java/j2ee Application Framework

2.0

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 IoC 컨테이너를 사용할수 있다. 하지만 당신은 Hibernate 통합 코드나 JDBC추상레이어를 사용하도록 선택할수 있다. Spring은 비침략적이도록 디자인되었고, 프레임워크에서 의존성이 의미하는것은 대개 아무것도 없다(또는 사용 영역에 의존해서 굉장히 최소한적이다).

이 문서는 Spring의 기능에 대한 참조가이드를 제공한다. 이 문서는 여전히 작업중이기 때문에 만약 당신이 어떠한 요청이나 언급을 가진다면 사용자 메일링 리스트나 지원 포럼 (http://forum.springframework.org/)에 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

배경

2004년초, Martin Fowler는 그의 사이트 독자들에거 물었다 : Inversion of Control에 관해 이야기할때 “질문은, 제어의 측면에서 무엇이 역행하는가?” 였다. 파울러는 개명(또는 용어 자체가 좀더 의미를 잘 설명하는)한 이론을 제안했고 의존성 삽입 라는 개념을 사용하기 시작했다. 그의 글는 Inversion of Control(IoC) 또는 Dependency Injection(DI) 뒤에 숨겨진 굉장한 아이디어들을 설명하기위해 계속된다.

IoC와 DI에 대한 어느정도 통찰력이 필요하다면 : http://martinfowler.com/articles/injection.html을 방문하라.

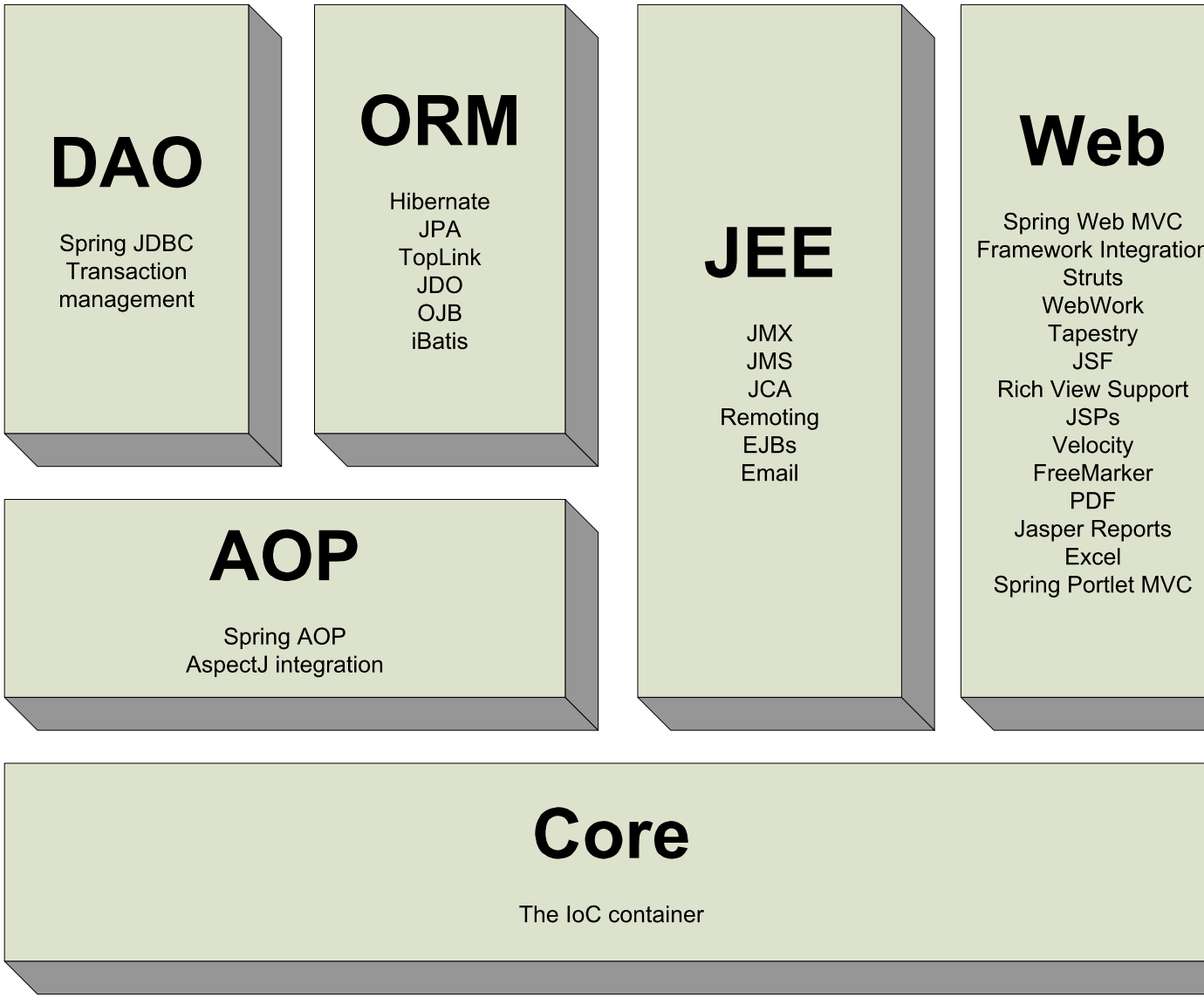
Java애플리케이션은 애플리케이션을 형성하는 다른것과 협력하는 많은 수의 객체로 구성된다. 애플리케이션내 객체는 스스로간에 의존성을 가진다고 말할수 있다.

Java언어와 플랫폼은 풍부한 기능의 애플리케이션 서버와 웹 프레임워크를 위해 원시타입과 클래스를 기본적으로 빌드하는 것으로 부터 방법을 분류하여 아키텍트를 위한 풍부한 기능을 제공하고 애플리케이션을 빌드한다. 부재로 인해 결정적으로 눈에 띄는 영역은 기본적인 빌딩 블럭과 그것을 조합한 것을 일관성있는 전체로 가지는 수단이다. 이 영역은 애플리케이션을 빌드하여 아키텍트와 개발자가 제공하는 것을 남는다. 여기에는 다양한 클래스와 객체 인스턴스를 조합한 비지니스를 돌리는 많은 수의 디자인 패턴이 있다. Factory, Abstract Factory, Builder, Decorator, 그리고 Service Locator 와 같은 디자인 패턴은 소프트웨어 개발 산업에서 널리 보급된 인식과 수용을 가진다. 이것들 모두는 매우 좋지만, 이러한 패턴은 패턴이 하는 것에 대한 상세설명과 함께 이 패턴이 가장 잘 적용되는 그리고 이 패턴이 할당된 애플리케이션의 문제에 대해 주어진 이름이 가장 좋은 구현형태를 나타낸다. 마지막 구문이 “패턴이 하는 것의 상세설명 ” 을 사용한다. 패턴책과 위키는 당신이 제거하고 곰곰히 생각하고 애플리케이션에 스스로 구현할수있는 형상화된 가장 좋은 상황의 목록이다.

Spring 프레임워크의 IoC컨포넌트는 다양한 개별 컨포넌트를 조합하는 형상화된 방법을 완전히 작동하는 애플리케이션에 제공하여 클래스, 객체 그리고 애플리케이션을 조합하는 서비스를 가져오는 기업적인 개념을 할당한다. Spring 프레임워크는 수많은 애플리케이션에 수년간 증명되고 디자인 패턴처럼 형상화된 가장 좋은 형태를 가진다. 그리고 당신이 아키텍트와 개발자처럼 제거하고 자체적인 애플리케이션에 통합할수 있는 첫번째 클래스객체처럼 이러한 패턴을 체계화한다. 이것은 견고하고 유지보수가 가능한 애플리케이션을 처리하기 위해 Spring프레임워크를 사용한 수많은 조직과 기구에 의해 입증된것처럼 정말로 매우 좋은 것이다.

1.1. 개요

Spring은 밑의 다이어그램에서 보여지는 7개의 모듈로 잘 조직된 많은 기능을 포함한다. 이 장은 순서대로 각 모듈을 언급한다.



Spring 프레임워크의 개요

Core 패키지는 프레임워크의 가장 기본적인 부분이고 IoC와 의존성 삽입(Dependency

Injection-DI)기능을 제공한다. 여기의 기본적인 개념은 프로그램에 따른 싱글톤의 필요성을 제거하는 factory패턴의 정교한 구현물을 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, 국제화(I18N)(예를 들어 resource bundle을 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.

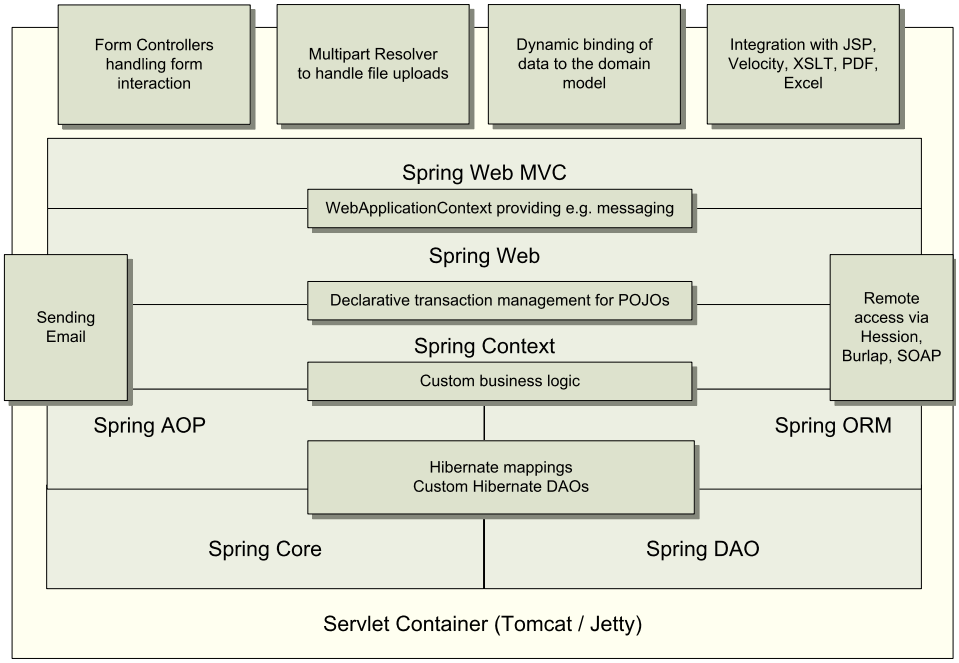
Spring의 AOP 패키지는 당신이 정의하는것을 허용하는 AOP 제휴 호환 aspect-지향 프로그래밍 구현물을 제공한다. 예를 들어 코드를 명백하게 분리하기 위한 메소드-인터셉터와 pointcut은 논리적으로 구별되어야 할 기능을 구현한다. 소스레벨 메터데이타 기능을 사용하여 당신은 .NET속성과 다소 비슷한 모든 종류의 행위적 정보를 당신의코드로 결합한다.

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

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

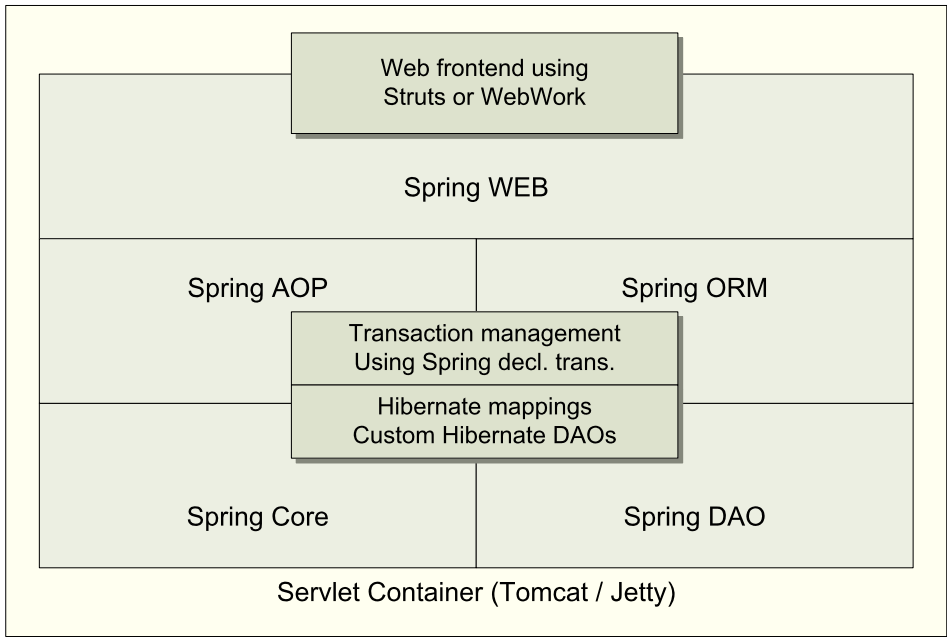
1.2. 사용 시나리오

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

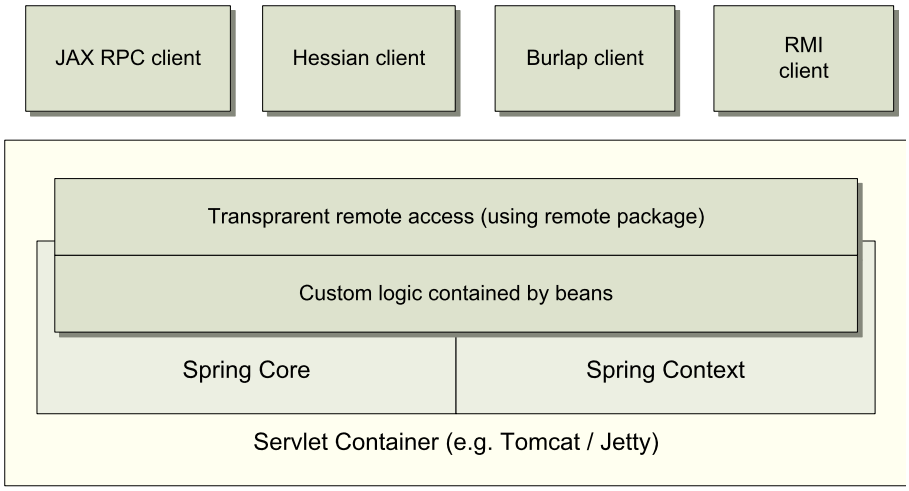
Spring의 선언적인 트랜잭션 관리 기능을 사용하여, 웹 애플리케이션은 EJB에 의해 제공되는것과 같은 컨테이너 관리 트랜잭션을 사용할때 되는것처럼 완벽하게 트랜잭션적이다. 당신의 모든 사용자 정의 비지니스 로직은 Spring의 의존성 삽입 컨테이너에 의해 관리되는 간단한 POJO를 사용해서 구현될수 있다. 메일을 보내거나 유효성체크를 위한 지원을 포함하는 서비스, 웹 레이어의 비의존성은 당신에게 유효성체크 규칙을 수행하기 위한 위치를 선택하도록 허용한다. Spring의 ORM지원은 JPA, Hibernate, JDO 그리고 iBATIS와 통합된다. 예를 들어 Hibernate를 사요할때 당신은 존재하는 Hibernate맵핑을 지속적으로 사용하고 표준 Hibernate SessionFactory설정을 사용할수 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

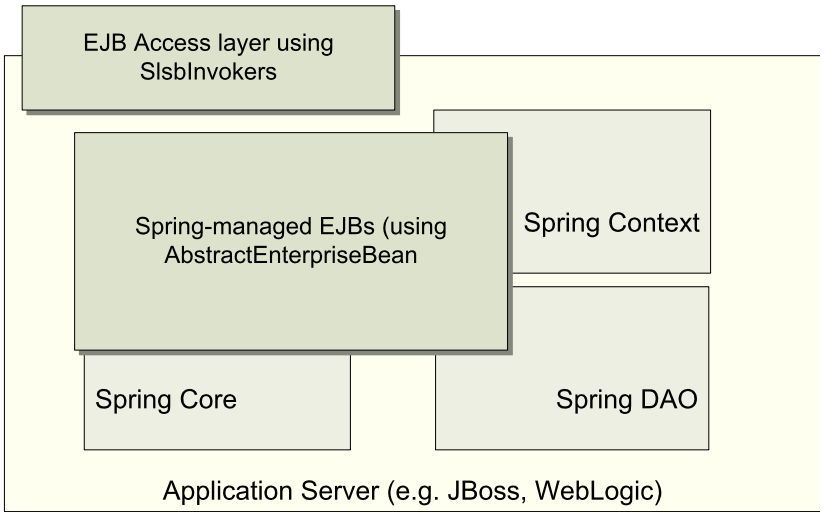
때때로 현재의 환경은 당신에게 다른 프레임워크로의 완벽한 교체를 허용하지 않는다. Spring은 이것내

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- 그리고 추상- 레이어를 제공한다.

Chapter 2. Spring 2.0에서 새로운 것

2.1. 소개

가끔 Spring프레임워크를 사용했다면 아마도 많은 변화가 있었음을 알아차렸을것이다.

> JDK 지원
>
> Spring프레임워크가 Java 1.3이후 모든 Java와 호환되도록 충고해달라. 비록 Java 1.3을 사용할때 사용하지 못하는 Spring프레임워크의 몇가지 향상된 기능이 있지만 이것은 1.3, 1.4, 그리고 1.5가 지원되는 것을 의미한다.

이 개정은 수많은 새 기능을 포함하고 현재 존재하는 많은 기능이 검토되고 향상되었다. 사실 Spring의 많은 부분은 Spring개발팀이 Spring릴리즈 버전을 증가시키길 결정할정도로 잘되어 있고 향상되었다. 그리고 Spring 2.0은 플로리다의 Spring Experience 컨퍼런스에서 2005년 12월에 발표되었다.

이 장은 Spring 2.0의 새롭고 향상된 기능을 진술하기 위해 가이드처럼 제공된다. 이것은 Spring아키텍트와 개발자가 Spring 2.0기능에 즉시 친숙해질수 있도록 해준다. 기능에 대해 좀더 깊이 있는 정보를 위해서, 이 장에서 링크되어 있는 관련부분을 참조하라.

아래에서 언급된 새롭고 향상된 몇가지 기능은 Spring 1.2.x릴리즈에 적용되었거나 적용될것이다. 이전버전으로 적용되었는지를 보기 위해 1.2.x릴리즈의 changelog를 보라.

2.2. Inversion of Control (IoC) 컨테이너

2.0에서 향상된 많은 부분은 Spring IoC컨테이너이다.

2.2.1. 좀더 쉬운 XML설정

Spring XML설정은 지금 좀더 쉬워졌다. XML스키마에 기초하여 새로운 XML설정 문법의 등장에 감사한다. Spring이 제공하는(그리고 Spring 팀은 그러한 문법이 xml을 다소 덜 장황하고 읽기 쉽도록 해주기 때문에 사용하도록 권한다.) 새로운 태그의 장점을 얻고자 할때는, Appendix A, XML 스키마-기반 설정를 보라.

