무시된다. 관리되지 않는 환경내에서 JmsTemplate을 사용할때 당신은 프라퍼티 SessionTransacted 와 SessionAcknowledgeMode의 사용을 통해 이러한 값들을 명시할수 있다. JmsTemplate 를 가지고 PlatformTransactionManager을 사용할때 템플릿은 언제나 주어진 트랜잭션적인 JMS세션이 될것이다.

15.3.3. 목적지(destination) 관리

ConnectionFactory와 같은 목적지(Destinations)는 JNDI에서 저장되고 가져올수 있는 객체를 처리하는 JMS이다. Spring 애플리케이션 컨텍스트를 설정할때 하나는 JMS목적지를 위한 당신의 객체 참조의 의존성삽입을 수행하기 위한 JNDI factory클래스인 JndiObjectFactoryBean을 사용할수 있다. 어쨌든 종종 이 전략은 애플리케이션내 많은 수의 목적지가 있거나 JMS제공자(provider)를 위한 유일한 향상된 목적지 관리 기능을 가진다면 다루기가 어렵다. 이러한 향상된 목적지 관리의 예제는 목적지의 동적 생성이 되거나 목적지의 구조적인 이름공간(namespace)을 위한 지원이 될것이다. JmsTemplate은 인터페이스 DestinationResolver의 구현물을 위한 JMS 목적지 객체의 목적지 이름의 분석을 위임한다. DynamicDestinationResolver는 JmsTemplate에 의해 사용되는 디폴트 구현물이고 동적 목적지 분석을 조정한다. JndiDestinationResolver는 또한 JNDI내 포함된 목적지를 위한 서비스 위치자와 같이 작동하고 DynamicDestinationResolver내 포함된 행위를 위해 선택적으로 의지한다(fall back).

매우 종종 JMS애플리케이션내 사용되는 목적지는 오직 수행시에만 알려지기 때문에 애플리케이션이 배치될때 관리적으로 생성될수 없다. 이것은 종종 잘 알려진 명명규칙에 따라 수행시 목적지를 생성하는 시스템 콤포넌트들의 상호작용간에 애플리케이션 로직을 공유하기 때문이다. 비록 동적 목적지의 생성이 JMS스펙의 일부는 아닐지라도 대부분의 업체는 이 기능을 제공하고 있다. 동적 목적지는 임시 목적지로부터 그것들과 달리 작동하는 사용자에 의해 정의된 이름을 가지고 생성되고 JNDI내 종종 등록되지 않는다. API는 목적지와 함께 속한 프라퍼티가 업체가 명시한 이후 제공자(privider)에서 제공자(provider)로의 다양한 동적 목적지를 생성하기 위해 사용된다. 어쨌든 업체에 의해 때때로 만들어진 간단한 구현물 선택은 JMS스펙내 경고를 무시하는것이고 디폴트 목적지 프라퍼티를 가진 새로운 목적지를 생성하기 위한 TopicSession의 메소드 createTopic(String topicName) 나 QueueSession의 메소드 createQueue(String queueName)를 사용하는것이다. 업체 구현물에 의존하여 DynamicDestinationResolver는 그 다음 하나를 분석하는 대신에 물리적인 목적지를 생성할수도 있다.

boolean타입의 프라퍼티인 PubSubDomain은 사용되는 JMS도메인이 무엇인지에 대해 아는 JmsTemplate를 설정하기 위해 사용된다. 디폴트에 의해 이 프라퍼티의 값은 false이고 사용될 점대점(point-to-point) 도메인, 큐(queue)를 표시한다. 1.0.2구현물에서 이 프라퍼티의 값은 만약 JmsTemplate의 send작업이 큐(queue)나 토픽(topic)으로 메시지를 전달할지 결정한다. 이 플래그(flag)는 1.1구현물을 위한 send작업에 영향을 주지 않는다. 어쨌든 두 구현물에서 이 프라퍼티는 DestinationResolver의 구현물을 통해 동적 목적지의 분석행위를 결정한다.

당신은 프라퍼티 DefaultDestination을 통해 디폴트 목적지를 가진 JmsTemplate을 설정할수 있다. 디폴트 목적지는 특정 목적지를 참조하지 않는 send와 receive작업이 사용될수 있다.

15.4. JmsTemplate 사용하기

JmsTemplate를 사용하여 시작하기 위해 당신은 JMS 1.0.2 구현물인 JmsTemplate102나 JMS 1.1 구현물이 JmsTemplate을 선택할 필요가 있다. 어떠한 버전을 제공할지 결정하기 위해 JMS제공자(provider)을 체크하라.

15.4.1. 메시지 보내기

JmsTemplate은 메시지를 보내기 위한 많은 편리한 메소드를 포함한다. javax.jms.Destination객체를 사용하여 목적지를 명시하는 send메소드가 있고 JNDI룩업으로 사용하기 위한 문자열을 사용하여 목적지를 정의한다. send메소드는 디폴트 목적지를 사용하는 목적지 인자를 가지지 않는다. 이것은 1.0.2 구현물을 사용하여 큐(queue)에 메세지를 보내는 예제이다.

```
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;
  }

}
```

이 예제는 제공되는 Session객체로부터 텍스트 메시지를 생성하기 위한 MessageCreator 콜백과 ConnectionFactory의 참조를 전달하여 생성되는 JmsTemplate, 메시지 도메인을 명시하는 boolean값을 사용한다. 0(zero) 인자 생성자와 setConnectionFactory/Queue 메소드는 제공되고 BeanFactory를 사용하여 인스턴스를 생성하기 위해 사용될수 있다. 메소드 simpleSend는 밑에서 보여지는 큐(queue)대신에 토픽(topic)으로 전달하기 위해 변경된다.

```
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");
    }
  });
}
```

애플리케이션 컨텍스트내 1.0.2를 설정할때 만약 큐(queue)나 토픽(topic)으로 전달하기를 원하는지 표시하기 위한 boolean프라퍼티인 PubSubDomain 프라퍼티의 값을 셋팅하는것을 기억하는것이 중요하다.

send(String destinationName, MessageCreator c) 메소드는 목적지의 문자열이름을 사용하여 당신에게 메시지를 보내도록 한다. 만약 이러한 이름들이 JNDI에 등록이 도어 있다면 당신은 JndiDestinationResolver의 인스턴스를 위한 템플릿의 DestinationResolver 프라퍼티를 셋팅해야만 한다.

만약 당신이 JmsTemplate을 생성하고 디폴트 목적지를 명시한다면 send(MessageCreator c)는 그 목적지로 메시지를 보낸다.

15.4.2. 동기적으로 받기(Receiving)

JMS가 전형적으로 비동기적인 처리와 관련되어 있지만 동기적으로 메시지를 소비하는것도 가능하다. 오버로드된 receive 메소드는 이 기능을 제공한다. 동기적으로 받는동안 쓰레드를 호출하는것은 메시지가 사용가능할때까지 블럭된다. 이것은 쓰레드를 호출하는것이 잠재적으로 무기한 블럭이 될수 있기 때문에 위험한 작업이 될수 있다. receiveTimeout 프라퍼티는 메시지를 기다리는것을 중지하기 전에 수령자(receiver)가 얼마나 오래 기다릴지를 명시한다.

15.4.3. 메시지 변환기(converter) 사용하기

도메인 모델 객체의 전송을 용이하게 하기 위해 JmsTemplate 은 메시지의 데이터내용을 위한 인자처럼 자바객체를 가지는 다양한 send메소드를 가진다. JmsTemplate 내 오버로드된 메소드인 convertAndSend 와 receiveAndConvert는 MessageConverter인터페이스의 인스턴스를 위한 변환처리를 위임한다. 이 인터페이스는 자바객체와 JMS메시지 사이의 변환을 위한 간단한 규칙을 정의한다. 디폴트 구현물인 SimpleMessageConverter 는 String 와 TextMessage, byte[] 와 BytesMesssage, java.util.Map 와 MapMessage간의 전환을 지원한다. 변환기를 사용하여 당신의 애플리케이션 코드는 JMS를 통해 전달하고 받는 비지니스 객체에 집중할수 있고 JMS메시지처럼 표현되는 방식의 상세화에 괴로워하지 않게된다.

현재 모래상자(sandbox)는 자바빈과 MapMessage간의 변환을 위한 반영(reflection)을 사용하는 MapMessageConverter을 포함한다. 당신이 스스로 구현할수 있는 다른 인기있는 구현물에 대한 선택사항은 객체를 표현하는 TextMessage을 생성하기 위한 JAXB, Castor, XMLBeans, 또는 XStream과 같은 존재하는 XML마셜링(marshalling) 패키지를 못쓰게 만드는(bust) 변환기이다.

변환기 클래스내부에서 일반적으로 캡슐화할수 없는 메시지 프라퍼티, 헤더, 그리고 몸체(body)의 셋팅을 받아들이기 위해 MessagePostProcessor 인터페이스는 당신에게 이것이 보내기 전 변환된 후 메시지에 접근하도록 한다. 아래의 예제는 java.util.Map 이 메시지로 변환된 후 메시지 헤더와 프라퍼티를 변경하는 방법을 보여준다.

```
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;
        }
    });
}
```

이것은 이 형태의 메시지를 보여준다.

```
MapMessage={
  Header={
    ... standard headers ...
    CorrelationID={123-00001}
  }
  Properties={
    AccountID={Integer:1234}
  }
  Fields={
    Name={String:Mark}
    Age={Integer:35}
  }
}
```

15.4.4. SessionCallback 과 ProducerCallback

send작업이 많은 공통적인 사용 시나리오를 다루는 동안 당신이 JMS세션이나 MessageProducer에서 다중 작업을 수행하고자 원하는 때가 있다. SessionCallback 과 ProducerCallback는 JMS세션과 Session/MessageProducer를 짝으로(pair) 드러낸다. JmsTemplate의 execute() 메소드는 이러한 콜백 메소드를 수행한다.

Chapter 16. EJB에 접근하고 구현하기

가벼운 컨테이너처럼 Spring은 종종 EJB대체물로 검토된다. 우리는 대부분의 애플리케이션과 사용상황에서 많은 것을 할것이라도 믿는다. ORM그리고 JDBC접근, 트랜잭션 영역내 풍부한 지원 기능과 조합된 컨테이너와같은 Spring은 EJB컨테이너와 EJB를 통해 유사한 기능을 구현하는 것보다 좀더 나은 선택이다.

어쨌든 Spring을 사용하는 것이 당신에게 EJB를 사용하는것을 제한하지는 않는다는 것을 아는게 중요하다. 사실 Spring은 EJB에 대한 접근, 구현 그리고 그것들내 기능을 사용하는것을 좀더 쉽게 만들어 준다. 추가적으로 EJB에 의해 제공되는 서비스에 접근하기 위해 Spring을 사용하는 것은 변경되기 위한 클라이언트 코드없이 local EJB, remote EJB또는 갖가지 POJO사이에 나중에 투명하게 교체될 그런 서비스의 구현을 허용한다.

이 장에서 우리는 Spring이 어떻게 당신이 EJB에 접근하고 구현하는것을 돕는지 본다. Spring은 비상태유지(stateless) 세션빈(SLSBs)에 접근할때 특별한 값을 제공한다. 그래서 우리는 이것을 언급함으로써 시작할것이다.

16.1. EJB에 접근하기

16.1.1. 개념

local또는 remote 비상태유지(stateless) 세션빈의 메소드를 호출하기 위해, 클라이언트 코드는 (local또는 remote)EJB Home객체를 얻기위해 대개 JNDI룩업을 수행해야만 한다. 그 다음 실질적인 (local또는 remote)EJB객체를 얻기 위해 객체의 'create' 메소드 호출을 사용한다. 하나 이상의 메소드는 EJB에서 호출된다.

반복적인 하위레벨 코드를 파하기 위해 많은 EJB애플리케이션은 서비스 위치자(Locator)와 비지니스 위임 패턴을 사용한다. 클라이언트 코드 도처에 JDNI룩업을 하는것보다 더 좋다. 하지만 그들의 일반적인 구현물은 명백한 단점을 가진다. 예를 들면

☒ EJB를 사용한 전형적인 코드는 서비스 위치자나 비지니스 위임 패턴에 의존한다. 하지만 이것은 테스트를 좀더 어렵게 한다.

☒ 비지니스 위임이 없이 사용되는 서비스 위치자 패턴의 경우, 애플리케이션 코드는 여전히 EJB Home의 create()메소드를 호출하는것으로 끝이 나고 결과 예외를 다룬다. 게다가 이것은 EJB API와 EJB프로그래밍 모델의 복잡함에 묶인다.

☒ 비지니스 위임 패턴을 구현하는 것은 일반적으로 우리가 EJB의 같은 메소드를 간단히 호출하는 다양한 메소드를 써야만 하는 위치의 명백한 코드 중복의 결과를 낳는다.

Spring접근법은 대개 코드가 없는 비지니스 위임처럼 작동하는 Spring ApplicationContext나 BeanFactory내 설정되는 프록시 객체의 생성과 사용을 허용한다. 당신은 실제값을 추가하지 않는다면 다른 서비스 위치자, 다른 JNDI 룩업 또는 손으로 작성된 비지니스 위임내 중복 메소드를 사용할 필요가 없다.

16.1.2. local SLSBs에 접근하기

우리가 local EJB를 사용할 필요가 있는 웹 컨트롤러를 가지고 있다고 가정하자. 우리는 최상의 상황을

따를것이고 EJB 비지니스 메소드 인터페이스 패턴을 사용할것이다. 그래서 EJB의 local 인터페이스는 EJB 성격이 아닌 비지니스 메소드 인터페이스를 확장한다. 이 비지니스 메소드 인터페이스를 MyComponent라고 부르자.

```
public interface MyComponent {
    ...
}
```

(비지니스 메소드 인터페이스 패턴을 위한 중요한 이유중 하나는 local 인터페이스내 메소드 시그너처와 bean구현 클래스사이 동기화가 자동적이라는 것을 확인하는 것이다. 다른 이유는 이것이 나중에 우리는 위해 그렇게 하도록 만들때 서비스의 POJO로 교체하는것을 좀더 쉽게 만든다는 것이다.) 물론 우리는 local home인터페이스를 구현할 필요가 필요가 있을것이다. 그리고 SessionBean과 MyComponent비지니스 메소드 인터페이스를 구현하는 bean구현 클래스를 제공한다. 지금 우리의 웹 티어 컨트롤러를 EJB구현물로 연결하는 필요가 있는 자바코드만이 컨트롤러의 MyComponent타입의 setter메소드를 드러낸다. 이것은 컨트롤러내 인스턴스 변수처럼 참조를 저장할것이다.

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

우리는 컨트롤러내 어떠한 비지니스 메소드내 인스턴스 변수를 순차적으로 사용할수 있다. 지금 우리가 Spring ApplicationContext 나 BeanFactory밖에서 컨트롤러 객체를 얻는다고 가정하자. 우리는 EJB프록시 객체가 될 LocalStatelessSessionProxyFactoryBean를 설정하는 같은 컨텍스트를 사용할수 있다. 프록시의 설정은 그리고 컨트롤러의 myComponent 프라퍼티의 셋팅은 다음처럼 설정 항목으로 한다.

```
<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>
```

Spring AOP프레임워크의 도움으로 비록 당신이 이러한 결과를 즐기기 위해 AOP개념으로 작업을 강제로 하지 않더라도 이 상황뒤에는 마법같은 일이 많다. myComponent bean정의는 비지니스 메소드 인터페이스를 구현하는 EJB를 위한 프록시를 생성한다. EJBlocal home은 시작시 캐시된다. 그래서 하나의 JNDI룩업만이 있다. 각각의 EJB가 호출되는 시점에 프록시는 local EJB의 create()메소드를 호출하고 EJB의 관련된 비지니스 메소드를 호출한다.

myController bean정의는 프록시를 위한 컨트롤러의 myController 프라퍼티를 셋팅한다.

EJB접근 기법은 애플리케이션 코드의 굉장한 단순화를 초래한다. 웹 티어 코드(또는 다른 EJB클라이언트 코드)는 EJB사용의 의존성을 가지지 않는다. 만약 우리가 POJO나 모의(mock)객체 또는 다른 테스트 스텁(stub)을 가진 EJB참조를 교체하기를 원한다면 우리는 자바코드의 한줄의 변경도 없이 myComponent bean정의를 간단하게 변경할수 있다. 추가적으로 우리는 JNDI룩업을 한줄도 쓰지 않거나 우리의 애플리케이션의 일부처럼 다른 EJB 관련 코드를 쓰지 않는다.

실제 애플리케이션에서 벤치마크와 경험은 이 접근법(대상 EJB 반영적인 호출의 포함하는)의 의 성능 오버헤드가 최소이고 일반적인 사용에서 측정불가능이라는 것을 표시한다. 애플리케이션 서버내 EJB구조와

관련된 비용때문에 우리는 EJB를 위한 잘 정의된 호출을 만드는것을 원하지 않는다는것을 기억하라.