관련노트에서, XML스키마에 기초한 설정의 장점을 가질수 없다면 당신이 참조하도록 Spring 2.0을 위한 새롭고, 업데이트된 DTD가 있다. DOCTYPE선언은 편리성을 위해 아래에 포함되어 있지만, 흥미를 가진 독자는 Spring 2.0배포판의 'dist/resources' 디렉토리에 포함된 'spring-beans-2.0.dtd' DTD를 읽을것이다.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
```

2.2.2. 새로운 bean scope

Spring의 이전버전은 정확히 두개의 bean scope(singleton 과 prototype)를 위한 IoC컨테이너 레벨의 지원을 가졌다. Spring 2.0은 Spring이 배치된 환경에 의존하는 추가적인 scope(예를 들면, 웹환경에서

request와 session scope)를 제공할 뿐아니라 Spring 사용자가 자체적인 scope를 통합할수 있도록 'hooks'를 제공하여 이것을 향상시켰다.

singleton- 과 prototype- scope의 bean을 위한 참조(그리고 내부) 구현물이 변경되었더라도 변경사항은 마지막 사용자에게는 완전히 그대로이다. 현재 가지고 있던 설정은 변경할 필요가 없고 문제가 되지 않는다.

새롭게 추가된것과 이전의 scope는 Section 3.5, "Bean scopes" 에서 상세히 다루어진다.

2.2.3. 확장가능한 XML작성

XML설정은 작성하기 쉬울뿐 아니라 확장또한 가능하다.

'확장가능한(extensible)'은 애플리케이션 개발자나 이종 프레임워크또는 제품업체가 자체적인 태그를 Spring설정파일에 넣을수 있다는 것을 의미한다. 이것은 당신에게 자체적인 컴포넌트의 특정 설정을 위한 자체적인 도메인 정의 언어를 가지도록 해준다.

사용자정의 Spring태그를 구현하는 것은 모든 애플리케이션 개발자나 프로젝트에 Spring을 사용하는 전사적 아키텍트에 흥미로운 일은 아니다. 우리는 다른 업체가 Spring설정파일에서 사용하기 위한 사용자정의 설정 태그를 개발하는데 흥미를 가지길 기대한다.

확장가능한 설정 기법은 Appendix B, 확장가능한 XML제작에서 좀더 상세하게 다루어진다.

2.3. 관점(Aspect) 지향 프로그래밍 (AOP)

Spring 2.0은 AOP에 대해 많이 향상되었다. Spring AOP프레임워크 자체는 XML내 설정하는게 더 쉽다. 그리고 결과적으로 덜 장황하다. Spring 2.0은 AspectJ pointcut언어와 @AspectJ aspect 선언 스타일과 통합되었다. Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍는 이 새로운 지원을 언급한다.

2.3.1. 좀더 쉬운 AOP XML설정

Spring 2.0은 정규의 Java객체에 의해 지원되는 정의 aspect를 위한 새로운 스키마 지원을 소개한다. 이 지원은 AspectJ pointcut 언어의 장점을 가지고 완전한 타입의(typed) advice를 제공한다(캐스팅 이나 Object[]처리를 하지 않는다). 이 지원에 대한 상세한 정보는 Section 6.3, "Schema-based AOP support" 에서 볼수 있다.

2.3.2. @AspectJ aspects 를 위한 지원

Spring 2.0은 @AspectJ 어노테이션을 사용하여 정의된 aspect를 지원한다. 이러한 aspect는 AspectJ와 Spring AOP간에 공유될수 있다. @AspectJ aspect를 위한 지원은 Section 6.2, "@AspectJ 지원" 에서 언급된다.

2.4. 미들 티어

2.4.1. XML내 선언적인 트랜잭션의 좀더 쉬운 설정법

Spring 2.0내에서 트랜잭션을 설정하는 방법은 꽤 변경되었다. 1.2.x스타일의 설정은 계속 유효하게 사용할수 있지만, 새로운 스타일은 다소 덜 장황하고 추천되는 스타일이다. Spring 2.0은 Spring 컨테이너에 의해 생성되지는 않는 객체를 생성하기 위해 사용할수 있는 AspectJ aspect를 가진다.

Chapter 9, 트랜잭션 관리은 모든 상세한 내용을 포함하고 있다.

2.4.2. JPA

Spring 2.0은 할당하는 것과 이것의 일반적인 사용 패턴의 범위에서 Spring의 JDBC추상 레이어와 유사한 JPA 추상레이어를 다룬다.

만약 당신이 퍼시스턴스 레이어의 충추로 JPA구현물을 사용하는데 관심을 가진다면, Section 12.6, “JPA” 부분에서 상세한 Spring지원을 볼수 있다.

2.4.3. 비동기 JMS

Spring 2.0이전, Spring의 JMS지원은 전송(sending)메시지에 제한되었다. 이 기능(JmsTemplate에 의해 캡슐화되는)이 좋기는 하지만, 비동기적인 메시지의 생성(production)과 소비(consumption)와 같은 완전한 JMS스택을 할당하지는 않는다.

그리고 Spring 2.0은 Section 19.4.2, “비동기적인 수령 - 메시지-기반 POJO” 에서 상세히 언급되는것처럼 비동기적인 형태로 메시지의 소비를 위한 완전한 지원을 다룬다.

2.4.4. JDBC

Spring JDBC추상 프레임워크의 영역에 몇가지 주목할만한 새로운 클래스가 있다. 첫번째는 전통적인 JDBC statement의 (‘?’) 인자에 반대되는 명명 파라미터를 사용하여 JDBC statement프로그래밍을 위한 지원을 제공하는 NamedParameterJdbcTemplate이다.

또다른 새로운 클래스인 SimpleJdbcTemplate은 JdbcTemplate를 사용하는 것을 목표로한다. 사용하기 좀더 쉬운 JdbcTemplate는 java 5를 사용할때만 사용가능하다.

2.5. Web 티어

웹 티어 지원은 Spring 2.0에서 크게 향상되었고 확장되었다.

2.5.1. Spring MVC를 위한 form태그 라이브러리

Spring MVC를 위한 풍부한 JSP태그 라이브러리가 Spring사용자로부터 가장 많이 투표된 JIRA이슈였다.

Spring 2.0은 Spring MVC를 사용할때 좀더 쉬운 JSP페이지를 만들수 있도록 완전한 기능의 JSP태그 라이브러리를 가진다. Spring팀은 JIRA이슈에 관심을 가진 모든 개발자를 만족시킬수 있으리라 확신한다. 새로운 태그 라이브러리는 Section 13.9, “Spring의 폼 태그 라이브러리 사용하기. ” 에서 다루어진다. 새로운 태그의 모든 참조는 Appendix E, spring-form.tld에서 볼수 있다.

2.5.2. Spring내 기능위주의 디폴트

많은 프로젝트를 위해, 확립된 규칙을 고수하고 합리적인 디폴트를 가지는 것이 필요하다. 설정을 넘어서는 규칙의 생각은 현재 Spring MVC에서 명시적으로 지원한다. 이것이 의미하는 것은 당신이 Controllers를 위한 명명규칙의 세트를 수립한다면, 당신은 핸들러 맵핑, view resolver, ModelAndView, 등등을 셋업하기 위해 필수인 설정의 양을 크게 줄인다. 이것은 빠른 프로토타이핑을 위해서는 굉장한 이익이 되고 코드기반으로 일관성을 부여할수 있다.

Spring MVC의 설정을 넘어선 규칙 지원은 Section 13.11, “설정에 대한 규칙” 에서 상세히 설명된다.

2.5.3. 포틀릿 프레임워크

Spring 2.0은 Spring MVC프레임워크와 비슷한 개념의 포틀릿 프레임워크를 가진다. Spring 포틀릿 프레임워크의 상세한 정보는 Chapter 16, 포틀릿(Portlet) 통합에서 볼수 있다.

2.6. 그외 모든것들

이 마지막 부분은 Spring 2.0에서 새롭고 향상된 모든것을 포함하고 위 부분에서 포함되지 않는것들이다.

2.6.1. 동적(Dynamic) 언어 지원

Spring 2.0은 Java가 아닌 다른 언어로 작성된 bean을 지원한다. 지원되는 동적 언어는 JRuby, Groovy 그리고 BeanShell이다. 이 동적 언어지원은 Chapter 24, 동적 언어 지원에서 상세히 다루어진다.

2.6.2. JMX

Spring프레임워크는 Notifications(통지)를 위한 지원을 가진다. 이것은 MBeanServer로 MBean의 등록행위를 선언적으로 제어하는 것이 가능하다.

☒ection 20.8, “통지”

☒

2.6.3. 작업 스케줄링

Spring 2.0은 작업의 스케줄링에 대한 추상화를 제공한다. 흥미를 가지는 개발자를 위해, Section 23.4, “Spring TaskExecutor 추상화” 는 상세한 정보를 포함한다.

2.6.4. Java 5 지원

당신이 Java 5를 사용하는 몇 안되는 개발 프로젝트에 투입되었다면, 당신은 Java 5를 위한 몇가지 지원을 가지는 Spring 2.0가 마음에 들것이다. 아래는 Spring Java 5 기능을 가리키는 링크이다.

☒ection 9.5.8, “AspectJ와 @Transactional 를 사용하기”

☒ection 6.8.1, “Using AspectJ to dependency inject domain objects with Spring”

☒ection 6.2, “@AspectJ 지원”

Section 25.3.1, “@Required”

Section 11.2.3, “SimpleJdbcTemplate”

2.7. Spring 2.0으로 이전하기

이 마지막부분은 Spring 1.2.x에서 Spring 2.0으로 이전하는 동안 발생하는 이슈를 상세하게 다룬다.

Spring 1.2 애플리케이션에서 Spring 2.0으로 업그레이드하는 것은 Spring 2.0 jar파일을 애플리케이션의 디렉토리 구조에서 적절한 위치에 간단히 두어야만 한다.

마지막 문장의 키워드는 물론 “해야만 한다는 것이다”. 업그레이드는 당신의 코드에서 사용하는 Spring API가 얼마나 많은지에 대해 항상 균일하거나 의존하지 않는다. Spring 2.0은 많은 클래스와 Spring 1.2 코드기반에서 deprecated된것으로 표시된 메소드를 제거했다. 그래서 이러한 클래스와 메소드를 사용한다면, 당신은 물론 대안이 되는 클래스와 메소드를 사용해야만 할것이다.

설정에 관해서, Spring 1.2.x 스타일의 XML설정은 Spring 2.0라이브러리에서 100% 완전하게 호환된다. 물론 Spring 1.2.x DTD를 여전히 사용한다면, Spring 2.0기능의 새로운 장점(scopes 와 좀더 쉬운 AOP 그리고 트랜잭션 설정)을 몇가지 얻을수 없을것이다.

제안되는 이전 전략은 이 릴리즈에 존재하는 향상된 코드로부터 이득을 얻기 위해 Spring 2.0 jar를 두는것이다. 증가된 기초에서, 당신은 Spring 2.0의 새로운 기능과 설정을 사용하여 시작하는 것을 선택한다. 예를 들어, 당신은 새로운 Spring 2.0 스타일로 aspect를 설정하는 것을 시작하도록 선택할수 있다. 이것은 예전 스타일인 Spring 1.2.x설정(1.2.x DTD를 참조하는)을 사용하는 설정의 90%정도 완전히 유효하고 Spring 2.0의 새로운 설정(2.0 DTD 나 XSD를 참조하는)을 사용하여 다른 10%정도가 유효하다. XML설정을 업그레이드하기 위해 Spring 2.0라이브러리를 두도록 강요하지는 않는다.

2.7.1. 변경사항

변경사항의 포괄적인 목록을 위해, Spring 프레임워크 2.0 배포판의 가장 상위 디렉토리에 위치한 'changelog.txt' 파일을 보라.

2.7.1.1. Jar 패키징

Spring 프레임워크 jar파일의 패키징은 1.2.x 와 2.0 릴리즈간에 상당히 변경되었다. 특히, JDO, Hibernate 2/3, TopLink ORM통합 클래스를 위한 전용 jar파일이 존재한다. 그것들은 더이상 핵심 'spring.jar' 파일에 포함되지 않는다.

2.7.1.2. XML 설정

Spring 2.0 은 이전 버전의 DTD보다 좀더 풍부한 형태의 Spring XML 메타데이터 포맷을 언급하는 XSD를 가진다. 예전의 DTD는 여전히 완벽히 지원된다. 하지만 가능하다면 당신은 bean정의 파일의 가장 상위에 XSD파일의 참조를 두도록 권유한다.

변경된 사항은 bean 범위(scope)가 정의되는 방법이다. Spring 1.2 DTD를 사용한다면, 'singleton' 속성을 계속 사용할수 있다. 당신은 어쨌든 'singleton' 속성의 사용을 허락하지 않는 새로운 Spring 2.0 DTD를 참조하도록 선택할수 있다. 하지만 bean생명주기 범위를 정의하는 'scope' 속성을 사용하라.

2.7.1.3. Deprecated된 클래스와 메소드

@deprecated로 표시된 이전의 많은 클래스와 메소드가 Spring 2.0에서 제거되었다. Spring팀은 2.0을 새로운 시작으로 표시하기로 결정했다. 그리고 deprecated된 '결과'는 코드기반에 영향을 주는것을 지속하는 대신에 현재 삭제된다.

이전에 언급한것처럼, 변경의 포괄적인 목록을 위해, Spring 프레임워크 2.0 배포판의 가장 상위 디렉토리에 위치한 'changelog.txt' 파일을 보라.

다음의 클래스/인터페이스는 Spring 2.0 코드기반으로부터 모두 제거되었다.

☒ ResultReader : RowMapper 인터페이스를 대신 사용하라.

☒ BeanReferenceFactoryBean : 사용가능한 별칭 기법을 대신 사용하라.

☒ BeanDefinitionRegistryBuilder : BeanDefinitionReaderUtils 클래스의 편리한 메소드를 대신 사용하라.

☒ BeanFactoryBootstrap : BeanFactoryLocator나 사용자정의 부트스트랩(bootstrap) 클래스를 대신 사용하는 것을 고려하라.

☒ RequestUtils : ServletRequestUtils 를 대신 사용하라.

2.7.1.4. Apache OJB

Apache OJB를 위한 지원이 Spring 소스트리에서 완전히 제거되었다. Apache OJB통합 라이브러리는 여전히 사용가능하다. 하지만 Spring Modules 프로젝트에 있는 새로운 곳에서 찾을수 있다.

2.7.1.5. iBatis

iBATIS SQL Maps 1.3을 위한 지원이 모두 제거되었다. 필요하다면 iBATIS SQL Maps 2.0/2.1로 업그레이드하라.

2.8. 업데이트된 샘플 애플리케이션

많은 수의 샘플 애플리케이션은 Spring 2.0의 새롭고 향상된 기능을 보여주기 위해 업데이트되었다. 그래서 그 애플리케이션들을 자세히 살펴보라. 이미 언급한 샘플 애플리케이션은 Spring배포판('spring-with-dependecies.[zip|tar.gz]')의 'samples'에서 찾을수 있다.

언급한 배포판은 많은 수의 소위 전시용(showcase)이라 불리는 애플리케이션을 포함한다. 이 전시용 애플리케이션은 범위에 제한되고 Spring 2.0의 새로운 기능에 대한 예제를 제공한다. 당신은 이 전시용 애플리케이션에서 일부 코드를 얻을수 있고 사용할수 있다.

2.9. 향상된 문서

Spring 참조문서는 Spring 2.0의 새로운 기능을 반영하기 위해 업데이트되었다.

문서에서 에러가 없도록 확인하는 동안 몇가지 에러가 생겼다. 어떤 오타나 심각한 에러가 발견된다면, 이슈를 올려서 Spring 팀에 에러를 알려달다.

Part I. Core Technologies

This initial part of the reference documentation covers all of those technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework's Inversion of Control (IoC) container. A thorough treatment of the Spring Framework's IoC container is closely followed by comprehensive coverage of Spring's Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand, and which successfully addresses the 80% sweet spot of AOP requirements in Java enterprise programming.

Coverage of Spring's integration with AspectJ (currently the richest - in terms of features - and certainly most mature AOP implementation in the Java enterprise space) is also provided.

Finally, the adoption of the test-driven-development (TDD) approach to software development is certainly advocated by the Spring team, and so coverage of Spring's support for integration testing is covered (alongside best practices for unit testing). The Spring team have found that the correct use of IoC certainly does make both unit and integration testing easier (in that the presence of setter methods and appropriate constructors on classes makes them easier to wire together on a test without having to set up service locator registries and suchlike)... the chapter dedicated solely to testing will hopefully convince you of this as well.

- Chapter 3, IoC 컨테이너
- Chapter 4, 자원
- Chapter 5, 유효성체크(Validation), 데이터-바인딩, BeanWrapper, 와 PropertyEditors
- Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍
- Chapter 7, Spring AOP APIs
- Chapter 8, Testing

Chapter 3. IoC 컨테이너

3.1. 소개

이 장은 제어의 역전(IoC) 개념의 Spring 프레임워크의 구현물을 다룬다. [1]IoC는 Spring 프레임워크가 제공하는 많은 주변 기능들의 전반적인 토대가 되며, 풍부하면서도 개념적으로는 매우 단순한 이 기술을 순서대로 철저하게 다룰 것이다.

> 어느 인터페이스?
>
> 사용자는 BeanFactory 나 ApplicationContext 중 어느것이 특별한 상황에서 사용하기 위해 가장 적합한지 확실하지 못한다. 대개 J2EE환경에서 대부분의 애플리케이션을 개발할때, 사용하기 가장 좋은 선택은 ApplicationContext이다. 이는 BeanFactory의 모든 기능을 제공하면서 또한 일반적으로 많은 이들이 원하는 몇몇 기능을 사용하는데 있어 좀더 선언적인 접근방식(declarative approach)을 제공하기도 한다.

org.springframework.beans과 org.springframework.context 패키지는 Spring 프레임워크의 IoC컨테이너를 위한 기초를 제공한다. BeanFactory 인터페이스는 어떠한 성질의 객체라도 관리할 수 있는 고급 설정 기법을 제공한다. ApplicationContext 인터페이스는 BeanFactory 위에 구축되었고(BeanFactory를 상속받음), Spring의 AOP기능, 메시지 자원 핸들링(국제화에 사용됨), 이벤트 위임, 웹 애플리케이션에서 사용하기 위한 WebApplicationContext와 같은 특정 애플리케이션 계층의 컨텍스트를 더 쉽게 통합할 수 있는 등의 다른 기능들을 가지고 있다.

다시 말해, BeanFactory는 설정 프레임워크와 기본 기능을 제공하는 반면에, ApplicationContext는 좀더 전사적 중심(enterprisse-centric)의 기능을 거기에 보태고 있다. ApplicationContext는 BeanFactory의 완벽한 수퍼세트(BeanFactory의 기능을 모두 포함하면서, 다른 기능을 더 추가한 상태)이다. BeanFactory의 모든 기능과 행태는 ApplicationContext에도 마찬가지로 적용된다고 보면 된다.

이 장은 BeanFactory 와 ApplicationContext 모두에 적용하는 기본 개념을 다루는 첫번째 부분 과 ApplicationContext 인터페이스에만 적용되는 기능을 다루는 두번째 부분의 두개의 부분으로 나뉘었다.

3.2. 기초 - 컨테이너와 beans

Spring에서, 애플리케이션의 뼈대를 형성하고 Spring IoC컨테이너에 의해 관리되는 객체를 beans라고 한다. bean은 간단히 객체라 할 수 있으며, Spring IoC container에 의해 일반적으로 인스턴스화되고 조합되고 관리된다. 그 외에는 bean에는 별반 특별할 것이 없다(이는 모든 다른 점에 있어서 여러분의 어플리케이션에 있는 아마도 많은 객체들중의 하나일 뿐이다). 이러한 bean들과 그것들간의 의존성은 컨테이너에 의해 사용되는 설정 메타데이터에 의해 반영된다.

> 왜.. bean 인가?
>
> 'component' 나 'object'가 아닌 'bean' 이라는 이름을 사용한 까닭은 Spring 프레임워크 자체의 기원에 유래를 두고 있다(Spring은 부분적으로는 Enterprise JavaBeans의 복잡성에 대한 대안으로 나왔다).

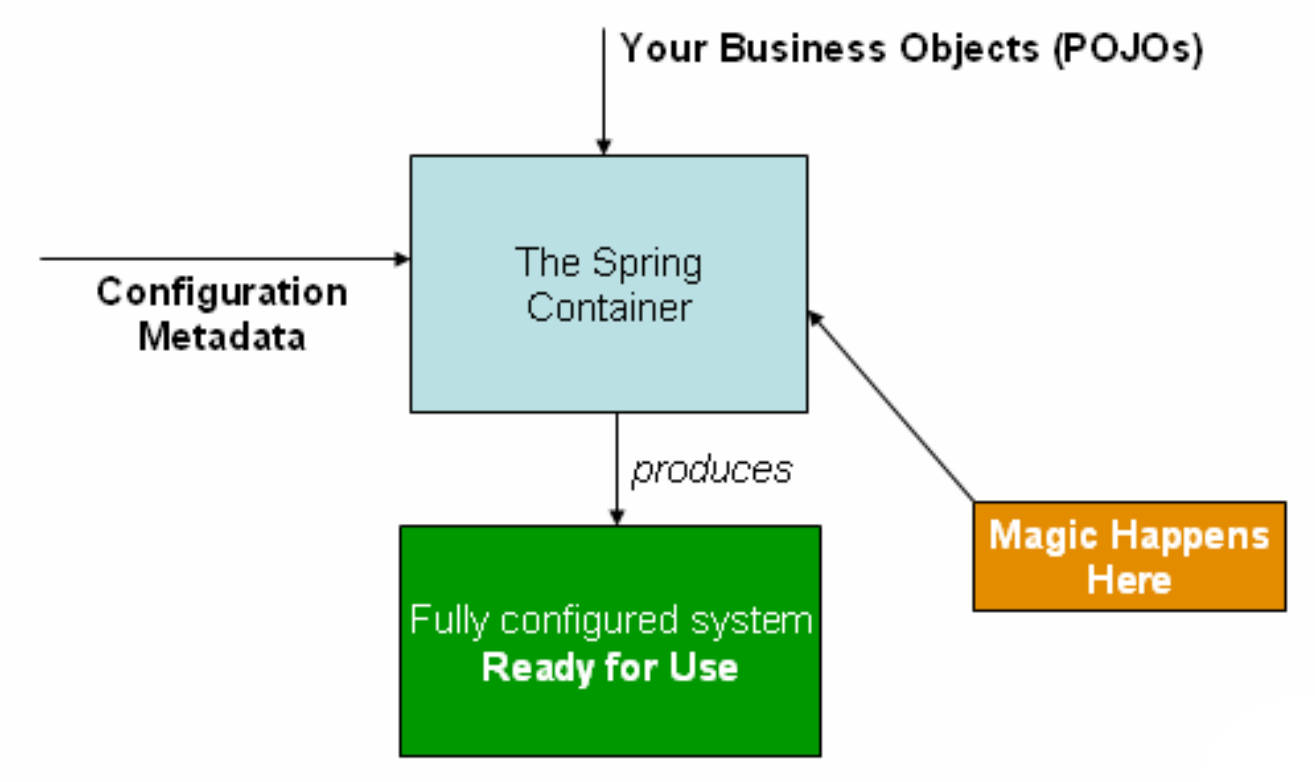
[1]배경를 보라.

3.2.1. 컨테이너

org.springframework.beans.factory.BeanFactory는 앞서말한 bean을 포함하고 관리하는 책임을 지는 Spring IoC 컨테이너의 실제 표현이다.

BeanFactory은 Spring에서 핵심 IoC 컨테이너 인터페이스이다. 이것은 어플리케이션의 객체를 인스턴스화하거나 소싱(다른곳에서 생성된 객체를 가져오기)하고 해당 객체를 설정하고 각 객체간의 의존성을 조합하는 책임을 맡고 있다.

Spring과 함께 바로 제공되는 뛰어난 BeanFactory 인터페이스의 많은 구현(implementations)이 있다. 가장 공통적으로 사용되는 BeanFactory 구현은 XmlBeanFactory 클래스이다. 이 구현을 통해 XML로 애플리케이션을 구성하는 객체과 그 객체들간의 의심할 여지없이 풍부한 상호의존 관계를 표현할 수 있다. XmlBeanFactory는 이 XML 설정 메타데이터를 가지고 완전하게 설정된 시스템이나 애플리케이션을 생성한다.



Spring IoC 컨테이너

3.2.1.1. 설정 메타데이터

위 이미지에서 볼수 있는 것처럼, Spring IoC 컨테이너는 설정 메타데이터의 몇가지 형태를 지원한다; 이 설정 메타데이터는 Spring 컨테이너에 "[애플리케이션내 객체를]인스턴스화, 설정, 그리고 조합" 하는 방법을 지정하는 방법에 지나지 않는다. 이 설정 메타데이터는 대개 간단하고 직관적인 XML형태로 제공된다. XML-기반의 설정 메타데이터를 사용할때, 당신은 Spring IoC 컨테이너가 관리해주길 원하는 bean을 위한 bean정의를 작성하고, 컨테이너가 할일을 하게 두면 된다.

Note

XML-기반의 메타데이터는 설정 메타데이터의 가장 일반적으로 사용되는 형태이다. 그렇다고 이것이 사용가능한 유일한 메타데이터의 형태는 아니다. Spring IoC 컨테이너 자체는 설정 메타데이터의 형태와는 전혀 결합되지 않는다.(decoupled)

개발시에는 XML, Java 프라퍼티 형태, 또는 프로그램으로 처리(Spring의 public API를 사용하여)하는 설정 메타데이터를 제공할수 있다. XML-기반 설정 메타데이터 형태는 매우 간단하다. 그래서 이 장의 나머지는 핵심 개념과 Spring IoC 컨테이너의 기능을 전달하기 위해 XML 형태를 사용할 것이다.

> Resource
>
> 일단 IoC에 대한 기본을 공부한 뒤에, Chapter 4, 자원 에 언급된, Spring의 리소스(Resource) 추상화에 대해 배우는것이 유용할 것이다.
>
> ApplicationContext 생성자에 제공되는 위치 경로 혹은 경로은 Java CLASSPATH에서부터 로컬 파일시스템과 같이 다양한 외부 자원으로 부터 설정 메타데이터를 읽어들일 수 있도록 하는 리소스 문자열이다.

광범위한 주요 애플리케이션 작성 시나리오에서는, Spring IoC 컨테이너의 하나 이상의 인스턴스를 생성하기 위해서 사용자가 명시적으로 코딩할 필요가 없음을 주의하기 바란다. 예를 들어, 웹 애플리케이션의 경우, 애플리케이션의 관련 web.xml 파일내에 간단히 8줄 정도의 반복적으로 사용되는 J2EE 웹 서술자 XML 구문으로 충분할것이다(Section 3.9.4, “웹 애플리케이션을 위한 편리한 ApplicationContext 인스턴스화” 를 보라).

가장 기본적인 수준으로는, Spring IoC 컨테이너 설정은 컨테이너가 관리해야만 하는 적어도 하나의 bean 정의로 구성되지만, 보통은 1개 이상의 bean을 정의한다. XML-기반의 설정 메타데이터를 사용할때, 이러한 bean은 가장 상위 레벨의 <beans/>요소내부에 하나 혹은 그 이상의 <bean/> 요소로 설정된다.

이러한 bean 정의는 애플리케이션을 구성하는 실질적인 객체에 대응한다. 일반적으로 이것은 서비스 계층 객체, 데이터 접근 객체(DAO) 혹은 Struts Action과 같은 표현 객체, Hibernate SessionFactory와 같은 하부 기반 구조를 이루는 객체, JMS Queue 참조 객체 등을 위한 bean정의를 가질것이다. (bean이 정의할 수 있는 범위는 물론 제한이 없고, 단지 애플리케이션의 범위와 복잡성에 의해 제한될 뿐이다).

아래에서 XML-기반의 설정 메타데이터의 기본 구조의 예제를 볼 수 있다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  <bean id="..." class="...">
    <!-- 이 bean을 위한 도우미와 설정이 여기에 온다. -->
  </bean>
  <bean id="..." class="...">
    <!-- 이 bean을 위한 도우미와 설정이 여기에 온다.  -->
  </bean>
  <!--더 많은 bean 정의들이 있을 수 있다... -->
</beans>
```

3.2.2. 컨테이너 인스턴스화하기

Spring IoC 컨테이너를 인스턴스화하는 것은 쉽다. 아래에서 그 방법을 보자.

```
Resource resource = new FileSystemResource("beans.xml");
       BeanFactory factory = new XmlBeanFactory(resource);
```

... 또는...

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

... 또는...

```
ApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml", "applicationContext-part2.xml"});
        //물론 ApplicationContext는
        BeanFactory이기도 하다
        BeanFactory factory = (BeanFactory) context;
```

3.2.2.1. XML-기반 설정 메타데이터 작성하기

컨테이너 정의를 여러 XML파일로 분리하는 것이 종종 도움이 될 수도 있다. 그리고서 이러한 XML 파일들로 설정된 애플리케이션 컨텍스트를 적재하는 한가지 방법은 여러 리소스 경로를 파라미터로 받는 애플리케이션 컨텍스트 생성자를 사용하는 것이다. bean factory를 사용하여, bean 정의 리더(reader)는 순서대로 각각의 파일로부터 bean 정의를 읽는데 여러번 사용될수 있다.

일반적으로, Spring팀은 위와 같은 접근 방식을 선호하는데, 그것은 컨테이너 설정 파일이 다른 파일들과 조합된다는 사실을 각 설정파일이 서로 알필요가 없어도 되기 때문이다. 다른 방법으로는 다른 파일(혹은 파일들)로 부터 bean 정의를 적재하기 위해 하나 혹은 그 이상의 <import/> 요소를 사용하는 것이다. <import/> 요소는 import를 수행하는 파일에서 <bean/> 요소가 나오기 이전에 두어야만 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="..."/>

        </beans>
```

이 예제에서, 외부 bean정의가 3개의 파일(services.xml, messageSource.xml과 themeSource.xml)로부터 적재된다. 모든 위치 경로는 import를 수행하는 정의파일에 대해 상대적이다. 그래서 이 경우에 messageSource.xml 과 themeSource.xml이 import 하는 파일의 위치 아래의 resources 에 있어야만 하는 반면에 services.xml은 import를 수행하는 파일과 같은 디렉토리나 클래스패스 경로내 두어야만 한다. 여기 보듯이 맨 앞의 슬래쉬(/)는 실제로는 무시된다. 하지만 상대경로를 나타내려 하기 때문에 비록 슬래쉬가 무시되더라도 슬래쉬 없이 사용하는 것이 더 나은 이다.

import될 파일의 내용은 <beans/>요소를 가장 상위 레벨에 포함하는 스키마나 DTD에 따라 완전히 유효한 XML bean정의 파일이어야만 한다.

3.2.3. beans

이전에 언급된것처럼, Spring IoC 컨테이너는 하나 혹은 그 이상의 bean을 관리한다. 이러한 bean은 컨테이너에 제공된 bean을 가진 설정 메타데이터내 정의(대개 XML <bean/> 정의의 형태로)된 지시사항을 사용하여 생성된다.

컨테이너 내부에서는, 이러한 bean 정의는 다른 여러 정보들 중에서도 특히 다음의 메타데이터를 포함하는 BeanDefinition객체로 표현된다.

☒ 패키지명을 포함하는 클래스명 : 이것은 보통 정의된 bean의 실제 구현 클래스이다. 하지만 bean이 일반적인 생성자를 사용하지 않고 정적(static) factory 메소드를 호출하여 인스턴스화된다면, 이것은 실제로는 factory 클래스의 클래스명이 될것이다.

☒ bean행위적 설정 요소는 bean이 컨테이너 내에서 어떻게 행위를 하는지를 지정한다(이를 테면 프로토타입 또는 싱글톤, autowiring 모드, 의존성 체크 모드, 초기화(initialization)와 제거(destruction)메소드).

☒ 새롭게 생성된 bean내에 셋팅할 생성자 인자와 프라퍼티 값. 예를 들면 커넥션 풀을 관리하는 빈 안에서 (생성자의 인자이거나 프라퍼티로 값으로 지정되는)커넥션의 갯수나 커넥션 풀 최대 크기와 같은 값들이 될 것이다.

☒ 작업을 하는 bean을 위해 필요한 다른 bean, 이를 테면 협력자(collaborators - 혹은 의존성dependency이라고 불리기도 한다).

위에서 나열된 개념들은 각각의 bean정의를 구성하는 프라퍼티의 묶음으로 직접적으로 바뀌게 된다. 그러한 프라퍼티 중의 몇몇은 각각에 대한 좀더 상세한 문서 링크를 포함해서 아래에 목록을 제시하였다.

Table 3.1. Bean 정의

이름	좀더 상세한 정보
class	Section 3.2.3.2, “bean 인스턴스 생성하기”
name	Section 3.2.3.1, “bean 명명하기”
scope	Section 3.5, “Bean scopes”
생성자 인자들	Section 3.3.1, “의존성 삽입하기”
프라퍼티	Section 3.3.1, “의존성 삽입하기”
autowiring 모드	Section 3.3.6, “Autowiring 협력자”
의존성 체크 모드	Section 3.3.7, “의존성을 위한 체크”
게으른 초기화 모드	Section 3.3.5, “Lazily-instantiating beans”
초기화 메소드	Section 3.6.1, “Lifecycle 인터페이스”
제거(destruction) 메소드	Section 3.6.1, “Lifecycle 인터페이스”

지정된 bean을 생성하는 방법을 나타내는 bean정의 외에도, 어떤 BeanFactory 구현물은 factory외부에서 (사용자 코드에 의해)생성되어 존재하는 객체를 등록할 수 있도록 해준다. DefaultListableBeanFactory 클래스는 registerSingleton(..) 메소드를 통해 그런 기능을 지원한다. 일반적인 애플리케이션은 메타데이터 bean정의를 통해 정의된 bean들을 가지고 단독으로 작동한다.

3.2.3.1. bean 명명하기

bean 명명 규칙

(적어도 Spring개발팀 사이에서는)bean의 이름을 지을 때, 인스턴스 필드명은 표준 Java 명명규칙을 사용하도록 한다. 그건 바로 bean의 이름은 소문자로 시작하고 그 이후는 camel-case(첫번째 단어는 소문자로 시작하고 두번째 단어는 대문자로 시작)방식을 따른다. 예를들면 (따옴표는 제외하고) 'accountManager', 'accountService', 'userDao', 'loginController' 등이 될 것이다.

bean을 명명하는 알관성 있는 방법을 적용하는 것은 설정을 좀더 읽기 쉽고 이해하기 쉽도록 만들어준다. 이러한 표준 명명 규칙을 적용하는 것은 그닥 어려운 일이 아니다. Spring AOP를 사용한다면 명명 규칙 적용은 이름으로 관련성을 맺은 bean의 묶음에 advice를 적용할 때 매우 큰 수고를 덜어 줄 수 있다.

모든 bean은 하나 또는 그 이상의 id(식별자 혹은 이름이라고 불리기도 하며, 이 용어들은 모두 같은 것을 나타낸다) 를 가진다. 이러한 id는 bean을 적재하고 있는 컨터이너 내에서 유일해야만 한다. bean은 대개 오직 하나의 id를 가지지만 하나 이상의 id를 가진다면, 추가적인 것들은 본질적으로는 별칭으로 간주할 수 있다.

XML-기반의 설정 메타데이터를 사용할때, bean의 식별자를 명시하기 위해 'id' 나 'name' 속성을 사용한다. 'id' 속성은 정확하게 하나의 id를 명시하도록 허용한다. 그리고 실제 XML요소의 ID 속성이고 XML파서는 다른 요소가 id를 참조할때 몇가지 추가적인 유효성 검사를 할수 있다. 이것은 bean id를 명시하기 위해 권장하는 방법이다. 하지만 XML 스펙에서는 XML ID에 적법한 문자가 제한돼 있다. 이것은 보통 큰 제약은 아니지만, 이러한 특수한 XML 문자중 하나를 사용할 필요가 있거나 bean에 대해 다른 별칭을 사용하길 원한다면, 하나 혹은 그 이상의 id를 사용하는 대신 'name' 속성에서 쉼표 (,), 세미콜론 (;), 또는 공백으로 구분해서 나타내면된다.

bean을 위해 name을 제공하는 것이 필수가 아니다. name이 명시적으로 제공되지 않는다면, 컨테이너는 bean을 위해 (유일한) name을 생성할것이다. bean을 위한 name을 제공하지 않는 까닭은 나중에 언급할 것이다(한가지 사용예로 내부(inner) beans이 있다).

3.2.3.1.1. bean에 별칭정하기

bean 정의 내부에서는, id 속성을 이용해서 최대 한 개의 이름과 그리고 alias속성을 통해 여러개의 다른 이름들을 조합하여 bean에 한 개 이상의 이름을 지정할 수 있다. 이러한 모든 이름은 동일한 bean에 대응하는 별칭으로 간주되며, 한 애플리케이션 내에서 각각의 컴포넌트가 컴포넌트 자체에 명시된 bean이름을 사용하여 공통적인 의존성을 참조하도록 하는 것과 같은 몇가지 상황에서 유용하게 쓰일 수 있다.

하지만 bean을 정의 할 때 모든 별칭을 함께 명시하는 것 만으로는 부족할 때도 있다. 때로는 다른 곳에서 정의된 bean을 위한 별칭을 지정하는 것이 바람직한 경우도 있다. 이것은 XML-기반의 설정 메타데이터에서, 독립적인 <alias/> 요소를 사용하면 가능하다. 예를 들어:

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

이 경우, 동인한 컨테이너 내에 있는 'fromName'이라는 이름의 bean은 별칭을 정의한 뒤에는 'toName'으로 참조할 수 있다.

구체적인 예를들면, XML 일부분에서 컴포넌트 A가 componentA-dataSource라고 불리는 DataSource bean을 정의하는 경우에, 컴포넌트 B는 그 DataSource를 XML 설정의 다른 부분에서 componentB-dataSource와 같이 참조할 것이다. 주 애플리케이션인 MyApp는 자체적인 XML의 일부를 정의하고 3개의 부분 모두로 부터 최종적인 애플리케이션 컨텍스트 조합해낸다. 그리고 그 DataSource를

myApp-dataSource로 참조할것이다. 이 시나리오는 MyApp XML 일부에 다음의 독립적인 별칭을 추가하여 쉽게 다룰수 있다.

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

이제 각각의 컴포넌트와 주 애플리케이션은 유일하면서 다른 정의들과 충돌하지 않을 것이 보장된 이름(사실상 네임스페이스가 있는것이다)을 통해서 dataSource를 참조하면서도, 여전히 그것들은 같은 bean을 참조하는 것이다.

3.2.3.2. bean 인스턴스 생성하기

Spring IoC 컨테이너 관한한, bean 정의는 기본적으로 하나 혹은 그 이상의 실제 객체를 생성하기 위한 조리법 같은 것이다. 컨테이너는 요청이 들어오면 지정된 bean을 위한 조리법을 찾아보고, 해당 bean정의에 의해 캡슐화된 설정 메타데이터를 사용하며 계속 진행해서, 설정 사항들을 반영한 실물 객체를 생성한다. 이번 섹션은 어플리케이션 개발자가 Spring IoC 컨테이너에게 객체가 어떤 타입(혹은 클래스)으로 인스턴스화되는지, 그리고 최종적으로 생성될 객체를 어떻게 인스턴스화하는지 지정하는 방법에 대해 다룰 것이다.

XML 기반의 설정 메타데이터를 사용한다면, <bean/> 요소의 'class' 속성을 사용하여 인스턴스를 생성할 객체의 타입(혹은 클래스)을 명시할수 있다. 이 'class' 속성(내부적으로는 최종적으로 BeanDefinition 인스턴스의 Class프라퍼티가 된다)은 일반적으로 필수 (Section 3.2.3.2.3, "인스턴스 factory 메소드를 사용하여 인스턴스화 생성하기" 와 Section 3.7, "Bean정의 상속" 에 두가지 예외가 있다)이고 두가지 목적 중 하나를 위해 사용된다. 컨테이너 자체가 생성자를 호출('new'를 사용하는 Java코드와 어떤 면에서는 동일한 방식으로)하여 bean을 직접 생성하는 대부분의 일반적인 경우에서는 class 속성에는 생성될 bean의 클래스를 기입한다. 컨테이너가 bean을 생성하기 위해 클래스의 정적(static), factory 메소드를 호출하는 다소 덜 일반적인 상황에서는 class 프라퍼티는 객체를 생성하기 위해 호출되는 static factory메소드를 포함하는 실제 클래스를 기입한다 (static factory메소드의 호출로부터 반환되는 객체의 타입은 class 속성에 지정한 것과 같은 클래스이거나 완전히 다른 클래스여도 상관없다).

3.2.3.2.1. 생성자로 인스턴스 생성하기

생성자를 통한 접근방법으로 bean을 생성할때, 모든 일반적인 클래스는 Spring과 호환되며 사용하는데 문제가 없다. 다시말해, 생성할 클래스는 어떠한 특정 인터페이스를 구현하거나 특정 형태로 코딩할 필요가 없다. 단지 bean클래스를 명시하는 것만으로 충분하다. 하지만 지정된 bean을 위해 어떤 타입의 IoC를 사용할것이냐에 따라서 기본(아무것도 하지 않는) 생성자가 필요할 수도 있다.

추가적으로, Spring IoC 컨테이너는 진정한 JavaBeans만을 관리하는 것으로 제한 받지 않고, 사실상 당신이 원하는 그 어떤 클래스라도 관리할 수 있다. Spring을 사용하는 대부분의 사람들은 실제 JavaBean(기본 생성자와 프라퍼티에 따라서 적절한 setter와 getter를 가지는)을 컨테이너 내에 두길 선호하지만, 좀더 색다르고 bean 스타일이 아닌 클래스도 가능하다. 예를들어, 만약 JavaBean 스펙을 따르지 않는 예전에 만들어진 connection pool을 사용할 필요가 있다하도, Spring은 역시나 그것도 잘 관리한다.

XML-기반 설정 메타데이터를 사용할때 당신은 다음처럼 bean클래스를 명시할 수 있다. :

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

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

생성자에 인자를 제공(필요할 경우에)하는 기법이나, 객체를 생성한 이후에 객체 인스턴스에 프라퍼티 값을 설정하는 방법은 곧 설명할 것이다

3.2.3.2.2. static factory 메소드를 사용하여 인스턴스 생성하기

static factory 메소드를 사용해서 생성할 bean을 정의할 때는 클래스를 명시하는 class 속성에 static factory메소드를 포함하고 있는 클래스를 지정하고, factory-method라는 또 다른 속성에 factory 메소드의 이름을 명시할 필요가 있다. Spring은 그 메소드를 (나중에 알아볼 예정인 추가적인 인자를 함께 전달하는 방법과 함께) 호출할 것이고, 일반적으로 생성자를 통해 생성된 것과 다를 바 없이 사용할 수 있는 생생한 객체를 돌려 받게 될 것이다. 이러한 bean 정의 방식은 예전 방식의 코드에서 static factory를 호출할 때 사용할 수 있다.

다음 예제에서 factory 메소드를 호출하여 생성되는 bean을 정의하는 방법을 볼 수 있다. 이 bean 정의가 반환될 객체의 타입(클래스)를 명시하지 않고, 오직 factory 메소드를 가지고 있는 클래스만을 명시함을 주의해서 보라. 이 예제에서, createInstance() 메소드는 꼭 static 메소드이어야만 한다.

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

factory메소드에 (추가적인)인자를 제공하는 기법이나 factory로부터 객체 인스턴스를 생성한 후에 그 인스턴스에 프라퍼티를 설정하는 것은 곧 설명할 것이다.

3.2.3.2.3. 인스턴스 factory 메소드를 사용하여 인스턴스화 생성하기

static factory method를 통해 인스턴스를 생성하는 것과 유사한 방법으로, 인스턴스 factory 메소드를 사용하여 인스턴스화를 생성한다는 것은 이미 컨테이너에 존재하는 bean의 factory 메소드를 호출하여 새로운 bean을 만들어 냄을 의미한다.

이 기법을 사용하기 위해, 'class' 속성은 설정하면 안되고 동일(혹은 부모/조상) 컨테이너에 있는 factory메소드를 가진 bean의 이름을 'factory-bean' 속성을 명시해야만 한다. factory메소드 자체는 여전히 'factory-method' 속성을 통해 설정하면 된다(아래의 예제에서 보이는것처럼).