JNDI룩업에 관련되는 한가지 주의사항이 있다. bean컨테이너에서 이 클래스는 대개 싱글톤처럼(이것을 프로토타입으로 만들기 위한 이유는 없다) 사용되는것이 가장 좋다. 어쨌든 bean컨테이너가 싱글톤(XML ApplicationContext 변형을 하는것처럼) 을 먼저 인스턴스화 한다면 당신은 EJB컨테이너가 대상 EJB를 로드하기 전에 bean컨테이너가 로드된다면 문제를 가지게 된다. 그것은 JNDI룩업이 이 클래스의 init메소드내 수행될것이고 캐시되지만 EJB는 대상 위치에 여전히 바운드되지 않을것이기 때문이다. 해결법은 이 factory객체를 미리 인스턴스화하지 않지만 첫번째 사용시 이것이 생성되는것을 허용한다. XML컨테이너에서 이것은 lazy-init 속성을 통해 컨트롤된다.

비록 이것이 Spring사용자에게 중요 관심사가 되지는 않을것이지만 EJB를 사용한 프로그램에 따른 AOP작업을 하는것은 LocalSlsbInvokerInterceptor를 찾는것을 원할것이다.

16.1.3. remote SLSB에 접근하기

remote EJB에 접근하는것은 local EJB에 접근하는것과 비교해서 SimpleRemoteStatelessSessionProxyFactoryBean를 사용하는것을 제외하고 기본적으로 같다. 물론 Spring을 사용하든 사용하지 않든 remote호출은 의미적으로 적용한다. 다른 컴퓨터내 다른 VM안에서 객체의 메소드를 호출하는것은 때때로 사용 시나리오와 실패(failure)핸들링의 개념으로 다르게 처리된다.

Spring의 EJB 클라이언트 지원은 Spring을 사용하지 않는 접근법에 비해 하나 이상의 장점을 추가한다. 대개 EJB를 local이나 remote로 호출하는 것들간에는 쉽게 진행하고 되돌리기 위한 EJB클라이언트 코드를 위해서는 문제가 있다. 이것은 local인터페이스 메소드가 호출되지 않는동안 remote인터페이스 메소드가 RemoteException를 던지는 것을 명시해야하고 클라이언트 코드는 이것을 다루어야 하기 때문이다. remote EJB로 옮겨질 필요가 있는 local EJB를 위해 쓰여진 클라이언트 코드는 일반적으로 remote 예외를 위한 핸들링을 추가하기 위해 변경되고 local EJB로 옮겨질 필요가 있는 remote EJB를 위해 쓰여진 클라이언트 코드는 remote 예외의 많은 필요없는 핸들링이 수행되지만 같은코드로 그대로 유지될수 있거나 그 코드를 제거하기 위해 변경될 필요가 있다. Spring remote EJB프록시를 사용하여 당신은 당신의 비지니스 메소드 인터페이스내 던져지는 RemoteException을 선언하는것을 대신할수 있고 EJB코드를 구현한다. RemoteException를 던지는 것을 제외하면 동일한 remote 인터페이스를 가지고 그들이 같다면 두개의 인터페이스를 자동적으로 처리하기 위한 프록시에 의존한다. 클라이언트 코드는 체크된 RemoteException을 다루지 않는다. 어떠한 실질적인 RemoteException은 EJB호출이 RuntimeException의 하위클래스인 체크되지 않은 RemoteAccessException 클래스처럼 다시 던져질것이다. 대상 서비스는 그 다음 클라이언트 코드의 인식및 처리가 없이 local EJB나 remote EJB(또는 POJO) 구현물 사이에서 교체될것이다. 물론 이것은 옵션적이다. 당신의 비지니스 인터페이스내 선언된 RemoteExceptions으로 부터 당신을 정지시키는것은 아무것도 없다.

16.2. Spring의 편리한 EJB구현물 클래스를 사용하기.

Spring은 또한 당신이 EJB를 구현하도록 도와주는 편리한 클래스를 제공한다. EJB가 트랜잭션 설정과 원격작업을 책임지도록 놔둔체 POJO내 EJB뒤에 비지니스 로직을 두는 좋은 상황을 만들기 위해 디자인되었다.

비상태유지(stateless) 또는 상태유지(stateful) 세션빈, 또는 메시지빈을 구현하기 위해, 당신은 AbstractStatelessSessionBean, AbstractStatefulSessionBean, 그리고 AbstractMessageDrivenBean/AbstractJmsMessageDrivenBean으로 부터 반복적으로 당신의 구현 클래스를 끌어낸다.

구현물을 명확한 자바 서비스 객체로 실질적으로 위임하는 비상태유지(stateless) 세션빈을 시험하는것을 검토하라. 우리는 비지니스 인터페이스를 가진다.

```
public interface MyComponent {
    public void myMethod(...);
    ...
}
```

We have the plain java implementation object:

```
public class MyComponentImpl implements MyComponent {
    public String myMethod(...) {
        ...
    }
    ...
}
```

그리고 마지막으로 비상태유지(stateless) 세션빈 자체:

```
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(...);
    }
    ...
}
```

Spring EJB지원 기초 클래스는 그들의 생명주기처럼 BeanFactory(또는 이 경우 ApplicationContext의 하위클래스)를 디폴트로 생성하고 로드함으로써 이루어진다. 그 다음 EJB(예를 들면 POJO서비스 객체를 얻기 위한 위의 코드내에서 사용된것처럼)에 사용가능하게 된다. 로딩은 BeanFactoryLocator의 하위클래스인 전략(strategy)객체를 통해 이루어진다. BeanFactoryLocator의 실질적인 구현은 디폴트인 JNDI환경 변수(EJB의 경우, java:comp/env/ejb/BeanFactoryPath)처럼 명시된 자원 위치로부터 ApplicationContext을 생성하는 ContextJndiBeanFactoryLocator에 의해 사용된다. 만약 BeanFactory/ApplicationContext 로딩 전략을 변경할 필요가 없다면 디폴트 BeanFactoryLocator구현물은 setBeanFactoryLocator()메소드를 호출하거나 setSessionContext()내, 또는 EJB의 실질적인 생성자내에서 오버라이드되어 사용된다. 좀더 다양한 정보를 위해서는 JavaDoc를 보라.

JavaDoc에서 언급된것처럼 상태유지(stateful) 세션빈은 그들의 생명주기의 일부처럼 수동적이고 재활성화되기 위해 기대되고 EJB컨테이너에 의해 저장되지 않을수 있기 때문에 수동적이고 활성화된 BeanFactory를 로드하지않고 다시 로드하기 위한 ejbPassivate 과 ejbActivate로 부터 unloadBeanFactory() 와 loadBeanFactory를 수동으로 호출할 non-serializable한 BeanFactory/ApplicationContext 인스턴스를 사용한다.

EJB의 사용을 위해 ApplicationContext를 로드하기 위한 ContextJndiBeanFactoryLocator의 디폴트 사용법은 몇몇 상황을 위해 적절하다. 어쨌든 모든 EJB가 자신이 복사본을 가진후 ApplicationContext가 많은 수의 bean을 로드하거나 그러한 bean의 초기화가 시간을 소비하거나 메모리 집중적일때 문제가 있다. 이 경우 사용자는 디폴트 ContextJndiBeanFactoryLocator사용법을 오버라이드하길 원할것이고 다중 EJB또는 다른 클라이언트에 의해 사용되기 위한 공유 BeanFactory나 ApplicationContext를 로드하고 사용할수 있는 ContextSingletonBeanFactoryLocatore와 같은 다른 BeanFactoryLocator을 사용한다. 이것을 하는것은 EJB를 위해 유사한 코드를 추가함으로써 비교적 간단하다.

```
/**
 * 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);
}
```

그것들의 사용법에 대한 좀더 다양한 정보를 위해서 BeanFactoryLocator 와 ContextSingletonBeanFactoryLocatore를 위한 관련 JavaDoc를 보라.

Chapter 17. Spring을 사용한 원격(Remoting)및 웹서비스

17.1. 소개

원격 지원을 위한 Spring통합 클래스는 다양한 기술을 사용한다. 원격 지원은 당신의 (Spring) POJO에 의해 구현되는 원격-가능 서비스의 개발을 쉽게 한다. 현재 Spring은 4가지 원격 기술을 지원한다.

☒ 원격 메소드 호출 (RMI). RmiProxyFactoryBean 과 RmiServiceExporter의 사용을 통해 Spring은 전통적인 RMI(java.rmi.Remote 인터페이스와 java.rmi.RemoteException을 가지는)와 RMI 호출자(어떠한 자바 인터페이스를 가진)를 통한 투명한 원격 모두 지원합니다.
☒ Spring의 HTTP 호출자. Spring은 어떤 자바 인터페이스(RMI 호출자와 같은)를 지원하는 HTTP를 통한 자바 직렬화를 허용하는 특별한 원격 전략을 제공한다. 관련 지원 클래스는 HttpInvokerProxyFactoryBean 과 HttpInvokerServiceExporter이다.
☒ Hessian. HessianProxyFactoryBean과 HessianServiceExporter를 사용하여 Caucho에 의해 제공되는 가벼운 바이너리 HTTP기반 프로토콜을 사용하는 당신의 서비스를 투명하게 드러낼수 있다.
☒ Burlap. Burlap은 Hessian을 위한 Caucho의 XML기반의 대안이다. Spring은 BurlapProxyFactoryBean 과 BurlapServiceExporter 같은 지원 클래스를 제공한다.
☒ JAX RPC. Spring은 JAX-RPC를 통한 웹서비스를 위한 원격 지원을 제공한다.
☒ JMS (TODO).

Spring의 원격 기능에 대해 언급하는 동안 우리는 다음의 도메인 모델과 관련 서비스를 사용할것이다.

```
// 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
  }
}
```

우리는 RMI를 사용하여 원격 클라이언트에 서비스를 드러내길 시작하고 RMI를 사용한 장애에 대해 조금 이야기할것이다. 우리는 Hessian을 위한 예제를 보여줄 것이다.

17.2. RMI를 사용한 서비스 드러내기

RMI를 위한 Spring의 지원을 사용하여, 당신은 RMI내부구조를 통해 당신의 서비스를 투명하게 드러낼수 있다. 이 셋업 후 당신은 보안 컨텍스트 위임이나 원격 트랜잭션 위임을 위한 표준적인 지원이 없다는 사실을 제외하고 기본적으로 remote EJB와 유사한 설정을 가진다. Spring은 RMI호출자를 사용할때 추가적인 호출 컨텍스트와 같은 것을 위한 고리(hooks)를 제공한다. 그래서 당신은 예를 들어 보안 프레임워크나 사용자정의 보안 증명에 붙일수 있다.

17.2.1. RmiServiceExporter를 사용하여 서비스 내보내기

RmiServiceExporter를 사용하여, 우리는 RMI객체처럼 AccountService객체의 인터페이스를 드러낼수 있다. 인터페이스는 RmiProxyFactoryBean을 사용하거나 전통적인 RMI서비스의 경우 일반적인 RMI를 통해 접근될수 있다. RmiServiceExporter는 RMI호출자를 통해 RMI가 아닌 서비스의 발생을 명시적으로 지원한다.

우리는 먼저 Spring BeanFactory내 우리의 서비스를 셋업한다.

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

그리고 나서 우리는 RmiServiceExporter를 사용하여 우리의 서비스를 드러낼것이다.

```
<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>
```

당신이 볼수 있는 것처럼, 우리는 RMI등록(registry)을 위한 포트를 오버라이딩한다. 종종, 당신의 애플리케이션 서버는 RMI등록을 유지하고 그것을 방해하지 않는것이 현명하다. 게다가 서비스 이름은 서비스를 바인드 하기 위해 사용된다. 서비스는 rmi://HOST:1199/AccountService에 비인드될것이고 우리는 클라이언트측에서 서비스로 링크하기 위해 이 URL을 사용할것이다.

노트 : 우리는 하나의 프라퍼티를 남겨두었다. 이를테면 servicePort이고 디폴트로에 의해 0이된다. 이것은 서비스와 통신하기 위해 사용될 익명 포트를 의미한다. 당신은 다른 포트를 명시할수 있다.

17.2.2. 클라이언트에서 서비스 링크하기

우리의 클라이언트는 계좌(accounts)를 관리하기 위한 AccountService을 사용하는 간단한 객체이다.

```
public class SimpleObject {
  private AccountService accountService;
  public void setAccountService(AccountService accountService) {
    this.accountService = accountService;
  }
}
```

클라이언트에서 서비스에 링크하기 위해, 우리는 간단한 객체와 서비스 링크 설정을 포함하는 분리된 bean factory를 생성할 것이다.

```
<bean class="example.SimpleObject">
        <property name="accountService"><ref bean="accountService"/></property>
</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>
```

그것은 우리가 클라이언트에서 원격 계좌(account)서비스를 지원하기 위해 해야 할 필요가 있는 모든것이다. Spring은 호출자를 투명하게 생성하고 RmiServiceExporter를 통해 계좌 서비스를 원격적으로 가능하게 한다. 클라이언트에서 우리는 RmiProxyFactoryBean를 사용하여 이것을 링크한다.

17.3. HTTP를 통해 서비스를 원격으로 호출하기 위한 Hessian 이나 Burlap을 사용하기.

Hessian은 바이너리 HTTP-기반 원격 프로토콜을 제공한다. 이것은 Caucho에 의해 생성되었고 Hessian자체에 대한 좀더 상세한 정보는 http://www.caucho.com에서 찾을수 있다.

17.3.1. Hessian을 위해 DispatcherServlet을 묶기.

Hessian은 HTTP를 통해 통신하고 사용자정의 서블릿을 사용해서도 그렇게 한다. Spring의 DispatcherServlet 원리를 사용하여, 당신의 서비스를 드러내는 서블릿을 쉽게 묶을수 있다. 먼저 우리는 애플리케이션내 새로운 서블릿을 생성해야만 한다. (이것은 web.xml으로 부터 인용한다.)

```
<servlet>
```

```
        <servlet-name>remoting</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
        <servlet-name>remoting</servlet-name>
        <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

당신은 아마도 Spring의 DispatcherServlet 원리에 익숙하고 만약 그렇다면 당신은 WEB-INF 디렉토리내 (당신의 서블릿 이름 뒤) remoting-servlet.xml라는 이름의 애플리케이션 컨텍스트를 생성할것이다. 애플리케이션 컨텍스트는 다음 부분에서 사용될것이다.

17.3.2. HessianServiceExporter를 사용하여 bean을 드러내기

remoting-servlet.xml 라고 불리는 새롭게 생성된 애플리케이션 컨텍스트내에서 우리는 당신의 서비스를 내보내는 HessianServiceExporter를 생성할것이다.

```
<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>
```

지금 우리는 클라이언트에서 서비스를 링크할 준비가 되어있다. 명시된 핸들러 맵핑은 없고 서비스를 향한 요청 URL을 맵핑한다. 그래서 BeanNameUrlHandlerMapping은 사용될것이다. 나아가 서비스는 bean이름(http://HOST:8080/remoting/AccountService)을 통해 표시되는 URL에 내보내어질것이다.

17.3.3. 클라이언트의 서비스로 링크하기

HessianProxyFactoryBean을 사용하여 우리는 클라이언트에서 서비스로 링크할수 있다. 같은 원리는 RMI예제처럼 적용한다. 우리는 분리된 bean factory나 애플리케이션 컨텍스트를 생성하고 SimpleObject가 계좌를 관리하기 위해 AccountService를 사용하는 다음 bean을 따른다.

```
<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. Burlap 사용하기

우리는 Hessian의 XML기반의 대응물인 Burlap을 여기서 상세하게 언급하지 않을것이다. Hessian처럼 정확하게 같은 방법으로 설정되고 셋업된 후 변형물은 위에서 설명된다. Hessian 이라는 단어를 Burlap으로 교체하고 당신은 모든것을 셋팅한다.

17.3.5. Hessian 이나 Burlap을 통해 드러나는 서비스를 위한 HTTP 기본 인증 적용하기

Hessian 과 Burlap 장점중 하나는 두가지 프로토콜이 모두 HTTP-기반이기 때문에 우리가 HTTP 기본 인증을 쉽게 적용할수 있다는 것이다. 당신의 일반적인 HTTP서버의 보안 기법은 web.xml 보안 기능을 사용하여 쉽게 적용될수 있다. 대개 당신은 여기서 사용자별 보안 증명을 사용하지 않지만 공유된 증명은 Hessian/BurlapProxyFactoryBean 레벨(JDBC 데이터소스와 유사한)에서 정의된다.

```
<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>
```

이것은 우리가 BeanNameUrlHandlerMapping을 명시적으로 언급하고 오직 관리자만을 허용하는 인터셉터를 셋팅하며 이 애플리케이션 컨텍스트내에서 언급된 bean을 호출하는 작업을 수행하는 예제이다.

노트 : 물론, 이 예제는 보안 내부구조의 유연한 종류를 보여주지 않는다. 보안이 관련된 만큼 좀더 많은 옵션을 위해서 Spring을 위한 Acegi 보안 시스템을 보라. 그것은 http://acegisecurity.sourceforge.net에서 찾을수 있다.

17.4. HTTP호출자를 사용하여 서비스를 드러내기

그들 자체의 얼마 안되는 직렬화 기법을 사용하는 가벼운 프로토콜인 Burlap 과 Hessian이 상반된것처럼 Spring HTTP호출자는 HTTP를 통해 서비스를 드러내기 위한 표준적인 자바 직렬화 기법을 사용한다. 이것은 당신의 인자와 반환 타입이 Hessian 과 Burlap이 사용하는 직렬화 기법을 사용하여 직렬화될수 없는 복합(complex)타입이라면 커다란 장점을 가진다. (원격 기술을 선택할때 좀더 많은 숙고사항을 위해서 다음 부분을 참조하라.)

Spring은 HTTP호출을 수행하거나 Commons HttpClient를 위해 J2SE에 의해 제공되는 표준적인 기능을

사용한다. 만약 당신이 좀더 향상되었거나 사용하기 쉬운 기능이 필요하다면 후자를 사용하라. 좀더 많은 정보를 위해서는 jakarta.apache.org/commons/httpclient [http://jakarta.apache.org/commons/httpclient]을 참조하라.

17.4.1. 서비스 객체를 드러내기

Hessian 이나 Burlap을 사용하는 방법과 매우 유사한 서비스 객체를 위한 HTTP 호출자 내부구조를 셋업하라. Hessian지원이 HessianServiceExporter를 제공하는 것처럼 Spring Http호출자 지원은 org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter라고 불리는 것을 제공한다. (위에서 언급된) AccountService을 드러내기 위해 다음 설정이 대체될 필요가 있다.

```
<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. 클라이언트에서 서비스 링크하기

다시 클라이언트로부터 서비스를 링크하는것은 Hessian 이나 Burlap을 사용할때 이것을 하는 방법과 매우 유사하다. 프록시를 사용하여 Spring은 내보내어진 서비스를 위한 URL을 위해 당신의 호출을 HTTP POST요청으로 번역할수 있을것이다.

```
    <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>
```

전에 언급된것처럼, 당신은 사용하고자 하는 HTTP클라이언트를 선택할수 있다. 디폴트에 의하면 HttpInvokerProxy가 J2SE HTTP기능을 사용한다. 하지만 당신은 httpInvokerRequestExecutor 프라퍼티를 셋팅하여 Commons HttpClient를 사용할수 있다.

```
<property name="httpInvokerRequestExecutor">
        <bean class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerRequestExecutor"/>
</property>
```

17.5. 웹 서비스

Spring은 다음을 위한 지원을 가진다.

☒
위의 지원 목록 다음으로, 당신은 XFire xfire.codehaus.org [http://xfire.codehaus.org]를 사용하여 웹 서비스를 드러낼수 있다. XFire는 Codehaus에서 현재 개발중인 가벼운 SOAP라이브러리이다.

17.5.1. JAX-RPC를 사용하여 서비스를 드러내기

Spring은 JAX-RPC 서블릿 endpoint구현물(ServletEndpointSupport)을 위한 편리한 base클래스이다. 우리의 AccountService를 드러내기 위해 우리는 Spring의 ServletEndpointSupport클래스를 확장하고 여기서 언제나 비지니스 레이어로 호출을 위임하는 비지니스 로직을 구현한다.

```
/**
 * 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);
    }

}
```

우리의 AccountServletEndpoint는 Spring의 기능에 접근하기 위해 허용하는 Spring컨텍스트처럼 같은 웹 애플리케이션내에서 수행할 필요가 있다. Axis의 경우, AxisServlet정의를 web.xml로 복사하고 "server-config.wsdd" 내 endpoint를 셋업한다.(또는 배치툴을 사용한다.) Axis를 사용한 웹 서비스처럼 드러나는 OrderService가 있는 샘플 애플리케이션 JPetStore를 보라.

17.5.2. 웹 서비스에 접근하기

Spring은 웹 서비스 프록시인 LocalJaxRpcServiceFactoryBean 과 JaxRpcPortProxyFactoryBean을 생성하기 위한 두가지 factory bean을 가진다. 전자는 JAX-RPC서비스 클래스만을 반환할수 있다. 후자는 우리의 비지니스 서비스 인터페이스를 구현하는 프록시를 반환할수 있는 완전한 버전이다. 이 예제에서 우리는 이전 단락내 나오는 AccountService Endpoint를 위한 프록시를 생성하기 위해 나중에 사용된다. 당신은 Spring이 조금의 코딩을 요구하는 웹 서비스를 위한 대단한 지원을 가진다는 것을 볼것이다. 대부분의 마법은 대개 Spring설정 파일내에서 이루어진다.

```
<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>
```

serviceInterface는 클라이언트가 사용할 원격 비지니스 인터페이스이다. wsdlDocumentUrl은 WSDL파일을 위한 URL이다. Spring은 JAX-RPC서비스를 생성하기 위해 시작시 이것이 필요하다. namespaceUri는 .wsdl파일내 targetNamespace에 대응한다. serviceName은 .wsdl파일내 서비스 이름에 대응한다. portName은 .wsdl파일내 포트명에 대응한다.

웹 서비스에 접근하는 것은 우리가 RemoteAccountService인터페이스처럼 이것을 드러내는 bean factory를 가지는것처럼 매우 쉽다. 우리는 Spring내 이것을 묶을수 있다.

```
    <bean id="client" class="example.AccountClientImpl">
        ...
        <property name="service">
            <ref bean="accountWebService"/>
        </property>
    </bean>
```

그리고 클라이언트 코드로 부터 우리는 이것이 RemoteException을 던지는것을 제외하고 일반 클래스인것처럼 웹 서비스에 접근할수 있다.

```
public class AccountClientImpl {

    private RemoteAccountService service;

    public void setService(RemoteAccountService service) {
        this.service = service;
    }

    public void foo() {
       try {
            service.insertAccount(...);
        } catch (RemoteException e) {
            // ouch
            ...
        }
     }

}
```

우리는 Spring이 관련된 체크되지 않은 RemoteAccessException으로의 자동변환을 지원하기 때문에 체크된 RemoteException을 제거할수 있다. 이것은 우리가 비-RMI인터페이스 또한 제공하는것을 요구한다. 우리의 설정은 다음과 같다.

```
    <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>
```

serviceInterface는 비-RMI 인터페이스를 위해 변경된다. 우리의 RMI 인터페이스는 portInterface 프라퍼티를 사용하여 정의된다. 우리의 클라이언트 코드는 java.rmi.RemoteException을 피할수 있다.

```
public class AccountClientImpl {

    private AccountService service;

    public void setService(AccountService service) {
        this.service = service;
    }

    public void foo() {
        service.insertAccount(...);
     }

}
```

17.5.3. Register Bean 맵핑

Account와 같은 정보를 넘어 복합 객체를 이동시키기 위해 우리는 클라이언트 측에서 bean맵핑을 등록해야만 한다.

Note

서버측에서 Axis를 사용하여 등록된 bean맵핑은 server-config.wsdd에서 언제나 수행된다. 우리는 클라이언트 측에서 bean맵핑을 등록하기 위해 Axis를 사용할것이다. 이것을 하기 위해 우리는 Spring Bean factory의 하위클래스를 만들 필요가 있고 프로그램에 따라 bean맵핑을 등록한다.

```
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. 자체적인 핸들러 등록하기

이 장에서 우리는 SOAP메시지를 정보를 통해 보내기 전에 코드를 사용자정의 할수 있는 웹 서비스

프록시를 위한 javax.rpc.xml.handler.Handler를 등록할 것이다. javax.rpc.xml.handler.Handler는 콜백 인터페이스이다. jaxrpc.jar내 제공되는 편리한 base클래스인 javax.rpc.xml.handler.GenericHandler가 있다.

```
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;
    }

}
```

우리의 AccountHandler를 JAX-RPC 서비스에 등록하는 것이 필요하다. 그래서 메시지가 정보를 통해 전달되기 전에 handleRequest을 호출할것이다. Spring은 이 시점에 핸들러를 등록하기 위한 선언적인 지원을 가지지 않는다. 그래서 우리는 프로그램마다 다른 접근법을 사용해야만 한다. 어쨌든 Spring은 우리가 이 bean factory를 확장하고 postProcessJaxRpcService 메소드를 오버라이드 할수 있는 것처럼 이것을 쉽게 만든다.

```
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);
    }

}
```

그리고 마지막으로 우리는 factory bean을 사용하기 위해 Spring설정을 변경하는 것을 기억해야만 한다.

```
    <bean id="accountWebService" class="example.AccountHandlerJaxRpcPortProxyFactoryBean">
        ...
    </bean>
```

17.5.5. XFire를 사용하여 웹 서비스를 드러내기

XFire는 Codehaus에서 호스팅되는 가벼운 SOAP라이브러리이다. 현 시점(2005년 3월)에, XFire는 여전히 개발중이다. 비록 Spring지원이 안정적이라고 하더라도 대부분의 기능은 나중에 추가될것이다. XFire를 드러내는 것은 당신이 WebApplicationContext에 추가할 RemoteExporter-스타일의 bean으로

조합된 XFire를 가진 XFire 컨텍스트를 사용하는 것이다.

당신이 서비스를 드러내는 것을 허용하는 모든 메소드처럼 당신은 드러낼 서비스를 포함하는 관련된 WebApplicationContext를 가진 DispatcherServlet을 생성해야 한다.

```
<servlet>
  <servlet-name>xfire</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
```

당신은 XFire설정을 링크해야만 한다. 이것은 ContextLoaderListener(또는 서블릿)가 가지는 contextConfigLocations 컨텍스트 파라미터에 컨텍스트 파일을 추가하는것이다. 설정 파일은 XFire jar파일내 위치하고 물론 애플리케이션의 클래스패스에 위치할수도 있다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:org/codehaus/xfire/spring/xfire.xml
  </param-value>
</context-param>

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

당신이 서블릿 맵핑(위에서 선언된 XFire서블릿을 위한 /* 맵핑)을 추가한 후에 당신은 오직 XFire를 사용하는 서비스를 드러내기 위한 추가적인 bean을 추가해야만 한다. 예를 들어 당신은 xfire-servlet.xml을 다음에 추가하라.

```
<beans>
  <bean name="/Echo" class="org.codehaus.xfire.spring.XFireExporter">
    <property name="service">
      <ref bean="echo"/>
    </property>
    <property name="serviceInterface">
      <value>org.codehaus.xfire.spring.Echo</value>
    </property>
    <property name="serviceBuilder">
      <ref bean="xfire.serviceBuilder"/>
    </property>
    <!-- the XFire bean is wired up in the xfire.xml file you've linked in earlier
    <property name="xfire">
      <ref bean="xfire"/>
    </property>
  </bean>

  <bean id="echo" class="org.codehaus.xfire.spring.EchoImpl"/>
</beans>
```

XFire는 나머지를 다룬다. 이것은 당신의 서비스 인터페이스를 분석하고 이것으로 부터 WSDL을 생성한다.

이 문서의 일부는 XFire사이트로부터 가져왔다. XFire와 Spring통합에 대한 좀더 상세한 정보는 docs.codehaus.org/display/XFIRE/Spring [http://docs.codehaus.org/display/XFIRE/Spring]를 보라.

17.6. 자동-탐지(Auto-detection)는 원격 인터페이스를 위해 구현되지 않는다.

구현된 인터페이스의 자동-탐지가 원격 인터페이스에는 발생하지 않는 가장 중요한 이유는 원격 호출자를 위해 너무 많은 문이 열리는 것을 피하는 것이다. 대상 객체는 호출자에게 드러내는 것을 원하지 않는 InitializingBean 이나 DisposableBean처럼 내부 콜백 인터페이스를 구현해야만 한다.

대상에 의해 구현된 모든 인터페이스를 가진 프록시를 제공하는 것은 로컬의 경우 언제나 문제가 되지 않는다. 하지만 원격 서비스를 내보낼때 당신은 원격 사용의 경향이 있는 특정 작업을 가진 특정 서비스 인터페이스를 보여야만 한다. 내부 콜백 인터페이스외에도 대상은 원격 노출의 경향이 있는 것중 하나를 가진 다중 비지니스 인터페이스를 구현해야만 한다. 이러한 이유로 우리는 명시되는 서비스 인터페이스를 요구한다.

이것은 설정의 편리함과 내부 메소드의 뜻하지 않는 노출의 위험사이의 거래이다. 서비스 인터페이스를 명시하는 것은 많은 노력이 필요하지 않고 당신을 특정 메소드의 제어된 노출에 관련된 안전적인 쪽에 두게된다.

17.7. 기술을 선택할때 고려사항.

여기에 표시된 각각 그리고 모든 기술은 결점을 가진다. 당신이 기술을 선택할때 당신이 드러내는 서비스와 당신이 정보를 통해 보낼 객체중 필요한 것을 주의깊게 검토해야만 한다.

RMI를 사용할때, 당신이 RMI 소통을 관통(tunneling)하지 않는 한 HTTP프로토콜을 통해 객체에 접근하는 것은 불가능하다. RMI는 정보를 통해 직렬화가 필요한 복합 데이터 모델을 사용할때 중요한 완전한 객체 직렬화를 지원하는 상당히 무거운 프로토콜이다. 어쨌든 RMI-JRMP는 자바 클라이언트에 묶인다. 이것은 자바-대-자바 원격 솔루션이다.

Spring의 HTTP호출자는 만약 당신이 HTTP-기반 원격이 필요하지만 자바 직렬화에 의존한다면 좋은 선택이다. 이것은 수송기처럼 HTTP를 사용하는 RMI호출자를 가진 기본 내부구조를 공유한다. HTTP호출자는 자바-대-자바 원격에 제한을 가지지 않을뿐 아니라 클라이언트측과 서버측 모두 제한을 가하지 않는다. (후자는 비-RMI인터페이스를 위해 Spring의 RMI호출자에 적용한다.)

Hessian 그리고/또는 Burlap은 명시적으로 비-자바 클라이언트를 허용하기 때문에 이종 환경내에서 작동할때 명백한 값을 제공한다. 어쨌든 비-자바 지원은 여전히 제한된다. 알려진 문제는 늦게 초기화하는 collection으로 조합된 Hibernate객체의 직렬화를 포함한다. 만약 당신이 그러한 데이타 모델을 가진다면 Hessian대신에 RMI나 HTTP호출자를 사용하는 것을 검토하라.

JMS는 서비스의 클러스터(clusters)를 제공하기 위해 유용할수 있고 로드 밸런싱, 발견(discovery) 그리고 자동 대체(failover)를 다루기 위해 JMS 브로커(broker)를 허용한다. 디폴트에 의해 자바 직렬화는 JMS원격을 사용하지만 JMS제공자가 서버가 다른 기술로 구현되는것을 허용하는 XStream과 같은 포맷팅을 묶기 위한 다른 기법을 사용할수 있을때 사용된다.

EJB는 표준적인 권한(role)-기반 인증과 인증, 그리고 원격 트랜잭션 위임을 지원하는 면에서 RMI를 능가하는 장점을 가진다. 이것은 비록 핵심 Spring에 의해 제공되지는 않지만 보안 컨텍스트 위임을

지원하는 RMI호출자나 HTTP호출자를 얻는것은 가능하다. 써드 파티나 사용자정의 솔루션내 플러그인하기 위한 선호하는 고리(hooks)가 있다.

Chapter 18. Spring 메일 추상 계층을 사용한 이메일 보내기

18.1. 소개

Spring 은 전자메일을 보내기 위한 높은 수준의 추상화를 제공하는데, 이것은 사용자들이 기반 메일링 시스템에 대한 명세서가 필요 없도록 해주며, 고객을 대신하여 낮은 레벨의 리소스 핸들링에 대한 책임을 진다.

18.2. Spring 메일 추상화 구조

Spring 메일 abstraction 계층의 메인 패키지는 org.springframework.mail 패키지이다. 이것은 메일을 보내기 위한 주된 인터페이스인 MailSender와, from, to, cc, subject, text와 같은 간단한 메일의 속성들을 캡슐화하는 값객체(value object)인 SimpleMailMessage를 포함하고 있다. 이 패키지는 MailException.을 루트로 하는 체크된 예외들의 계층을 포함하고 있는데, 이 예외들은 낮은 레벨의 메일 시스템 예외들에 대해 보다 상위의 추상화를 제공한다. 메일 예외계층에 대한 더 많은 정보는 JavaDocs을 참조하길 바란다.

Spring은 또한 MIME 메시지와 같이 JavaMail의 특징에 특화된 MailSender의 서브 인터페이스인 org.springframework.mail.javamail.JavaMailSender를 제공한다. Spring은 또한 JavaMail MIME 메시지들의 준비를 위한 callback 인터페이스인 org.springframework.mail.javamail.MimeMessagePreparator를 제공한다.