```
<!-- createInstance()라는 factory 메소드를 포함하고 있는 factory bean-->
<bean id="myFactoryBean" class="...">
...
</bean>

<!-- factory bean을 통해 생성할 bean -->
<bean id="exampleBean"
  factory-bean="myFactoryBean"
  factory-method="createInstance"/>
```

비록 여전히 bean프라퍼티를 설정하는 메카니즘을 논의할 예정이긴 하지만, 이러식의 접근 방법은 factory bean 자체를 DI를 통해 설정하고 관리할 수 있음을 알 수 있게 해준다.

3.2.4. 컨테이너 사용하기

@@@3.2.4 Using the Container@@@

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");
        BeanFactory factory = new XmlBeanFactory(is);
```

getBean(String)를 사용하여 당신의 bean인스턴스를 가져올수 있다. BeanFactory의 클라이언트측 시각은 놀라울정도로 간단한다. BeanFactory 인터페이스는 호출할 클라이언트를 위해 오직 6개의 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.

☒ Class getType(String name): 주어진 이름을 가진 bean의 Class를 반환한다. 주어진 이름을 가진 bean이 없다면, NoSuchBeanDefinitionException 가 던져질것이다.

☒ boolean isSingleton(String): 주어진 이름하에 등록된 bean정의나 bean인스턴스가 싱글톤인지 아닌지 조사한다(싱글톤과 같은 bean범위는 later에서 설명된다.). 만약 주어진 이름에 관련된 bean이 발견되지 않는다면 NoSuchBeanDefinitionException가 던져질것이다.

☒ String[] getAliases(String): 만약 bean정의내 어떠한 것도 명시되어 있다면 주어진 bean이름을 위한 별칭을 반환한다.

3.3. 의존성

당신의 전형적인 기업용 애플리케이션은 한개의 객체(또는 Spring내 bean)로 만들어지지는 않는다. 가장 간단한 애플리케이션조차도 최종 사용자가 응집성있는 애플리케이션처럼 보는것을 표현하기 위해 함께 작동하는 소량의 객체를 가진다는것을 의심할 필요가 없을것이다. 다음 부분은 독립적인 많은 수의 bean정의를 정의하는 것으로부터 객체가 몇가지 목표(대개 최종사용자가 원하는 것을 수행하는 애플리케이션)를 달성하기 위해 함께 작동하는 전체적으로 실제화된 애플리케이션까지 수행하는 방법을 설명한다.

3.3.1. 의존성 삽입하기

Dependency Injection (DI) 배후의 기본개념은 생성자의 인자, factory메소드에 대한 인자, 또는 factory메소드로부터 생성되거나 반환된후 객체 인스턴스에 셋팅되는 프라퍼티를 통해서만 의존성을 정의하는 객체이다. 이것은 bean을 생성할때 이러한 의존성을 실제로 삽입하기 위한 컨테이너의 작업이다. 이것은 본질적으로 반대이고 나아가 이름으로는 Inversion of Control (IoC) 이다. bean자체는 인스턴스화를 제어하거나 클래스의 직접적 생성을 사용하여 자체적으로 의존성을 위치시키거나 서비스 위치자 패턴와 같은 것이다.

DI개념을 적용될때 코드가 좀더 깔끔해지도록 사용법이 분명해진다. 그것들과 함께 제공되지만 bean이 의존성을 룩업할때 디커플링의 좀더 높은 단계에 도착하는 것이 좀더 쉽다.(그리고 추가적으로 의존성이 위치한 곳과 실제 클래스가 하는 것이 무엇인지 알지 못한다.)

이전 단락에서 보여준것처럼, DI는 두가지의 형태로 존재한다. 명명하여 Setter 삽입 과 생성자(Constructor) 삽입이다.

3.3.1.1. Setter 삽입

Setter-기반의 DI는 인자가 없는 생성자나 인자가 없는 static factory 메소드가 bean을 인스턴스화하기 위해 호출된후 bean의 setter메소드를 호출하여 실제화된다.

setter삽입을 사용하여 의존성이 삽입되는 클래스의 예제를 아래에서 보자. 이 클래스에 대해 특별한 것은 없다. 이것은 일반적인 Java이다.

```
public class SimpleMovieLister {

// the SimpleMovieLister has a dependency on the MovieFinder
private MovieFinder movieFinder;

// a setter method so that the Spring container can 'inject' a MovieFinder
public void setMoveFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
}

// business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

3.3.1.2. 생성자 삽입

생성자-기반의 DI는 각각의 협력자는 표시하는 많은 수의 인자를 가진 생성자를 호출하여 실제화된다. 추가적으로, bean을 생성하기 위한 특정 인자를 가진 static factory메소드를 호출하는 것은 대부분 동등하게 간주될수 있고 이 텍스트의 나머지는 생성자를 위한 인자와 static factory메소드를 위한 인자를 검토할것이다.

생성자 삽입을 사용하여 의존성을 삽입할수 있는 클래스의 예제를 아래에서 보라. 다시, 이 클래스에 대한 특별한 것은 없다.

```
public class SimpleMovieLister {

// the SimpleMovieLister has a dependency on the MovieFinder
private MovieFinder movieFinder;

// a constructor so that the Spring container can 'inject' a  MovieFinder
public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
}

// business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

생성자 또는 setter-기반의 DI?

Spring팀은 대개 setter삽입의 사용을 지지한다. 많은수의 생성자의 인자는 특히 몇가지 프라퍼티가

선택적일때 다루기 어려울수 있다. setter메소드의 존재는 재설정(재삽입)되기 위해 다루기 쉬운 클래스의 객체를 만든다(JMX MBeans을 통한 관리를 위해 특별히 사용케이스를 강요한다.)

생성자-삽입은 몇가지 순수주의자(그리고 중은 이유를 가지고)에 의해 선호된다. 모든 객체의 의존성을 제공하는 것은 객체가 전체적으로 초기화가 안된 상태로 클라이언트 코드에 결코 반환되지 않는다는 것을 의미한다. 대조적인 면은 객체가 재-설정(또는 재-삽입)을 위해 다루는 것이 다소 어렵게 된다는 것이다.

여기에는 단단히 고정된 규칙이 없다. 어떠한 타입의 DI를 사용하는 것이 특정 클래스를 위해 현명하다. 때때로, 소스를 가지지 않는 이기종 클래스를 다룰때, 선택은 이미 당신에게 달렸다. 기존 클래스는 어떠한 setter메소드도 나타내지 않는다. 그리고 생성자 삽입은 DI타입에서만 사용가능할것이다.

BeanFactory는 의존성을 bean에 삽입하기 위한 이러한 모든 형태를 지원한다(사실 몇가지 의존성이 생성자 접근법을 통해 제공된 후 setter-기반의 의존성을 삽입하는 것을 지원한다). BeanDefinition의 형태인 의존성을 위한 설정은, 한 형태의 프라퍼티를 다른 형태로 변환하는 방법을 아는 PropertyEditor 인스턴스와 함께 사용된다. 어쨌든, 대부분의 Spring사용자는 직접 이러한 클래스를 다루지 않고 클래스의 인스턴스로 내부적으로 변환하기 위해 Spring IoC컨테이너 인스턴스를 로드하기 위해 사용되는 XML정의 파일을 사용할것이다.

bean의존성 해석은 다음처럼 발생한다.

1. BeanFactory는 모든 bean을 언급하는 설정으로 생성되고 초기화된다(대개의 Spring사용자는 XML형태의 설정파일을 지원하는 BeanFactory 나 ApplicationContext 구현물을 사용한다.)

2. 각각의 bean은 프라퍼티, 생성자의 인자, 또는 대개의 생성자 대신에 사용되는 static-factory메소드의 인자의 형태로 표시되는 의존성을 가진다. 이러한 의존성은 bean이 실제로 생성될때 bean에 제공될것이다.

3. 각각의 프라퍼티 또는 생성자의 인자는 실제 셋팅되기 위한 값의 정의이거나 컨테이너내 다른 bean에 대한 참조이다.

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.

Spring이 컨테이너가 생성될때 컨테이너내 각각의 bean의 설정의 유효성을 체크하는 것을 실제화하는 것이 중요하다. It is important to realize that Spring validates the configuration of each bean in a container as the container is created, including the validation that properties which are bean references are actually referring to valid beans (i.e. the beans being referred to are also defined in the container. However, the bean properties themselves are not set until the bean is actually created. For that which are singleton-scoped and set to be pre-instantiated (such as singleton beans in an ApplicationContext), creation happens at the time that the container is created, but otherwise this is only when the bean is requested. When a bean actually has to be created, this will potentially cause a graph of other beans to be created, as its dependencies and its dependencies' dependencies (and so on) are created and assigned.

Circular dependencies

If you are using predominantly constructor injection it is possible to write and configure your classes and beans such that an unresolvable circular dependency scenario is created.

Consider the scenario where you have class A, which requires an instance of class B to be provided via constructor injection, and class B, which requires an instance of class A to be provided via constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container will detect this circular reference at runtime, and throw a BeanCurrentlyInCreationException.

One possible solution to this issue is to edit the source code of some of your classes to be configured via setters instead of via constructors. Another solution is not to use constructor injection and stick to setter injection only.

You can generally trust Spring to do the right thing. It will detect mis-configuration issues, such as references to non-existent beans and circular dependencies, at container load-time. It will actually set properties and resolve dependencies (i.e. create those dependencies if needed) as late as possible, which is when the bean is actually created. This does mean that a Spring container which has loaded correctly, can later generate an exception when you request a bean, if there is a problem creating that bean or one of its dependencies. This could happen if the bean throws an exception as a result of a missing or invalid property, for example. This potentially delayed visibility of some configuration issues is why ApplicationContext implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you find out about configuration issues when the ApplicationContext is created, not later. If you wish, you can still override this default behavior and set any of these singleton beans to lazy-initialize (i.e. not be pre-instantiated).

Finally, if it is not immediately apparent, it is worth mentioning that when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being passed (via one of the DI flavors) to the dependent bean. This means that if bean A has a dependency on bean B, the Spring IoC container will totally configure bean B prior to invoking (for example) the attendant setter method on bean A; you can read 'totally configure' to mean that the bean will be instantiated (if not a pre-instantiated singleton), all of its dependencies will be set, and the relevant lifecycle methods (such as a configured init method or the IntializingBean callback method will all be invoked.

3.3.1.3. 몇몇 예제들

먼저, setter-기반의 DI를 위한 XML-기반의 설정 메타데이터를 사용하는 예제이다. 아래에서 몇가지 bean정의를 명시하는 Spring XML설정파일의 작은 일부를 보자.

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

  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</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.

이제는 생성자-기반의 DI를 사용하는 예제이다. 아래는 생성자의 인자와 관련 Java클래스를 명시하는 XML설정의 일부이다.

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

  <!-- constructor injection using the nested <ref/> element -->
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>

  <!-- constructor injection using the neater 'ref' attribute -->
  <constructor-arg ref="yetAnotherBean"/>

  <constructor-arg type="int" value="1"/>
</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 인자로 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. bean정의의 생성자의 인자에서 잠재적인 모호함이 없다면, bean정의에 정의된 생성자의 인자의 순서는 초기화될때 적절한 생성자에 제공될 인자의 순서이다. 다음의 클래스를 검토하라.

```
package x.y;

public class Foo {

public Foo(Bar bar, Baz baz) {
        // ...
}
}
```

There is no potential for ambiguity here (assuming of course that Bar and Baz classes are not related in an inheritance hierarchy). Thus the following configuration will work just fine, and one does not need to specify the constructor argument indexes and / or types explicitly... it just plain works as one would expect it to.

```
<beans>
<bean name="foo" class="x.y.Foo">
        <constructor-arg>
                <bean class="x.y.Bar"/>
        </constructor-arg>
        <constructor-arg>
                <bean class="x.y.Baz"/>
        </constructor-arg>
</bean>
</beans>
```

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true<value>, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class, which is used for the following two sections:

```
package examples;

public class ExampleBean {

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

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

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

Tip

생성자의 인자 인덱스를 명시하는것은 생성자 IoC를 수행하는 방법이 선호된다.

3.3.3. 상세화된 bean프라퍼티와 생성자의 인자

앞 부분에서 언급된것처럼 bean프라퍼티와 생성자의 인자는 다른 관리빈(협력자), 또는 인라인으로 정의된 값의 참조처럼 정의될수 있다. Spring의 XML-기반의 설정 메타데이터는 이러한 목적을 위한 <property/> 와 <constructor-arg/>요소내에서 많은 수의 하위요소를 지원한다.

3.3.3.1. 순수값(원시, Strings, 등등)

<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.

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

3.3.3.1.1. idref 요소

idref 요소는 컨테이너내 다른 bean의 id를 (<constructor-arg/> 나 <property/>요소에)전달하기 위한 에러를 검사(error-proof)하는 방법이다.

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

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

이것은 수행시 다음의 조각들과 동일하다.

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

<bean id="client" 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, 'client' bean의 'targetName'프라퍼티에 전달되는 값에서 수행되는 유효성체크는 없다. 어느 활자는 'client' bean이

인스턴스화될때 발견된다. 'client' bean이 prototype bean이라면, 이 활자(와 결과 예제)는 컨테이너가 실제로 배치된후에 발견될것이다.

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">
   <!-- a bean with an id of 'theTargetBean' must exist, else an XML exception will be thrown -->
   <idref local="theTargetBean"/>
</property>
```

예제의 방법에 의해, 값을 가져오는 <idref/> 요소의 한가지 공통된 위치는 ProxyFactoryBean bean 정의내 AOP 인터셉터의 설정내이다. 인터셉터 이름을 명시할때 <idref/> 요소를 사용한다면, 무심코 인터셉터의 id를 잘못쓰는 경우는 없다.

3.3.3.2. 다른 bean에 대한 참조

ref요소는 <constructor-arg/> 또는 <property/> 정의 요소내부에 허용되는 마지막 요소이다. 이것은 컨테이너에 의해 관리되는 다른 bean에 대해 참조되기 위해 명시된 프라퍼티의 값을 셋팅하는데 사용된다. 앞부분에서 언급했던것 처럼 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 컨테이너(같은 XML파일이든 아니든)나 부모 컨테이너내에서 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 컨테이너의 부모 컨테이너내 있을 bean을 위해 생성될 참조를 허용한다. 'parent' 속성의 값은 아마 대상 bean의 'id' 속성이나 'name' 속성내 값중에 하나와 같을것이고 대상 bean은 최근것을 위해 부모 컨테이너내 있어야만 한다. bean참조 형태의 가장 중요한 사용은 몇몇 프록시의 순서대로 부모 컨텍스트내 존재하는 bean을 포장하기 위해 필요할때이고 초기의 객체는 그것을 포장하기 위해 필요하다.

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
    <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService"  <-- notice that the name of this bean is the same as the name of the 'parent' bean
```

```
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/>  <-- notice how we refer to the parent bean
    </property>
  <!-- insert other configuration and dependencies as required as here -->
</bean>
```

('parent' 속성의 사용은 전혀 공통적이지 않다.)

3.3.3.3. 내부 bean

<property/> 나 <constructor-arg/> 요소내부의 <bean/> 요소는 내부 bean이라 불리는 것을 정의하기 위해 사용된다. 내부 bean정의는 정의된 id나 name을 필요로 하지 않고 언급된 id나 name값은 컨테이너에 의해 무시되기 때문에 id나 name값을 명시하지 않는 것이 가장 좋다.

내부 bean의 예제를 아래에서 보라.

```
<bean id="outer" class="...">
  <!-- instead of using a reference to a target bean, simply define the target inline -->
  <property name="target">
    <bean class="com.mycompany.Person"> <!-- this is the inner bean -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

내부 bean의 특별한 경우, 'singleton' 플래그와 'id' 또는 'name'속성은 사실상 무시된다. 내부 bean은 언제나 익명이고 언제나 prototypes과 같이 범위화된다. 둘러싸인 bean보다는 협력하는 bean으로 내부 bean을 삽입하는 것은 가능하지 않다는 것을 노트하라.

3.3.3.4. collection

<list/>, <set/>, <map/>, 과 <props/>요소는 정의되고 셋팅되는 Java Collection List, Set, Map, and Properties의 프라퍼티와 인자를 허용한다.

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@somecompany.org</prop>
        <prop key="support">support@somecompany.org</prop>
        <prop key="development">development@somecompany.org</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>
                <value>yup an entry</value>
            </key>
```

```
            <value>just some string</value>
        </entry>
        <entry>
            <key>
                <value>yup a ref</value>
            </key>
            <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>
```

map key의 value나 value, 또는 set value는 다음의 어떤 요소도 될수 있다는 것을 노트하라.

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

3.3.3.4.1. Collection 병합

Spring 2.0에서, 컨테이너는 collection의 병합또한 지원한다. 이것은 애플리케이션 개발자에게 부모 스타일의 <list/>, <map/>, <set/> 또는 <props/>요소를 정의하고 부모 collection으로 부터 상속하거나 값을 덮는 자식 스타일의 <list/>, <map/>, <set/> 또는 <props/>요소를 가지는 것을 허용한다. 예를 들면, 자식 collection의 값은 자식 collection요소가 부모 collection내 명시된 값을 덮는 부모와 자식 collection 요소의 병합으로 부터 얻어지는 결과가 될것이다.

병합에 대한 이 섹션은 부모-자식 bean기법의 사용하게 한다. 이 개념은 아직 소개되지 않아서, 부모와 자식 bean정의의 개념에 친숙하지 않은 독자는 지속되기 전에 관련 섹션을 읽기를 바란다(Section 3.7, “Bean정의 상속” 를 보라).

예제는 이 기능을 보여주기 위한 가장 좋은 소스를 제공한다.

```
<beans>
<bean id="parent" abstract="true" class="example.ComplexObject">
<property name="adminEmails">
        <props>
                <prop key="administrator">administrator@somecompany.com</prop>
                <prop key="support">support@somecompany.com</prop>
        </props>
</property>
</bean>
<bean id="child" parent="parent">
<property name="adminEmails">
        <!-- the merge is specified on the *child* collection definition -->
        <props merge="true">
                <prop key="sales">sales@somecompany.com</prop>
                <prop key="support">support@somecompany.co.uk</prop>
        </props>
</property>
</bean>
<beans>
```

child bean정의의 adminEmails 프라퍼티의 <props/>요소에서 merge=true속성을 사용하자. child bean이

컨테이너에 의해 실질적으로 분석되고 인스턴스화될때, 결과 인스턴스는 부모의 adminEmails collection을 가진 자식의 adminEmails collection의 병합의 결과를 포함하는 adminEmails Properties collection을 가질것이다.

```
administrator=administrator@somecompany.com
sales=sales@somecompany.com
support=support@somecompany.co.uk
```

자식 Properties collection의 값 세트가 부모 <props/>로부터 모든 프라퍼티 요소를 상속하는 방법에 주의하라. support 값을 위한 자식 값이 부모 collection내 부수적인 값을 덮는 방법에 주의하라.

이 병합행위는 <list/>, <map/>, 그리고 <set/> collection타입에 유사하게 적용한다. <list/>요소의 특정경우에, 이 의미는 List collection타입과 관련된다. 이를테면, value의 ordered collection의 개념은 유지관리된다. 부모값은 모든 자식 목록의 값에 선행한다. Map, Set, 그리고 Properties collection타입의 경우, 컨테이너에 의해 내부적으로 사용되는 Map, Set 그리고 Properties구현물 타입에 관련된 collection타입을 위한 영향을 끼친다.

마지막으로, 병합 지원에 대한 몇몇 사소한 노트는 순서대로이다. 첫째 다른 collection타입간(이를 테면, Map 과 List)의 병합을 할수 없다. 그리고 하나가 그렇게 시도한다면 Exception을 던질것이다. 그리고 이 경우에는, 알기쉽다. 'merge' 속성은 더 낮은 레벨, 상속된, 자식 정의..에서 명시되어야만 한다. 부모 collection정의의 'merge'속성을 명시하는 것은 예측가능하고 바람직한 병합의 결과를 만들지 않을것이다. 그리고 마지막으로, 이 병합 기능은 오직 Spring 2.0(과 그 이후 버전)에서만 사용가능하다.

3.3.3.4.2. Strongly-typed collection (Java5+ only)

If you are one of the lucky few to be using Java5 (Tiger), you will be aware that it is possible (and I daresay recommended) to have strongly typed collections. That is, it is possible to declare a Collection type such that it can only contain String elements (for example).

If you are using Spring to dependency inject a strongly-typed Collection into a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed Collection instances will be converted to the appropriate type prior to being added to the Collection.

An example will make this clear; consider the following class definition, and it's attendant (XML) configuration...

```
public class Foo {

private Map<String, Float> accounts;

public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
}

}
```

```
<beans>
<bean id="foo" class="x.y.Foo">
        <property name="accounts">
                <map>
                        <entry key="one" value="9.99"/>
                        <entry key="two" value="2.75"/>
                        <entry key="six" value="3.99"/>
                </map>
        </property>
```

```
</bean>
</beans>
```

When the 'accounts' property of the 'foo' bean is being prepared for injection, the generics information about the element type of the strongly-typed Map<String, Float> is actually available via reflection, and so Spring's type conversion infrastructure will actually recognize the various value elements as being of type Float and so the string values '9.99', '2.75', and '3.99' will be converted into an actual Float type.

3.3.3.5. null

<null/> 요소는 null 값을 다루기 위해 사용된다. Spring은 프라퍼티를 위한 빈 인자를 빈 문자열처럼 처리한다. 다음의 XML-기반의 설정 메타데이터 조각은 email프라퍼티를 빈 String값으로 셋팅되도록 한다.

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

이것은 다음의 Java코드인 exampleBean.setEmail("")과 동등하다. 특별한 <null> 요소는 아마도 null값을 표시하는데 사용될것이다.

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

위 설정은 다음의 자바코드인 exampleBean.setEmail(null) 와 동등하다.

3.3.3.6. XML-기반의 설정 메타데이터 간략화

이것은 value이나 bean참조를 설정하기 위해 필요한 공통사항이다. 완전한 형태의 <value/> 와 <ref/>요소를 사용하는것보다 다소 덜 장황하게 간략화한 몇가지 형태가 존재한다. <property/>, <constructor-arg/>, 그리고 <entry/>요소 모두 완전한 형태의 <value/> 요소 대신에 사용된 'value' 속성을 지원한다. 그러므로 다음은

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

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

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

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 유사하게 간략화된 'ref' 속성을 지원한다. 그러므로 다음은

```
<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 동등하다. <ref local="xxx"> 를 위한 간략화된 형태는 없다. 엄격한 local ref를 강요하기 위해 당신은 긴 형태를 사용해야만 한다.

마지막으로 entry 요소는 'key'/'key-ref' 와 'value'/'value-ref'속성의 형태로 map의 키 그리고/또는 값을 명시하기 위한 간략화된 형태를 허용한다. 그러므로 다음은

```
<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.

3.3.3.7. 혼합된 프라퍼티 명

복합적이거나 내포된 프라퍼티명은 bean프라퍼티를 셋팅할때 완전히 규칙에 따른다(legal)는 것을 알아두라. 마지막 프라퍼티명을 제외한 경로의 모든 컴포넌트는 null이 아니다. 예를 들면, 이 bean정의내에서:

```
<bean id="foo" class="foo.Bar">
  <property name="fred.bob.sammy" value="123" />
</bean>
```

foo bean은 bob 프라퍼티를 가지는 fred프라퍼티를 가진다. 그리고 bob 프라퍼티는 sammy프라퍼티를

가지고 마지막 sammy 프라퍼티는 123값으로 셋팅된다. 이 작업을 위해, foo의 fred프라퍼티, 그리고 fred의 bob프라퍼티는 bean이 생성된 후에 null이 아니어야만 한다. 그렇지 않으면 NullPointerException예외가 던져질것이다.

3.3.4. depends-on 사용하기

대부분의 상황을 위해 bean이 다른 것들의 의존성이라는 사실은 하나의 bean이 다른것의 프라퍼티처럼 셋팅한다는 사실에 의해 간단하게 표현된다. 이것은 XML-기반의 설정 메타데이터내 <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' 속성은 이 초기화된 요소를 사용하는 bean이전에 초기화되기 위한 하나 이상의 bean을 명시적으로 강제하기 위해 사용된다. 하나의 bean에 의존성을 표시하기 위한 'depends-on' 속성을 사용하는 예제를 아래에서 보라.

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

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

다중 bean에 의존성을 표시할 필요가 있다면, 콤마, 공백 그리고 세미콜론과 같은 모든 유효한 구분자를 사용하여 'depends-on'속성의 값으로 bean이름의 구분된 목록을 제공할수 있다. 많은 수의 bean에 의존성을 표시하기 위해 'depends-on'를 사용하는 예제를 아래에서 보라.

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

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

3.3.5. Lazily-instantiating beans

The default behavior for ApplicationContext implementations is to eagerly pre-instantiate all singleton beans at startup. Pre-instantiation means that an ApplicationContext implementation instance will eagerly create and configure all of it's singleton beans as part of its initialization process. This is generally a good thing, because it means that any errors in the configuration or in the attendant environment will be discovered immediately (as opposed to possibly hours or even days down the line).

However, there are times when this behavior is not what is wanted. If you do not want a singleton bean to be pre-instantiated when using an ApplicationContext implementation, you can (on a bean-definition by bean-definition basis) selectively control this by marking a bean definition as lazy-initialized. A lazily-initialized bean indicates to the IoC container whether or not a bean instance should be created at startup or when it is first requested.

When configuring beans via XML, this lazy loading is controlled by the 'lazy-init' attribute on the <bean/> element; to wit:

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true">
    <!-- various properties here... -->
</bean>

<bean name="not.lazy" class="com.foo.AnotherBean">
    <!-- various properties here... -->
</bean>
```

When the above configuration is consumed by an ApplicationContext implementation, the bean named 'lazy' will not be eagerly pre-instantiated when the ApplicationContext is starting up, whereas the 'not.lazy' bean will be eagerly pre-instantiated.

One thing to understand about lazy-initialization is that even though a bean definition may be marked up as being lazy-initialized, if the lazy-initialized bean is the dependency of a singleton bean that is not lazy-initialized, when the ApplicationContext is eagerly pre-instantiating the singleton, it will (of course) have to satisfy all of said singletons dependencies, one of which will be the lazy-initialized bean! So don't be confused if the IoC container creates one of the beans that you have explicitly configured as lazy-initialized at startup; all that means is that the lazy-initialized bean probably is being injected into a non-lazy-initialized singleton bean elsewhere in your configuration.

It is also possible to control lazy-initialization at the container level by using the 'default-lazy-init' attribute on the <beans/> element; to wit:

```
<beans default-lazy-init="true">
    <!-- no beans will be eagerly pre-instantiated... -->
</beans>
```

3.3.6. Autowiring 협력자

Spring IoC컨테이너는 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. [2] XML-기반의 설정 메타데이터를 사용할때, bean정의를 위한 autowire모드는 <bean/> 요소의 autowire 속성을 사용하여 명시된다. 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 컨테이너를 조사하고 autowire될 필요가 있는 프라퍼티와 같은 이름의 bean을 찾는다. 예를 들면 당신이 만약 이름에 의해 autowire하기 위해 셋팅하는 bean정의를 가지고 이것이 master

[2] Section 3.3.1, "의존성 삽입하기" 를 보라.