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. Spring 메일 추상화 사용하기

OrderManager라는 비즈니스 인터페이스가 있다고 가정해보자.

```
public interface OrderManager {

    void placeOrder(Order order);
```

```
}
```

그리고, 주문번호를 가진 이메일 메시지가 생성되어 그 주문을 한 고객에게 보내져야만 한다는 유스케이스가 있다고도 가정해보자. 그렇다면 이 목적을 달성하기 위해 우리는 MailSender와 SimpleMailMessage를 사용할 것이다.

일반적으로, 우리는 비즈니스 코드에서 인터페이스들을 사용하게 될 것이고, Spring IoC 컨테이너로 하여금 우리에게 필요한 모든 협력자(구현 클래스)들을 다루도록 할 것임을 전제하고 있다.

아래에 OrderManager의 구현클래스가 있다.

```
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());
        }
    }
}
```

그리고 위의 코드에 대한 bean 정의는 다음과 같을 것이다.

```
<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>
```

이제 MimeMessagePreparator callback 인터페이스를 사용한 OrderManager의 구현클래스를 보자. 아래의 경우 JavaMail MimeMessage를 사용할 수 있도록 mailSender 속성이 JavaMailSender 타입이라는 점에 주의해야 한다.

```
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());
        }
    }
}
```

만약 당신이 JavaMail MimeMessage의 모든 것을 사용하려 한다면, MimeMessagePreparator를 이용할 수 있다.

위의 메일코드는 단지 대조적인 하나의 방법이며, (Spring 메일 추상화를 사용한 메일코드는) 임의의

Spring AOP를 통한 리팩토링을 완벽하게 지원하고 있기 때문에, OrderManager 타겟에 쉽게 적용될 수 있다. 이 문제에 대해서는 AOP 챕터를 보도록 하라.

18.3.1. 플러그인할 수 있는 MailSender 구현클래스들

Spring 은 2개의 MailSender 구현클래스를 부수적으로 가지는데, 하나는 JavaMail 구현체이고 다른 하나는 http://servlets.com/cos (com.oreilly.servlet)에 포함된 Jason Hunter 의 MailMessage에 기반한 구현 클래스이다. 더 많은 정보는 JavaDocs을 참조하도록 하라.

18.4. JavaMail MimeMessageHelper 사용하기

JavaMail message를 다룰 때 가장 간편한 컴포넌트들 가운데 하나는 org.springframework.mail.javamail.MimeMessageHelper이다. 이것은 당신으로 하여금 귀찮은 javax.mail.internet 클래스들의 API들을 사용하지 않도록 도와준다. 가능한 2개의 시나리오는 다음과 같다.

18.4.1. 간단한 MimeMessage 를 생성하고 보내기

MimeMessageHelper를 사용하면, MimeMessage를 셋업하고 보내는 것이 매우 간단해진다.

```
// 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. 첨부파일들과 inline 리소스들을 보내기

이메일은 첨부파일뿐만 아니라 멀티파트 메시지 속의 inline 리소스들을 허용한다. inline 리소스들은 예를 들어 이미지, 스타일시트와 같이 당신이 메시지 속에서 사용하려고 하지만 첨부파일로 명시되는 것은 원하지 않는 리소스들을 말한다. 다음의 코드는 MimeMessageHelper를 사용하여 inline 이미지를 이메일에 딸려 보내는 방법을 보여준다.

```
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 리소스들은 위에서 본 바와 같이 (위의 경우 identifier1234) Content-ID를 사용하여 mime message에 첨부된다. 당신이 텍스트와 리소스를 추가하는 순서는 매~우 중요하다. 먼저 텍스트를 추가하고 이후에 리소스를 추가해야 한다. 만약 당신이 다른 방식으로 한다면, 결코 동작하지 않을 것이다!

Chapter 19. Quartz 혹은 Timer 를 사용한 스케쥴링

19.1. 소개

Spring은 스케쥴링을 지원하는 통합 클래스들을 제공한다. 현재적으로, Spring은 1.3 이후버전 JDK의 일부분인 Timer와 Quartz 스케쥴러 (http://www.quartzscheduler.org)를 지원하고 있다. 이 두개의 스케쥴러들은 각각 Timer 혹은 Triggers에 대한 선택적 참조를 가지는 FactoryBean을 사용하여 세팅된다. 게다가 당신이 타겟 object의 메써드를 편리하게 호출할 수 있도록 도와주는 Quartz 스케쥴러와 Timer에 대한 편의 클래스를 제공한다.(이것은 일반적인 MethodInvokingFactoryBeans와 비슷하다.)

19.2. OpenSymphony Quartz 스케쥴러 사용하기

Quartz는 Triggers, Jobs 그리고 모든 종류의 jobs를 인식하고 있는 JobDetail를 사용한다. Quartz에 깔려 있는 기본적인 개념을 알고 싶다면, http://www.opensymphony.com/quartz를 찾아보길 바란다. 편리한 사용을 위해서, Spring은 Spring 기반 어플리케이션 내에서 Quartz의 사용을 손쉽게 만들어주는 두 개의 클래스들을 제공한다.

19.2.1. JobDetailBean 사용하기

JobDetail 객체는 job을 실행하기 위해 필요한 모든 정보를 가지고 있다. Spring은 소위 JobDetailBean이라고 불리는 클래스를 제공하는데, 이것은 JobDetail을 합리적인 디폴트값을 가진 실질적인 JavaBean 객체로 만들어준다. 다음의 예제를 보도록 하자.

```
<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>
```

위의 job detail bean은 job(ExampleJob)을 실행하기 위한 모든 정보를 가지고 있다. 타임아웃은 job data map으로 기술되었다. job data map은 (실행시 넘겨지는) JobExecutionContext를 통해 이용할 수 있지만, JobDetailBean 역시 job data map으로부터 실질적인 job의 프라퍼티들을 매핑할 수 있다. 때문에 이러한 경우, 만약 ExampleJob이 timeout이라는 프라퍼티를 가지고 있다면, JobDetailBean은 그것을 자동으로 적용할 것이다.

```
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
  }
}
```

당신은 job detail bean의 모든 부가적인 세팅들 역시 마찬가지로 이용할 수 있다.

주의: name과 group 프라퍼티를 사용함으로써, 당신은 job의 name과 group을 변경할 수 있다. default로 job의 이름은 job detail bean의 이름과 동일하다.(위의 예에서는 exampleJob이 된다.)

19.2.2. MethodInvokingJobDetailFactoryBean 사용하기

종종 당신은 특정한 객체의 메써드를 호출할 필요가 있을 것이다. 당신은 MethodInvokingJobDetailFactoryBean을 사용하여 다음과 같이 할 수 있다.

```
<bean id="methodInvokingJobDetail"
  class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject"><ref bean="exampleBusinessObject"/></property>
    <property name="targetMethod"><value>doIt</value></property>
</bean>
```

위의 예는 (아래에 있는) exampleBusinessObject의 doIt을 호출하는 것을 의미한다.

```
public class BusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

MethodInvokingJobDetailFactoryBean을 사용할 때, 메써드를 호출할 한줄짜리 jobs를 생성할 필요가 없으며, 당신은 단지 실질적인 비지니스 객체를 생성해서 그것을 묶기만 하면된다.

default로는 Quartz Jobs는 비상태이며, 상호 작용하는 jobs의 가능성을 가진다. 만약 당신이 동일한 JobDetail에 대해 두 개의 triggers를 명시한다면, 첫번째 job이 끝나기 이전에 두번째가 시작할지도 모른다. 만약 JobDetail 객체가 상태 인터페이스를 구현한다면, 이런 일은 발생하지 않을 것이다. 두번째 job은 첫번째가 끝나기 전에는 시작하지 않을 것이다. MethodInvokingJobDetailFactoryBean를 사용한 jobs가 동시작용하지 않도록 만들기 위해서는, concurrent 플래그를 false로 세팅해주어야 한다.

```
<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>
```

주의: 기본적으로 jobs는 concurrent 옵션에 따라 실행될 것이다.

19.2.3. triggers 와 SchedulerFactoryBean을 사용하여 jobs를 묶기

우리는 job details과 jobs를 생성했고, 당신이 특정 객체의 메써드를 호출할 수 있도록 하는 편의클래스 bean을 살펴보았다. 물론, 우리는 여전히 jobs를 그자체로 스케쥴할 필요가 있다. 이것은 triggers와 SchedulerFactoryBean을 사용하여 이루어진다. 여러가지 triggers는 Quartz 내에서 이용할 수 있다. Spring은 편의를 위해 2개의 상속받은 triggers를 기본적으로 제공한다.: CronTriggerBean과 SimpleTriggerBean이 그것이다.

Triggers는 스케쥴될 필요가 있다. Spring은 triggers를 세팅하기 위한 프라퍼티들을 드러내는 SchedulerFactoryBean을 제공하고 있다. SchedulerFactoryBean은 그 triggers와 함께 실질적인 jobs를 스케쥴한다.

다음 두가지 예를 보자.

```
<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, 이제 우리는 두 개의 triggers를 세팅했다. 하나는 10초 늦게 실행해서 매 50초마다 실행될 것이고, 다른 하나는 매일 아침 6시에 실행될 것이다. 모든 것을 완료하기 위해서, 우리는 SchedulerFactoryBean을 세팅해야 한다.

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <ref local="cronTrigger"/>
      <ref local="simpleTrigger"/>
```

```
    </list>
  </property>
</bean>
```

당신이 세팅할 수 있는 더욱 많은 속성들이 SchedulerFactoryBean에 있다. 이를테면, job details에 의해 사용되는 calendars라던가, Quartz를 커스터마이징할 수 있게 하는 프라퍼티같은 것들이 말이다. 더 많은 정보를 위해서는 JavaDoc(http://www.springframework.org/docs/api/org/springframework/scheduling/quartz/SchedulerFact 참조하도록 해라.

19.3. JDK Timer support 사용하기

Spring에서 스케쥴링 업무를 처리하는 또다른 방법은 JDK Timer 객체들을 사용하는 것이다. Timers 자체에 대한 더 많은 정보는 http://java.sun.com/docs/books/tutorial/essential/threads/timer.html에서 찾아볼 수 있다. 위에서 살펴 본 기본개념들은 Timer support에도 마찬가지로 적용된다. 당신은 임의의 timers를 생성하고 메써드들을 호출하기 위해 timer를 사용한다. TimerFactoryBean을 사용하여 timers를 묶는다.

19.3.1. 임의의 timers 생성하기

당신은 TimerTask를 사용하여 임의의 timer tasks를 생성할 수 있다. 이것은 Quartz jobs와 유사하다

```
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
  }
}
```

이것을 묶는 것 역시 간단하다:

```
<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>
```

task를 단지 한번만 실행하고자 한다면, period 속성을 -1(혹은 다른 음수값으)로 바꿔주면 된다.

19.3.2. MethodInvokingTimerTaskFactoryBean 사용하기

Quartz support와 비슷하게, Timer 역시 당신이 주기적으로 메써드를 호출할 수 있도록 하는 요소들을 기술한다.

```
<bean id="methodInvokingTask"
  class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject"><ref bean="exampleBusinessObject"/></property>
    <property name="targetMethod"><value>doIt</value></property>
</bean>
```

위의 예제는 (아래와 같은) exampleBusinessObject에서 호출되는 doIt에서 끝날 것이다

```
public class BusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

ScheduledTimerTask가 언급된 위의 예제의 참조값을 methodInvokingTask로 변경하면 이 task가 실행될 것이다.

19.3.3. 감싸기 : TimerFactoryBean을 사용하여 tasks를 세팅하기

TimerFactoryBean은 실질적인 스케쥴링을 세팅한다는 같은 목적을 제공한다는 점에서 Quartz의 SchedulerFactoryBean과 비슷하다. TimerFactoryBean는 실질적인 Timer를 세팅하고 그것이 참조하고 있는 tasks를 스케쥴한다. 당신은 대몬 쓰레드를 사용할 것인지 말것인지를 기술할 수 있다.

```
<bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">
  <property name="scheduledTimerTasks">
    <list>
      <!-- see the example above -->
      <ref local="scheduledTask"/>
    </list>
  </property>
</bean>
```

끝!

Chapter 20. JMX 지원

20.1. 소개

Spring내 JMX지원은 당신에게 쉽게 기능을 제공하고 당신의 Spring애플리케이션을 JMS내부구조와 투명하게 통합한다. 특별히 Spring JMS는 4가지의 핵심적인 기능을 제공한다.

☒ JMS MBean처럼 Spring bean의 자동 등록

☒ 당신 bean의 관리 인터페이스를 제어하기 위한 유연한 기법

☒ 원격, JSR-160 연결자를 넘어선 MBean의 명시적인 제시(exposure)

☒ local과 remote MBean자원 모두의 간단한 프록시화

이러한 기능들은 Spring이나 JMS인터페이스 그리고 클래스를 위한 애플리케이션 컴포넌트를 커플링하지 않는 작업을 위해 다지인되었다. 물론 당신 애플리케이션 클래스의 대부분을 위해 Spring JMX기능의 장점을 가져가기 위한 Spring이나 JMS를 인식할 필요는 없다.

20.2. 당신의 bean을 JMX로 내보내기(Exporting)

Spring JMX 프레임워크내 핵심(core) 클래스는 MBeanExporter이다. 이 클래스는 당신의 Spring bean을 가져오고 그것들을 JMX MBeanServer에 등록하는 책임이 있다. 예를 들어, 밑에서 보여지는 간단한 bean클래스를 보자.

```
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

    private String name;

    private int age;

    private boolean isSuperman;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int add(int x, int y) {
        return x + y;
    }
```

```
    public void dontExposeMe() {
        throw new RuntimeException();
    }

}
```

JMX MBean의 속성과 작동(operation)처럼 이 bean의 프라퍼티와 메소드를 드러내기 위해 당신은 당신의 설정파일내 MBeanExporter 클래스의 인스턴스를 간단하게 설정하고 밑에서 보여지는것처럼 bean으로 전달한다.

```
<beans>
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>
</beans>
```

여기서 중요한 정의는 exporter bean이다. beans 프라퍼티는 JMX MBeanServer에 내보내져야하는 당신의 bean이 어떠한 것인지 MBeanExporter에 알리기 위해 사용된다. beans 프라퍼티는 Map의 타입이고 게다가 당신은 내보낼 bean을 설정하기 위한 <map> 과 <entry> 태그들을 사용한다. 디폴트 설정에서 Map의 항목의 key는 항목의 값인 bean을 위한 ObjectName처럼 사용된다. 이 행위는 XXX부분에서 언급된것처럼 변경될수 있다.

testBean bean를 설정하는것은 ObjectNamebean:name=testBean1 하위의 JMX MBean처럼 드러난다. bean의 모든 public성격의 프라퍼티는 속성과 모든 public성격의 메소드(Object내 정의된)가 작동(operation)처럼 드러나는 만큼 드러난다.

20.2.1. MBeanServer 생성하기

위에서 보여지는 설정은 애플리케이션이 이미 실행중인 하나의 MBeanServer를 가진 환경에서 실행중이라는것을 가정한다. 이 경우 Spring은 실행중인 MBeanServer를 할당하고 그것과 함께 당신의 bean을 등록할것이다. 이것은 당신의 애플리케이션이 자신만의 MBeanServer를 가진 톰캣이나 IBM웹스피어와 같은 컨테이너내부에서 실행중일때 유용하다.

어쨌든, 이 접근법은 독립형(standalone) 환경이나 MBeanServer를 제공하지 않는 컨테이너내부에서 실행될때는 필요없다. 이것을 극복하기 위해 당신은 당신의 설정을 위한 org.springframework.jmx.support.MBeanServerFactoryBean의 인스턴스를 추가하여 선언적으로 MBeanServer인스턴스를 생성할수 있다. 당신은 또한 MBeanServer가 MBeanExporter의 서버 프라퍼티를 셋팅하기 위해서 MBeanServerFactoryBean를 사용하는지 확인할수 있다. 이것은 밑에서 보여준다.

```
<beans>
```

```
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="server">
      <ref local="mbeanServer"/>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"/>
</beans>
```

여기, MBeanServer의 인스턴스는 MBeanServerFactoryBean에 의해 생성되고 서버 프라퍼티를 통해 MBeanExporter에 제공된다. 당신이 당신 자신의 MBeanServer를 제공할때 MBeanServer는 수행을 할당하기 위해 시도하지 않을것이다. 이 작업을 정확하게 하기 위해 당신은 클래스패스에 JMX구현물의 위치시켜야만 한다.

20.2.2. 늦게 초기화되는(Lazy-Initialized) MBeans

만약 당신이 MBeanExporter을 가진 bean을 설정한다면 그것은 늦은(lazy) 초기화를 위해 설정되고 그 다음 MBeanExporter는 이 규칙을 깨지 않을것이고 bean을 인스턴스화하는것을 피할것이다. 대신 이것은 MBeanServer을 가지고 프록시를 등록할것이고 발생한 프록시의 첫번째 호출까지 BeanFactory로 부터 bean을 얻는것을 미룰것이다.