모드	설명
	프라퍼티(이것은 setMaster(...) 메소드를 가진다.)를 포함한다면 Spring은 master라는 이름의 bean정의를 찾을것이고 프라퍼티를 셋팅하기 위해 이것을 사용한다.
byType	컨테이너내 프라퍼티 타입의 bean이 정확하게 하나 있다면 프라퍼티를 autowire가 되도록 허용한다. 만약 하나 이상이 있다면 치명적인 예외가 던져지고 이것은 bean을 위한 byType autowiring을 사용하지 않는것을 나타낸다. 만약 대응되는 bean이 없다면 아무것도 발생하지 않는다. 프라퍼티가 셋팅되지 않는다. 만약 이 기능을 바라지 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-arg 셋팅내 명시적인 의존성이 언제나 autowiring을 오버라이드한다는것에 주의하라. 소위 간단하다고 불리는 원시(primitives), Strings, 그리고Classes(그리고 이러한 간단한 프라퍼티의 배열.)(이것은 디자인되고 기능에 추가될것이다.)와 같은 프라퍼티를 자동으로 묶는(autowire)것이 현재 가능하지 않다는 것을 알라. Autowire 행위는 모든 autowiring가 완성된후 수행될 의존성 체크와 조합될수 있다.

다양한 장점과 autowiring의 단점을 이해하는 것은 중요하다. 몇가지 장점은 다음을 포함한다.

☒ Autowiring은 요구되는 설정의 양을 명백하게 감소시킨다. 어쨌든 bean 템플릿(이 장 도처에서 언급된)의 사용과 같은 기법은 이점에서 가치있다.

☒ Autowiring은 객체가 나타나는 것처럼 그것 자체를 최신으로 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

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

☒ wiring정보는 아마도 Spring컨테이너로부터 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.

3.3.6.1. Excluding a bean from being available for autowiring

You can also (on a per bean basis) totally exclude a bean from being an autowire candidate. When configuring beans using Spring's XML format, the 'autowire-candidate' attribute of the <bean/> element can be set to 'false'; this has the effect of making the container totally exclude that specific bean definition from being available to the autowiring infrastructure.

This can be useful when you have a bean that you absolutely never ever want to have injected into other beans via autowiring. It does not mean that the excluded bean cannot itself be configured using autowiring... it can, it is rather that it itself will not be considered as a candidate for autowiring other beans.

3.3.7. 의존성을 위한 체크

Spring IoC컨테이너는 컨테이너로 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-check'속성을 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 bean ㄷ처럼 협력자를 제외한 모든 것)을 위해 수행되는 의존성 체크.
object	협력자를 위해 수행되는 의존성 체크.
all	협력자, 원시타입 그리고 collection을 위해 수행되는 의존성 체크.

Java 5를 사용하고 소스레벨의 어노테이션을 사용한다면, 관심을 가지고 Section 25.3.1, "@Required" 부분을 보게 될것이다.

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.

```
// a class that uses a stateful inner class to do some data processing
package fiona.apple;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;

public class CommandProcessor implements BeanFactoryAware {

   // instances of this class are stateful, and hence best deployed as prototypes
   public static final class ThreadedCommandHelper implements CommandHelper, Runnable {

      private Object command;

      public Object process(Object command) {
         this.command = command;
         new Thread(this).start();
         // harvest and return result of processing...
         return ...;
      }

      public void run() {
         // do the processing of the command...
      }
   }

   private BeanFactory beanFactory;

   public Object process(Object command) {
       // grab a new instance of the appropriate CommandHelper
      return createHelper().process(command);
   }

   protected CommandHelper createHelper() {
       return (CommandHelper) this.beanFactory.getBean("command.helper");
   }

   public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
      this.beanFactory = beanFactory;
   }
}
```

이것은 bean코드가 인식을 하고 Spring에 커플링이 되기 때문에 대개 바람직한 해결법은 아니다. Spring IoC컨테이너의 향상된 기능인 메소드 삽입은 몇몇 다른 시나리오에 따라 깔끔한 형태로 다루어지는 사용 상황을 허용한다.

3.4.1. 룩업(Lookup) 메소드 삽입

> 음.. 이러한 종류가 아닌...
>
> Tapestry가 구현물로 런타임시 오버라이드할 추상 setter와 getter를 작성하는 Tapestry 4.0의 페이지

당신은 이 블로그 글에서 메소드 삽입을 위한 동기에 대해 좀더 많은 것을 읽을수 있다.

룩업 메소드 삽입은 컨테이너내 다른 명명된 bean를 룩업하는 결과를 반환하는 컨테이너내 관리빈의 메소드를 오버라이드하기 위해 컨테이너의 기능을 적용한다. 룩업은 위에서 언급(비록 싱글톤이 될수 있더라도)된 시나리오처럼 프로토타입(prototype) bean이 될것이다. Spring은 CGLIB 라이브러리를 통해 바이트코드 생성을 사용하여 동적으로 생성된 하위클래스가 메소드를 오버라이드하는것을 통해 이 삽입을 구현한다.

당신이 이전이 코드 조각(CommandProcessor 클래스)으로부터 코드를 본다면, 이것은 createHelper() 메소드의 구현물을 동적으로 제공하는 Spring컨테이너이다. 당신의 CommandProcessor 클래스는 아래에서 볼수 있는것처럼, Spring의존성을 가지지 않는다.

```
package fiona.apple;

// no more Spring imports!

public class CommandProcessor {

   // instances of this class are stateful, and hence best deployed as prototypes
   public static final class ThreadedCommandHelper implements CommandHelper, Runnable {

      private Object command;

      public Object process(Object command) {
         this.command = command;
         new Thread(this).start();
         // harvest and return result of processing...
         return ...;
      }

      public void run() {
         // do the processing of the command...
      }
   }

   public Object process(Object command) {
       // grab a new instance of the appropriate CommandHelper
      return createHelper().process(command);
   }

    // mmm, but where is the implementation of this method?
   protected abstract CommandHelper createHelper();

}
```

(위 코드조각내 CommandProcessor)삽입된 메소드를 포함하는 클라이언트 클래스에서, '삽입된' 메소드는 다음 형태의 시그너처를 가져야만 한다.

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

만약 메소드가 추상적이라면, Spring이 생성한 하위클래스는 메소드를 구현할것이다. 그렇지 않다면, Spring이 생성한 하위클래스가 원래의 클래스내 정의된 구현된 메소드를 오버라이드할것이다. 예제를 보자.

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="statefulCommandHelper"
      class="fiona.apple.CommandProcessor$ThreadedCommandHelper" scope="prototype"/>
```

```
<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandProcessor" class="fiona.apple.CommandProcessor">
  <lookup-method name="createHelper" bean="statefulCommandHelper"/>
</bean>
```

commandProcessor로 확인된 bean은 statefulCommandHelper bean의 새로운 인스턴스가 필요할때마다 자체적인 createHelper메소드를 호출할것이다. bean을 디플로이하는 사람은 프로토타입으로 statefulCommandHelper bean을 디플로이하는 것을 주의해야만 한다는 것을 노트하는 것이 중요하다. 싱글톤으로 디플로이된다면(명시적이거나, 이 플래그에 디폴트로 true로 셋팅되는 것에 의존하여), statefulCommandHelper의 같은 인스턴스는 매번 반환될것이다.

룩업 메소드 삽입은 생성자와 setter삽입모두 조합될수 있다.

Please be aware that in order for this dynamic subclassing to work, you will need to have the CGLIB jar(s) on your classpath. Additionally, the class that the Spring container is going to subclass cannot be final, and the method that is being overridden cannot be final either. Also, testing a class that has an abstract method can be somewhat odd in that you will have to subclass the class yourself and supply a stub implementation of the abstract method. Finally, beans that have been the target of method injection cannot be serialized.

The interested reader may also find the ServiceLocatorFactoryBean (in the org.springframework.beans.factory.config package) to be of use... the approach is similar to that of the ObjectFactoryCreatingFactoryBean, but it allows you to specify your own lookup interface as opposed to having to use a Spring-specific lookup interface such as the ObjectFactory. Consult the (copious) Javadocs for the ServiceLocatorFactoryBean for a full treatment of this alternative approach (that does reduce the coupling to Spring).

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.

XML-기반의 설정 메타데이터를 사용할때, 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"/>
```

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.

3.5. Bean scopes

When you create a bean definition (typically in an XML configuration file) what you are actually creating is (loosely speaking) a recipe or template for creating actual instances of the objects defined by that bean definition. The fact that a bean definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the scope of the objects created from a particular bean definition. This approach is very powerful and gives you the flexibility to choose the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of which three are available only if you are using a web-aware Spring ApplicationContext).

The scopes supported out of the box are listed below:

Table 3.4. Bean scopes

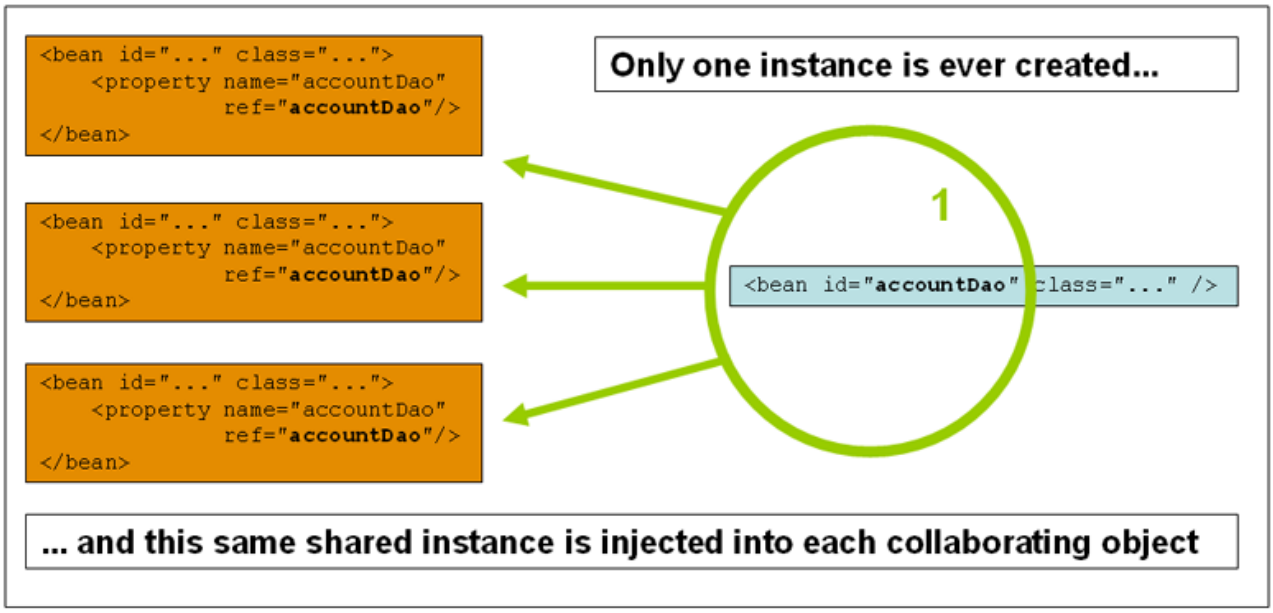
Scope	Description
singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; i.e. each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
global session	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

3.5.1. The singleton scope

When a bean is a singleton, only one shared instance of the bean will be managed and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that bean definition (or recipe). This single instance will be stored in a singleton cache, and all subsequent requests and references for that named bean will result in the cached object instance being returned.

The following diagram illustrates the Spring singleton scope.



Please be aware that Spring's concept of a singleton bean is quite different from the Singleton pattern as defined in the seminal Gang of Four (GoF) patterns book. The classic GoF Singleton hardcodes the scope of an object such that one and only one instance of a particular class will ever be created per ClassLoader. The scope of the Spring singleton is best described as per container and per bean. This means that if you define one bean for a particular class in a single Spring container, then the Spring container will create one and only one instance of the class defined by that bean definition.

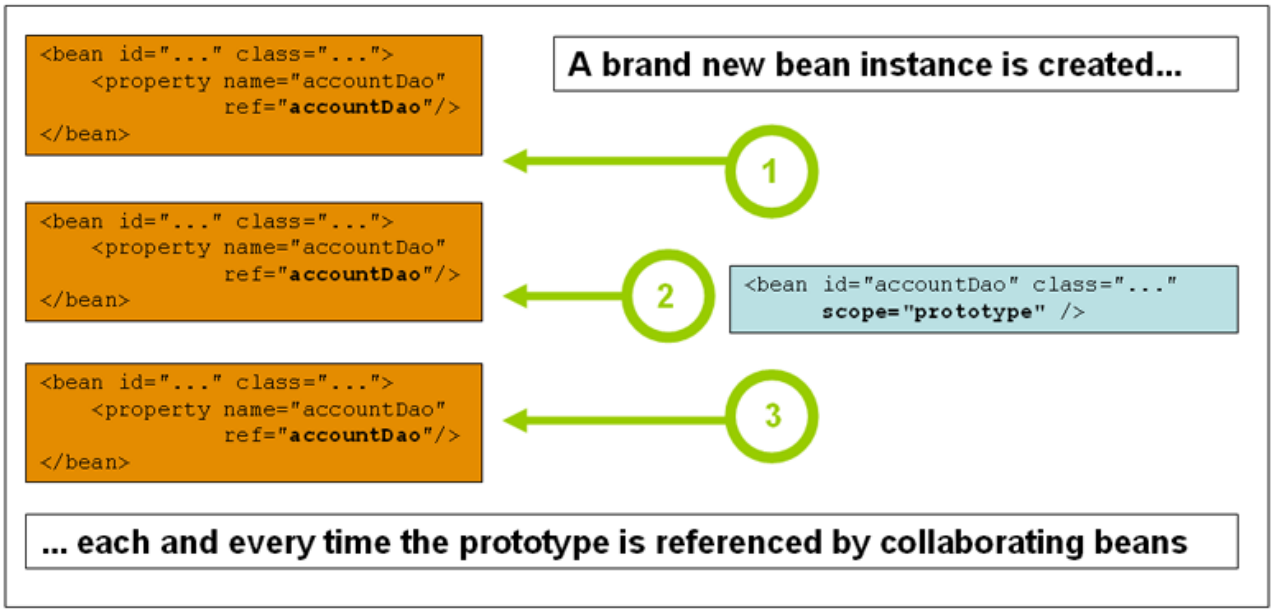
The singleton scope is the default scope in Spring. To define a bean as a singleton in XML, you would write configuration like so:

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

<!-- the following is equivalent, though redundant (and preserved for backward compatibility) -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>
```

3.5.2. The prototype scope

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made (that is, it is injected into another bean or it is requested via a programmatic getBean() method call on the container). As a rule of thumb, you should use the prototype scope for all beans that are stateful, while the singleton scope should be used for stateless beans.

The following diagram illustrates the Spring prototype scope. Please note that a DAO would not typically be configured as a prototype, since a typical DAO would not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.



To define a bean as a prototype in XML, you would write configuration like so:

```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<!-- the following is equivalent too (and preserved for backward compatibility) -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>
```

There is one quite important thing to be aware of when deploying a bean in the prototype scope, in that the lifecycle of the bean changes slightly. Spring cannot (and hence does not) manage the complete lifecycle of a prototype bean: the container instantiates, configures, decorates and otherwise assembles a prototype object, hands it to the client and then has no further knowledge of that prototype instance. This means that while initialization lifecycle callback methods will be (and are) called on all objects regardless of scope, in the case of prototypes, any configured destruction lifecycle callbacks will not be called. It is the responsibility of the client code to clean up prototype scoped objects and release any expensive resources that the prototype bean(s) are holding onto. (One possible way to get the Spring container to release resources used by singleton-scoped beans is through the use of a bean post processor which would hold a reference to the beans that need to be cleaned up.)

In some respects, you can think of the Spring container's role when talking about a prototype-scoped bean as somewhat of a replacement for the Java 'new' operator. Any lifecycle aspects past that point have to be handled by the client. The lifecycle of a bean in a Spring IoC container is further described in the section entitled Section 3.6.1, "Lifecycle 인터페이스".

Backwards compatibility note: specifying the lifecycle scope in XML

If you are referencing the 'spring-beans.dtd' DTD in a bean definition file(s), and you are being explicit about the lifecycle scope of your bean(s) you must use the "singleton" attribute to express the lifecycle scope (remembering that the singleton lifecycle scope is the default). If you are referencing the 'spring-beans-2.0.dtd' DTD or

the Spring 2.0 XSD schema, then you will need to use the "scope" attribute (because the "singleton" attribute was removed from the definition of the new DTD and XSD files in favour of the "scope" attribute).

To be totally clear about this, this means that if you use the "singleton" attribute in an XML bean definition then you must be referencing the 'spring-beans.dtd' DTD in that file. If you are using the "scope" attribute then you must be referencing either the 'spring-beans-2.0.dtd' DTD or the 'spring-beans-2.0.xsd' XSD in that file.

3.5.3. The other scopes

The other scopes, namely request, session, and global session are for use only in web-based applications (and can be used irrespective of which particular web application framework you are using, if indeed any). In the interest of keeping related concepts together in one place in the reference documentation, these scopes are described here.

Note

The scopes that are described in the following paragraphs are only available if you are using a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you try using these next scopes with regular Spring IoC containers such as the XmlBeanFactory or ClassPathXmlApplicationContext, you will get an IllegalStateException complaining about an unknown bean scope.

3.5.3.1. Initial web configuration

In order to effect the scoping of beans at the request, session, and global session level (i.e. web-scoped beans), some minor initial configuration is required before you can set about defining your bean definitions. Please note that this extra setup is not required if you just want to use the 'standard' scopes; i.e. singleton and prototype.

Now as things stand, there are a couple of ways to effect this initial setup depending on your particular servlet environment. If you are using a Servlet 2.4+ web container, then you need only add the following ContextListener to the XML declarations in your web applications 'web.xml' file.

```
<web-app>
  ...
  <listener>
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
  </listener>
  ...
</web-app>
```

If you are using an older web container (before Servlet 2.4), you will need to use a (provided) javax.servlet.Filter implementation. Find below a snippet of XML configuration that has to be included in the 'web.xml' file of your web application if you want to have access to web-scoped beans (the filter settings depend on the surrounding web application configuration and so you will have to change them as appropriate).

```
<web-app>
  ..
```

```
  <filter>
    <filter-name>requestContextFilter</filter-name>
    <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>requestContextFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

That's it. The RequestContextListener and RequestContextFilter classes both do exactly the same thing, namely bind the HTTP request object to the Thread that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

3.5.3.2. The request scope

Consider the following bean definition:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

With the above bean definition in place, the Spring container will create a brand new instance of the LoginAction bean using the 'loginAction' bean definition for each and every HTTP request. That is, the 'loginAction' bean will be effectively scoped at the HTTP request level. You can change or dirty the internal state of the instance that is created as much as you want, safe in the knowledge that other requests that are also using instances created off the back of the same 'loginAction' bean definition will not be seeing these changes in state since they are particular to an individual request. When the request is finished processing, the bean that is scoped to the request will be discarded.

3.5.3.3. The session scope

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

With the above bean definition in place, the Spring container will create a brand new instance of the UserPreferences bean using the 'userPreferences' bean definition for the lifetime of a single HTTP Session. In other words, the 'userPreferences' bean will be effectively scoped at the HTTP Session level. Just like request-scoped beans, you can change the internal state of the instance that is created as much as you want, safe in the knowledge that other HTTP Session instances that are also using instances created off the back of the same 'userPreferences' bean definition will not be seeing these changes in state since they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session will also be discarded.

3.5.3.4. The global session scope

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

The global session scope is similar to the standard HTTP Session scope (described immediately above), and really only makes sense in the context of portlet-based web applications. The portlet specification defines the notion of a global Session that is shared amongst all of the various portlets that make up a single portlet web application. Beans defined at the global session scope are scoped (or bound) to the lifetime of the global portlet Session.

Please note that if you are writing a standard Servlet-based web application and you define one or more beans as having global session scope, the standard HTTP Session scope will be used, and no error will be raised.

3.5.3.5. Scoped beans as dependencies

Being able to define a bean scoped to a HTTP request or Session (or indeed a custom scope of your own devising) is all very well, but one of the main value-adds of the Spring IoC container is that it manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject a bean that, for the sake of argument is scoped at the HTTP request scope, into another bean, you will need to inject an AOP proxy in place of the scoped bean. That is to say, you need to inject a proxy object that exposes the same public interface as the scoped object, but that is smart enough to be able to retrieve the real, target object from the relevant scope (for example a HTTP request) and delegate method calls onto the real object.

Note

You do not need to use the <aop:scoped-proxy/> in conjunction with beans that are scoped as singletons or prototypes. It is an error to try to create a scoped proxy for a singleton bean (and the resulting BeanCreationException will certainly set you straight in this regard).

Let's look at the configuration that is required to effect this; the configuration is not hugely complex (it takes just one line), but it is important to understand the "why" as well as the "how" behind it.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <!-- a HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

        <!-- this next element effects the proxying of the surrounding bean -->
        <aop:scoped-proxy/>
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.foo.SimpleUserService">

        <!-- a reference to the proxied 'userPreferences' bean -->
        <property name="userPreferences" ref="userPreferences"/>

    </bean>
</beans>
```

To create a proxy to a scoped bean using XML-based configuration, you need only to insert a child <aop:scoped-proxy/> element into a scoped bean definition (you may also need the CGLIB library on your classpath so that the container can effect class-based proxying; you will also need to be using XSD based configuration). The above XML configuration demonstrated the "how"; now for the "why". So, just why do you need this <aop:scoped-proxy/> element in the definition of beans scoped at the request, session, and globalSession level? The reason is best explained by picking apart the following bean definition (please note that the following 'userPreferences' bean definition as it stands is incomplete):

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

From the above configuration it is evident that the singleton bean 'userManager' is being injected with a reference to the HTTP Session-scoped bean 'userPreferences'. The salient point here is that the 'userManager' bean is a singleton... it will be instantiated exactly once per container, and its dependencies (in this case only one, the 'userPreferences' bean) will also only be injected once. This means that the 'userManager' will (conceptually) only ever operate on the exact same 'userPreferences' object, i.e. the one that it was originally injected with. This is not what you want when you inject a HTTP Session-scoped bean as a dependency into a collaborating object. What we do want is a single 'userManager' object, and then, for the lifetime of a HTTP Session, we want to see and use a 'userPreferences' object that is specific to said HTTP Session.

Rather what you need then is to inject some sort of object that exposes the exact same public interface as the UserPreferences class (ideally an object that is a UserPreferences instance) and that is smart enough to be able to go off and fetch the real UserPreferences object from whatever underlying scoping mechanism we have chosen (HTTP request, Session, etc.). We can then safely inject this proxy object into the 'userManager' bean, which will be blissfully unaware that the UserPreferences reference that it is holding onto is a proxy. In the case of this example, when a UserManager instance invokes a method on the dependency-injected UserPreferences object, it is really invoking a method on the proxy... the proxy will then go off and fetch the real UserPreferences object from (in this case) the HTTP Session, and delegate the method invocation onto the retrieved real UserPreferences object.

That is why you need the following, correct and complete, configuration when injecting request-, session-, and globalSession-scoped beans into collaborating objects:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

3.5.4. Custom scopes

As of Spring 2.0, the bean scoping mechanism in Spring is extensible. This means that you are not limited to just the bean scopes that Spring provides out of the box; you can define your own scopes, or even redefine the existing scopes (although that last one would probably be considered bad practice - please note that you cannot override the built-in singleton and prototype scopes).

Scopes are defined by the org.springframework.beans.factory.config.Scope interface. This is the interface that you will need to implement in order to integrate your own custom scope(s) into the Spring container. The interface itself is quite simple, with two methods to get and remove an object from/to an underlying storage mechanism respectively. Possible custom scopes are beyond the scope of this reference manual. You may wish to look at the Scope implementations that are supplied with Spring for an idea of how to go about implementing your own.

The remainder of this section details how, after you have written and tested one or more custom Scope implementations, you then go about making the Spring container aware of your new scope. The central method to register a new Scope with the Spring container is declared on the ConfigurableBeanFactory interface (implemented by most of the concrete BeanFactory implementations that ship with Spring); this central method is displayed below:

```
void registerScope(String scopeName, Scope scope);
```

The first argument to the registerScope(..) method is the unique name associated with a scope; examples of such names in the Spring container itself are 'singleton' and 'prototype'. The second argument to the registerScope(..) method is an actual instance of the custom Scope implementation that you wish to register and use.

Let's assume that you have written your own custom Scope implementation, and you have registered it like so:

```
// note: the ThreadScope class does not exist; I made it up for the sake of this example
Scope customScope = new ThreadScope();
beanFactory.registerScope("thread", scope);
```

You can then create bean definitions that adhere to the scoping rules of your custom Scope like so:

```
<bean id="..." class="..." scope="thread"/>
```

If you have your own custom Scope implementation(s), you are not just limited to only programmatic registration of said custom scope(s). You can also do the Scope registration declaratively, using a custom BeanFactoryPostProcessor implementation, the CustomScopeConfigurer class. The BeanFactoryPostProcessor interface is one of the primary means of extending the Spring IoC container, and is described in a later section of this very chapter.

The declarative registration of custom Scope implementations using the CustomScopeConfigurer class is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
```

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

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread" value="com.foo.ThreadScope"/>
            </map>
        </property>
    </bean>

    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>

</beans>
```

The CustomScopeConfigurer also allows you to specify actual Class instances as entry values, as well as actual Scope implementation instances; see the Javadocs for the CustomScopeConfigurer class for details.

3.6. bean의 성질을 커스터마이징하기.

3.6.1. Lifecycle 인터페이스

Spring은 컨테이너내 당신의 bean 행위를 변경하기 위한 다양한 표시자(marker)인터페이스를 제공한다. 그것들은 InitializingBean 과 DisposableBean을 포함한다. 이 인터페이스를 구현하는 것은 bean에게 초기화와 파괴화(destruction)의 작업을 수행하도록 허용하는 전자를 위한 afterPropertiesSet()을 후자를 위해 destroy()를 호출함으로써 컨테이너내 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 3.8, “컨테이너 확장 지점” 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.

3.6.1.1. 콜백 초기화하기

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

```
void afterPropertiesSet() throws Exception;
```

대개 InitializingBean 인터페이스의 사용은 제거될수 있다. (그리고 Spring에 코드를 불필요하게 결합한 후 억제된다.). bean정의는 명시되는 일반적인 초기화 메소드를 위한 지원을 제공한다. XML-기반의 설정 메타데이터의 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
    }
}
```

Is exactly the same as...

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

```
public class AnotherExampleBean implements InitializingBean {

    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

하지만 Spring에 코드를 결합하지는 않는다.

3.6.1.2. 콜백 파괴하기

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

```
void destroy() throws Exception;
```

DisposableBean 표시자(marker) 인터페이스의 사용은 제거될수 있다. (그리고 Spring에 코드를 불필요하게 결합한 후 억제된다.). bean정의는 명시되기 위한 일반적인 파괴(destroy) 메소드를 위한 지원을 제공한다. XML-기반의 설정 메타데이터의 경우에, <bean/>의 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 releasing pooled connections)
    }
}
```

Is exactly the same as...

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

```
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

하지만 Spring에 코드를 결합하지는 않는다.

3.6.1.2.1. Default initialization & destroy methods

When you are writing initialization and destroy method callbacks that do not use the Spring-specific InitializingBean and DisposableBean callback interfaces, one (in the experience of this author) typically finds oneself writing methods with names such as init(), initialize(), dispose(), etc. The names of such lifecycle callback methods are (hopefully!) standardized across a project so that developers on a team all use the same method names and thus ensure some level of consistency.

The Spring container can now be configured to 'look' for named initialization and destroy callback method names on every bean. This means that you as an application developer can simply write your application classes, use a convention of having an initialization callback called init(), and then (without having to configure each and every bean with, in the case of XML-based configuration, an 'init-method="init"' attribute) be safe in the knowledge that the Spring IoC container will call that method when the bean is being created (and in accordance with the standard lifecycle callback contract described previously).

Let's look at an example to make the use of this feature completely clear. For the sake of the example, let us say that one of the coding conventions on a project is that all initialization callback methods are to be named init() and that destroy callback methods are to be called destroy(). This leads to classes like so...

```
public class DefaultBlogService implements BlogService {

private BlogDao blogDao;

public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
}

// this is (unsurprisingly) the initialization callback method
public void init() {
        if (this.blogDao == null) {
                throw new IllegalStateException("The [blogDao] property must be set.");
        }
}
}
```

The attendant XML configuration for the above class, and making use of the by-convention initialization callback method configuration, would look like so:

```
<beans default-init-method="init">

<bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
</bean>
```

```
</beans>
```

Notice the use of the 'default-init-method' attribute on the top-level <beans/> element. The presence of this attribute means that the Spring IoC container will recognize a method called 'init' on beans as being the initialization method callback, and when a bean is being created and assembled, if the bean's class has such a method, it will be invoked at the appropriate time.

Destroy method callbacks are configured similarly (in XML that is) using the 'default-destroy-method' attribute on the top-level <beans/> element.

The use of this feature can save you the (small) housekeeping chore of specifying an initialization and destroy method callback on each and every bean, and it is great for enforcing a consistent naming convention for initialization and destroy method callbacks (and consistency is something that should always be aimed for).

One final word... let's say you want to use this feature, but you have some existing beans where the underlying classes already have for example initialization callback methods that are named at variance with the convention. You can always override the default by specifying (in XML that is) the method name using the 'init-method' and 'destroy-method' attributes on the <bean/> element itself.

3.6.1.2.2. Shutting down the Spring IoC container gracefully in non-web applications

Note

This next section does not apply to web applications (in case the title of this section did not make that abundantly clear). Spring's web-based ApplicationContext implementations already have code in place to handle shutting down the Spring IoC container gracefully when the relevant web application is being shutdown.

If you are using Spring's IoC container in a non-web application environment, for example in a rich client desktop environment, and you want the container to shutdown gracefully and call the relevant destroy callbacks on your singleton beans, you will need to register a shutdown hook with the JVM. This is quite easy to do (see below), and will ensure that your Spring IoC container shuts down gracefully and that all resources held by your singletons are released (of course it is still up to you to both configure the destroy callbacks for your singletons and implement such destroy callbacks correctly).

So to register a shutdown hook that enables the graceful shutdown of the relevant Spring IoC container, you simply need to call the registerShutdownHook() method that is declared on the AbstractApplicationContext class. To wit...