20.2.3. MBean의 자동 등록

MBeanExporter와 이미 유효한 MBean을 통해 내보내어진 bean은 Spring으로 부터 더이상의 조정(intervention) 없는 MBeanServer처럼 등록된 이미 유효한 MBean이다. MBean은 autodetect 프라퍼티를 true로 셋팅하여 MBeanExporter에 의해 자동적으로 감지될수 있다.

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="autodetect">
    <value>true</value>
    </property>
</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.export.TestDynamicMBean"/>
```

여기에, spring:mbean=true 라고 불리는 bean은 벌써 유효한 JMX MBean이고 Spring에 의해 자동적으로 등록된것이다. 디폴트에 의하면 JMX등록을 위해 자동감지된 bean은 ObjectName처럼 사용된 bean이름을 가진다. 이 행위는 XXX부분에서 상세화된것처럼 오버라이드될수 있다.

20.3. 당신 bean의 관리 인터페이스를 제어하기

이전의 예제에서 당신은 모든 public성격의 프라퍼티와 메소드를 가진 bean의 관리 인터페이스를 조금 제어했다. 이 문제를 해결하기 위해 Spring JMX는 당신 bean의 관리 인터페이스를 제어하기 위한 포괄적이고 확장가능한 기법을 제공한다.

20.3.1. MBeanInfoAssembler 인터페이스

이 장면뒤에 MBeanExporter는 드러난 각각의 bean의 관리 인터페이스를 명시하는 책임을 지닌 org.springframework.jmx.export.assembler.MBeanInfoAssembler 인터페이스의 구현물을 위임한다. 디폴트 구현물인 org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler는 당신이 이전 예제에서 본것처럼 모든 public성격의 프라퍼티와 메소드를 드러내는 인터페이스를 간단하게 정의한다. Spring은 소스레벨 메타데이타(metadata)나 어느 임의의 인터페이스를 사용하여 관리 인터페이스를 제어하도록 허용하는 MBeanInfoAssembler 인터페이스의 두개의 추가적인 구현물을 제공한다.

20.3.2. 소스레벨 메타데이타(metadata) 사용하기

MetadataMBeanInfoAssembler를 사용하여 당신은 소스레벨 메타데이타를 사용하는 당신의 bean을 위한 관리 인터페이스를 정의할수 있다. 메타데이타 읽기는 org.springframework.jmx.export.metadata.JmxAttributeSource인터페이스에 의해 캡슐화된다. 특히 Spring JMX는 Commons Attributes를 위한 AttributesJmxAttributeSource인터페이스와 JDK 5.0 annotations을 위한 AnnotationsAttributeSource인터페이스의 두가지 구현물을 위한 지원을 제공한다. MetadataMBeanInfoAssembler는 정확하게 기능을 위한 JmxAttributeSource의 구현물이 설정되어야만 한다. 이 예제를 위해 우리는 Commons Attributes 메타데이타 접근법을 사용할것이다.

JMX로 내보내기위한 bean을 표시하기 위해 당신은 ManagedResource 속성으로 bean의 클래스에 주석(annotate)을 달아야 한다. Commons Attributes 메타데이타 접근법의 경우 이 클래스는 org.springframework.jmx.metadata 패키지에서 찾을수 있다. 당신이 작동(operation)처럼 드러내기를 바라는 각각의 메소드는 ManagedOperation속성으로 표시되어야만 하고 당신이 드러내길 바라는 각각의 프라퍼티는 ManagedAttribute속성으로 표시되어야 한다. 프라퍼티를 표시할때 당신은 쓰기전용(write-only)이나 읽기전용(read-only) 속성을 위한 getter이나 setter를 생략할수 있다.

밑의 예제는 당신이 좀더 먼저 본 JmxTestBean 클래스가 Commons Attributes 메타데이타로 표시된다는것을 보여준다.

```
package org.springframework.jmx;

/**
 * @@org.springframework.jmx.export.metadata.ManagedResource
 *  (description="My Managed Bean", objectName="spring:bean=test",
 *  log=true, logFile="jmx.log", currencyTimeLimit=15, persistPolicy="OnUpdate",
 *  persistPeriod=200, persistLocation="foo", persistName="bar")
 *
 */
public class JmxTestBean implements IJmxTestBean {

  private String name;

  private int age;

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *   (description="The Age Attribute", currencyTimeLimit=15)
```

```
   */
  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *  (description="The Name Attribute",  currencyTimeLimit=20,
   *   defaultValue="bar", persistPolicy="OnUpdate")
   */
  public void setName(String name) {
    this.name = name;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *   (defaultValue="foo", persistPeriod=300)
   */
  public String getName() {
    return name;
  }


  /**
   * @@org.springframework.jmx.export.metadata.ManagedOperation
   *  (description="Add Two Numbers Together")
   */
  public int add(int x, int y) {
    return x + y;
  }

  public void dontExposeMe() {
    throw new RuntimeException();
  }
}
```

여기서 당신은 JmxTestBean 클래스가 ManagedResource 속성으로 표시되고 이 ManagedResource 속성이 프라퍼티의 세트로 설정된다. 이러한 프라퍼티는 MBeanExporter에 의해 생성되는 MBean의 다양한 양상(aspect)로 설정하기 위해 사용될수 있고 Section 20.3.4, “소스레벨 메타데이타 타입들” 에서 나중에 좀더 상세하게 설명된다.

당신은 ManagedAttribute속성으로 표시되는 age와 name속성들 모두 알릴것이지만 age 프라퍼티의 경우 오직 getter만이 표시된다. 이것은 속성처럼 관리 인터페이스내 포함되기 위한 이헌 프라퍼티 모두를 야기할것이다. 그리고 age속성은 읽기전용이다.

마지막으로 당신은 dontExposeMe() 메소드가 표시되지 않기 때문에 add(int, int)메소드는 ManagedOperation속성으로 표시된다는것을 알릴것이다. 이것은 MetadataMBeanInfoAssembler를 사용할때 오직 하나의 작업(operation)인 add(int, int)을 포함하기 위한 관리 인터페이스를 야기할것이다.

아래의 코드는 당신이 MetadataMBeanInfoAssembler를 사용하기 위한 MBeanExporter를 설정하는 방법을 보여준다.

```
<?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="exporter" class="org.springframework.jmx.export.MBeanExporter">
```

```
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="assembler">
      <ref local="assembler"/>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>

  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource"/>

  <bean id="assembler" class="org.springframework.jmx.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">
      <ref local="attributeSource"/>
    </property>
  </bean>
</beans>
```

여기서 당신은 MetadataMBeanInfoAssembler가 AttributesJmxAttributeSource의 인스턴스로 설정되고 assembler프라퍼티를 통해 MBeanExporter로 전달하는것을 볼수 있다. 이것은 당신의 Spring-노출 MBean을 위한 데타데이타-기반 관리 인터페이스의 장점을 가져오기 위해 요구되는 모든것이다.

20.3.3. JDK 5.0 Annotations 사용하기

관리 인터페이스 정의를 위한 JDK 5.0 annotation의 사용을 가능하게 하기 위해 Spring은 Commons Attribute속성 클래스와 그것들을 읽기 위해 MBeanInfoAssembler을 허용하는 JmxAttributeSource, AnnotationsJmxAttributeSource의 구현물을 반영하는 annotation의 세트를 제공한다.

밑의 예제는 관리 인터페이스가 정의된 JDK 5.0 annotation을 가진 bean을 보여준다.

```
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(objectName="bean:name=testBean4", description="My Managed Bean", log=true,
    logFile="jmx.log", currencyTimeLimit=15, persistPolicy="OnUpdate", persistPeriod=200,
    persistLocation="foo", persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

   private String name;

  private int age;

  @ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)
  public int getAge() {
    return age;
```

```
  }

  public void setAge(int age) {
    this.age = age;
  }

  @ManagedAttribute(description="The Name Attribute",
      currencyTimeLimit=20,
      defaultValue="bar",
      persistPolicy="OnUpdate")
  public void setName(String name) {
    this.name = name;
  }

  @ManagedAttribute(defaultValue="foo", persistPeriod=300)
  public String getName() {
    return name;
  }

  @ManagedOperation(description="Add Two Numbers Together")
  public int add(int x, int y) {
    return x + y;
  }

  public void dontExposeMe() {
    throw new RuntimeException();
  }
}
```

당신이 볼수 있는것처럼 메타데이타 정의의 기본적인 문법보다 조금 변경된것을 볼수 있다. 그 장면뒤에서 이 접근법은 JDK 5.0 annotation이 Commons Attributes에 의해 사용되는 클래스로 형변환되기 때문에 수행시 조금더 느리다. 어쨌든 이것은 한번에 한한(one-off) 비용이고 JDK 5.0 annotations은 당신에게 컴파일시각 체크(compile-time checking)의 이득을 부여한다.

20.3.4. 소스레벨 메타데이타 타입들

다음의 소스레벨 메타데이타 타입들은 Spring JMX내에서 사용되기 위해 사용가능하다.

Table 20.1. 소스-레벨 메타데이타 타입들

목적	Commons Attributes 속성	JDK 5.0 Annotation	Annotation/Attribute 타입
자원을 관리하는 JMX처럼 클래스의 모든 인사턴스를 표시하기(Mark)	ManagedResource	@ManagedResource	Class
JMX작동(operation)처럼 메소드 표시하기	ManagedOperation	@ManagedOperation	Method
JMX속성의 절반처럼 getter이나 setter 표시하기	ManagedAttribute	@ManagedAttribute	Method (only getters and setters)
작동 파라미터를 위한	ManagedOperationParameter	@ManagedOperationParameter	Method

목적	Commons Attributes 속성	JDK 5.0 Annotation	Annotation/Attribute 타입
설명 정의하기		와 @ManagedOperationParameters	

다음의 설정 파라미터는 소스레벨 메타데이타 타입들의 사용을 위해 사용가능하다.

Table 20.2. 소스레벨 메타데이타 파라미터

파라미터	설명	적용
objectName	관리되는 자원의 ObjectName를 결정하기 위한 MetadataNamingStrategy에 의해 사용	ManagedResource
description	자원, 속성 또는 작동(operation)의 친숙한 설명 셋팅하기	ManagedResource, ManagedAttribute, ManagedOperation, ManagedOperationParameter
currencyTimeLimit	currencyTimeLimit 서술자 필드의 값을 셋팅하기	ManagedResource, ManagedAttribute
defaultValue	defaultValue 서술자 필드의 값을 셋팅하기	ManagedAttribute
log	log 서술자 필드의 값을 셋팅하기	ManagedResource
logFile	logFile 서술자 필드의 값을 셋팅하기	ManagedResource
persistPolicy	persistPolicy 서술자 필드의 값을 셋팅하기	ManagedResource
persistPeriod	persistPeriod 서술자 필드의 값을 셋팅하기	ManagedResource
persistLocation	persistLocation 서술자 필드의 값을 셋팅하기	ManagedResource
persistName	persistName 서술자 필드의 값을 셋팅하기	ManagedResource
name	작동(operation) 파라미터의 표시명(display name)을 셋팅하기	ManagedOperationParameter
index	작동(operation) 파라미터의 인덱스 셋팅하기	ManagedOperationParameter

20.3.5. AutodetectCapableMBeanInfoAssembler 인터페이스

좀더 간단한 설정을 위해, Spring은 MBean자원의 자동감지를 위한 지원을 추가하는 MBeanInfoAssembler 인터페이스를 확장하는 AutodetectCapableMBeanInfoAssembler인터페이스를 소개한다. 만약 당신이 AutodetectCapableMBeanInfoAssembler의 인스턴스를 사용해서 MBeanExporter을 설정한다면 이것은 JMX에 드러내기 위한 bean의 포함(inclusion)에 '결정(vote)'하는것을 허용한다.

특별히, AutodetectCapableMBeanInfo 의 구현물만이 ManagedResource속성으로 표시되는 bean을 포함하기 위해 결정(vote)할 MetadataMBeanInfoAssembler이다. 이 경우 디폴트 접근법은 이것과 같은 설정의 결과를 보이는 ObjectName처럼 bean이름을 사용하는 것이다.

```
<?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="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler">
      <ref local="assembler"/>
    </property>
  </bean>

  <bean id="bean:name=testBean1" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>

  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource"/>

  <bean id="assembler" class="org.springframework.jmx.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">
      <ref local="attributeSource"/>
    </property>
  </bean>
</beans>
```

이 설정내에서 MBeanExporter에 전달되는 bean은 없다. 어쨌든 JmxTestBean은 이것이 ManagedResource 속성으로 표시된 이후 등록될것이고 MetadataMBeanInfoAssembler는 이것을 감지하고 이것을 포함하기 위해 결정한다. 이 접근법이 가진 문제점은 JmxTestBean의 이름이 현재 비지니스 의미(meaning)를 가진다는것이다. 당신은 Section 20.4, "당신의 bean을 위한 ObjectName 제어하기" 에서 정의되는것처럼 ObjectName 생성을 위한 디폴트 행위를 변경하여 이 문제를 해결할수 있다.

20.3.6. 자바 인터페이스를 사용하여 관리 인터페이스 정의하기

MetadataMBeanInfoAssembler에 추가적으로, Spring은 당신에게 인터페이스의 집합(collection)내 정의된 메소드의 세트에 기반하여 드러나는 메소드와 프라퍼티를 강요하는것을 허용하는 InterfaceBasedMBeanInfoAssembler을 포함한다.

비록 MBean을 드러내기 위한 표준적인 기법이 인터페이스와 간단한 명명 개요(scheme)를 사용하는 것이라고 하더라도 InterfaceBasedMBeanInfoAssembler는 당신에게 하나의 인터페이스보다 많은 수를 사용하는것을 허용하고 MBean인터페이스를 구현하는 당신의 bean을 위한 필요성을 제거하는 명명규칙을 위한 필요성을 제거하여 이 기능을 확장한다.

당신이 좀더 일찍 본 JmxTestBean클래스를 위한 관리 인터페이스를 정의하기 위해 사용되는 이 인터페이스를 보라.

```
public interface IJmxTestBean {

  public int add(int x, int y);

  public long myOperation();

  public int getAge();

  public void setAge(int age);

  public void setName(String name);

  public String getName();
}
```

이 인터페이스는 JMX MBean의 작동(operation)과 속성처럼 드러날 메소드와 프라퍼티를 정의한다. 밑의 코드는 관리 인터페이스를 위한 정의처럼 이 인터페이스를 사용하기 위해 Spring JMX를 설정하는 방법을 보여준다.

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
        <property name="managedInterfaces">
          <value>org.springframework.jmx.IJmxTestBean</value>
        </property>
      </bean>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>

</beans>
```

여기서 당신은 어느 bean을 위한 관리 인터페이스를 생성할때 InterfaceBasedMBeanInfoAssembler가 IJmxTestBean 인터페이스를 사용하기 위해 설정되는것을 볼수 있다. InterfaceBasedMBeanInfoAssembler에 의해 처리되는 bean이 JMX 관리 인터페이스를 생성하기 위해 사용되는 인터페이스를 구현하는것을 요구하지 않는 것을 이해하는것은 중요하다.

위 경우에, IJmxTestBean 인터페이스는 모든 bean을 위한 모든 관리 인터페이스를 생성하기 위해 사용된다. 많은 경우 이것은 바람직한 행위가 아니며 당신은 다른 bean을 위해 다른 인터페이스를 사용하길

원할것이다. 이 경우 당신은 각각의 항목의 key는 bean이름이고 각각의 항목의 값은 bean사용을 위한 인터페이스 이름의 콤마로 분리된 리스트인 interfaceMappings 프라퍼티를 통해 InterfaceBasedMBeanInfoAssembler에 Properties를 전달할수 있다.

만약 managedInterfaces 나 interfaceMappings 프라퍼티를 통해 명시된 관리 인터페이스가 없다면 InterfaceBasedMBeanInfoAssembler는 bean에 반영할것이고 관리 인터페이스를 생성하기 위한 bean에 의해 구현되는 모든 인터페이스를 사용할것이다.

20.3.7. MethodNameBasedMBeanInfoAssembler 사용하기

MethodNameBasedMBeanInfoAssembler는 당신에게 속성과 작동(operation)처럼 JMX에 드러날 메소드명의 리스트를 명시하는것을 허용한다. 밑의 코드는 이것을 위한 샘플 설정을 보여준다.

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
        <property name="managedMethods">
          <value>add,myOperation,getName,setName,getAge</value>
        </property>
      </bean>
    </property>
</bean>
```

여기서 당신은 add 와 myOperation 메소드가 JMX작동(operation)처럼 드러나고 getName, setName 그리고 getAge 메소드가 선호되는 JMX속성의 절반처럼 드러날것을 볼수 있다. 위 코드에서 메소드 맵핑은 JMX에 드러나는 bean에 적용한다. bean기초에 의해 bean에서 드러나는 메소드를 제어하는 것은 메소드명의 리스트를 위한 bean이름을 맵핑하기 위한 MethodNameMBeanInfoAssembler의 methodMappings프라퍼티를 사용한다.

20.4. 당신의 bean을 위한 ObjectName 제어하기

MBeanExporter 는 이것이 등록하는 각각의 bean을 위한 ObjectName를 얻기 위해 ObjectNamingStrategy의 구현물로 위임한다. 디폴트 구현물인 KeyNamingStrategy는 디폴트에 의해 ObjectName처럼 beans 의 key Map을 사용할것이다. 추가적으로 KeyNamingStrategy는 ObjectName을 분석하기 위한 Properties 파일내 항목을 위한 beans의 key Map을 맵핑할수 있다. KeyNamingStrategy에 추가적으로 Spring은 두가지(bean의 동일성에 기반하는 ObjectName을 빌드하는 IdentityNamingStrategy와 ObjectName을 얻기 위한 소스레벨 메타데이타를 사용하는 MetadataNamingStrategy)의 추가적인 ObjectNamingStrategy 구현물을 제공한다.

20.4.1. Properties로 부터 ObjectName 읽기

당신은 당신 자신의 KeyNamingStrategy 인스턴스를 설정할수 있고 bean key를 사용하는것보다 Properties 인스턴스로부터 ObjectName를 읽어서 이것을 설정한다. KeyNamingStrategy는 bean key에 대응하는 key를 가진 Properties내 항목을 위치시키는 시도를 할것이다. 만약 어떠한 항목이 발견되지 않거나 Properties 인스턴스가 null이라면 그 bean key는 자체적으로 사용된다.

밑의 코드는 KeyNamingStrategy를 위한 샘플 설정을 보여준다.

```
<?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="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="namingStrategy">
      <ref local="namingStrategy"/>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.KeyNamingStrategy">
    <property name="mappings">
      <props>
        <prop key="testBean">bean:name=testBean1</prop>
      </props>
    </property>
    <property name="mappingLocations">
      <value>names1.properties,names2.properties</value>
    </property>
  </bean
</beans>
```

여기서 KeyNamingStrategy의 인스턴스는 mapping프라퍼티에 의해 정의된 Properties 인스턴스로 부터 통합된 Properties 인스턴스와 mapping 프라퍼티에 의해 정의된 경로내 위치한 프라퍼티 파일들로 설정된다. 이 설정에서 testBean bean은 이 항목이 bean key에 대응되는 key를 가진 Properties 인스턴스내 항목이 된 이후 ObjectName bean:name=testBean1가 주어질것이다.

Properties 인스턴스내 발견될수 있는 항목이 없다면 bean key는 ObjectName처럼 사용된다.

20.4.2. MetadataNamingStrategy 사용하기

MetadataNamingStrategy 는 ObjectName을 생성하기 위한 각각의 bean의 ManagedResource속성의 objectName프라퍼티를 사용한다. 밑의 코드는 MetadataNamingStrategy를 위한 설정을 보여준다.

```
<?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="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean">
          <ref local="testBean"/>
```

```
        </entry>
      </map>
    </property>
    <property name="namingStrategy">
      <ref local="namingStrategy"/>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>

  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource"/>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource">
      <ref local="attributeSource"/>
    </property>
  </bean>
</beans>
```

20.5. JSR-160 연결자(Connectors)로 당신의 bean을 내보내기

원격 접속을 위해 Spring JMX 모듈은 서버측과 클라이언트측 연결자를 생성하기 위한 org.springframework.jmx.support패키지내 두가지의 FactoryBean 구현물을 제공한다.

20.5.1. 서버측 연결자(Connectors)

Spring JMX가 JSR-160 JMXConnectorServer를 생성, 시작 그리고 드러내기 위해 다음의 설정을 사용한다.

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean"/>
```

디폴트에 의해 ConnectorServerFactoryBean은 "service:jmx:jmxmp://localhost:9875"를 바운드하는 JMXConnectorServer를 생성한다. serverConnector bean은 로컬호스트, 9875포트의 jmxmp 프로토콜을 통해 클라이언트로 로컬 MBeanServer를 드러낸다. JMXMP 프로토콜은 JSR 160에 의해 선택적으로 표시된다는것을 알라. 현재 인기있는 오픈소스 구현물인 MX4J와 자바 5.0에 의해 제공되는 것은 JMXMP를 지원하지 않는다.

다른 URL을 명시하고 MBeanServer를 가진 JMXConnectorServer를 등록하는 것은 serviceUrl 과 objectName 프라퍼티를 사용한다.

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName">
      <value>connector:name=rmi</value>
    </property>
    <property name="serviceUrl">
      <value>service:jmx:rmi://localhost:9875</value>
    </property>
</bean>
```

만약 objectName 프라퍼티가 셋팅된다면 Spring은 ObjectName하위의 MBeanServer으로 당신의 연결자를 자동적으로 등록할것이다. 밑의 예제는 당신이 JMXConnector를 생성할때 ConnectorServerFactoryBean로 전달할수 있는 파라미터의 전체 세트를 보여준다.

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName">
      <value>connector:name=iiop</value>
    </property>
    <property name="serviceUrl">
      <value>service:jmx:iiop://localhost:9875</value>
    </property>
    <property name="threaded">
      <value>true</value>
    </property>
    <property name="daemon">
      <value>true</value>
    </property>
    <property name="environment">
      <map>
        <entry key="someKey">
          <value>someValue</value>
        </entry>
      </map>
    </property>
</bean>
```

이러한 프라퍼티에 대한 좀더 상세한 정보를 JavaDoc을 보라.

20.5.2. 클라이언트측 연결자

MBeanServer을 가능하게 하는 원격 JSR-160을 위해 MBeanServerConnection을 생성하는 것은 밑에서 보여지는것처럼 MBeanServerConnectionFactoryBean을 사용한다.

```
<bean id="clientConnector" class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl">
      <value>service:jmx:rmi://localhost:9875</value>
    </property>
</bean>
```

20.5.3. Burlap/Hessian/SOAP 곳곳의 JMX

JSR-160 은 클라이언트와 서버간에 이루어지는 통신의 방법을 위한 확장을 허락한다. 위 예제는 JSR-160(IIOP 와 JRMP) 와 선택적인 JMXMP에 의해 요구되는 필수 RMI-기반의 구현물을 사용하는것이다. 다른 제공자(provider)와 MX4J [http://mx4j.sourceforge.net]와 같은 구현물을 사용하여 당신은 SOAP, Hessian, 간단한 HTTP나 SSL곳곳의 Burlap, 그리고 다른것들같은 프로토콜의 장점을 가질수 있다.

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName">
    <value>connector:name=burlap</value>
  </property>
  <property name="serviceUrl">
    <value>service:jmx:burlap://localhost:9874</value>
  </property>
</bean>
```

MX4J가 사용되는 이 예제를 위한 좀더 다양한 정보를 위해서는 MX4J문서를 보라.

20.6. 프록시를 통해서 MBean에 접속하기

Spring JMX는 당신에게 로컬또는 원격 MBeanServer내 등록된 MBean에 대한 호출 경로를 재정의하는 프록시를 생성하는것을 허용한다. 이러한 프록시는 당신에게 당신의 MBean과 상호작용할수 있는 것을 통해 표준적인 자바 인터페이스를 제공한다. 밑의 코드는 로컬 MBeanServer내에서 실행중인 MBean을 위한 프록시를 설정하는 방법을 보여준다.

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName">
      <value>bean:name=testBean</value>
    </property>
    <property name="proxyInterface">
      <value>org.springframework.jmx.IJmxTestBean</value>
    </property>
</bean>
```

여기서 당신은 프록시가 ObjectName(bean:name=testBean)하위에 등록된 MBean을 위해 생성되는것을 볼수 있다. 프록시가 구현할 인터페이스의 세트는 proxyInterfaces 프라퍼티에 의해 제어되고 MBean의 작동(operation)과 속성을 위한 인터페이스의 메소드와 프라퍼티를 맵핑하기 위한 규칙은 InterfaceBasedMBeanInfoAssembler에 의해 사용되는 규칙과 같다.

MBeanProxyFactoryBean은 MBeanServerConnection을 통해 접근가능한 MBean을 위한 프록시를 생성할수 있다. 디폴트에 의해 로컬 MBeanServer는 위치되고 사용되지만 당신은 이것을 오버라이드 할수 있고 원격 MBean을 위한 프록시 지점(pointing)을 위해 허용하는 원격 MBeanServer를 위한 MBeanServerConnection 지점(pointing)을 제공할수 있다.

```
<bean id="clientConnector" class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl">
      <value>service:jmx:rmi://remotehost:9875</value>
    </property>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName">
      <value>bean:name=testBean</value>
    </property>
    <property name="proxyInterface">
      <value>org.springframework.jmx.IJmxTestBean</value>
    </property>
</bean>
```

여기서 당신은 우리가 MBeanServerConnectionFactoryBean을 사용하여 원격 머신을 위한 MBeanServerConnection 지점을 생성하는것을 볼수 있다. 이 MBeanServerConnection은 server 프라퍼티를 통해 MBeanProxyFactoryBean으로 전달된다. 생성된 프록시는 MBeanServerConnection를 통해 MBeanServer로 모든 호출을 전달할것이다.

Chapter 21. Testing

21.1. 단위 테스팅

이 매뉴얼은 스프링 기반 어플리케이션을 위한 효과적인 단위 테스트를 작성하는 데는 별 도움을 주지 못할 것이다.

의존성 주입(Dependency Injection)의 주된 이익들 중 하나는 당신의 코드가 전통적인 J2EE 개발방법에 비해 컨테이너에 덜 의존적이라는 점이다.

당신의 어플리케이션을 구성하는 POJO는 Spring 혹은 어떠한 다른 컨테이너 없이 new(생성자) 실행을 사용하여 간단하게 초기화된 객체들을 가지고, JUnit 테스트에서 테스트 가능해야만 한다. 당신은 당신의 코드를 독립적으로 테스트하기 위해, 가짜(mock) 객체들 혹은 많은 다른 가치있는 테스팅 기법들을 사용할 수 있다. 만약 당신이 Spring을 둘러싼 구조적인 장점--예를 들어, EJB 없는 J2EE에서의 장점--을 따라가고자 한다면, 당신은 완벽한 계층화 만들기가 또한 테스팅을 굉장히 쉽게 해준다는 사실을 알게될 것이다. 예를 들어, 당신은 단위 테스트 중에 퍼시스턴트 데이터들에 억세스할 필요없이, DAO 인터페이스들을 stub 혹은 mock 함으로써 서비스 계층 객체들을 테스트할 수 있을 것이다.

진정한 단위 테스트는 굉장히 빨리 실행될 것이다. 왜냐하면, 어플리케이션 서버, 데이터베이스, ORM 툴 등 셋업해야 할 실행시 하부구조가 아무것도 없기 때문이다. 따라서 진정한 단위 테스트는 당신의 생산성을 높혀줄 것이다.

21.2. 통합 테스팅

그러나, 당신의 어플리케이션 서버에 대한 개발없이 몇몇 통합 테스팅을 실행할 수 있는 것은 매우 중요하다. 이것은 다음과 같은 것들을 테스트할 것이다.

☒ 당신의 Spring 컨텍스트들을 올바르게 연결하기

☒ JDBC 혹은 ORM 툴을 사용한 데이터 접근--정확한 SQL문. 예를 들어, 당신은 DAO 구현 클래스들을 테스트할 수 있다.

그래서 Spring은 spring-mock.jar.에서 통합 테스팅을 위한 훌륭한 지원을 제공한다. 이것은 Cactus와 같은 툴을 사용한 컨테이너 내부 테스팅에 비해 훨씬 뛰어난 대안으로 여겨질 수 있다.

org.springframework.test 패키지는 Spring 컨테이너를 사용하지만 어플리케이션 서버 또한 다른 배포된 환경에 종속되지 않는, 통합 테스팅을 위한 훌륭한 상위 클래스들을 제공한다. 그런 테스트들은 다른 특별한 배포단계 없이 JUnit-심지어 IDE 내에서도--에서 실행될 수 있다. 그것들은 unit 테스트보다는 늦게 실행되겠지만, Cacus 테스트 또는 어플리케이션 서버 배포판에 의존적인 원격 테스트보다는 훨씬 빠를 것이다.

이 패키지의 상위 클래스들은 다음의 기능들을 제공한다:

☒ 컨텍스트 캐싱

☒ 테스트 클래스들에 대한 의존성 주입

☒ 테스트에 적합한 트랜잭션 관리

☒ 테스팅에 유용한 상속받은 인스턴스 변수들

2004년 말이래 많은 수의 인터페이스와 다른 프로젝트들은 이러한 접근방식의 강력함과 유용함을 보여왔다. 기능에 있어서의 몇가지 중요한 부분들을 상세하게 살펴보도록 하자.

21.2.1. 컨텍스트 관리와 캐싱

org.springframework.test 패키지는 Spring 컨텍스트의 일관된 로딩과 로딩된 컨텍스트를 캐싱을 제공한다. 후자는 매우 중요한데, 왜냐하면 만약 당신이 큰 프로젝트에서 일을하고 있다면, 스타트업 시간은 이슈가 될 것이기 때문이다.--이것은 Spring 그 자체의 오버헤드때문이 아니라 Spring 컨테이너에 의해 초기화되는 객체들이 그 자체로 초기화에 시간이 걸리기 때문이다. 예를 들어, 50-100개의 하이버네이트 매핑 파일들을 가진 프로젝트는 그것들을 로드하는데 10-20초 가량 소요되고, 매 테스트 케이스를 실행하기 전의 비용은 굉장한 생산성 감소를 초래하게 될 것이다.

따라서, AbstractDependencyInjectionSpringContextTests 는 컨텍스트들의 위치를 제공하는 하위 클래스들이 반드시 구현해야 하는 abstract protected method를 가진다.

```
protected abstract String[] getConfigLocations();
```

이것은 어플리케이션을 설정하기 위해 사용되는 컨텍스트 위치들--전형적으로 클래스패쓰에 위치하는--의 리스트를 제공한다. 이것은 web.xml 혹은 다른 배포 설정에서 기술되는 설정 위치들의 리스트와 동일하거나 거의 비슷할 것이다.

디폴트로, 한번 로드되면, 설정들의 세트는 매 테스트 케이스를 위해 재사용될 것이다. 그래서, 셋업 비용은 단지 한번만 발생되고 뒤이은 테스트 실행은 보다 빠를 것이다.

테스트가 설정 위치를 지저분하게하는 정말 드문 경우, --예를 들어, 빈 정의 혹은 어플리케이션 객체의 상태를 수정 등으로 인해-- 리로딩을 요청하게 될 때, 당신은 AbstractDependencyInjectionSpringContextTests 에 있는 setDirty() 메써드를 호출할 수 있다. 이것은 다음 테스트 케이스를 실행하기 전에 설정을 리로드하고 어플리케이션 컨텍스트를 재생성하게 한다.

21.2.2. 테스트 클래스 인스턴스들의 의존성 주입

AbstractDependencyInjectionSpringContextTests (와 하위클래스들)가 당신의 어플리케이션 컨텍스트를 로드할 때, 그것들은 선택적으로 당신의 테스트 클래스들의 인스턴스들을 세터 주입을 통해 설정할 수 있다. 당신이 해야 할 일은 인스턴스 변수들과 그에 일치하는 세터들을 정의하는 것 뿐이다. AbstractDependencyInjectionSpringContextTests는 getConfigLocations() 메써드에서 기술된 설정 파일들의 세트에서 일치되는 객체들을 자동으로 위치시킬 것이다.

상위 클래스들은 타입에 의한 자동묶기를 사용한다. 그래서 만약 당신이 동일한 타입의 빈 정의들을 여러개 가진다면, 당신은 그러한 특정한 빈들을 위해 이 접근방식을 사용할 수 없을 것이다. 이런 경우, 당신은 상속된 applicationContext 인스턴스 변수를 사용할 수 있으며, getBean()을 사용하여 명백하게 룩업할 수 있을 것이다.

만약 당신이 당신의 테스트 케이스들에 적용된 세터 주입을 원하지 않는다면, 세터들을 아무것도 정의하지 않거나 org.springframework.test 패키지의 클래스 계층의 root인, AbstractSpringContextTests를 상속받도록 하라. 이 클래스는 단지 Spring 컨텍스트들을 로드하기 위핸 편의 메써드들만을 가지고 있으며 아무런

의존성 주입도 실행하지 않는다.

21.2.3. 트랜잭션 관리

실제 데이터베이스에 접근하는 테스트에 있어 한 가지 일반적인 문제점은 데이터베이스 상태에 테스트가 영향을 끼친다는 점이다. 심지어 당신이 개발 데이터베이스를 사용할 때에도, 상태변화는 이후의 테스트들에 영향을 끼칠지 모른다.