```
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

public static void main(final String[] args) throws Exception {
        AbstractApplicationContext ctx
                = new ClassPathXmlApplicationContext(new String []{"beans.xml"});
```

```
        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...
}
}
```

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

3.6.2.1. BeanFactoryAware

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

```
public interface BeanFactoryAware {

    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

이것은 bean에게 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.

BeanFactoryAware 기반 접근법에 동등하게 영향을 끼치는 대체가능한 선택사항은 org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean를 사용하는 것이다.(이것은 여전히 Spring에 대한 커플링을 제거하기 않는다. 하지만 BeanFactoryAware기반 접근법만큼 IoC의 핵심개념에는 위배되지 않는다는 것을 기억하라.)

ObjectFactoryCreatingFactoryBean은 순서대로 bean룩업에 영향을 끼치기 위해 사용될수 있는 객체(factory)에 대한 참조를 반환하는 FactoryBean 구현물이다. ObjectFactoryCreatingFactoryBean 클래스는 자체적으로 실질적으로 삽입되는 클라이언트 bean이 ObjectFactory 인터페이스의 인스턴스인 BeanFactoryAware인터페이스를 구현한다. 이것은 Spring의 종속적인 인터페이스(그리고 나아가 Spring으로부터 완전히 디커플링되지는 않는)이지만, 클라이언트는 bean룩업에 영향을 주는 ObjectFactory의 getObject()메소드를 사용할수 있다(반환되는 ObjectFactory구현물아래에서 이름(name)으로 bean을 실질적을 룩업하는 BeanFactory로 위임한다.). 애플리케이션 개발자가 할 필요가 있는것은 룩업되는 bean의 이름을 가지는 ObjectFactoryCreatingFactoryBean를 제공하는 것이다. 예제를 보자.

```
package x.y;

public class NewsFeed {

    private String news;

    public void setNews(String news) {
        this.news = news;
    }

    public String getNews() {
        return this.toString() + ": '" + news + "'";
    }
}
```

```
package x.y;

import org.springframework.beans.factory.ObjectFactory;

public class NewsFeedManager {

    private ObjectFactory factory;

    public void setFactory(ObjectFactory factory) {
        this.factory = factory;
    }

    public void printNews() {
        // here is where the lookup is performed; note that there is no
        // need to hardcode the name of the bean that is being looked up...
        NewsFeed news = (NewsFeed) factory.getObject();
        System.out.println(news.getNews());
    }
}
```

ObjectFactoryCreatingFactoryBean 접근법을 사용하여 위 클래스를 묶는 XML설정을 아래에서 보자.

```
<beans>
    <bean id="newsFeedManager" class="x.y.NewsFeedManager">
        <property name="factory">
            <bean
class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">
                <property name="targetBeanName">
                    <idref local="newsFeed" />
                </property>
            </bean>
        </property>
    </bean>
    <bean id="newsFeed" class="x.y.NewsFeed" scope="prototype">
        <property name="news" value="... that's fit to print!" />
    </bean>
</beans>
```

그리고 이것은 newsFeed bean의 새로운 인스턴스(prototype)가 NewsFeedManager의 printNews() 메소드내부 삽입된 ObjectFactory에 대한 각각의 호출을 위해 실질적으로 반환되는 것을 테스트하기 위한 작은 프로그램이다.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.NewsFeedManager;

public class Main {

    public static void main(String[] args) throws Exception {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        NewsFeedManager manager = (NewsFeedManager) ctx.getBean("newsFeedManager");
        manager.printNews();
        manager.printNews();
    }
}
```

위 실행중인 프로그램으로부터의 결과는 다음과 같은 것을 보여줄것이다.(물론 결과는 당신의 머신에 따라 다양할것이다.)

```
x.y.NewsFeed@1292d26: '... that's fit to print!'
```

```
x.y.NewsFeed@5329c5: '... that's fit to print!'
```

3.6.2.2. 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.

3.7. Bean정의 상속

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. XML-기반의 설정 메타데이터를 사용할때, 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"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
      class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBean" init-method="initialize">

  <property name="name" value="override"/>
  <!-- the age property value of 1 will be inherited 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 init, destroy메소드와/또는 정적 factory메소드가 명시된다면 그것들은 관련된 부모 셋팅을 오버라이드할것이다.

남은 셋팅들은 언제나 자식 정의로부터 가져올것이다.: depends on, autowire 모드, 의존성 체크, 싱글톤, scope, 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" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
    parent="inheritedTestBeanWithoutClass" init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit the value of 1 from the parent bean definition-->
</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.

ApplicationContexts(BeanFactories 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 'abstract'속성값을 true로 셋팅해야만 하는 반면에 애플리케이션 컨텍스트는 이것을 실질적으로 미리 인스턴스화할것이라는 것은 중요(적어도 싱글톤 bean을 위해서)하다.

3.8. 컨테이너 확장 지점

Spring 프레임워크의 IoC컴포넌트는 확장을 위해 디자인되었다. 애플리케이션 개발자를 위해 다양한 BeanFactory 나 ApplicationContext 구현 클래스의 하위클래스를 만들필요가 없다. Spring IoC컨테이너는 특별한 통합 인터페이스의 구현물에 삽입하여 크게 확장될수 있다. 다음의 몇가지 부분은 이러한 다양한 통합 인터페이스를 상세하게 다룬다.

3.8.1. BeanPostProcessors로 bean 커스터마이징하기

우리가 볼 첫번째 확장지점은 BeanPostProcessor 인터페이스이다. The first extension point that we will look at is the BeanPostProcessor interface. This interface defines a number of callback methods that you as an application developer can implement in order to provide your own (or override the containers default) instantiation logic, dependency-resolution logic, and so forth. If you want to do some custom logic after the Spring container has finished instantiating, configuring and otherwise initializing a bean, you can plug in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessors if you wish. You can control the order in which these BeanPostProcessors execute by setting the 'order' property (you can only set this property if the BeanPostProcessor implements the Ordered interface; if you write your own BeanPostProcessor you should consider implementing the Ordered interface too); consult the Javadocs for the BeanPostProcessor and Ordered interfaces for more details.

Note

BeanPostProcessors operate on bean (or object) instances; that is to say, the Spring IoC container will have instantiated a bean instance for you, and then

BeanPostProcessors get a chance to do their stuff.

If you want to change the actual bean definition (i.e. the recipe that defines the bean), then you rather need to use a BeanFactoryPostProcessor (described below in the section entitled Section 3.8.2, “Customizing configuration metadata with BeanFactoryPostProcessors” .

Also, BeanPostProcessors are scoped per-container. This is only relevant if you are using container hierarchies. If you define a BeanPostProcessor in one container, it will only do its stuff on the beans in that container. Beans that are defined in another container will not be post-processed by BeanPostProcessors in another container, even if both containers are part of the same hierarchy.

The org.springframework.beans.factory.config.BeanPostProcessor interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container (see below for how this registration is effected), for each bean instance that is created by the container, the post-processor will get a callback from the container both before any container initialization methods (such as afterPropertiesSet and any declared init method) are called, and also afterwards. The post-processor is free to do what it wishes with the bean instance, including ignoring the callback completely. A bean post-processor will typically check for marker interfaces, or do something such as wrap a bean with a proxy; some of the Spring AOP infrastructure classes are implemented as bean post-processors and they do this proxy-wrapping logic.

It is important to know that a BeanFactory treats bean post-processors slightly differently than an ApplicationContext. An ApplicationContext will automatically detect any beans which are defined in the configuration metadata which is supplied to it that implement the BeanPostProcessor interface, and register them as post-processors, to be then called appropriately by the container on bean creation. Nothing else needs to be done other than deploying the post-processor in a similar fashion to any other bean. On the other hand, when using a BeanFactory implementation, bean post-processors explicitly have to be registered, with code like this:

```
ConfigurableBeanFactory factory = new XmlBeanFactory(...);

        // now register any needed BeanPostProcessor instances
        MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();
        factory.addBeanPostProcessor(postProcessor);

        // now start using the factory
```

This explicit registration step is not convenient, and this is one of the reasons why the various ApplicationContext implementations are preferred above plain BeanFactory implementations in the vast majority of Spring-backed applications, especially when using BeanPostProcessors.

Note

You typically don’t want to have BeanPostProcessors marked as being lazily-initialized. If they are marked as such, then the Spring container will never instantiate them, and thus they won’t get a chance to apply their custom logic. If you are using the ’default-lazy-init’ attribute on the declaration of your <beans/> element, be sure to mark your various BeanPostProcessor bean definitions with ’lazy-init="false"’.

Find below some examples of how to write, register, and use BeanPostProcessors in the context of an ApplicationContext.

3.8.1.1. Example: Hello World, BeanPostProcessor-style

This first example is hardly compelling, but serves to illustrate basic usage. All we are going to do is code a custom BeanPostProcessor implementation that simply invokes the toString() method of each bean as it is created by the container and prints the resulting string to the system console. Yes, it is not hugely useful, but serves to get the basic concepts across before we move into the second example which is actually useful.

Find below the custom BeanPostProcessor implementation class definition:

```
package scripting;

	import org.springframework.beans.factory.config.BeanPostProcessor;
	import org.springframework.beans.BeansException;

	public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

		// simply return the instantiated bean as-is
		public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
			return bean; // we could potentially return any object reference here...
		}

		public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
			System.out.println("Bean '" + beanName + "' created : " + bean.toString());
			return bean;
		}
	}
```

Here is the attendant XML-based configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:lang="http://www.springframework.org/schema/lang"
	xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

    <lang:groovy id="messenger"
          script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
    </lang:groovy>

    <!--
        when the above bean ('messenger') is instantiated, this custom
        BeanPostProcessor implementation will output the fact to the system console
     -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

Notice how the InstantiationTracingBeanPostProcessor is simply defined; it doesn't even have a name, and because it is a bean it can be dependency injected just like any other bean. (The above configuration also just so happens to define a bean that is backed by a Groovy script. The Spring 2.0 dynamic language support is detailed in the chapter entitled Chapter 24, 동적

언어 지원.)

Find below a small driver script to exercise the above code and configuration;

```
import org.springframework.context.ApplicationContext;
        import org.springframework.context.support.ClassPathXmlApplicationContext;
        import org.springframework.scripting.Messenger;

        public final class Boot {

                public static void main(final String[] args) throws Exception {
                        ApplicationContext ctx = new ClassPathXmlApplicationContext("scripting/beans.xml");
                        Messenger messenger = (Messenger) ctx.getBean("messenger");
                        System.out.println(messenger);
                }
        }
```

The output of executing the above program will be (something like) this:

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961
        org.springframework.scripting.groovy.GroovyMessenger@272961
```

3.8.1.2. Example: The RequiredAnnotationBeanPostProcessor

Using marker interfaces or annotations in conjunction with a custom BeanPostProcessor implementation is a common means of extending the Spring IoC container. This next example is a bit of a cop-out, in that you are directed to the section entitled Section 25.3.1, "@Required" which demonstrates the usage of a custom BeanPostProcessor implementation that that ships with the Spring distribution which ensures that JavaBean properties on beans that are marked with an (arbitrary) annotation are actually (configured to be) dependency-injected with a value.

3.8.2. Customizing configuration metadata with BeanFactoryPostProcessors

The next extension point that we will look at is the org.springframework.beans.factory.config.BeanFactoryPostProcessor. This semantics of this interface are similar to the BeanPostProcessor, with one major difference. BeanFactoryPostProcessors operate on bean definitions (i.e. the configuration metadata that is supplied to a container); that is to say, the Spring IoC container will allow BeanFactoryPostProcessors to read the configuration metadata and potentially change it before the container has actually instantied any other beans.

You can configure multiple BeanFactoryPostProcessors if you wish. You can control the order in which these BeanFactoryPostProcessors execute by setting the 'order' property (you can only set this property if the BeanFactoryPostProcessor implements the Ordered interface; if you write your own BeanFactoryPostProcessor you should consider implementing the Ordered interface too); consult the Javadocs for the BeanFactoryPostProcessor and Ordered interfaces for more details.

Note

If you want to change the actual bean instances (i.e. the objects that are created from the configuration metadata), then you rather need to use a BeanPostProcessor (described above in the section entitled Section 3.8.1, "BeanPostProcessors로 bean

커스터마이징하기" .

Also, BeanFactoryPostProcessors are scoped per-container. This is only relevant if you are using container hierarchies. If you define a BeanFactoryPostProcessor in one container, it will only do its stuff on the bean definitions in that container. Bean definitions in another container will not be post-processed by BeanFactoryPostProcessors in another container, even if both containers are part of the same hierarchy.

A bean factory post-processor is executed manually (in the case of a BeanFactory) or automatically (in the case of a ApplicationContext) to apply changes of some sort to the configuration metadata that defines a container. Spring includes a number of pre-existing bean factory post-processors, such as PropertyResourceConfigurer and PropertyPlaceHolderConfigurer, both described below, and BeanNameAutoProxyCreator, which is very useful for wrapping other beans transactionally or with any other kind of proxy, as described later in this manual. The BeanFactoryPostProcessor can be used to add custom property editors.

In a BeanFactory, the process of applying a BeanFactoryPostProcessor is manual, and will be similar to this:

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

        // 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);
```

This explicit registration step is not convenient, and this is one of the reasons why the various ApplicationContext implementations are preferred above plain BeanFactory implementations in the vast majority of Spring-backed applications, especially when using BeanFactoryPostProcessors.

An ApplicationContext will detect any beans which are deployed into it which implement the BeanFactoryPostProcessor interface, and automatically use them as bean factory post-processors, at the appropriate time. Nothing else needs to be done other than deploying these post-processor in a similar fashion to any other bean.

Note

Just as in the case of BeanPostProcessors, you typically don't want to have BeanFactoryPostProcessors marked as being lazily-initialized. If they are marked as such, then the Spring container will never instantiate them, and thus they won't get a chance to apply their custom logic. If you are using the 'default-lazy-init' attribute on the declaration of your <beans/> element, be sure to mark your various BeanFactoryPostProcessor bean definitions with 'lazy-init="false"'.

3.8.2.1. 예제: PropertyPlaceholderConfigurer

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

허용하는데 유용하다.

위치유지자(placeholder)값과 함께 DataSource가 정의된 다음의 XML-기반의 설정 메타데이터 일부를 검토하라. 우리는 외부 Properties파일로부터 몇몇 프라퍼티를 설정할것이다. 실행시 우리는 데이터소스의 몇몇 프라퍼티를 대체할 메타데이터를 위한 PropertyPlaceholderConfigurer을 적용할것이다.

```
<bean id="dataSource" destroy-method="close"
                class="org.apache.commons.dbcp.BasicDataSource">
            <property name="driverClassName" value="$ {jdbc.driverClassName}"/>
            <property name="url" value="$ {jdbc.url}"/>
            <property name="username" value="jdbc.username"/>
            <property name="password" value="$ {jdbc.password}"/>
        </bean>
```

실질적인 값들은 표준적인 Java Properties형태로 다른 파일로부터 나온다.

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

PropertyPlaceHolderConfigurer는 명시하는 Properties 파일내 프라퍼티를 찾지 않는다. 하지만 사용하고자 하는 프라퍼티를 찾지 못한다면 Java System 프라퍼티를 체크한다. 이 행위는 설정자의 systemPropertiesMode 프라퍼티를 셋팅하여 사용자정의 될수 있다. 이것은 3개의 값을 가진다. 하나는 언제나 오버라이드하는 설정자를 말하고, 하나는 결코 오버라이드하지 않도록하고, 하나는 프라퍼티가 명시된 프라퍼티 파일내에서 찾을수 없을때만 오버라이드하도록 한다. 좀더 많은 정보를 위해서는 PropertiesPlaceholderConfigurer를 위한 Javadoc을 보라.

3.8.2.2. 예제: PropertyOverrideConfigurer

PropertyOverrideConfigurer, 다른 bean factory 후-처리자는 PropertyPlaceholderConfigurer와 비슷하지만 후자와는 대조적으로 원래의 정의는 디폴트 값을 가지거나 bean프라퍼티를 위한 값을 가질수 없다. 만약 오버라이딩된 Properties 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 PropertyOverrideConfigurer가 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.

마지막 프라퍼티를 제외한 경로의 모든 컴포넌트가 이미 null이 아닌 만큼 복합 프라퍼티명또한

지원된다(추측컨데 생성자에 의해 초기화된). 이 예제에서:

```
foo.fred.bob.sammy=123
```

foo bean의 fred프라퍼티의 bob프라퍼티의 sammy프라퍼티는 123으로 셋팅되어 있다..

3.8.3. FactoryBeans를 사용하여 인스턴스화 로직을 사용자정의하기

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

FactoryBean 인터페이스는 Spring IoC 컨테이너 인스턴스화 로직에 대한 접속가능한 지점이다. 많은 XML에 반대로 Java로 좀더 잘 표현되는 몇가지 복잡한 초기화 코드를 가진다면, 당신은 클래스내부에서 복잡한 초기화를 작성하고 컨테이너로 사용자정의 FactoryBean를 삽입하는 자체적인 FactoryBean을 생성할수 있다.

FactoryBean 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.

FactoryBean 개념과 인터페이스는 Spring프레임워크내 많은 부분에서 사용된다. 이 글이 작성되는 시점에 Spring자체에 포함된 FactoryBean 인터페이스의 구현물이 50개 이상이 있다.

마지막으로, 때때로 bean이 아닌 실제 FactoryBean 인스턴스 자체를 위한 컨테이너를 요청할 필요가 있다. BeanFactory(ApplicationContext를 포함해서)의 getBean 메소드를 호출할때 '&'를 가진 bean id를 앞에 붙여서 수행될것이다. 그래서 myBean id를 가진 FactoryBean을 위해, 컨테이너의 getBean("myBean")를 호출하는 것은 FactoryBean의 산출물을 반환할것이다. 하지만 getBean("&myBean")을 호출하는 것은 FactoryBean 인스턴스 자체를 반환할것이다.

3.9. ApplicationContext

beans 패키지는 종종 프로그램마다 다른 방식으로 관리와 bean을 변경하기 위한 기초적인 기능을 제공하는 동안 context 패키지는 좀 더 프레임워크 기반 형태로 BeanFactory기능을 강화시키는 ApplicationContext를 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 ApplicationContext인터페이스이다. BeanFactory인터페이스에서의 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.

3.9.1. MessageSources를 사용하여 국제화하기

ApplicationContext인터페이스는 MessageSource라고 불리는 인터페이스를 확장해서 메시징(i18n또는 국제화)기능을 제공한다. HierarchicalMessageSource와 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 resource bundle을 가진다고 가정한다. ResourceBundles을 통한 메시지를 분석하는 JDK표준적인 방법을 사용하여 메시지를 분석하기 위한 요청이 다루어질것이다. 이 예제의 목적을 위해, 위 resource bundle파일의 두개의 내용을 가정해보자.

```
# in 'format.properties'
    message=Alligators rock!
```

```
# in 'exceptions.properties'
    argument.required=The '{0}' argument is required.
```

MessageSource 기능을 수행하기 위한 몇몇 (명백히 자명한) 드라이버 코드는 아래와 같을수 있다. 모든 ApplicationContext구현물이 MessageSource이고 그래서 MessageSource인터페이스로 변환할수 있다는 것을 기억하라.

```
public static void main(String[] args) {
            MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
            String message = resources.getMessage("message", null, "Default", null);
            System.out.println(message);
    }
```

위 프로그램의 수행으로 부터 결과 출력물은 다음과 같을 것이다..

```
Alligators rock!
```

목록화하기 위해, MessageSource는 'beans.xml'라고 명명된 파일내 정의되었다(이 파일은 classpath의 가장 상위에 존재한다.). 'messageSource' bean정의는 basenames프라퍼티를 통해 많은 resource bundle을 참조한다. 목록내에서 basenames프라퍼티로 전달되는 3개의 파일은 당신의 classpath의 가장 상위에 존재한다. (각각 format.properties, exceptions.properties, 그리고 windows.properties 라는 파일명을 가진다).

다른 예제를 보자. 그리고 이 시점에 우리는 메시지 검색을 위한 인자를 전달할 것이다. 이 인자들은 문자열로 변환되고 검색 메시지내 placeholder로 삽입된다. 이것은 예제와 함께 설명되는게 가장 좋다.

```
<beans>

        <!-- this MessageSource is being used in a web application -->
        <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
            <property name="baseName" value="WEB-INF/test-messages"/>
        </bean>

        <!-- let's inject the above MessageSource into this POJO -->
        <bean id="example" class="com.foo.Example">
            <property name="messages" ref="messageSource"/>
        </bean>

    </beans>
```

```
public class Example {

        private MessageSource messages;
```

```
        public void setMessages(MessageSource messages) {
            this.messages = messages;
        }

        public void execute() {
            String message = this.messages.getMessage("argument.required",
                new Object [] {"userDao"}, "Required", null);
            System.out.println(message);
        }

    }
```

execute() 메소드의 호출의 결과는 다음과 같을것이다.

```
The 'userDao' argument is required.
```

국제화(i18N)를 고려해볼때, Spring의 다양한 MessageResource구현물은 같은 로케일 분석과 표준적인 JDK ResourceBundle처럼 되돌리기(fallback)규칙을 따른다. 짧게 말해, 이미 정의된 예제 'messageSource'로 계속할것이다. 만약 당신이 영국(en-GB) 로케일에 대해 메시지를 해석하길 원한다면, 각각 format_en_GB.properties, exceptions_en_GB.properties, 그리고 windows_en_GB.properties라고 불리는 파일을 생성할 것이다.

로케일 분석은 애플리케이션의 주위환경에 의해 대개 관리된다. 비록 이 예제의 목적을 위해, 우리는 영국 메시지를 분석하기를 원하는 로케일을 직접 명시할것이다.

```
# in 'exceptions_en_GB.properties'
    argument.required=Ebagum lad, the '{0}' argument is required, I say, required.
```

```
public static void main(final String[] args) {
        MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
        String message = resources.getMessage("argument.required",
            new Object [] {"userDao"}, "Required", Locale.UK);
        System.out.println(message);
    }
```

위 프로그램으의 수행으로 부터 결과 출력물은 다음과 같을 것이다..

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

MessageSourceAware인터페이스는 정의된 MessageSource에 대한 참조를 얻기 위해 사용될수 있다. MessageSourceAware인터페이스를 구현하는 ApplicationContext로 정의된 어느 bean은 생성되고 설정될때 애플리케이션 컨텍스트의 MessageSource를 주입할것이다.

3.9.2. 이벤트

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 3.5. Built-in Events

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(ApplicationEventMulticaster 구현물을 통한 대안의 이벤트 배포 전략을 제공하는 것이 가능하다.). 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"/>
      </bean>
```

Now, let's look at the actual classes:

```
public class EmailBean implements ApplicationContextAware {

    private List blackList;
        private ApplicationContext ctx;

    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 {

        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.

3.9.3. 하위 레벨 자원(resource)에 대한 편리한 접근

애플리케이션 컨텍스트의 최적화된 사용법과 이해를 위해, 사용자는 Chapter 4, 자원에서 언급된것처럼 Spring의 Resource추상화 자체에 친숙해야만 한다.

애플리케이션 컨텍스트는 Resource를 로드하기 위해 사용될수 있는 ResourceLoader이다. Resource는 본질적으로는 classpath, 파일시스템 위치, 표준적인 URL로 언급할수 있는 어떠한 지점, 그리고 다양한 변형형태를 포함하는 투명한 형태의 위치로부터 하위 레벨의 resource를 얻기 위해 사용될수 있는 java.net.URL이다(사실, 이것은 적절한 위치의 URL을 포장하고 사용한다.). resource위치 문자열이 어떤 특별한 접두사없는 간단한 경로라면, resource이 유래된 지점은 실질적 애플리케이션 컨텍스트 타입에 특성화되고 적절하게된다.

애플리케이션 컨텍스트로 배치된 bean은 애플리케이션 컨텍스트 자체가 ResourceLoader로 전달되는 초기화시점에 자동적으로 콜백되기 위한 특별한 표시자(marker) 인터페이스인 ResourceLoaderAware를 구현한다.

bean은 정적 resource에 접근하기 위해 사용되는 Resource타입의 프라퍼티를 나타내고 프라퍼티는 다른 프라퍼티처럼 주입될것이다. bean을 배치하는 사람은 간단한 문자열 경로처럼 Resource 프라퍼티를 명시하고 텍스트 문자열을 실질적인 Resource객체로 변환하기 위해 컨텍스트에 의해 자동으로 등록된 특별한 자바빈 PropertyEditor에 의존한다.

위치 경로나 ApplicationContext생성자에 제공되는는 경로는 실질적으로는 resource문자열이고 특정 컨텍스트 구현물에 적절하게 처리된 간단한 형태이다(이를테면 ClassPathXmlApplicationContext는 classpath위치로 간단한 위치경로를 처리한다.). 하지만 실질적 컨텍스트 타입에 따라 classpath나 URL로부터 정의의 로딩을 강요하기 위한 특정 접두사와 함께 사용될것이다.

3.9.4. 웹 애플리케이션을 위한 편리한 ApplicationContext 인스턴스화

프로그램으로 처리해서 종종 생성될 BeanFactory와 반대로, ApplicationContext 인스턴스는 예를 들어

ContextLoader를 사용하여 선언적으로 생성될수 있다. 물론 당신은 ApplicationContext구현물중 하나를 사용하여 프로그램으로 처리하여 ApplicationContext인스턴스를 생성할수 있다. 먼저 ContextLoader 인터페이스와 그것의 구현물을 시험해보자.

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 above 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.

3.10. Glue 코드와 좋지않은 싱글톤

애플리케이션내 코드의 대부분은 이것이 생성될때 컨테이너에 의해 자신만의 의존성이 제공되고 컨테이너를 완벽하게 자각하지 못하는 Spring IoC 컨테이너에 코드가 제공되는 DI스타일로 작성되는 것이 가장 좋다. 어쨌든 다른 코드와 함께 묶기 위한 때때로 필요한 코드의 작은 glue레이어를 위해 Spring IoC컨테이너에 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 Spring IoC컨테이너에 접근하는 싱글톤 스타일을 사용한다. inversion of control은 여전히 코드의 대부분을 위해 달성된다.(객체는 컨테이너로 부터 나온다.). 게다가 대부분의 코드는 컨테이너나 이것이 접근되는 방법을 자각하지 않는다. 그리고 모든 이익을 가지는 다른 코드로 부터 커플링되지 않은체로 남게된다. EJB는 Spring IoC컨테이너로 부터 나오는 명확한 자바 구현물 객체를 위해 위임하는 스텁/프록시 접근법을 사용할수 있다. Spring IoC컨테이너 자체가 이론상 싱글톤이 되지 않는 동안 이것은 비-싱글톤 Spring IoC컨테이너를 사용하는 각각의 bean을 위해 메모리 사용이나 초기화 시점(Hibernate SessionFactory처럼 Spring IoC컨테이너내 bean을 사용할 때)의 개념에서 비현실적일수 있다.

다른 예제처럼, 다중 레이어(이를테면, 다양한 JAR파일들, EJB들, 그리고 EAR처럼 패키지된 WAR파일)의 복잡한 J2EE애플리케이션에서 이것 자체의 Spring IoC 컨테이너정의(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 Spring IoC 컨테이너를 간단하게 생성하기 위한 접근법이 선호된다. 모든 Spring IoC 컨테이너 구현물은 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 Spring IoC 컨테이너를 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 SingletonBeanFactoryLocator와 같은 클래스는 웹 애플리케이션 Spring IoC 컨테이너의 부모처럼 사용될수 있는 효과적인 싱글톤형태로 다중 구조적 Spring IoC 컨테이너 로드를 요구하기 위해 사용될수 있다. 그 결과는 아래쪽의 레이어를 위한 bean정의가 필요할때만 오직 한번 로드되는것이다.

3.10.1. 싱글톤 헬퍼 클래스 사용하기

당신은 각각의 JavaDoc를 봐서 SingletonBeanFactoryLocator 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. 자원

4.1. 소개

자바의 표준 java.net.URL인터페이스와 다양한 URL접두사를 위한 표준 핸들러는 불행하게도 하위-레벨 자원에 모두 접근하기 위해 충분하지 않다. classpath나 ServletContext에 상대적인 위치로부터 얻을 필요가 있는 자원에 접근하기 위해 사용될 표준 URL 구현물이 없다. 특정 URL접두사(http:와 같은 접두사를 위해 존재하는 핸들러와 유사한)를 위한 새로운 핸들러를 등록하는 것이 가능한 반면에, 이것은 대개 처리하기 어렵다. 그리고 URL 인터페이스는 여전히 가리키는 자원의 존재여부를 체크하기 위한 메소드와 같은 몇몇 필요한 함수가 빈약하다.

4.2. Resource 인터페이스

Spring의 Resource인터페이스는 하위-레벨 자원에 대한 추상 접근을 위해 좀더 많은 능력을 가지는 인터페이스이다.