또한, 데이터 삽입, 수정 등 많은 동작들은 트랜잭션 외부에서 이루어지거나 검증될 수 없을 것이다.

org.springframework.test.AbstractTransactionalDataSourceSpringContextTests 상위 클래스(와 그 하위 클래스들)는 이러한 필요를 충적시키기 위해 존재한다. 기본적으로, 이 클래스들은 개별 테스트 케이스를 위해 트랜잭션을 생성하고 롤백한다. 당신은 간단하게 트랜잭션의 존재를 확인할 수 있는 코드를 쓰기만 하면 된다. 만약 당신이 트랜잭션적으로 프락시된 객체를 당신의 테스트 내에서 호출한다면, 그 객체들은 그들의 트랜잭션적인 문법에 따라 올바르게 동작할 것이다.

AbstractTransactionalSpringContextTests 는 어플리케이션 컨텍스트 내에 정의된 PlatformTransactionManager 빈에 의존한다. 타입에 의해 자동으로 묶어주기 때문에 이름은 중요한 것이 아니다.

일반적으로 당신은 하위 클래스인 AbstractTransactionalDataSourceSpringContextTests를 상속할 것이다. 이것은 또한 DataSource 빈 정의--이것 역시 아무 이름이어도 상관없다.--가 설정들 내에 존재해야만 한다. 이것은 편리한 질의를 위해 유용한 JdbcTemplate 인스턴스를 생성하고, 선택된 테이블들의 내용들을 삭제하기에 편리한 메써드들을 제공한다. (트랜잭션은 기본적으로 롤백된다는 점을 기억하라. 왜냐하면 그것이 안전하기 때문이다.)

만약 당신이 트랜잭션이 커밋되기를 원한다면,--드물지만, 예를 들어 만약 당신이 데이터베이스에 데이터를 입력시키는 특별한 테스트를 원한다면 유용할 것이다.-- 당신은 AbstractTransactionalSpringContextTests로부터 상속받은 setComplete() 메써드를 호출할 수 있다. 이것은 롤백 대신에 트랜잭션이 커밋되도록 할 것이다.

또한 테스트 케이스가 끝나기 전에 트랜잭션을 종료시키는 편리한 기능이 있는데, endTransaction() 메써드를 호출하면 된다. 이것은 기본적으로 트랜잭션을 롤백시키며, 오로지 이전에 setComplete() 가 호출되었을 때만 커밋시킨다. 이 기능은 만약 당신이 이를테면, 웹 혹은 트랜잭션 외부의 원격티어에서 사용될 하이버네이트 매핑된 객체들과 같이, 연결이 끊어진 데이터 객체들의 동작을 테스트하고자 할 때 유용하다. 종종, lazy 로딩 에러는 UI 테스팅을 통해서만 발견되는데; 만약 당신이 endTransaction() 를 호출한다면, 당신은 JUnit 테스트 수트를 통해 UI의 올바른 동작을 검증할 수 있을 것이다.

이러한 테스트 지원 클래스들은 하나의 데이터베이스와 함께 동작하는 것으로 설계되었다.

21.2.4. 편리한 변수들

당신이 org.springframework.test 패키지를 상속한다면, 당신은 다음의 protected 인스턴스 변수들에 접근할 수 있을 것이다.

- applicationContext (ConfigurableApplicationContext): AbstractDependencyInjectionSpringContextTests 로부터 상속받았다. 명시적인 빈 룩업을 수행하거나 총괄적으로 컨텍스트의 상태를 테스트할 때 이것을 사용하라.
- jdbcTemplate: AbstractTransactionalDataSourceSpringContextTests 로부터 상속받았다. 상태를 확인하기 위해 질의를 할 때 유용하다. 예를 들어, 당신이 객체를 생성하고, 그것을 ORM 툴을 사용하여 저장하는

어플리케이션 코드에 대한 테스팅 이전/이후에, 그 데이터가 데이터베이스에 나타나는지를 검증하기 위해 질의할 수 있을 것이다. (Spring은 그 쿼리가 동일 트랜잭션 범위에서 실행되는 것을 보증할 것이다.) 당신은 이것이 올바르게 작동되기 위해, ORM 툴이 그것의 변화들을 flush하도록 호출할 필요가 있을 것이다. 예를 들어, 하이버네이트 Session 인터페이스의 flush() 메써드를 사용해서 말이다.

종종 당신은 통합 테스트를 위해 많은 테스트들에서 사용되는 보다 유용한 변수들을 제공하는 어플리케이션-포괄 상위 클래스를 제공할 것이다.

21.2.5. 예시

Spring 배포판에 포함된 PetClinic 샘플 어플리케이션은 이러한 테스트 상위 클래스들의 사용을 설명해준다.(Spring 1.1.5 이상 버전)

대부분의 테스트 기능은 AbstractClinicTests에 포함되었고, 부분적인 내역들은 아래에서 보이는 대로이다.

```
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());
}
```

노트:

☒ 이 테스트 케이스는 이것은 의존성 주입과 트랜잭션적인 동작을 위해 org.springframework.AbstractTransactionalDataSourceSpringContextTests 를 상속받았다.

☒ clinic 인스턴스 변수--테스트될 어플리케이션 객체--는 setClinic() 메써드를 통해 의존성 주입에 의해 세팅된다.

☒ testGetVets() 메써드는 테스트될 어플리케이션 코드의 올바른 동작을 검증하기 위해 JdbcTemplate 변수를 사용하는 방법을 보여준다. 이것은 더욱 강력한 테스트들을 가능하게 하고 정확한 테스트 데이터에 대한 의존성을 줄여준다. 예를 들어, 당신은 테스트들을 중단하지 않고도 데이터베이스에 부가적인 row들을 추가할 수 있다.

☒ 데이터베이스를 사용하는 많은 통합 테스트들처럼, AbstractClinicTests 에서의 테스트들의 대부분은 테스트 케이스들이 실행되기 전 데이터베이스 내에 이미 존재하는 최소량의 데이터베이스에 의존한다.

그러나, 당신은 또한 당신의 테스트 케이스들 내에서 --물론, 하나의 트랜잭션 내에서-- 데이터베이스에 입력하는 것을 선택할 수도 있다.

PetClinic 어플리케이션은 3가지 데이터 접근 기술들을 제공한다.--JDBC, 하이버네이트, 그리고 아파치 OJB. 그래서 AbstractClinicTests 는 컨텍스트 위치들을 기술하지 않는다. 이것은 AbstractDependencyInjectionSpringContextTests의 필수적인 protected abstract 메써드를 구현하는 하위 클래스들에 따라 다르다..

예를 들어, PetClinic 테스트의 JDBC 구현은 다음의 메써드를 포함한다.

```
public class HibernateClinicTests extends AbstractClinicTests {

   protected String[] getConfigLocations() {
      return new String[] {
         "/org/springframework/samples/petclinic/hibernate/applicationContext-hibernate.xml"
      };
   }
}
```

PetClinic은 매우 간단한 어플리케이션이기 때문에, 단 하나의 Spring 설정 파일만이 존재한다. 물론, 보다 복잡한 어플리케이션들은 일반적으로 Spring 설정을 여러 개의 파일들로 쪼갤 것이다.

최하위 클래스에서 정의되는 대신, 설정 위치들은 모든 어플리케이션-특화 통합 테스트를 위한 일반적인 베이스 클래스에서 종종 기술된다. 이것은 또한 유용한 --자연스럽게 의존성 주입에 의해 제공되는--하이버네이트를 사용하는 어플리케이션의 경우의 HibernateTemplate와 같은인스턴스 변수들을 추가할 것이다.

가능한 한, 당신은 배포될 환경에서와 동일한 Spring 설정 파일들을 통합 테스트에서도 가져야만 한다. 유일하게 다른점이라면 데이터베이스 커넥션 풀링과 트랜잭션 하부구조와 관련된 부분들 뿐이다. 만약 당신이 고성능(? full-blown) 어플리케이션 서버에 배포할 것이라면, 당신은 아마도 그것의 (JNDI를 통해 가능한) 커넥션 풀과 JTA 구현을 사용할 것이다. 따라서 제품 내에서, 당신은 DataSource와 JtaTransactionManager를 위해 JndiObjectFactoryBean을 사용할 것이다. JNDI와 JTA는 컨테이너 외부 통합 테스트에서는 가능하지 않을 것이다. 따라서, 당신은 반드시 Commons DBCP BasicDataSource와 DataSourceTransactionManager 혹은 HibernateTransactionManager와 같은 조합을 사용해야 할 것이다. 당신은 어플리케이션 서버와 로컬 설정 사이의 선택을 가지는 이러한 다양한 동작들을, 테스트와 제품 환경들 사이에 변화가 없는 모든 다른 설정들로부터 분리하여, 하나의 XML 파일에 분해할 수 있다.

21.2.6. 통합 테스트 실행하기

통합 테스트는 자연적으로 평범한 유닛 테스트들에 비해 보다 환경적인 의존성을 가진다.그런 통합 테스팅은 유닛 테스팅의 대체물이 아니라 테스팅의 부가적인 형태이다.

주된 의존성은 전형적으로 어플리케이션에 의해 사용되는 완전한 스키마를 포함하는 개발 데이터베이스에 대한 것이다. 이것은 아마도 또한, DBUnit같은 툴에 의해 셋업되거나 데이터베이스 툴 셋을 사용하여 임포트된, 테스트 데이타를 포함한다.

Appendix A. spring-beans.dtd

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

<!--
	Spring XML Beans DTD
	Authors: Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu

	이것은 Spring BeanFactory가 관리하고 XmlBeanDefinitionReader(DefaultXmlBeanDefinitionParser를 가진)가
	읽는 자바빈 객체의 명명공간(namespace)을 생성하는 간단하고 일관적인 방법을 정의한다.

	대부분의 Spring기능을 사용하는 문서타입은 bean factory에 기초를 둔 웹 애플리케이션 컨텍스트를 포함한다.

	이 문서내 각각의 "bean" 요소는 자바빈을 정의한다. 대개 bean클래스가 자바빈 프라퍼티와/또는 생성자의
	인자에 따라 명시된다.

	bean인스턴스는 "싱글톤" (공유 인스턴스) or "프로토타입"(독립적인 인스턴스)이 될수 있다. 더 많은
	범위(scope)는 핵심 BeanFactory 구조의 상위에 내장되도록 가정되어서 이것의 일부가 아니다.

	bean사이의 참조는 제안된다. 이를테면 같은 factory(또는 조상(ancestor) factory)내 다른 bean을 위한 참조를
	위해 자바빈 프라퍼티나 생성자의 인자를 셋팅한다.

	bean참조의 대안처럼 "내부 bean정의"가 사용될수 있다. 내부 bean정의의 싱글톤 플래그는 효과적으로 무시된다.
	내부 bean은 대개 익명 프로토타입이다.

	bean프라퍼티 타입이나 생성자의 인자타입처럼 리스트, 세트, maps, 그리고 java.util.Properties를 위한 지원이 있다.

	포맷이 간단하고, DTD가 충분하고 이 시점에 스키마가 필요하지 않다.

	DTD를 따르는 XML문서는 다음의 doctype를 선언해야만 한다.

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


<!--
	문서의 가장 상위(root). 문서는 bean 정의만, import만, 또는 둘의 혼합을 포함할수 있다. (대개 import가 먼저이다)
-->
<!ELEMENT beans (
	description?,
	(import | alias | bean)*
)>

<!--
	모든 bean정의를 위한 디폴트 값. "bean"레벨에 오버라이드될수 있다. 상세사항을 위해서 속성 정의를 보라.
-->
<!ATTLIST beans default-lazy-init (true | false) "false">
<!ATTLIST beans default-dependency-check (none | objects | simple | all) "none">
<!ATTLIST beans default-autowire (no | byName | byType | constructor | autodetect) "no">

<!--
	요소는 인코딩된 요소의 목적을 설명하는 정보성 텍스트를 포함. 언제나 선택사항임.
	XML bean정의 문서의 사용자 문서를 위해 가장 먼서 사용.
-->
<!ELEMENT description (#PCDATA)>


<!--
	import하기 위한 XML bean정의를 명시.
-->
<!ELEMENT import EMPTY>
```

```
<!--
	import하기 위한 XML bean정의 파일의 상대적인 자원 위치.
	예를 들어 "myImport.xml", "includes/myImport.xml" 또는 "../myImport.xml".
-->
<!ATTLIST import resource CDATA #REQUIRED>


<!--
	다른 정의 파일내 위치할수 있는 bean을 위한 별칭을 정의.
-->
<!ELEMENT alias EMPTY>

<!--
	별칭을 정의하기 위한 bean의 이름
-->
<!ATTLIST alias name CDATA #REQUIRED>

<!--
	bean의 위해 정의하는 별칭이름.
-->
<!ATTLIST alias alias CDATA #REQUIRED>


<!--
	하나의 bean을 정의

	bean정의는 생성자의 인자, 프라퍼티 값, 룩업 메소드, 그리고 교체된 메소드를 위한 내포된 태그를 포함한다.
	같은 bean에서 생성자 삽입(constructor injection)과 setter삽입(setter injection)의 혼합은 명시적으로 지원된다.
-->
<!ELEMENT bean (
	description?,
	(constructor-arg | property | lookup-method | replaced-method)*
)>

<!--
	참조 체크를 가능하게 하는 id에 의해 구별될수 있는 bean.

	유효한 XML id에는 제약이 있다. 만약 당신이 XML id처럼 적합하지 않은 이름을 사용하는 자바코드내
	당신의 bean을 참조하길 원한다면 선택사항인 "name" 속성을 사용하라. 아무것도 주어지지 않았다면
	bean클래스 이름은 id (만약 그 이름을 가진 bean이 이미 있다면 "#2"처럼 접두사 적인 카운터를
	가진)처럼 사용된다.
-->
<!ATTLIST bean id ID #IMPLIED>

<!--
	선택사항. id내 하나 이상의 별칭을 생성하기 위해 사용될수 있다. 다중 별칭은 많은 수의 공간이나 콤마에
	의해 분리될수 있다.
-->
<!ATTLIST bean name CDATA #IMPLIED>

<!--
	각각의 bean정의는 자식 bean정의를 위한 부모처럼 제공되는 상황을 제외하고 클래스의 완전한 형태의
	이름을 명시해야만 한다.
-->
<!ATTLIST bean class CDATA #IMPLIED>

<!--
	선택적으로 부모 bean정의를 명시한다.

	아무것도 명시되지 않는다면 부모의 bean클래스가 사용되지만 이것을 오버라이드할수 있다.
	나중의 경우 자식 클래스는 부모 클래스와 호환되어야만 한다. 이를테면 부모 클래스의 프라퍼티값과
	생성자의 인자값을 받을수 있어야만 한다.

	자식 bean정의는 새로운 값을 추가하는 선택사항을 가지고 생성자의 인자값, 프라퍼티 값 그리고 부모로부터
	오버라이드된 메소드를 상속할것이다. 만약 init 메소드, destroy 메소드, factory bean 그리고/또는 factory 메소드가
```

```
        명시되었다면 그것들은 부모 셋팅에 관련하여 오버라이드 할것이다.

        남아있는 셋팅은 자식 정의(autowire모드, 의존성 체크, 싱글톤, 늦은 초기화(lazy init)에 의존하여)로 부터 가져올것이다.
-->
<!ATTLIST bean parent CDATA #IMPLIED>

<!--
        이 bean이 "abstract"인지에 대한 값. 견고한 자식 bean정의를 위한 부모처럼 제공되지만 자체적으로는 인스턴스화되지 않는다.
        디폴트는 false. 특정 bean을 인스턴스화 하는것을 시도하지 않도록 bean factory에게 알리기 위해서는 true로 명시하라.
-->
<!ATTLIST bean abstract (true | false) "false">

<!--
        이 bean이 "singleton"(id를 가진 getBean()을 호출하여 반환될 공유 인스턴스)이거나 getBean()을 호출하여 독립적인
        인스턴스를 만드는 "prototype"인지에 대한 값. 디폴트는 singleton.

        singleton이 가장 공통적으로 사용된다. 그리고 멀티-쓰레드 서비스 객체를 위해 이상적이다.
-->
<!ATTLIST bean singleton (true | false) "true">

<!--
        bean이 늦게(lazily) 초기화되는지에 대한 값.
        false라면 이것은 singletons의 초기화를 수행하는 bean factory에 의해 시작될때 인스턴스화될것이다.
-->
<!ATTLIST bean lazy-init (true | false | default) "default">

<!--
        bean프라퍼티를 "autowire"할지에 대한 제어를 하는 선택적인 속성.
        이것은 XML bean정의 파일내 명시적으로 코드될 필요가 없는 bean 참조내 마법적인 처리이다.
        하지만 Spring은 의존성을 해결한다.

        5가지의 모드가 있다.

        1. "no"
        전통적인 Spring 디폴트. 마법같은 wiring이 없다. bean참조는 <ref> 요소를 통해 XML파일내 정의되어야만
        한다. 우리는 대부분의 경우 좀더 명시적으로 문서를 만드는것처럼 이것을 추천한다.