```
public interface Resource extends InputStreamSource {

    boolean exists();

    boolean isOpen();

    URL getURL() throws IOException;

    File getFile() throws IOException;

    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();
}
```

```
public interface InputStreamSource {

    InputStream getInputStream() throws IOException;

}
```

Resource 인터페이스로부터 몇몇 매우 중요한 메소드는:

☒ getInputStream(): 자원의 위치를 정하고 연다. 자원을 읽기 위한 InputStream을 반환한다. 각각의 호출은 refresh된 InputStream을 반환한다. 스트림을 닫기 위한 호출자의 책임을 가진다.

☒ exists(): 자원이 물리적인 형태로 존재하는지에 대한 boolean값 표시를 반환.

☒ isOpen(): 이 자원이 열린 스트림을 다루는지에 대한 boolean값 표시를 반환. 만약 true라면, InputStream은 여러번 읽을수 없고 반드시 한번 읽고 난후 자원 부족을 피하기 위해 닫아야만 한다. InputStreamResource예외를 가지고 평소의 자원 구현을 위해서는 false로 두라.

☒ getDescription(): 자원을 이용해서 작업할때 에러출력을 위해 사용되기 위한 상세설명을 반환한다.

이것은 종종 전체경로의 파일명이나 실질적인 URL이다.

밑바닥에 깔린 구현물은 호환가능하고 이 함수들을 지원한다면 다른 메소드는 당신에게 자원을 표현하는 실질적인 자원의 URL이나 File객체를 얻도록 허용한다.

Resource 추상화는 자원이 필요할때 많은 메소드 시그너처내 인자타입처럼 Spring자체에 광범위하게 사용된다. 몇몇 Spring API(다양한 ApplicationContext구현물의 생성자처럼)내 다른 메소드들은 context구현물에 적절한 Resource를 생성하기 위해 사용되는 있는 그대로이거나(unadorned) 간단한 형태인 String을 가지거나 String경로의 특별한 접두사를 통해, 생성되고 사용되어야하는 특별한 Resource구현물을 명시하기 위한 호출자를 허용한다.

당신의 코드가 Spring의 다른 부분에 대해 알지않고 다루지 않을때 조차도 자원에 접근하기 위해, Resource는 많은 Spring과 사용되고 Spring에 의해 사용된다. 이것은 당신 자신의 코드에 의해 일반적인 유틸리티 클래스처럼 사용되기 위해 매우 유용하다. 이것이 당신의 코드를 Spring에 커플링하는 동안, 이것은 URL을 위해 더 많은 능력을 가진 대체물처럼 제공되고 이 목적을 위해 당신이 사용할 다른 라이브러리에 동일할수 있는 유틸리티 클래스의 작은 세트를 위해 이것을 커플링한다.

이것은 Resource가 기능을 대체하지 않는다는 것을 아는것이 중요하다. 예를들어, UrlResource는 URL을 포장하고 포장된 URL을 사용한다.

4.3. 내장된 Resource구현물

Spring외부에서 일관적으로 제공되는 많은 수의 내장된 Resource구현물이 있다.

4.3.1. UrlResource

UrlResource는 java.net.URL을 포장한다. 그리고 파일, HTTP target, FTP target 등등과 같은 URL을 통해 대개 접근가능한 객체에 접근하기 위해 사용된다. 모든 URL은 적절히 표준화된 접두사가 하나의 URL타입대 다른것을 표시하기 위해 사용되는것처럼 표준적인 String표현을 가진다. 이것은 파일시스템 경로에 접근하기 위한 file:, HTTP프로토콜을 통해 자원에 접근하기 위한 http:, FTP를 통해 자원에 접근하기 위한 ftp: 등을 포함한다.

UrlResource는 UrlResource생성자를 사용하여 명시적으로 자바코드에 의해 생성된다. 하지만 당신이 경로를 표현하는 String인자를 가지는 API메소드를 호출할수 있을때 무조건적으로 생성될것이다. 후자의 경우를 위해, JavaBeans PropertyEditor는 생성하기 위한 Resource의 타입을 결정할것이다. 만약 경로 문자열이 classpath:와 같이 잘 알려진 접두사를 몇개 포함한다면, 그 접두사를 위한 적절히 특별한 Resource를 생성할것이다. 어쨌든, 접두사를 인지하지 않는다면, 이것은 표준 URL문자이고 UrlResource를 생성할것이라는 것을 가정할것이다.

4.3.2. ClassPathResource

클래스는 classpath로부터 얻어지는 자원을 표현한다. 이것은 주어진 클래스로더이거나 로딩하는 자원을 위해 주어진 클래스인 쓰레드 컨텍스트 클래스 로더를 사용한다.

이 Resource의 구현물은 클래스 경로 자원이 파일시스템내 위치하지만, jar내 위치하는 classpath자원을 위한 것이 아니고 파일시스템에 확장(서블릿 엔진에 의하거나 환경이 무엇이듯)되지 않았다면 java.io.File처럼 분석을 지원한다. 이것은 java.net.URL처럼 언제나 분석을 지원한다.

ClassPathResource는 ClassPathResource 생성자를 사용하여 명시적으로 자바 코드에 의해 생성된다. 하지만 경로를 표현하는 String 인자를 가지는 API메소드를 호출할때 무조건적으로 생성될것이다. 후자의 경우를 위해, JavaBeans PropertyEditor는 문자열 경로에서 특별한 접두사 classpath:를 인지하고 이 경우 ClassPathResource를 생성한다.

4.3.3. FileSystemResource

이것은 java.io.File가 다루기 위한 Resource구현물이다. 이것은 File과 URL처럼 분석을 명백하게 지원한다.

4.3.4. ServletContextResource

이것은 웹 애플리케이션 root디렉토리내 상대적인 경로를 해석하는 ServletContext자원을 위한 Resource 구현물이다.

이것은 스트림접근과 URL접근을 언제나 지원하지만, 웹 애플리케이션 저장고(archive)가 확장되고 자원이 파일시스템에서 물리적일때만 java.io.File접근을 허용한다. 이것처럼 확장되거나 파일시스템에 있는지, JAR나 DB와 같은 것으로부터 직접 접근되는지는 Servlet컨테이너에 실질적으로 의존한다.

4.3.5. InputStreamResource

주어진 InputStream을 위한 Resource 구현물이다.이것은 적용가능한 특정 Resource구현물이 없을 경우에만 사용된다. 특별히, 가능하다면 ByteArrayResource나 파일-기반 Resource구현물을 선호하라.

다른 Resource구현물과 반대로, 이것은 이미 열려있는 자원을 위한 설명자(descriptor)이다. 그러므로 isOpen()에서 "true"를 반환한다. 만약 당신이 어딘가에 자원 설명자를 유지할 필요가 있거나 스트림을 여러번 읽을 필요가 있다면 이것을 사용하지 말라.

4.3.6. ByteArrayResource

이것은 주어진 byte배열을 위한 Resource 구현물이다. 이것은 주어진 byte배열을 위한 ByteArrayInputStreams을 생성한다.

이것은 한번 사용하는 InputStreamResource를 재정렬하지 않고 주어진 byte배열로부터 내용을 로드하기 위해 유용하다.

4.4. ResourceLoader 인터페이스

ResourceLoader 인터페이스는 Resources를 반환할수 있는 객체에 의해 구현된다.

```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

모든 애플리케이션 컨텍스트는 ResourceLoader를 구현한다. 그러므로 모든 애플리케이션 컨텍스트는 Resource를 얻기 위해 사용된다.

당신이 특정 애플리케이션 컨텍스트에서 getResource()를 호출하고 명시된 위치경로가 특정 접두사를 가지지 않을때, 당신은 특정 애플리케이션 컨텍스트에 적절한 Resource타입으로 돌아갈것이다. 이를테면 당신이

ClassPathXmlApplicationContext를 요청할때가 그렇다.

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt);
```

반환되는 것은 ClassPathResource가 될것이다. 만약 같은 메소드가 FileSystemXmlApplicationContext인스턴스에 대해 수행된다면, 당신은 FileSystemResource를 돌려받게 될것이다. WebApplicationContext를 위해서, 당신은 ServletContextResource를 돌려받게 될것이다.

그 자체로, 당신은 특정 애플리케이션 컨텍스트를 위해 적절한 형태로 자원을 로드할수있다.

반면에, 당신은 애플리케이션 컨텍스트 타입에 상관없이 특별한 classpath: 접두사를 명시하여 ClassPathResource를 사용하도록 강요할지도 모른다.

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt);
```

표준 java.net.URL 접두사를 명시하여 UrlResource를 사용하도록 강요한다.

```
Resource template = ctx.getResource("file:/some/resource/path/myTemplate.txt);
```

```
Resource template = ctx.getResource("http://myhost.com/resource/path/myTemplate.txt);
```

다음의 테이블은 String을 Resource로 변환하기 위한 전략을 보여준다.

Table 4.1. 자원(Resource) 문자열

접두사	예제	상세설명
classpath:	classpath:com/myapp/config.xml	classpath로 부터 로드
file:	file:/data/config.xml	파일시스템으로부터 URL처럼 로드[a]
http:	http://myserver/logo.png	URL처럼 로드
(none)	/data/config.xml	기초적인 ApplicationContext에 달려있다.

[a]다음(Section 4.7.3, “FileSystemResource 경고(caveats)”)에서 다른 사항을 보라.

4.5. ResourceLoaderAware 인터페이스

ResourceLoaderAware 인터페이스는 ResourceLoader와 제공되는 객체를 위한 특별한 표시자 인터페이스이다.

```
public interface ResourceLoaderAware {
    void setResourceLoader(ResourceLoader resourceLoader);
}
```

클래스가 ResourceLoaderAware를 구현하고 애플리케이션 컨텍스트로 배치했을때(Spring관리 bean처럼), 이것은 애플리케이션 컨텍스트에 의해 ResourceLoaderAware로 인지된다. 애플리케이션 컨텍스트는 인자로 자체를 제공하는 setResourceLoader(ResourceLoader)를 호출할것이다(기억하라. Spring내 모든 애플리케이션 컨텍스트는 ResourceLoader 인터페이스를 구현한다는 것을 기억하라).

물론, ApplicationContext이 ResourceLoader이기 때문에, bean은 ApplicationContextAware를 구현할수 있고 자원을 로드하기 위해 직접 컨텍스트로 제공된것을 사용한다. 하지만 대개, 그 모든것이 필요하다면 특별한 ResourceLoader인터페이스를 사용하는게 더 좋다. 코드는 전체 Spring ApplicationContext가 아니고 유틸리티 인터페이스일수 있는 자원로딩 인터페이스에 커플링될것이다.

4.6. 프라퍼티로 Resource 셋팅하기

bean자체가 동적인 처리의 몇가지 정렬을 통해 자원 경로를 판단하고 제공할려고 한다면, 이것은 아마도 bean이 자원을 로드하기 위한 ResourceLoader인터페이스를 사용하도록 하는것이다. 몇몇 정렬된 템플릿을 로드하는 예제처럼, 어느 특정 템플릿이 사용자의 역활에 의존할 필요가 있다고 생각해보자. 만약 자원이 정적이라면, 이것은 ResourceLoader인터페이스의 사용을 완전히 제거하고 필요한 Resource프라퍼티를 드러낸다. 그리고 그것들은 삽입될것이다.

이러한 프라퍼티로 삽입하기 위해 해야하는 것은 모든 애플리케이션 컨텍스트를 등록하고 String경로를 Resource객체로 변환할수 있는 특별한 JavaBeans PropertyEditor를 사용하는 것이다. 그래서 myBean이 Resource타입의 템플릿 프라퍼티를 가진다면, 이것은 다음처럼 자원을 위한 간단한 문자열로 설정될수 있다.

```
bean id="myBean" class="...">
  <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

자원 경로가 어떤 접두사를 가지지 않아서, 애플리케이션 컨텍스트 자체는 ResourceLoader처럼 사용될것이고, 자원 자체는 컨텍스트 타입에 적절히 의존하는 ClassPathResource, FileSystemResource, ServletContextResource등등을 통해 로드된다.

만약 특별한 Resource타입이 사용되도록 해야할 필요가 있다면, 접두사가 사용될것이다. 다음의 두가지 예제는 ClassPathResurce와 UrlResource(파일시스템 파일에 접근하기 위해 사용된다.)가 사용되도록 하는 방법을 보여준다.

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">
```

```
<property name="template" value="file:/some/resource/path/myTemplate.txt"/>
```

4.7. 애플리케이션 컨텍스트와 Resource 경로들

4.7.1. 애플리케이션 컨텍스트 생성하기

애플리케이션 컨텍스트 생성자(특별한 애플리케이션 컨텍스트 타입을 위한)는 대개 컨텍스트의 정의를 만드는 XML파일과 같은 자원의 위치 경로로 문자열이나 문자열의 배열을 가진다.

그러한 위치가 접두사를 가지지 않을때, 특정 Resource타입은 그 경로로부터 빌드되고 관계되고 특정

애플리케이션 컨텍스트에 적절한 정의를 로드하기 위해 사용된다. 예를 들어, 당신이 다음처럼 ClassPathXmlApplicationContext를 생성한다면:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

ClassPathResource가 사용될것처럼 정의는 classpath로부터 로드될것이다. 하지만 당신이 다음처럼 FilleSystemXmlApplicationContext를 생성한다면:

```
ApplicationContext ctx =
    new FileSystemClassPathXmlApplicationContext("conf/appContext.xml");
```

현재 작업중인 디렉토리에 상대적으로 파일시스템 위치로부터 정의가 로드될것이다.

위치 경로에서 특정 classpath접두사나 표준 URL접두사를 사용하는 것은 정의를 로드하기 위해 생성된 Resource 디폴트 타입을 오버라이드할것이다. 그래서 이 FileSystemXmlApplicationContext는

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

는 classpath로부터 실질적으로 정의를 로드할것이다. 어쨌든, 이것은 여전히 FileSystemXmlApplicationContext이다. 만약 ResourceLoader처럼 순차적으로 사용했다면, 접두사를 가지지 않는 경로는 파일시스템 경로처럼 처리될것이다.

4.7.1.1. ClassPathXmlApplicationContext 인스턴스 생성하기

ClassPathXmlApplicationContext 는 편리한 초기화를 가능하게 해주는 많은 수의 생성자를 보여준다. 기본적인 생각은 XML파일 자체(경로 정보를 가질 필요없이)의 파일명을 포함하는 문자열 배열을 제공하고 물론 Class를 제공한다. ClassPathXmlApplicationContext는 제공되는 클래스로부터 경로 정보를 얻어낸다.

예제는 이것을 명백하게 만들어준다. 디렉토리 레이아웃을 생각해보자.

```
com/
foo/
services.xml
daos.xml
MessengerService.class
```

'services.xml' 과 'daos.xml' 내 정의된 bean을 구성하는 ClassPathXmlApplicationContext 인스턴스는 다음처럼 인스턴스화될것이다.

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

Please do consult the Javadocs for the ClassPathXmlApplicationContext class for details of the various constructors.

4.7.2. 애플리케이션 컨텍스트 생성자 자원 경로내 와일드카드

애플리케이션 컨텍스트 생성자내 자원경로값은 대상 자원에 대한 일대일 맵핑이 되는 간단한 경로이거나 대안으로 특별한 "classpath*:" 접두사나/또는 내부 Ant스타일의 정규표현식(Spring의 PathMatcher 유틸리티를 사용하여 일치하는)를 포함할것이다. 후자모두 효과적인 와일드카드이다.

이 기법을 사용하는 것은 컴포넌트 스타일의 애플리케이션을 조합할때이다. 모든 컴포넌트는 잘 알려진 위치경로에 컨텍스트 정의 조각을 '발행(publish)' 할수 있고 마지막 애플리케이션 컨텍스트가 classpath*:를 통해 접두사로 붙여진 같은 경로를 사용하여 생성될때, 모든 컴포넌트 조각은 자동적으로 올려질것이다.

와일드카드를 붙이는 것은 특별히 애플리케이션 컨텍스트 생성자내 자원경로를 사용하고(또는 PathMatcher 유틸리티 클래스를 직접 구조적으로 사용하는 때이다) 생성시 해석된다. Resource 타입 자체를 사용하는 것은 아무 상관이 없다. 실제 Resource를 하나의 자원에 대한 자원점으로 생성하기 위해 classpath*: 접두사를 사용하는 것은 불가능하다.

4.7.2.1. Ant-스타일의 패턴

경로 위치가 Ant스타일의 패턴을 포함할때, 이를테면

```
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
```

... 해석자는 좀더 복잡하지만 와일드카드를 해석하고자 절차(procedure)를 정의한다. 와일드카드가 아닌 부분을 경로로 지정하고 URL을 얻어서 Resource를 만든다. 이 URL이 "jar:" URL이나 컨테이너 특유의 변수(이를테면 웹로직에서는 "zip:", 웹스피어에서는 "wsjar" 등등)가 아니라면, java.io.File이 얻어지고 파일시스템을 찾아서 와일드카드를 해석하기 위해 사용된다. jar URL의 경우, 해석자는 java.net.JarURLConnection을 얻거나 수동으로 jar URL을 파싱하고 와일드카드를 해석하기 위해 jar파일의 내용을 조사한다.

4.7.2.1.1. 이식성의 함축(Implication)

명시된 경로가 file URL(기본 ResourceLoader가 파일시스템이기 때문에 명시적이거나 절대적으로)이라면, 와일드카드는 완벽하게 이식가능한 형태로 작업하는 것을 보장한다.

명시된 경로가 클래스패스 경로라면, 해석자는 Classloader.getResource() 호출을 통해 마지막 와일드카드가 아닌 경로 조각 URL을 얻어야만 한다. 이것은 경로 노드(끝의 파일명이 아닌)이기 때문에 정렬된 URL이 반환되는 것을 실제로 정의(ClassLoader javadoc에서)하지 않았다. 사실, 이것은 언제나 디렉토리를 표현하는 java.io.File이다. 여전히 여기에는 이 작업에 대한 이식성이 고려된다.

jar URL이 마지막의 와일드카드를 가지지 않는 구문을 얻는다면, 해석자는 이것으로부터 java.net.JarURLConnection를 얻을수 있거나 jar에 내용을 알수 있도록 jar URL을 수동으로 파싱하고 와일드카드를 해석할수 있어야만 한다. 이것은 대부분의 환경에서 작동할것이지만 실패할수도 있다. 우리는 특정 환경에서 jar로 부터 자원의 와일드카드를 해석할수 있는지 테스트하길 강력히 권한다.

4.7.2.2. classpath*: 접두사

XML기반 애플리케이션 컨텍스트를 생성할때, 위치 문자열은 특별한 classpath*: 접두사를 사용할것이다.

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

이 특별한 접두사는 주어진 이름으로 얻는 모든 classpath자원을 명시(내부적으로, 이것은 ClassLoader.getResources(...)호출을 통해 본질적으로 발생한다.)하고 마지막 애플리케이션 컨텍스트 정의를 형성하기 위해 병합된다.

Classpath*: 이식성

와일드카드 클래스패스는 참조하는 클래스로더의 getResources() 메소드에 의존한다. 대부분의 애플리케이션 서버는 자체적인 클래스로더 구현물을 제공한다. 행위는 jar파일을 다룰때 특히 다를것이다. classpath* 가 작동하는지를 테스트 하기 위한 간단한 테스트는 클래스패스내 jar파일로부터 파일을 로드하기 위해 클래스로더를 사용(getClass().getClassLoader().getResources("<someFileInsideTheJar>"))하는 것이다. 같은 이름을 가지지만 다른 위치에 있는 파일로 이것을 테스트해보라. 적절하지 않은 경우에 결과는 반환된다. 클래스로더 행위에 영향을 끼치는 셋팅을 위해 애플리케이션 서버 문서를 체크하라.

"classpath*:" 접두사는 예를 들어 "classpath*:META-INF/*-beans.xml"와 같이 위치경로의 나머지에서 PathMatcher 패턴으로 조합될수 있다. 이 경우, 해석전략은 지극히 단순하다. ClassLoader.getResources()호출이 클래스 로더 구조에서 자원을 일치시키는 모든것을 얻기 위해 마지막 와일드카드가 아닌 경로조각에서 사용된다.

4.7.2.3. 와일드카드에 관련된 다른 노트

Ant스타일로 조합되었을때 실제 대상 파일이 파일시스템에 존재하지 않더라도 "classpath*:"가 패턴이 시작되기 전 적어도 하나의 root디렉토리로 확실하게 작동할것이다. 이것은 "classpath*:*.xml"과 같은 패턴이 jar파일의 가장 상위에서가 아닌 확장된 디렉토리의 가장 상위에서 파일을 가져올 것을 의미한다. JDK ClassLoader.getResources() 메소드내 제한은 오직 전달된 공백 문자열(검색을 위한 잠재적인 가장 상위를 표시하는)을 위한 파일시스템 위치를 반환한다.

"classpath:"자원을 가진 Ant스타일의 패턴은 검색을 위한 가장 상위 패키지가 여러개의 클래스패스 위치에서 사용가능하다면 일치하는 자원을 찾기 위해 보장되지 않는다. 이것은 자원이 다음과 같기 때문이다.

```
com/mycompany/package1/service-context.xml
```

하나의 위치가 될것이다. 하지만 경로가 다음과 같을때

```
classpath:com/mycompany/**/service-context.xml
```

는 이것을 해석하도록 시도한다. 해석자는 getResource("com/mycompany")에 의해 반환되는 (첫번째) URL을 없앨것이다. 이 기본 패키지 노드가 여러개의 클래스로더 위치에 존재한다면, 실제 마지막 자원은 아래에 놓이게 되지 않을것이다. 그러므로, 가장 상위 패키지에 포함된 모든 클래스 패스 경로를 검색할 이런 경우 같은 Ant스타일 패턴을 가지고 "classpath*:"를 사용하라.

4.7.3. FileSystemResource 경고(caveats)

FileSystemApplicationContext에 첨부되지 않는 FileSystemResource는 절대경로와 상대경로를 처리할것이다(FileSystemApplicationContext는 실제로 ResourceLoader가 아니다.). 상대경로는 현재 작업중인 디렉토리에 대해 상대적이다. 반면에 절대경로는 파일시스템의 root에 대해 상대적이다.

이전버전과의 호환성의 이유로, 어쨌든 이것은 FileSystemApplicationContext이 ResourceLoader일때 변경한다. FileSystemApplicationContext는 슬래쉬(/)로 시작하거나 슬래쉬(/)를 사용하지 않거나 상대적인 모든 위치 경로를 처리하기 위해 모든 첨부된 FileSystemResources 간단히 고집한다. 실제로 이것은 다음과 동일하다.

```
ApplicationContext ctx =
    new FileSystemClassPathXmlApplicationContext("conf/context.xml");
```

```
ApplicationContext ctx =
    new FileSystemClassPathXmlApplicationContext("/conf/context.xml");
```

는 다음과 마찬가지이다(여기서 다른것은 하나는 상대경로이고 하나는 절대경로라는 것이다.).

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

비록 이것이 차이가 난다고 이해하더라도 하나는 상대적인것이고 하나는 절대적인것일뿐이다.

실제로, 절대적인 파일시스템 경로가 필요하더라도, 이것은 FileSystemResource/FileSystemXmlApplicationContext로 절대경로를 사용하는것을 잊어버리는것이 더 좋다. 그리고 file: URL접두사를 사용하여 UrlResource사용을 강요하라.

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:/some/resource/path/myTemplate.txt");
```

```
// force this FileSystemXmlApplicationContext to load it's definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:/conf/context.xml");
```

Chapter 5. 유효성체크(Validation), 데이터-바인딩, BeanWrapper, 와 PropertyEditors

5.1. 소개

비지니스 로직에서 유효성체크를 검토하는 것은 장,단점이 있고 Spring은 그것들중 하나도 배제하지 않는 유효성체크(그리고 데이터바인딩)을 위한 디자인을 제공한다. 특히 유효성체크는 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 validation 패키지를 구성하고, 처음에 사용되었던것 안에 구성되어있다. 그러나 MVC framework안에 제안되어있지는 않다.

BeanWrapper는 Spring 프레임워크에서 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 PropertyEditor를 사용한다. PropertyEditor의 개념은 JavaBeans 명세(specification)의 일부분이다. BeanWrapper와 DataBinder가 밀접하게 연관된 이후로, BeanWrapper일때, 또한 이번 장(chapter)내에 PropertyEditors 사용을 잘 설명한다.

5.2. Spring의 Validator인터페이스를 사용하여 유효성 체크하기

Spring의 기능인 Validator인터페이스는 당신이 객체의 유효성을 체크하기 위해 사용할수 있다. Validator인터페이스는 일관적이고 Errors객체라 불리는 것을 사용하여 작동한다. 반면에, 유효성을 체크하는 동안, validator는 Errors객체에 대한 유효성체크 실패를 보고할것이다.

작은 데이타 객체를 고려해보자.

```
public class Person {

  private String name;
  private int age;

  // the usual getters and setters...
}
```

org.springframework.validation.Validator를 사용하여 우리는 Person클래스를 위한 유효성체크 행위를 제공할것이다.

☒ supports(Class) - 이 Validator가 주어진 Class의 인스턴스를 체크하는가.?
☒ validate(Object, org.springframework.validation.Errors) - 주어진 객체에 대한 유효성 체크를 수행하고 유효성 에러발생시 주어진 Errors객체를 가진 등록자를 둔다.

특히 Spring이 제공하는 ValidationUtils를 알고 있을때 Validator를 구현하는 것은 상당히 일관적이다.

```
public class PersonValidator implements Validator {

    /**
    * This Validator validates just Person instances
    */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

당신이 볼수 있는것처럼, ValidationUtils클래스의 정적(static) rejectIfEmpty(..)메소드는 null이거나 공백일때 'name'프라퍼티를 제거하기 위해 사용된다. 이 예제가 제공하는 이외의 기능이 무엇인지 보기 위해 ValidationUtils를 위한 JavaDoc를 보라.

While it is certainly possible to implement a single Validator class to validate each of the nested objects in a rich object, it may be better to encapsulate the validation logic for each nested class of object in its own Validator implementation. A simple example of a 'rich' object would be a Customer that is composed of two String properties (a first and second name) and a complex Address object. Address objects may be used independant of Customer objects, and so a distinct AddressValidator has been implemented. If you want your CustomerValidator to reuse the logic contained within the AddressValidator class without recourse to copy-n-paste you can dependency-inject or instantiate an AddressValidator within your CustomerValidator, and use it like so:

```
public class CustomerValidator implements Validator {

   private final Validator addressValidator;

   public UserValidator(Validator addressValidator) {
      if (addressValidator == null) {
          throw new IllegalArgumentException("The supplied [Validator] is required and must not be null.");
      }
      if (!addressValidator.supports(Address.class)) {
          throw new IllegalArgumentException(
            "The supplied [Validator] must support the validation of [Address] instances.");
      }
      this.addressValidator = addressValidator;
   }

    /**
    * This Validator validates Customer instances, and any subclasses of Customer too
    */
   public boolean supports(Class clazz) {
      return Customer.class.isAssignableFrom(clazz);
   }

   public void validate(Object target, Errors errors) {
      ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");
```

```
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");
        Customer customer = (Customer) target;
        try {
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);
        } finally {
            errors.popNestedPath();
        }
    }
}
```

유효성체크 에러는 validator에 전달되는 Errors객체를 보고한다. Spring Web MVC의 경우 당신은 error메시지를 점검하기 위해 spring:bind태그를 사용할수 있다. 하지만 물론 당신은 errors객체 자체를 점검할수 있다. 제공하는 메소드는 매우 일관적이다. 좀더 많은 정보는 JavaDoc에서 찾을수 있다.

5.3. error메시지를 위한 코드를 분석하기

우리는 데이터바인딩(databinding) 과 유효성체크(validation)에 대해 말했다. 유효성체크 에러에 대응하는 메시지를 출력하는 것은 우리가 언급할 필요가 있는 마지막 사항이다. 위에서 보여준 예제에서, 우리는 name 그리고 age필드를 제거한다. 만약 MessageSource를 사용한다면, 우리는 error메시지를 출력할것이다. 필드를 제거(이 경우 'name'과 'age')할때 주어진 error코드를 사용할것이다. 당신이 rejectValue나 Errors인터페이스로부터 다른 reject메소드중 하나를 호출(직접적이거나 직접적이지 않거나 ValidationUtils클래스 예제를 사용하여)할때, 근본적인 구현물은 당신이 전달한 코드뿐 아니라 많은 수의 추가적인 error코드를 등록할것이다. 등록한 error코드는 사용한 MessageCodesResolver에 의해 판단된다. 디폴트로 DefaultMessageCodesResolver가 사용되어, 당신이 준 코드를 가진 메시지를 등록할 뿐 아니라 reject메소드에 전달하는 필드명을 포함하는 메시지의 예제를 위해 DefaultMessageCodesResolver가 사용된다. too.darn.old코드 이외에 rejectValue("age", "too.darn.old")를 사용하여 필드를 제거하는 경우에, Spring은 too.darn.old.age 과 too.darn.old.age.int를 등록할것이다(첫번째는 필드명을 포함하고 두번째는 필드의 타입을 포함할것이다.).

MessageCodesResolver와 디폴트 전략에 대한 좀더 많은 정보가 각각 MessageCodesResolver와 DefaultMessageCodesResolver를 위한 JavaDoc에서 찾을수 있다.

5.4. Bean 조작(manipulation)과 BeanWrapper

org.springframework.beans 패키지는 Sun에서 제공하는 JavaBeans 표준을 고수한다(adhere). JavaBean은 인수가 없는 디폴트 구성자로된 간단한 클래스이고, bingoMadness이란 이름의 속성(property)은 setter setBingoMadness(..) 과 getter getBingoMadness()을 가진 네이밍 규칙을 따른다. JavaBeans과 명세서에 관한 더 많은 정보를 위해서, Sun의 웹사이트(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 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.)

5.4.1. 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 5.1. 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

```
BeanWrapper company = BeanWrapperImpl(new Company());
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue value = new PropertyValue("name", "Some Company Inc.");
company.setPropertyValue(value);

// ok, let's create the director and tie it to the company:
BeanWrapper jim = BeanWrapperImpl(new Employee());
jim.setPropertyValue("name", "Jim Stravinsky");
company.setPropertyValue("managingDirector", jim.getWrappedInstance());

// retrieving the salary of the managingDirector through the company
Float salary = (Float) company.getPropertyValue("managingDirector.salary");
```

5.4.2. 내장 PropertyEditors 구현물

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.) 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 객체의 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 5.2. 내장 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 파싱하라. 클래스를 찾을수 없을때, IllegalArgumentException을 던진다. BeanWrapperImpl에 의해 디폴트로 등록된다.
CustomBooleanEditor	Boolean속성들을 위해 커스터마이즈할수 있는 속성 에디터(editor). BeanWrapperImpl에 의해 디폴트로 등록된다.그러나, 사용자정의 편집기에 의해 등록된 custom 인스턴스에를 오버라이드할 수 있다.
CustomCollectionEditor	Collection 형태의 목표가 주어졌을때 소스 Collection으로 전환하는, Collections을 위한 설정 editor.
CustomDateEditor	사용자정의 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. BeanWrapperImpl에 의해 디폴트로 등록. 그러나, 사용자정의 편집기로 등록된 사용자정의 인스턴스를 오버라이드할 수 있다.
FileEditor	Strings에서 java.io.File-객체들을 결정 가능. BeanWrapperImpl에 의해 디폴트로 등록된다.
InputStreamEditor	단방향 설정 에디터는 텍스트 문자열을 가질 수 있고, InputStream을 생성할 수 있다. (ResourceEditor와 Resource의 조정자를 통해서). 그래서 InputStream 프라퍼티들은 Strings으로 직접 셋팅될수도 있다. 디폴트 사용은 InputStream을 닫지 않는다는 것을 주의하라!. BeanWrapperImpl에 의해 디폴트로 등록된다.
LocaleEditor	Strings에서 Locale-객체들을 결정 가능하고 반대로 Locale의 toString() 메소드도 같은 기능을 제공한다(String 포맷은 [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.
StringTrimmerEditor	Property 에디터는 Strings을 정리 정돈한다. 선택적으로 널(null) 값으로 되어있는 변형된 비어있는 문자열로 인정한다. 디폴트에의해 등록되어지지 않는다. 사용자는 필요할때에 등록해야한다.

Class	설명
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 Font, Color, 모든 원시 형태들의 PropertyEditors를 포함하고 있는 sun.bean.editor를 나타낸다. 만약 다루는 클개스가 같은 패키지 안에 있고, 클래스가 같은 이름을 가지고 있고, 'Editor'가 추가되어있다면, 또한 표준 JavaBeans 기초구조는 (등록하는 절차 없이) 자동적으로 PropertyEditor를 발견한다는 것을 주의하라. 예를 들어 Foo 타입의 프라퍼티를 위한 PropertyEditor로 인식되고 사용되기 위한 FooEditor 클래스를 위해 충분한 다음의 클래스와 패키지 구조를 가질수 있다.

```
com
  chank
    pop
      Foo
      FooEditor    // the PropertyEditor for the Foo class
```

당신은 여기서 표준적인 BeanInfo JavaBean기법(여기서 놀랍지 않은 방법으로 언급된)을 사용할수 있다. 관련 클래스의 프라퍼티를 가진 하나 또는 그 이상의 PropertyEditor 인스턴스를 명시적으로 등록하기 위한 BeanInfo기법의 사용 예제를 아래에서 보라.

```
com
  chank
    pop
      Foo
      FooBeanInfo    // the BeanInfo for the Foo class
```

그리고 이것은 참조된 FooBeanInfo 클래스를 위한 Java소스코드이다. 이것은 Foo 클래스의 age 프라퍼티를 가진 CustomNumberEditor에 관계될것이다.

```
    public class FooBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Foo.class) {
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

5.4.2.1. 추가적인 사용자정의 PropertyEditors 등록하기

string값으로 bean프라퍼티를 셋팅할때, Spring IoC컨테이너는 이러한 String을 프라퍼티의 복잡한 타입으로 변환하기 위해 표준 JavaBeans PropertyEditors를 사용한다. Spring은 (예를 들어, string으로 표현되는 classname을 실제 Class객체로 변환하기 위해)사용자정의 PropertyEditors의 수를 미리 등록한다.

추가적으로, Java의 표준 JavaBeans PropertyEditor는 적절하게 명명되고 자동적으로 발견하는 지원을 제공하는 클래스와 같은 패키지내 위치한 클래스를 위한 PropertyEditor를 허용하는 기법을 룩업한다.

다른 사용자정의 PropertyEditors를 등록할 필요가 있다면, 사용가능한 다양한 기법이 있다. 대개 편리하거나 추천되지 않는 대부분의 수동 접근법은 BeanFactory 참조를 가지는 것을 가정하는 ConfigurableBeanFactory 인터페이스의 registerCustomEditor()메소드를 간단히 사용한다. 좀더 편리한 기법은 CustomEditorConfigurer라고 불리는 특별한 bean factory 후-처리자를 사용한다. 비록 bean factory 후-처리자는 BeanFactory 구현물과 반-수동으로 사용될수 있다. 이것은 내포된 프라퍼티 셋업을 가진다. 그래서 다른 bean에 유사한 형태로 배치되고 자동으로 감지되고 적용되는 ApplicationContext와 사용되는 것이 강력하게 추천된다.

모든 bean factory와 애플리케이션 컨텍스트는 프라퍼티 변환을 다루기 위해 BeanWrapper라고 불리는 몇가지의 사용을 통해 많은 수의 내장된 프라퍼티 편집기를 자동으로 사용한다. BeanWrapper가 등록한 표준 프라퍼티 편집기는 다음 장에서 목록화된다. 추가적으로, ApplicationContexts는 애플리케이션 컨텍스트 타입을 명시하기 위해 적절한 방법으로 자원 룩업을 다루는 추가적인 수의 편집기를 오버라이드하거나 추가한다.

표준 JavaBeans PropertyEditor 인스턴스는 string으로 표현되는 프라퍼티 값을 프라퍼티의 복잡한 타입으로 변환하기 위해 사용된다. bean factory 후-처리자인 CustomEditorConfigurer는 ApplicationContext를 위한 추가적인 PropertyEditor 인스턴스를 위해 편리하게 추가된 지원을 위해 사용된다.

사용자 클래스인 ExoticType를 보자. 다른 클래스 DependsOnExoticType는 프라퍼티로 셋팅되는 ExoticType가 필요하다.

```
package example;

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;
    }
}
```

When things are properly set up, we want to be able to assign the type property as a string, which a PropertyEditor will behind the scenes convert into a real ExoticType object:

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

The PropertyEditor implementation could look similar to this:

```
// converts string representation to ExoticType object
package example;

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 ApplicationContext로 새로운 PropertyEditor를 등록하기 위해 CustomEditorConfigurer를 사용하고 필요할때 사용할수 있을것이다.

```
<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"/>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

Chapter 6. Spring을 이용한 Aspect 지향 프로그래밍

6.1. 소개

Aspect-지향 프로그래밍(AOP)는 프로그램 구조에 대한 다른 방식의 생각을 제공함으로써 객체지향 프로그래밍(OOP)를 보완한다. 클래스들에 추가적으로, AOP는 aspects를 제공한다. aspects는 다중 타입과 객체에 영향을 주는 트랜잭션 관리처럼 관심사의 모듈화를 가능하게 한다(이러한 관심사는 종종 crosscutting 관심사라는 용어로 사용된다.)

Spring의 핵심이 되는 컴포넌트중 하나는 AOP 프레임워크이다. Spring IoC컨테이너가 AOP에 의존하지 않는 동안, 당신이 원하지 않는다면 AOP를 사용할 필요가 없다는 것을 의미한다. AOP는 미들웨어 솔루션의 기능을 제공하기 위해 Spring IoC를 보완할것이다.

> Spring 2.0 AOP
>
> Spring 2.0은 스키마-기반의 접근법이나 @AspectJ 어노테이션 스타일을 사용하여 사용자정의 aspect를 작성하는 간단하고 강력한 방법을 소개한다. 새로운 애플리케이션을 위해, 우리는 Java 5로 작성된 애플리케이션을 위해 @AspectJ 스타일을 사용하는 것을 추천하고 다른 경우 스키마-기반의 스타일을 추천한다. 이러한 스타일 모두 직조(weaving)를 위해 Spring AOP를 사용하는 동안 완전한 타입의 advice와 AspectJ pointcut언어의 사용을 제공한다.
>
> Spring 2.0 스키마와 @AspectJ 기반의 AOP지원은 이 장에서 언급된다. Spring 2.0은 Spring 1.2와의 이전 버전에 대한 호환성과 다음장에서 언급된 Spring 1.2 API에 의해 제공되는 좀더 낮은 레벨의 AOP지원을 유지한다.

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.

이 장의 첫번째 AOP 개념의 소개는 당신이 사용하기 위해 선택하는 aspect선언의 스타일이 무엇이든 읽기를 원할것이다. 이 장의 나머지는 Spring 2.0 AOP지원에 중점을 둘것이다. Spring 1.2 스타일의 AOP의 개요를 위해 다음장을 보라.
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.

6.1.1. 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: 다중 객체에 영향을 주는 concern의 모듈화. 트랜잭션 관리는 J2EE애플리케이션의 crosscutting concern의 좋은 예제이다. Spring AOP에서, aspect는 정규 클래스나 @Aspect 어노테이션(@Aspect 스타일)으로 주석처리된 정규 클래스를 사용하여 구현된다.

☒ Joinpoint: 메소드 수행이나 예외를 다루는 것과 같은 프로그램의 수행기간 동안의 point. Spring AOP에서, joinpoint는 언제나 메소드 호출을 나타낸다. Join point정보는 org.aspectj.lang.JoinPoint 타입의 파라미터를 선언하여 advice에서 사용가능하다.

☒ Advice: 특정 joinpoint에서 aspect에 의해 획득되는 액션. advice의 다른 타입은 "around," "before" 과 "after" advice를 포함한다. advice 타입은 밑에서 언급된다. Spring을 포함한 많은 AOP프레임워크는 인터셉터로 advice를 모델화하고 joinpoint "주위(around)"로 인터셉터의 묶음(chain)을 유지한다.

☒ Pointcut: join point에 일치하는 속성. advice는 pointcut표현과 관련있고 pointcut에 일치하는 join point에서 수행된다(예를 들어, 어떤 이름을 가진 메소드의 수행). pointcut표현에 의해 일치하는 join point의 개념은 AOP에 집중된다. Spring은 디폴트로 AspectJ pointcut언어를 사용한다.

☒ Introduction: (중간(inter)-타입 선언으로 알려진) 타입에서 추가적인 메소드나 필드의 선언. Spring은 프록시화된 객체를 위해 새로운 인터페이스(와 관련 구현물)를 소개한다. 예를 들어, 간단한 캐시를 위해, IsModified 인터페이스를 구현하는 bean을 만들기 위한 소개(introduction)를 사용한다.

☒ 대상 객체: 객체는 하나이상의 aspect에 의해 충고된다. 또한 advised 객체를 참조한다. Spring AOP가 런타임 프록시를 사용하여 구현되기 때문에, 이 객체는 언제나 프록시화된 객체가 될것이다.

☒ AOP 프록시: aspect 규칙(advise메소드수행과 기타등등)을 구현하기 위하여 AOP프레임워크에 의해 생성되는 객체. Spring에서. AOP프록시는 JDK 동적 프록시나 CGLIB 프록시가 될것이다.
노트 : 프록시 생성은 스키마-기반과 Spring 2.0에서 소개된 aspect선언의 @AspectJ스타일의 사용자에게 명백하다.

☒ Weaving: 다른 애플리케이션 타입이나 advised객체를 생성하기 위한 객체를 가지는 aspect 연결. 이것은 컴파일 시점(예를 들어, AspectJ 컴파일러를 사용하여), 로그시점 또는 런타임에 수행될수 있다. 다른 Java AOP프레임워크처럼 Spring은 런타임시 직조(weaving)를 수행한다.

advice 타입

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

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

☒ After throwing advice: 메소드가 예외를 던져서 빠져나갈때 수행되는 advice

☒ After (finally) advice: join point를 빠져나가는(정상적이거나 예외적인 반환) 방법에 상관없이 수행되는 advice.

☒ Around advice: 메소드 호출과 같은 joinpoint주위(surround)의 advice. 이것은 가장 강력한 종류의

advice이다. Around advice는 메소드 호출 전후에 사용자 정의 행위를 수행할수 있다. 그것들은 joinpoint를 처리하거나 자기 자신의 반환값을 반환함으로써 짧게 수행하거나 예외를 던지는 것인지에 대해 책임을 진다.

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 JoinPoint의 proceed()메소드를 호출할 필요가 없고 나아가 그것을 호출하는것을 실패할수도 있다.

Spring 2.0에서, 모든 advice파라미터는 정적으로 타입화된다. 그래서 객체 배열보다는 적절한 타입(예를 들면, 메소드 수행의 반환값의 타입)의 advice파라미터를 가지고 작업한다.

pointcut에 의해 일치하는 join point의 개념은 오직 인터셉션만을 제공하는 예전 기술과 구분되는 AOP의 핵심이다. pointcut은 OO구조의 단독으로 대상화되도록 해준다. 예를 들어, 선언적인 트랜잭션 관리를 제공하는 around advice는 다중 객체에 걸쳐있는 메소드에 적용될수 있다(서비스 레이어내 모든 비니지스 작동과 같은).

6.1.2. 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 수행 join point만을 지원한다(Spring bean의 메소드 수행을 충고하는). 필드 인터셉션은 구현되지 않았지만 핵심 Spring AOP API에 영향없이 추가될수 있도록 필드 인터셉션을 지원한다. advice필드 접근과 join point수정이 필요하다면, AspectJ와 같은 다른 언어를 고려하라.

Spring AOP접근법은 대부분의 다른 AOP프레임워크와 다르다. 목적은 완벽한 AOP구현물을 제공하는 것이 아니다(비록 Spring AOP는 상당히 가능함에도). 전사적인 애플리케이션에서 공통의 문제를 푸는것을 돕기 위해 AOP구현물과 Spring IoC간에 닫힌(close) 통합을 제공한다.

게다가, 예를 들어, Spring의 AOP기능은 Spring IoC컨테이너와 결합되어 사용된다. Aspect는 일반적인 bean정의 문법을 사용하여 설정된다(강력한 "autoproxing" 기능을 허용하기는 하나.). 이것은 다른 AOP구현물과 결정적으로 다른점이다. 매우 잘 구성된 객체을 충고(advise)하는 것처럼 Spring AOP로 쉽거나 효과적으로 할수 없는 것이 있다. AspectJ는 이런 경우 가장 훌륭한 선택이다. 어쨌든, Spring AOP가 AOP를 다루기 쉬운 J2EE애플리케이션에서 공통의 문제를 위해 훌륭한 해결법을 제공한다고 경험했다.

Spring AOP는 포괄적인 AOP솔루션을 제공하기 위해 AspectJ와 경쟁하지 않을것이다. 우리는 Spring과 같은 프록시-기반의 프레임워크와 AspectJ와 같은 성숙한 프레임워크가 가치있고 경쟁보다는 상호보완한다고 믿는다. Spring 2.0은 일관적인 Spring 기반의 애플리케이션 아키텍처에서 AOP의 모든 사용이 가능하도록 AspectJ로 Spring AOP와 IoC를 균일하게 통합한다. 이 통합은 Spring AOP API나 AOP제휴 API에 영향을 주지 않는다. Spring AOP는 이전 버전과의 호환성을 유지한다. Spring AOP API에 대해서는 다음 장을 보라.

6.1.3. 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, 그리고 당신이 왜 이렇게 하는것을 원하는지 설명할것이다.
Spring 2.0이상에서, Spring은 아마도 전체적으로 생성되는 클래스를 포함해서 AOP프록시의 추가적인 타입을 제공할것이다. 이것은 프로그래밍 모델에는 영향을 끼치지 않을것이다.

6.2. @AspectJ 지원

"@AspectJ" 는 Java 5 어노테이션으로 주석처리된 정규 Java클래스로 aspect를 선언하는 스타일을 언급한다. @AspectJ스타일은 AspectJ 5 릴리즈의 일부처럼 AspectJ 프로젝트에 의해 소개되었다. Spring 2.0은 pointcut 파싱과 일치를 위해 AspectJ에 의해 제공되는 라이브러리를 사용하여 AspectJ 5처럼 같은 어노테이션을 해석한다. AOP런타임은 여전히 순수한 Spring AOP이고 AspectJ 컴파일러나 직조자(weaver)에 대한 의존성이 없다.

AspectJ 컴파일러와 직조자(weaver)를 사용하는 것은 AspectJ언어를 완전하게 사용가능하도록 해준다. 그리고 이것은 Section 6.8, “Using AspectJ with Spring applications” 에서 언급된다.

6.2.1. @AspectJ 지원 가능하게 하기

Spring 설정에서 @AspectJ aspect를 사용하기 위해, 당신은 @AspectJ aspect와 aspect에 의해 충고되는지에 대한 유무에 기초한 autoproxying bean에 기초하도록 Spring AOP를 설정할 필요가 있다. autoproxying을 사용하여 우리는 bean이 하나 이상의 aspect에 의해 충고(advised)되는지 Spring이 판단하도록 한다. 이것은 메소드 호출을 가로채고 advice가 필요할때 수행되는지를 확인하는 bean을 위한 프록시를 자동으로 생성할것이다.

@AspectJ 지원은 Spring설정내부에 다음의 요소를 포함하는것으로 가능하게 된다.

```
<aop:aspectj-autoproxy/>
```

이것은 당신이 Appendix A, XML 스키마-기반 설정에서 언급된것처럼 스키마 지원을 사용한다는 것으로 추정한다. aop명명공간내 태그를 import하는 방법을 위해 Section A.2.6, “aop 스키마” 를 보라.

DTD를 사용한다면, 애플리케이션 컨텍스트에 다음의 정의를 추가하여 @AspectJ지원을 가능하게 할수 있다.

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator" />
```

당신의 애플리케이션의 클래스패스에 두개의 AspectJ라이브러리(aspectjweaver.jar 와 aspectjrt.jar)를 둘 필요가 있을것이다. 이러한 라이브러리는 AspectJ(1.5.1이나 그 이후버전이 필요)의 lib 디렉토리나

Spring-with-dependencies 배포판의 lib/aspectj 디렉토리에서 볼수 있다.

6.2.2. aspect 선언하기

@AspectJ 지원을 가능하게 하여, @AspectJ aspect(@Aspect 어노테이션을 가지는)인 클래스로 당신의 애플리케이션 컨텍스트내 정의된 bean은 Spring에 의해 자동적으로 감지되고 Spring AOP를 설정하기 위해 사용될것이다. 다음은 그다지 유용하지 않은 aspect를 위해 요구되는 최소한의 정의를 보여주는 예제이다.

애플리케이션 컨텍스트내 정규 bean정의는 @Aspect 어노테이션을 가지는 bean클래스를 가리킨다.

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
   <!-- configure properties of aspect here as normal -->
</bean>
```

그리고 NotVeryUsefulAspect 클래스 정의는 org.aspectj.lang.annotation.Aspect 어노테이션으로 주석처리된다.

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

aspect(@Aspect로 주석처리된 클래스)는 다른 클래스처럼 메소드와 필드를 가진다. 그것들은 또한 pointcut, advice, 그리고 introduction(내부타입)선언을 가진다.

6.2.3. pointcut 선언하기

pointcut이 join point를 판단하고 advice가 수행될때 제어를 가능하게 해주는 것을 것을 상기해보자. Spring AOP는 Spring bean을 위한 메소드 수행 join point만을 지원한다., 그래서 당신은 pointcut을 Spring bean의 메소드 수행을 일치시키는 것으로 생각할수 있다. pointcut선언은 두개의 부분을 가진다. 시그너처는 이름(name)과 어떤 파라미터를 포함하고 부수적인 pointcut표현은 메소드 수행을 실제로 판단한다. @AspectJ 어노테이션 스타일의 AOP에서, pointcut시그너처는 정규 메소드 정의에 의해 제공되고 pointcut표현은 @Pointcut 어노테이션을 사용하여 표시된다(pointcut시그너처 처럼 제공하는 메소드는 void 반환타입을 가져야만 한다).

예제는 pointcut시그너처와 pointcut표현간의 구분을 가지도록 도와줄것이다. 다음 예제는 'transfer'라는 이름의 메소드의 수행과 일치할 'anyOldTransfer' 라는 이름의 pointcut을 정의한다.

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

@Pointcut 어노테이션 값의 형태인 pointcut표현은 정규 AspectJ 5 pointcut표현이다. AspectJ의 pointcut언어의 완전한 언급을 위해서, AspectJ 프로그래밍 가이드를 보라(그리고 Java 5 기반의 확장을 위해서, AspectJ 5 Developers Notebook를 보라) 또는 Colyer가 쓴 "Eclipse AspectJ"나 Ramnivas Laddad가 쓴 "AspectJ in Action"중 하나를 보라.

6.2.3.1. 지원되는 Pointcut 지시자(Designators)

Spring AOP는 pointcut표현을 사용하기 위한 다음의 AspectJ pointcut지시자를 지원한다.

> 다른 pointcut 타입
>
> 완전한 AspectJ pointcut언어는 Spring에서 지원되지 않는 추가적인 pointcut지시자를 지원한다. 이것들은 : call, initialization, preinitialization, staticinitialization, get, set, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this, 와 @withincode이다. Spring AOP에 의해 해석되는 pointcut표현내 pointcut지시자를 사용하는 것은 IllegalArgumentException가 던져지는 결과를 만들것이다.
>
> Spring AOP에 의해 지원되는 pointcut지시자는 좀더 많은 AspectJ pointcut지시자와 "bean"과 같은 Spring특유의 지시자를 잠재적으로 지원 지원하기 위해 차후 릴리즈에서 확장될것이다.

☒ execution - 메소드 수행 join point를 일치시키기 위해, 이것은 Spring AOP와 작동할때 당신이 사용할 기본적인 pointcut 지시자이다.

☒ within - 어떤 타입내 join point에 일치하는것을 제한(메소드의 수행은 Spring AOP를 사용할때 일치되는 타입내에서 선언된다.)

☒ this - bean참조(Spring AOP프록시)가 주어진 타입의 인스턴스인 join point에 일치하는것을 제한(Spring AOP를 사용할때 메소드의 수행)

☒ target - 대상 객체(프록시된 애플리케이션 객체)가 주어진 타입의 인스턴스인 join point(Spring AOP를 사용할때 메소드의 수행)에 일치하는것을 제한.

☒ args - 인자가 주어진 타입의 인스턴스인 join point(Spring AOP를 사용할때 메소드의 수행)에 일치하는것을 제한.

☒ @target - 수행중인 객체의 클래스가 주어진 타입의 어노테이션을 가지는 join point(Spring AOP를 사용할때 메소드의 수행)에 일치하는것을 제한.

☒ @args - 전달된 실제 인자의 런타임 타입이 주어진 타입의 어노테이션을 가지는 join point(Spring AOP를 사용할때 메소드의 수행)에 일치하는것을 제한.

☒ @within - 주어진 어노테이션을 가지는 타입내 join point를 일치하는것을 제한(Spring AOP를 사용할때 주어진 어노테이션을 가진 타입으로 선언된 메소드의 수행)

☒ @annotation - join point(Spring AOP에서 수행되는 메소드)의 대상(subject)이 주어진 어노테이션을 가지는 join point에 일치하는것을 제한

Spring AOP는 오직 메소드 수행 join point에 일치하는 것을 제한하기 때문에, 위 pointcut지시자의 언급은 당신이 AspectJ 프로그래밍 가이드에서 찾는것보다 더 폭이 좁은 정의를 준다. 추가적으로, AspectJ 자체는 타입에 기반하는 구문을 가지고 수행시 'this' 와 'target' 두 join point는 같은 객체(메소드를 수행하는 객체)를 참조한다. Spring AOP는 프록시에 기반한 시스템이고 프록시 객체 자체('this' 로 바운드되는)와 프록시 배후의 대상 객체('target'로 바운드되는)를 구별한다.

6.2.3.2. pointcut 표현 조합하기

pointcut 표현은 '&&', '||' 그리고 '!' 를 사용하여 조합될수 있다. 이것은 이름(name)으로 pointcut표현을 참조하는 것이 가능하다. 다음 예제는 3개의 pointcut표현 즉 anyPublicOperation (메소드 수행 join point가 public메소드의 수행을 표현한다면 일치하는); inTrading (trading 모듈에서 메소드가

수행된다면 일치하는), 그리고 tradingOperation (메소드 수행이 trading 모듈내 public 메소드를 표현한다면 일치하는)을 보여준다.

```
    @Pointcut("execution(public * *(..))")
    private void anyPublicOperation() {}

    @Pointcut("within(com.xyz.someapp.trading..*")
    private void inTrading() {}

    @Pointcut("anyPublicOperation() && inTrading()")
    private void tradingOperation() {}
```

이것은 위에서 보여진것처럼 좀더 작은 명명된 컴포넌트가 없는 좀더 복잡한 pointcut표현을 빌드하기 위한 가장 좋은 예제이다. 이름(name)으로 pointcut를 참조할때, 대개의 Java 가시성(visibility) 규칙을 적용한다(당신은 같은 타입내 private pointcut, 구조내에서 protected pointcut, public pointcut등을 볼수 있다). 가시성(Visibility)은 pointcut 일치(matching)에 영향을 끼치지 않는다..

6.2.3.3. 공통 pointcut 정의 공유하기

기업용 애플리케이션으로 작업할때, 당신은 애플리케이션의 모듈과 다양한 aspect의 특정 작업 세트를 참조하길 원한다. 우리는 이러한 목적을 위해 공통적인 pointcut표현을 획득하는 "SystemArchitecture" aspect를 정의하는 것을 추천한다. 대개의 aspect는 다음과 같을것이다.

```
package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

  /**
   * A join point is in the web layer if the method is defined
   * in a type in the com.xyz.someapp.web package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.web..*)")
  public void inWebLayer() {}

  /**
   * A join point is in the service layer if the method is defined
   * in a type in the com.xyz.someapp.service package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.service..*)")
  public void inServiceLayer() {}

  /**
   * A join point is in the data access layer if the method is defined
   * in a type in the com.xyz.someapp.dao package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.dao..*)")
  public void inDataAccessLayer() {}

  /**
   * A business service is the execution of any method defined on a service
   * interface. This definition assumes that interfaces are placed in the
   * "service" package, and that implementation types are in sub-packages.
   *
   * If you group service interfaces by functional area (for example,
   * in packages com.xyz.someapp.abc.service and com.xyz.def.service) then
```

```
    * the pointcut expression "execution(* com.xyz.someapp..service.*.*(..))"
    * could be used instead.
    */
   @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
   public void businessService() {}

   /**
    * A data access operation is the execution of any method defined on a
    * dao interface. This definition assumes that interfaces are placed in the
    * "dao" package, and that implementation types are in sub-packages.
    */
   @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
   public void dataAccessOperation() {}

}
```

aspect에 정의된 pointcut은 당신이 pointcut표현이 필요한 어디에서도 참조될수 있다. 예를 들어, 서비스 레이어를 트랜잭션 성질을 가지도록 만들기 위해서, 당신은 다음처럼 작성할수 있다.

```
<aop:config>
  <aop:advisor
      pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
      advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
<tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

<aop:config> 와 <aop:advisor> 태그는 Section 6.3, “Schema-based AOP support” 에서 언급된다. 트랜잭션 태그는 Chapter 9, 트랜잭션 관리에서 언급된다.

6.2.3.4. 예제

Spring AOP 사용자는 수행(execution) pointcut 지시자를 사용하길 선호한다. 수행 표현의 형태는 :

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
          throws-pattern?)
```

모든 부분은 반환하는 타입 패턴(returning type pattern - 위 조각내 ret-타입 패턴), 이름 패턴(name pattern), 그리고 파라미터 패턴(parameters pattern)은 선택적으로 제외한다. 반환하는 타입 패턴은 메소드의 반환 타입이 일치되는 join point를 위한 순서에 따라야만 한다. 가장 빈번히 당신은 어떤 반환타입과 일치하는 반환하는 타입 패턴으로 *를 사용할것이다. 전체 경로를 가지는 타입명은 메소드가 주어진 타입을 반환할때만 일치할것이다. 이름 패턴은 메소드명과 일치한다. 당신은 이름 패턴의 모든것이나 일부로 * 와일드카드를 사용할수 있다. 파라미터 패턴은 다소 좀더 복잡하다. ()는 파라미터를 가지지 않는 메소드와 일치한다. 반면에 (..)는 여러개의 파라미터와 일치한다. 패턴 (*)는 어떤 타입의 하나의 파라미터를 가지는 메소드와 일치한다. (*,String)는 두개의 파라미터를 가지는 메소드와 일치한다. 첫번째는 어떤 타입이든 될수 있고, 두번째는 String이 되어야만 한다. 좀더 많은 정보를 위해 AspectJ 프로그래밍 가이드의 Language Semantics 부분을 보라.

공통 pointcut 표현의 몇가지 예제는 아래에 있다.

☒ public 메소드의 수행:

```
execution(public * *(..))
```

☒ "set"로 시작하는 이름을 가지는 메소드의 수행:

```
execution(* set*(..))
```

☒ AccountService 인터페이스가 정의한 메소드의 수행:

```
execution(* com.xyz.service.AccountService.*(..))
```

☒ 서비스 패키지내 정의된 메소드의 수행:

```
execution(* com.xyz.service.*.*(..))
```

☒ 서비스 패키지나 하위 패키지내 정의된 메소드의 수행:

```
execution(* com.xyz.service..*.*(..))
```

☒ 서비스 패키지내부의 join point(Spring AOP내 메소드 수행):

```
within(com.xyz.service.*)
```

☒ 서비스 패키지나 하위 패키지내부의 join point(Spring AOP내 메소드 수행:

```
within(com.xyz.service..*)
```

☒ 프록시가 AccountService 인터페이스를 구현하는 join point(Spring AOP내 메소드 수행):

```
this(com.xyz.service.AccountService)
```

'this' 는 바인딩 폼에서 좀더 공통적으로 사용된다. advice 몸체에서 사용가능한 프록시 객체를 만드는 방법을 위해 advice의 다음 부분을 보라.

☒ 대상 객체가 AccountService 인터페이스를 구현하는 join point(Spring AOP내 메소드 수행):

```
target(com.xyz.service.AccountService)
```

'target' 은 바인딩 폼에서 좀더 공통적으로 사용된다. advice몸체에서 사용가능한 대상 객체를 만드는 방법을 위해 advice의 다음 부분을 보라.

☒ 하나의 파라미터를 가지는 join point, 런타임시 전달되는 인자는 Serializable이다:

```
args(java.io.Serializable)
```

'args' 는 바인딩 폼에서 좀더 공통적으로 사용된다. advice몸체에서 사용가능한 메소드 인자를 만드는 방법을 위해 advice의 다음 부분을 보라.

이 예제에 주어진 pointcut은 execution(* *(java.io.Serializable))과 다르다: args설명(version)은 런타임시 전달된 인자가 Serializable라면 일치한다. execution설명(version)은 메소드 시그너처가 Serializable 타입의 하나의 파라미터를 선언한다면 일치한다.

☒ 대상 객체가 @Transactional 어노테이션을 가지는 join point(Spring AOP내 메소드 수행):

```
@target(org.springframework.transaction.annotation.Transactional)
```

'@target' 은 바인딩 폼에서 사용될수 있다. advice몸체에서 사용가능한 어노테이션 객체를 만드는 방법을 위해 advice의 다음 부분을 보라.

☒ 대상 객체의 선언된 타입이 @Transactional 어노테이션을 가지는 join point(Spring AOP내 메소드 수행):

```
@within(org.springframework.transaction.annotation.Transactional)
```

'@within' 은 바인딩 폼에서 사용될수 있다. advice몸체에서 사용가능한 어노테이션 객체를 생성하는 방법을 위해 advice의 다음 부분을 보라.

☒ 수행(executing) 메소드가 @Transactional 어노테이션을 가지는 join point(Spring AOP내 메소드 수행):

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

'@annotation' 은 바인딩 폼에서 사용될수 있다. advice몸체에서 사용가능한 어노테이션 객체를 생성하는 방법을 위해 advice의 다음 부분을 보라.

☒ 하나의 파라미터를 가지고 전달된 인자의 런타임 타입이 @Classified 어노테이션을 가지는 join point(Spring AOP내 메소드 수행):

```
@args(com.xyz.security.Classified)
```

'@args' 는 바인딩 폼에서 사용될수 있다. advice몸체에서 사용가능한 어노테이션 객체를 생성하는 방법을 위해 advice의 다음 부분을 보라.

6.2.4. advice 선언하기

advice는 pointcut표현과 관련되고 pointcut이 일치하는 before, after 또는 around 메소드 수행을 작동시킨다. pointcut 표현은 명명된 pointcut에 대한 간단한 참조나 선언된 pointcut표현이 될것이다.

6.2.4.1. Before advice

Before advice는 @Before 어노테이션을 사용하여 aspect내 선언된다.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

적절한 pointcut표현을 사용한다면, 우리는 위 예제를 다시 작성할수 있다.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("execution(* com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

6.2.4.2. After 반환하는(returning) advice

After 반환하는(returning) advice는 일치되는 메소드 수행이 대개 반환할때 수행한다. 이것은 @AfterReturning 어노테이션을 사용하여 선언된다.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

노트: 이것은 물론 같은 aspect내부에 여러개의 advice선언과 다른 멤버을 가지는 것이 가능하다. 우리는 논의내 이슈에 집중하기 위한 예제에서 하나의 advice선언을 보여준다.

언젠가 당신은 반환되는 실제값을 위해 advice몸체에 접근할 필요가 있다. 당신은 반환값을 바인딩하는 @AfterReturning 형태를 사용할수 있다.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    returning="retVal")
  public void doAccessCheck(Object retVal) {
    // ...
  }

}
```

returning속성에서 사용된 이름은 advice메소드내 파라미터의 이름과 일치한다. 메소드 수행이 반환할때, 반환값은 일치하는 인자값으로 advice메소드에 전달될것이다. returning절이 명시된 타입(Object의 경우, 어떠한 반환값도 일치할것이다.)의 값을 반환하는 메소드 수행에만 일치하도록 제한한다.