        2. "byName"
        프라퍼티 이름에 의한 autowiring. Cat클래스의 bean이 dog프라퍼티를 드러낸다면 Spring은 이것을 현재 factory내
        "dog" bean의 값으로 셋팅하는것을 시도할것이다. 이름에 의한 bean이 대응되는것이 없다면 아무것도 발생하지
        않는다. 이 경우 에러를 발생시키기 위해 dependency-check="objects" 를 사용하라.

        3. "byType"
        bean factory내 프라퍼티 타입의 bean이 정확하게 한개가 있다면 autowiring한다. 만약 하나 이상이 있다면
        치명적인 에러가 발생하고 당신은 bean을 위한 byType autowiring을 사용할수 없다. 만약 하나도 없다면
        어떠한 특별한 일도 발생하지 않는다. 이 경우 에러를 발생시키기 위해서는 dependency-check="objects"를 사용하라.

        4. "constructor"
        생성자의 인자를 위한 "byType"과 비슷하다. bean factory내 생성자의 인자 타입의 bean이 정확하게 하나가
        존재하는 상황이 아니라면 치명적인 에러가 발생한다.

        5. "autodetect"
        bean 클래스의 자체 분석을 통해 "constructor" 나 "byType"을 사용한다. 만약 디폴트 생성자가 발견된다면
        "byType"이 적용된다.

        뒤의 두가지는 PicoContainer와 유사하고 작은 이름공간(namespace)을 위한 설정을 위해 bean factory를
        간단하게 만든다. 하지만 좀더 큰 애플리케이션을 위해 표준적인 Spring 행위처럼 잘 작동하지는 않는다.

        명시적인 의존성이다. 이를테면 "property" 와 "constructor-arg" 요소는 언제나 autowiring를 오버라이드한다.
        autowire행위는 모든 autowiring이 완성된 후 수행될 의존성 체크와 조합될수 있다.
-->
<!ATTLIST bean autowire (no | byName | byType | constructor | autodetect | default) "default">

<!--
        이 프라퍼티에서 표현되는 모든 bean의존성이 만족하는지에 대해 체크할지 제어하는 선택적인 속성.
```

```
        디폴트는 의존성 체크를 하지 않는다.

        "simple" 원시 타입과 문자열 타입을 포함한 의존성 체크.
        "object" 는 협력자(factory내 다른 bean)
        "all" 위 두타입 모두를 포함하는 의존성 체크.
-->
<!ATTLIST bean dependency-check (none | objects | simple | all | default) "default">

<!--
        초기화시 이 bean이 의존하는 bean의 이름.
        bean factory는 bean이 초기화되기 전에 보장할것이다.

        의존성은 bean프라퍼티나 생성자의 인자를 통해 대개 표현된다. 이 프라퍼티는 시작시 정적이나
        데이터베이스 준비처럼 다른 종류의 의존성을 위해 필요할것이다.
-->
<!ATTLIST bean depends-on CDATA #IMPLIED>

<!--
        bean프라퍼티를 셋팅한 후 호출하기 위한 사용자정의 초기화 메소드의 이름을 위한 선택적인 속성.
        메소드는 인자를 가지지 않아야 한다. 하지만 어떤 예외를 던질것이다.
-->
<!ATTLIST bean init-method CDATA #IMPLIED>

<!--
        bean factory가 닫힐 때 호출하기 위한 사용자 정의 회수(destroy) 메소드의 이름을 위한 선택적인 속성.
        메소드는 인자를 가지지 않아야 한다. 하지만 어떤 예외를 던질것이다.
        메모 : 싱글톤 bean에서는 오직 한번만 호출된다.
-->
<!ATTLIST bean destroy-method CDATA #IMPLIED>

<!--
        객체를 생성하기 위해 사용하는 factory메소드의 이름을 명시하는 선택적인 속성.
        만일 인자를 가진다면 factory메소드를 위한 인자를 명시하기 위해 constructor-arg 요소를 사용하라.
        autowiring은 factory메소드에 적용하지 않는다.

        "class"속성이 존재한다면 factory메소드는 bean정의의 "class"속성에 의해 명시되는 클래스의 정적 메소드가
        될것이다. 예를 들어 factory메소드가 생성자의 대안으로 사용될때 종종 이것은 생성된 객체처럼 같은
        클래스가 될것이다.  어쨌든. 이것은 다른 클래스가 될수 있다. 이 경우 생성된 객체는 "class"속성내 명시된
        클래스가 되지 "않을"것이다. 이것은 FactoryBean행위와 유사하다.

        "factory-bean"속성이 존재한다면 "class"속성은 사용되지 않는다. 그리고 factory메소드가 명시된 bean이름을
        가진 getBean호출로 부터 반환된 객체의 인스턴스 메소드가 될것이다. factory bean은 싱글톤이나
        프로토타입처럼 정의될수 있다.

        factory메소드는 많은 수의 인자를 가질수 있다. autowiring은 지원되지 않는다. factory-method속성과 결합되는
        인덱스화된 constructor-arg을 사용하라.

        setter 삽입(injection)은 factory메소드와 결합되서 사용될수 있다. 메소드 삽입(injection)은 결합되어 사용될수
        없다. factory메소드가 컨테이너가 bean생성할때 사용될 인스턴스를 반환한다.
-->
<!ATTLIST bean factory-method CDATA #IMPLIED>

<!--
        factory-metohd사용을 위한 class속성에 대한 대안이다. 이것이 명시된다면 class속성은 사용되지 않는다.
        이것은 관련 factory메소드를 포함하는 현재 또는 상위 factory내 bean의 이름으로 셋팅될수 있다.
        이것은 factory자체가 의존성 삽입과 사용될 인스턴스(정적인것 보다) 메소드를 사용하여 설정되는것을 허용한다.
-->
<!ATTLIST bean factory-bean CDATA #IMPLIED>


<!--
        bean정의는 생성자의 인자를 하나를 명시하거나 그 이상을 명시할수 있다.
        이것은 "autowire constructor"에 대안으로 사용된다.
        Arguments correspond to either a specific index of the constructor argument
        list or are supposed to be matched generically by type.
```

```
        메모 : 하나의 일반적인 인자의 값은 여러번 잠재적으로 대응되더라도 한번만 사용된다.

        constructor-arg 요소는 정적이나 인스턴스 factory메소드를 사용하여 bean을 생성하기 위해 factory-method요소와
        결합되어 사용된다.
-->
<!ELEMENT constructor-arg (
        description?,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        constructor-arg태그는 생성자의 안자 리스트내 정확한 인덱스를 명시하기 위해 index속성을 선택적으로 가진다.
        모호함(이를테면 같은 타입의 두개의 인자의 경우)을 피하기 위해서만 필요하다.
-->
<!ATTLIST constructor-arg index CDATA #IMPLIED>

<!--
        constructor-arg태그는 선택적으로 생성자의 인자 타입을 정확하게 명시하기 위한 type속성을 가질수 있다.
        모호함(이를테면 문자열로 부터 형변환될수 있는 하나의 인자를 가지는 생성자)을 피하기 위해서만 필요하다.
-->
<!ATTLIST constructor-arg type CDATA #IMPLIED>

<!--
  자식 요소인 "ref bean="에 단순화(short-cut)된 대안.
-->
<!ATTLIST constructor-arg ref CDATA #IMPLIED>

<!--
  자식 요소인 "value"에 단순화(short-cut)된 대안.
-->
<!ATTLIST constructor-arg value CDATA #IMPLIED>


<!--
        bean정의는 프라퍼티를 가지지 않거나 그 이상을 가질수 있다. property요소는 bean클래스에 의해 드러나는
        자바빈 setter메소드와 일치한다. Spring은 원시타입, 같거나 관련된 factory내 다른 bean에 대한 참조, 리스트,
        map 그리고 프라퍼티를 지원한다.
-->
<!ELEMENT property (
        description?,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        property name 속성은 자바빈 프라퍼티의 이름이다. 이것은 자바빈 관례(conventions)를 따른다.
        "age"의 이름은 setAge()와 일치하고 선택적으로 getAge()메소드와 일치할것이다.
-->
<!ATTLIST property name CDATA #REQUIRED>

<!--
  자식 요소인 "ref bean="에 단순화(short-cut)된 대안.
-->
<!ATTLIST property ref CDATA #IMPLIED>

<!--
  자식 요소인 "value"에 단순화(short-cut)된 대안.
-->
<!ATTLIST property value CDATA #IMPLIED>


<!--
        룩업 메소드는 주어진 메소드를 오버라이드 하기 위한 IoC컨테이너를 야기한다. bean속성내
        주어진 이름으로 bean을 반환한다. 이것은 메소드 삽입(injection)의 형태이다. 이것은
        수행시 싱글톤이 아닌 인스턴스를 위한 getBean()호출을 만들기 위해 BeanFactoryAware인터페이스를
```

```
        구현하는것의 대안처럼 특별히 유용하다. 이 경우 메소드 삽입(injection)은 다소 덜 침략적인(invasive)
        대안이다.
-->
<!ELEMENT lookup-method EMPTY>

<!--
        룩업 메소드의 이름. 이 메소드는 인자를 가져서는 안된다.
-->
<!ATTLIST lookup-method name CDATA #IMPLIED>

<!--
        룩업 메소드가 해석하기 위한 현재 또는 상위 factory내 bean의 이름.
        모든 호출의 구별되는 인스턴스를 반환할 룩업 메소드의 경우 종종 bean은 프로토타입이 될것이다.
        이것은 한개의 쓰레드 객체를 위해 유용하다.
-->
<!ATTLIST lookup-method bean CDATA #IMPLIED>


<!--
        룩업 메소드 기법과 유사하다. replaced-method요소는 메소드를 오버라이딩 하는 IoC컨테이너를
        제어하기 위해 사용된다. 이 기법은 임의의 코드를 가진 메소드의 오버라이딩을 허용한다.
-->
<!ELEMENT replaced-method (
        (arg-type)*
)>

<!--
        IoC컨테이너에 의해 대체되는 구현물의 메소드 이름.
        만약 이 메소드가 오버라이드되지 않는다면 arg-type 하위 요소를 사용할 필요가 없다.
        만약 이 메소드가 오버라이드된다면 arg-type 하위 요소는 메소드를 위한 정의를 모두 오버라이드하기
        위해 사용되어야만 한다.
-->
<!ATTLIST replaced-method name CDATA #IMPLIED>

<!--
        현재또는 상위 factory내 MethodReplacer인터페이스 구현물의 bean이름.
        이것은 싱글톤이나 프로토타입 bean이 될수 있다. 만약 프로토타입이라면 새로운 인스턴스는 각각의
        메소드 대체를 위해 사용될것이다. 싱글톤 사용이 일반적이다.
-->
<!ATTLIST replaced-method replacer CDATA #IMPLIED>

<!--
        replaced-method의 하위요소는 메소드 오버라이드할때 대체되는 메소드를 위한 인자를 확인한다.
-->
<!ELEMENT arg-type (#PCDATA)>

<!--
        문자열처럼 오버라이드된 메소드 인자의 타입을 명시.
        편의를 위해 이것은 FQN의 부분 문자열이 될수 있다. 이를테면 다음의 모두는 "java.lang.String"와
        대응될것이다.
        - java.lang.String
        - String
        - Str

        인자의 숫자처럼 체크될것이다. 이 편의는 종종 타이핑을 줄이기 위해 사용될수 있다.
-->
<!ATTLIST arg-type match CDATA #IMPLIED>


<!--
        이 factory나 외부 factory(부모나 포함된 factory)내 다른 bean에 대한 참조를 정의.
-->
<!ELEMENT ref EMPTY>

<!--
```

```
	참조는 대상 bean의 이름을 명시해야만 한다. "bean"속성은 수행시 체크되기 위한 컨텍스트내 어떤
	bean의 이름을 참조할수 있다. "local"속성을 사용하는 local참조는 bean id들을 사용한다.
	그것들은 이 DTD에 의해 체크될수 있다. 게다가 같은 bean factory XML파일내 참조를 위해 선호된다.
-->
<!ATTLIST ref bean CDATA #IMPLIED>
<!ATTLIST ref local IDREF #IMPLIED>
<!ATTLIST ref parent CDATA #IMPLIED>


<!--
	이 factory나 외부 factory(부모또는 포함된 factory)내 다른 bean의 id가 되어야만 하는 문자열 프라퍼티
	값을 정의. 정규의 "value"요소는 같은 효과를 위해 대신 사용될수 있다. 이 경우 idref를 사용하는것은
	xml파서에 의해 local bean id의 휴효성 체크와 헬퍼 툴(helper tool)에 의한 이름 완성을 허용한다.
-->
<!ELEMENT idref EMPTY>

<!--
	ID refs는 대상 bean의 이름을 명시해야만 한다. "bean"속성은 bean factory구현물에 의해 수행시
	잠재적으로 체크될 컨텍스트내 어느 bean의 이름을 참조할수 있다. "local" 속성을 사용하는 local참조는
	bean id를 사용한다. 그것들은 이 DTD에 의해 체크될수 있다. 게다가 같은 bean factory XML파일내
	참조를 위해 선호된다.
-->
<!ATTLIST idref bean CDATA #IMPLIED>
<!ATTLIST idref local IDREF #IMPLIED>


<!--
	프라퍼티 값의 문자열 표현을 포함한다. 이 프라퍼티는 문자열이나 자바빈 PropertyEditor를 사용하여
	요구되는 타입으로 변환될수 있다. 애플리케이션 개발자가 문자열을 객체로 형변환할수 있는
	사용자정의 PropertyEditor구현물을 쓰는것이 가능하다.

	간단한 객체만 추천된다. 다른 bean에 대한 참조를 가지고 자바빈 프라퍼티를 활성화하여 좀더 복잡한
	객체를 설정하라.
-->
<!ELEMENT value (#PCDATA)>

<!--
	value태그는 값이 형변환될수 있는 정확한 타입을 명시하기 위한 선택적인 type속성을 가질수 있다.
	대상 프라퍼티나 생성자의 인자의 타입이 일반적(generic)일때만 필요하다. 예를 들어, 이 경우
	collection요소.
-->
<!ATTLIST value type CDATA #IMPLIED>

<!--
	자바 null값을 표시한다. 빈 "value"태그는 빈 문자열로 해석하기 때문에 특별히 PropertyEditor가 그렇게
	하지 않는다면 null값으로 해석하지 않을것이다.
-->
<!ELEMENT null (#PCDATA)>


<!--
	리스트는 다중 내부 bean, ref, collection, 또는 value요소를 포함할수 있다. 자바 list는 포함되지 않는다.
	자바 1.5내 일반적인 지원에 참조는 강력하게 형태화(typed)될것이다. 리스트는 또한 배열타입으로 맵핑될수
	있다. 필요한 규칙은 BeanFactory에 의해 자동적으로 수행된다.
-->
<!ELEMENT list (
	(bean | ref | idref | value | null | list | set | map | props)*
)>

<!--
	세트는 다중 내부 bean, ref, collection, 또는 value요소를 포함할수 있다. 자바 set은 포함되지 않는다.
	자바 1.5내 일반적인 지원에 참조는 강력하게 형태화(typed)될것이다.
-->
<!ELEMENT set (
	(bean | ref | idref | value | null | list | set | map | props)*
```

```
)>


<!--
	Spring map은 문자열 key로 객체를 맵핑한다. map은 아마도 비어있을것이다.
-->
<!ELEMENT map (
	(entry)*
)>

<!--
	map항목은 내부 bean, ref, value, 또는 collection이 될수 있다.
	항목의 key는 "key"속성이나 자식 요소에 의해 주어진다.
-->
<!ELEMENT entry (
  key?,
	(bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
	각각의 map요소는 속성이나 자식 요소처럼 이것의 key를 명시해야만 한다.
	key속성은 언제나 문자열값이다.
-->
<!ATTLIST entry key CDATA #IMPLIED>

<!--
	"ref bean=" 자식 요소를 가진 "key"요소에 단순화된 대안.
-->
<!ATTLIST entry key-ref CDATA #IMPLIED>

<!--
  자식 요소인 "value"에 단순화된 대안.
-->
<!ATTLIST entry value CDATA #IMPLIED>

<!--
	자식 요소인 "ref bean="에 단순화된 대안.
-->
<!ATTLIST entry value-ref CDATA #IMPLIED>

<!--
	key요소는 내부 bean, ref, value, 또는 colllection을 포함할수 있다.
-->
<!ELEMENT key (
	(bean | ref | idref | value | null | list | set | map | props)
)>


<!--
	props요소는 문자열이 되어야만 하는 값내에서 map요소와 다르다.
	props는 아마도 비어있을것이다.
-->
<!ELEMENT props (
	(prop)*
)>

<!--
	요소 내용은 프라퍼티의 문자열 값이다.
	여백은 전형적인 XML형태에 의해 야기되는 원치않는 여백을 피하기 위해 잘라낸다.
-->
<!ELEMENT prop (#PCDATA)>

<!--
	각각의 프라퍼티 요소는 이것의 key를 명시해야만 한다.
-->
```

```
<!ATTLIST prop key CDATA #REQUIRED>
```