6.2.4.3. After throwing advice

After throwing advice는 일치되는 메소드 수행이 예외를 던져서 빠져나갈때 수행한다. 이것은 @AfterThrowing 어노테이션을 사용하여 선언된다.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doRecoveryActions() {
    // ...
  }

}
```

종종 당신은 주어진 타입의 예외가 던져질때만 수행할 advice를 원하고 advice몸체에서 던져진 예외에 접근할 필요가 있다. 던져진 예외를 advice파라미터에 일치시키는 것을 제한하고 바인드하기 위해 throwing 속성을 사용하라. (원한다면, 다른 예외타입으로 Throwable를 사용하라.)

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    throwing="ex")
  public void doRecoveryActions(DataAccessException ex) {
    // ...
  }

}
```

throwing 속성에서 사용된 이름은 advice메소드내 파라미터의 이름과 일치해야만 한다. 메소드 수행이 예외를 던져서 빠져나갈때, 예외는 일치하는 인자값으로 advice메소드에 전달될것이다. throwing절은 명시된 타입(이 경우 DataAccessException)의 예외를 던지는 메소드 수행을 위해서만 일치하는 것을 제한한다.

6.2.4.4. After (finally) advice

After (finally) advice는 일치되는 메소드 수행이 아무리 빠져나가더라도 수행한다. 이것은 @After 어노테이션을 사용하여 선언된다. After advice는 일반적이고 예외를 반환하는 상태를 다루기 위해서 준비되어야만 한다. 이것은 자원을 풀어주기 위해 사용된다.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

  @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doReleaseLock() {
    // ...
  }

}
```

6.2.4.5. Around advice

advice의 마지막 종류는 around advice이다. Around advice는 일치되는 메소드 수행 "도처에(around)" 수행한다. 이것은 메소드를 수행하는 이전과 이후에 작업하는 기회를 가진다. Around advice는 쓰레드에 안전한 방법(예를 들어 타이머를 시작하고 종료하는)으로 메소드를 수행하기 전과 후에 상태를 공유할 필요가 있다면 종종 사용된다. 언제나 당신의 요구사항에 만족하는 최소한의 advice 형태를 사용하라(이를테면, 간단한 before advice가 그렇게 한다면 around advice를 사용하지 말라).

Around advice는 @Around 어노테이션을 사용하여 선언된다. advice메소드의 첫번째 파라미터는 ProceedingJoinPoint 타입이어야만 한다. advice의 몸체내부에서, ProceedingJoinPoint의 proceed()를 호출하는 것은 기초적인 메소드가 수행되도록 만든다. proceed 메소드는 Object[]로 전달되는 것을 호출할것이다. 이 배열의 값은 처리할때 메소드 수행에 인자로 사용될것이다.
The behavior of proceed when called with an Object[] is a little different than the behavior of proceed for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to proceed must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to proceed in a given argument position supplants the original value at the join point for the entity the value was bound to. (Don't worry if this doesn't make sense right now!). The approach taken by Spring is simpler and a better match to its proxy-based, execution only semantics. You only need to be aware of this difference if you compiling @AspectJ aspects written for Spring and using proceed with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
  }

}
```

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not. Note that proceed may be invoked once, many times, or not at all within the body of the around advice, all of these are quite legal.

6.2.4.6. Advice parameters

Spring 2.0 offers fully typed advice - meaning that you declare the parameters you need in the advice signature (as we saw for the returning and throwing examples above) rather than work with Object[] arrays all the time. We'll see how to make argument and other contextual values available to the advice body in a moment. First let's take a look at how to write

generic advice that can find out about the method the advice is currently advising.

6.2.4.6.1. Access to the current JoinPoint

Any advice method may declare as its first parameter, a parameter of type org.aspectj.lang.JoinPoint (please note that around advice is required to declare a first parameter of type ProceedingJoinPoint, which is a subclass of JoinPoint. The JoinPoint interface provides a number of useful methods such as getArgs() (returns the method arguments), getThis() (returns the proxy object), getTarget() (returns the target object), getSignature() (returns a description of the method that is being advised) and toString() (prints a useful description of the method being advised). Please do consult the Javadocs for full details.

6.2.4.6.2. Passing parameters to advice

We've already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of args. If a parameter name is used in place of a type name in an args expression, then the value of the corresponding argument will be passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of dao operations that take an Account object as the first parameter, and you need access to the account in the advice body. You could write the following:

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
        "args(account,..)")
public void validateAccount(Account account) {
  // ...
}
```

The args(account,..) part of the pointcut expression serves two purposes: firstly it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of Account; secondly, it makes the actual Account object available to the advice via the account parameter.

Another way of writing this is to declare a pointcut that "provides" the Account object value when it matches a join point, and then just refer to the named pointcut from the advice. This would look as follows:

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
          "args(account,..)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
  // ...
}
```

The interested reader is once more refered to the AspectJ programming guide for more details.

The proxy object (this), target object (target), and annotations (@within, @target, @annotation, @args) can all be bound in a similar fashion. The following example shows how you could match the execution of methods annotated with an @Auditable annotation, and extract the audit code.

First the definition of the @Auditable annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
        AuditCode value();
}
```

And then the advice that matches the execution of @Auditable methods:

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && " +
        "@annotation(auditable)")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

6.2.4.6.3. Determining argument names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in (advice and pointcut) method signatures. Parameter names are not available through Java reflection, so Spring AOP uses the following strategies to determine parameter names:

1. If the parameter names have been specified by the user explicitly, then the specified parameter names are used: both the advice and the pointcut annotations have an optional "argNames" attribute which can be used to specify the argument names of the annotated method - these argument names are available at runtime. For example:

```
@Before(
   value="com.xyz.lib.Pointcuts.anyPublicMethod() && " +
         "@annotation(auditable)",
   argNames="auditable")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

   If an @AspectJ aspect has been compiled by the AspectJ compiler (ajc) then there is no need to add the argNames attribute as the compiler will do this automatically.

2. Using the 'argNames' attribute is a little clumsy, so if the 'argNames' attribute has not been specified, then Spring AOP will look at the debug information for the class and try to determine the parameter names from the local variable table. This information will be present as long as the classes have been compiled with debug information ('-g:vars' at a minimum). The consequences of compiling with this flag on are: (1) your code will be slightly easier to understand (reverse engineer), (2) the class file sizes will be very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables will not be applied by your compiler. In other words, uou should encounter no difficulties building with this flag on.

3. If the code has been compiled without the necessary debug information, then Spring AOP will attempt to deduce the pairing of binding variables to parameters (for example, if only

one variable is bound in the pointcut expression, and the advice method only takes one parameter, the pairing is obvious!). If the binding of variables is ambiguous given the available information, then an AmbiguousBindingException will be thrown.

4. If all of the above strategies fail then an IllegalArgumentException will be thrown.

6.2.4.6.4. Proceeding with arguments

We remarked earlier that we would describe how to write a proceed call with arguments that works consistently across Spring AOP and AspectJ. The solution is simply to ensure that the advice signature binds each of the method parameters in order. For example:

```
@Around("execution(List<Account> find*(..)) &&" +
        "com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern)
throws Throwable {
  String newPattern = preProcess(accountHolderNamePattern);
  return pjp.proceed(new Object[] {newPattern});
}
```

In many cases you will be doing this binding anyway (as in the example above).

6.2.4.7. Advice ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so given two pieces of after advice, the one with the highest precedence will run second). For advice defined within the same aspect, precedence is established by declaration order. Given the aspect:

```
@Aspect
public class AspectWithMultipleAdviceDeclarations {

  @Pointcut("execution(* foo(..))")
  public void fooExecution() {}

  @Before("fooExecution()")
  public void doBeforeOne() {
    // ...
  }

  @Before("fooExecution()")
  public void doBeforeTwo() {
    // ...
  }

  @AfterReturning("fooExecution()")
  public void doAfterOne() {
    // ...
  }

  @AfterReturning("fooExecution()")
  public void doAfterTwo() {
    // ...
  }
```

```
}
```

then for any execution of a method named foo, the doBeforeOne, doBeforeTwo, doAfterOne, and doAfterTwo advice methods all need to run. The precedence rules are such that the advice will execute in declaration order. In this case the execution trace would be:

```
doBeforeOne
doBeforeTwo
foo
doAfterOne
doAfterTwo
```

In other words doBeforeOne has precedence over doBeforeTwo, because it was defined before doBeforeTwo, and doAfterTwo has precedence over doAfterOne because it was defined after doAfterOne. It's easiest just to remember that advice runs in declaration order ;) - see the AspectJ Programming Guide for full details.

When two pieces of advice defined in different aspects both need to run at the same join point, then unless you specify otherwise the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by implementing the org.springframework.core.Ordered interface in the aspect class. Given two aspects, the aspect returning the lower value from Ordered.getValue() has the higher precedence.

6.2.5. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the @DeclareParents annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface UsageTracked, and an implementation of that interface DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics via JMX for example).

```
@Aspect
public class UsageTracking {

  @DeclareParents(value="com.xzy.myapp.service.*+",
                  defaultImpl=DefaultUsageTracked.class)
  public static UsageTracked mixin;

  @Before("com.xyz.myapp.SystemArchitecture.businessService() &&" +
          "this(usageTracked)")
  public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
  }

}
```

The interface to be implemented is determined by the type of the annotated field. The value

attribute of the @DeclareParents annotation is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

6.2.6. Aspect instantiation models

This is an advanced topic...

By default there will be a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles :- Spring supports AspectJ's perthis and pertarget instantiation models (percflow, percflowbelow, and pertypewithin are not currently supported).

A "perthis" aspect is declared by specifying a perthis clause in the @Aspect annotation. Let's look at an example, and then we'll explain how it works.

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

  private int someState;

  @Before(com.xyz.myapp.SystemArchitecture.businessService())
  public void recordServiceUsage() {
    // ...
  }

}
```

The effect of the perthis clause is that one aspect instance will be created for each unique service object executing a business service (each unique object bound to 'this' at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it will execute at matched join points, but only when the service object is the one this aspect is associated with. See the AspectJ programming guide for more information on per-clauses.

The 'pertarget' instantiation model works in exactly the same way as perthis, but creates one aspect instance for each unique target object at matched join points.

6.2.7. Example

Now that you've seen how all the constituent parts work, let's put them together to do something useful!

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to

transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we'll need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks:

```
@Aspect
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
      this.order = order;
   }

   @Around("com.xyz.myapp.SystemArchitecture.businessService()")
   public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
      int numAttempts = 0;
      PessimisticLockingFailureException lockFailureException;
      do {
         numAttempts++;
         try {
            return pjp.proceed();
         }
         catch(PessimisticLockingFailureException ex) {
            lockFailureException = ex;
         }
      }
      while(numAttempts <= this.maxRetries);
      throw lockFailureException;
   }

}
```

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice. Notice that for the moment we're applying the retry logic to all businessService()s. We try to proceed, and if we fail with an PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.

The corresponding Spring configuration is:

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
```

```
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>
```

To refine the aspect so that it only retries idempotent operations, we might define an Idempotent annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
  ...
}
```

6.3. Schema-based AOP support

If you are unable to use Java 5, or simply prefer an XML-based format, then Spring 2.0 also offers support for defining aspects using the new "aop" namespace tags. The exact same pointcut expressions and advice kinds are supported as when using the @AspectJ style, hence in this section we will focus on the new syntax and refer the reader to the discussion in the previous section (Section 6.2, “@AspectJ 지원”) for a understanding of writing pointcut expressions and the binding of advice parameters.

To use the aop namespace tags described in this section, you need to import the spring-aop schema as described in Appendix A, XML 스키마-기반 설정. See Section A.2.6, “aop 스키마” for how to import the tags in the aop namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (you can have more than one <aop:config> element in an application context configuration). An <aop:config> element can contain pointcut, advisor, and aspect elements (and these must be declared in that order).

Warning

The <aop:config> style of configuration makes heavy use of Spring's auto-proxying mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use of BeanNameAutoProxyCreator or suchlike. The recommended usage pattern is to use either just the <aop:config> style, or just the AutoProxyCreator style.

6.3.1. Declaring an aspect

Using the schema support, an aspect is simply a regular Java object defined as a bean in your spring application context. The state and behavior is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.

An aspect is declared using the <aop:aspect> element, and the backing bean is referenced using the ref attribute:

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>

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

The bean backing the aspect ("aBean" in this case) can of course be configured and dependency injected just like any other Spring bean.

6.3.2. Declaring a pointcut

A pointcut can be declared inside an aspect, in which case it is visible only within that aspect. A pointcut can also be declared directly inside an <aop:config> element, enabling the pointcut definition to be shared across several aspects and advisors.

A pointcut representing the execution of any business service in the service layer could be defined as follows:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in Section 6.2, "@AspectJ 지원". If you are using the schema based declaration style with Java 5, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression, but this feature is not available on JDK 1.4 and below (it relies on the Java 5 specific AspectJ reflection APIs). On JDK 1.5 therefore, another way of defining the above pointcut would be:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>
```

Assuming you have a SystemArchitecture aspect as described in Section 6.2.3.3, "공통 pointcut 정의 공유하기".

Declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
          expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    ...

  </aop:aspect>

</aop:config>
```

When combining pointcut sub-expressions, '&&' is awkward within an XML document, and so the keywords 'and', 'or' and 'not' can be used in place of '&&', '||' and '!' respectively.

Note that pointcuts defined in this way are referred to by their XML id, and cannot define pointcut parameters. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

6.3.3. Declaring advice

The same five advice kinds are supported as for the @AspectJ style, and they have exactly the same semantics.

6.3.3.1. Before advice

Before advice runs before a matched method execution. It is declared inside an <aop:aspect> using the <aop:before> element.

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

Here dataAccessOperation is the id of a pointcut defined at the top (<aop:config>) level. To define the pointcut inline instead, replace the pointcut-ref attribute with a pointcut attribute:

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.

The method attribute identifies a method (doAccessCheck) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before a data access operation is executed (a method execution join point matched by the pointcut expression), the "doAccessCheck" method on the aspect bean will be invoked.

6.3.3.2. After returning advice

After returning advice runs when a matched method execution completes normally. It is declared inside an <aop:aspect> in the same way as before advice. For example:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the return value within the advice body. Use the returning attribute to specify the name of the parameter to which the return value should be passed:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      returning="retVal"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

The doAccessCheck method must declare a parameter named retVal. The type of this parameter constrains matching in the same way as described for @AfterReturning. For example, the method signature may be declared as:

```
public void doAccessCheck(Object retVal) {...
```

6.3.3.3. After throwing advice

After throwing advice executes when a matched method execution exits by throwing an exception. It is declared inside an <aop:aspect> using the after-throwing element:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      method="doRecoveryActions"/>
```

```
    ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the thrown exception within the advice body. Use the throwing attribute to specify the name of the parameter to which the exception should be passed:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      thowing="dataAccessEx"
      method="doRecoveryActions"/>

    ...

</aop:aspect>
```

The doRecoveryActions method must declare a parameter named dataAccessEx. The type of this parameter constrains matching in the same way as described for @AfterThrowing. For example, the method signature may be declared as:

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

6.3.3.4. After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the after element:

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
      pointcut-ref="dataAccessOperation"
      method="doReleaseLock"/>

    ...

</aop:aspect>
```

6.3.3.5. Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements (i.e. don't use around advice if simple before advice would do).

Around advice is declared using the aop:around element. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on

the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be calling passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds. See Section 6.2.4.5, "Around advice" for notes on calling proceed with an Object[].

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
      pointcut-ref="businessService"
      method="doBasicProfiling"/>

    ...

</aop:aspect>
```

The implementation of the doBasicProfiling advice would be exactly the same as in the @AspectJ example (minus the annotation of course):

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
}
```

6.3.3.6. Advice parameters

The schema based declaration style supports fully typed advice in the same way as described for the @AspectJ support - by matching pointcut parameters by name against advice method parameters. See Section 6.2.4.6, "Advice parameters" for details.

If you wish to explicity specify argument names for the advice methods (not relying on either of the detection strategies previously described) then this is done using the arg-names attribute of the advice element. For example:

```
<aop:before
  pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
  method="audit"
  arg-names="auditable"/>
```

The arg-names attribute accepts a comma-delimited list of parameter names.

Find below a slightly more involved example of the XSD-based approach that illustrates some around advice used in conjunction with a number of strongly typed parameters.

First off, the service interface and implementation of said interface that will be being advised:

```
package x.y.service;

public interface FooService {

   Foo getFoo(String fooName, int age);
}

// the attendant implementation (defined in another file of course)

public class DefaultFooService implements FooService {
```

```
    public Foo getFoo(String name, int age) {
        return new Foo(name, age);
    }
}
```

Next up is the (admittedly simple) aspect. Notice the fact that the profile(..) method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call: the presence of this parameter is an indication that the profile(..) is to be used as around advice:

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

    public Object profile(ProceedingJoinPoint call, String name, int age) throws Throwable {
        StopWatch clock = new StopWatch(
                "Profiling for '" + name + "' and '" + age + "'");
        try {
            clock.start(call.toShortString());
            return call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
    }
}
```

Finally, here is the XML configuration that is required to effect the execution of the above advice for a particular joinpoint:

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

    <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this is the actual advice itself -->
    <bean id="profiler" class="x.y.SimpleProfiler"/>

    <aop:config>
        <aop:aspect ref="profiler">

            <aop:pointcut id="theExecutionOfSomeFooServiceMethod"
                    expression="execution(* x.y.service.FooService.getFoo(String,int))
                    and args(name, age)"/>

            <aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"
                    method="profile"/>

        </aop:aspect>
    </aop:config>

</beans>
```

If we had the following driver script, we would get output something like this on standard output:

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.service.FooService;

public final class Boot {

   public static void main(final String[] args) throws Exception {
      BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
      FooService foo = (FooService) ctx.getBean("fooService");
      foo.getFoo("Pengo", 12);
   }
}
```

```
StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0
-----------------------------------------
ms     %      Task name
-----------------------------------------
00000  ?  execution(getFoo)
```

6.3.3.7. Advice ordering

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in Section 6.2.4.7, "Advice ordering". The precedence between aspects is determined by implementing the `Ordered` interface on the bean backing the aspect.

6.3.4. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the `aop:declare-parents` element inside an `aop:aspect` This element is used to declare that matching types have a new parent (hence the name). For example, given an interface `UsageTracked`, and an implementation of that interface `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics via JMX for example.)

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

  <aop:declare-parents
      types-matching="com.xzy.myapp.service.*+",
      implement-interface="UsageTracked"
      default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

  <aop:before
    pointcut="com.xyz.myapp.SystemArchitecture.businessService()
              and this(usageTracked)"
    method="recordUsage"/>

</aop:aspect>
```

The class backing the usageTracking bean would contain the method:

```
public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
}
```

The interface to be implemented is determined by implement-interface attribute. The value of the types-matching attribute is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

6.3.5. Aspect instantiation models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

6.3.6. Advisors

The concept of "advisors" is brought forward from the AOP support defined in Spring 1.2 and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean, and must implement one of the advice interfaces described in Section 7.3.2, “Spring내 Advice 타입”. Advisors can take advantage of AspectJ pointcut expressions though.

Spring 2.0 supports the advisor concept with the <aop:advisor> element. You will most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring 2.0. Here's how it looks:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

  <aop:advisor
      pointcut-ref="businessService"
      advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

As well as the pointcut-ref attribute used in the above example, you can also use the pointcut attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use

the order attribute to define the Ordered value of the advisor.

6.3.7. Example

Let's see how the concurrent locking failure retry example from Section 6.2.7, "Example" looks when rewritten using the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we'll need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks (it's just a regular Java class using the schema support):

```
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
      this.order = order;
   }

   public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
      int numAttempts = 0;
      PessimisticLockingFailureException lockFailureException;
      do {
         numAttempts++;
         try {
            return pjp.proceed();
         }
         catch(PessimisticLockingFailureException ex) {
            lockFailureException = ex;
         }
      }
      while(numAttempts <= this.maxRetries);
      throw lockFailureException;
   }

}
```

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry).

The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice method. We try to proceed, and if we fail with a PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.
This class is identical to the one used in the @AspectJ example, but with the annotations removed.

The corresponding Spring configuration is:

```
<aop:config>

  <aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">

    <aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    <aop:around
       pointcut-ref="idempotentOperation"
       method="doConcurrentOperation"/>

  </aop:aspect>

</aop:config>

<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
     <property name="maxRetries" value="3"/>
     <property name="order" value="100"/>
</bean>
```

Notice that for the time being we assume that all business services are idempotent. If this is not the case we can refine the aspect so that it only retries genuinely idempotent operations, by introducing an Idempotent annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and using the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

```
  <aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..)) and
                    @annotation(com.xyz.myapp.service.Idempotent)"/>
```

6.4. Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ declaration style, and XML? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

6.4.1. Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise domain objects, or any other object not managed by the Spring container, then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, call join points, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. If aspects play a large role in your design, and you are able to use the AspectJ Development Tools (AJDT) in Eclipse, then the AspectJ language syntax is the preferred option :- it's cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect weaving (linking) phase to your build scripts.

6.4.2. @AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, then you have a choice of @AspectJ or XML style. On balance we recommend use of the @AspectJ style if you are using Java 5. Clearly if you are not running on Java 5, then the XML style is the best choice. Details of the trade-offs between the XML and @AspectJ styles are discussed below.

The XML style will be most familiar to existing Spring users. It can be used with any JDK level (referring to named pointcuts from within pointcut expressions does still require Java 5 though) and is backed by genuine POJOs. When using AOP as a tool to configure enterprise services (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently) then XML can be a good choice. With the XML style it is arguably clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of how a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is more limited in what in can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style we can write something like:

```
@Pointcut(execution(* get*()))
public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ *(..))
public void operationReturningAnAccount() {}
```

```
@Pointcut(propertyAccess() && operationReturningAnAccount())
public void accountPropertyAccess() {}
```

In the XML style I can declare the first two pointcuts:

```
<aop:pointcut id="propertyAccess"
    expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
    expression="execution(org.xyz.Account+ *(..))"/>
```

but I cannot define the accountPropertyAccess pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ based approach. On balance we prefer the @AspectJ style whenever you have aspects that do more than simple "configuration" of enterprise services.

6.5. Mixing aspect types

It is perfectly possible to mix @AspectJ style aspects using the autoproxying support, schema-defined <aop:aspect> aspects, <aop:advisor> declared advisors and even proxies and interceptors defined using the Spring 1.2 style in the same configuration. All of these are implemented using the same underlying support mechanism and will co-exist without any difficulty.

6.6. Proxying mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. (JDK dynamic proxies are preferred whenever you have a choice).

If the target object to be proxied implements at least one interface then a JDK dynamic proxy will be used. All of the interfaces implemented by the target type will be proxied. If the target object does not implement any interfaces then a CGLIB proxy will be created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not just those implemented by its interfaces) you can do so. However, there are some issues to consider:

☒ Final methods cannot be advised, as they cannot be overriden

☒ You will need the CGLIB 2 binaries on your classpath, whereas dynamic proxies are available with the JDK

To force the use of CGLIB proxies set the value of the proxy-target-class attribute of the

<aop:config> element to true:

```
<aop:config proxy-target-class="true">

        ...

</aop:config>
```

To force CGLIB proxying when using the @AspectJ autoproxy support, set the proxy-target-class attribute of the <aop:aspectj-autoproxy> element as follows:

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

6.7. Programmatic creation of @AspectJ Proxies

In addition to declaring aspects in your configuration using either <aop:config> or <aop:aspectj-autoproxy> it is also possible programmatically to create proxies that advise target objects. For the full details of Spring's AOP API, see the next chapter. Here we want to focus on the ability to automatically create proxies using @AspectJ aspects.

The class org.springframework.aop.aspectj.annotation.AspectJProxyFactory can be used to create a proxy for a target object that is advised by one or more @AspectJ aspects. Basic usage for this class is very simple, as illustrated below. See the Javadocs for full information.

```
// create a factory that can generate a proxy for the given target object
AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object supplied must be an @AspectJ aspect
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

6.8. Using AspectJ with Spring applications

Everything we've covered so far in this chapter is pure Spring AOP. In this section, we're going to look at how you can use the AspectJ compiler/weaver instead of or in addition to Spring AOP if your needs go beyond the facilities offered by Spring AOP alone.

Spring ships with a small AspectJ aspect library (it's available standalone in your distribution as spring-aspects.jar, you'll need to add this to your classpath to use the aspects in it). Section 6.8.1, "Using AspectJ to dependency inject domain objects with Spring" and Section 6.8.2, "Other Spring aspects for AspectJ" discuss the content of this library and how you can use it. Section 6.8.3, "Configuring AspectJ aspects using Spring IoC" discusses how to dependency inject AspectJ aspects that are woven using the AspectJ compiler. Finally, Section 6.8.4, "Using AspectJ Load-time weaving (LTW) with Spring applications" provides an introduction to load-time weaving for Spring applications using AspectJ.

6.8.1. Using AspectJ to dependency inject domain objects with Spring

The Spring container instantiates and configures beans defined in your application context. It is also possible to ask a bean factory to configure a pre-existing object given the name of a bean definition containing the configuration to be applied. The spring-aspects.jar contains an annotation-driven aspect that exploits this capability to allow dependency-injection of any object. The support is intended to be used for objects created outside of the control of any container. Domain objects often fall into this category: they may be created programmatically using the new operator, or by an ORM tool as a result of a database query.

The DependencyInjectionInterceptorFactoryBean in the org.springframework.orm.hibernate.support package can be used to have Spring create and configure prototype domain objects for Hibernate (using either autowiring or named prototype bean definitions). The interceptor does not of course support configuration of objects that you create yourself programmatically rather than retrieve from the database. Similar techniques may be possible with other frameworks. As ever, be pragramatic and choose the simplest thing that meets your requirements. Please note that the aforementioned class does not ship with the Spring distribution. If you want to make use of it, you will have to download it from the Spring CVS repository and compile it yourself; the file itself can be found in the 'sandbox' directory of the Spring CVS repository tree.

The @Configurable annotation marks a class as eligible for Spring-driven configuration. In the simplest case it can be used just as a marker annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class Account {
    ...
}
```

When used simply as a marker interface in this way, Spring will configure new instances of the annotated type (Account in this case) using a prototypical bean definition with the same name as the fully-qualified type name (com.xyz.myapp.domain.Account). Since the default name for a bean is the fully-qualified name of its type, a convenient way to declare the prototype definition is simply to omit the id attribute:

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
  <property name="fundsTransferService" ref="fundsTransferService"/>
  ...
</bean>
```

If you want to explicitly specify the name of the prototype bean definition to use, you can do so directly in the annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")
public class Account {
    ...
}
```

Spring will now look for a bean definition named "account" and use that as a prototypical definition to configure new Account instances.

You can also use autowiring to avoid having to specify a prototypical bean definition at all. To have Spring apply autowiring use the autowire property of the @Configurable annotation: specify either @Configurable(autowire=Autowire.BY_TYPE) or @Configurable(autowire=Autowire.BY_NAME for autowiring by type or by name respectively.

Finally you can enable Spring dependency checking for the object references in the newly created and configured object by using the dependencyCheck attribute (for example: @Configurable(autowire=Autowire.BY_NAME,dependencyCheck=true)). If this attribute is set to true, then Spring will validate after configuration that all properties (that are not primitives or collections) have been set.

Merely using the annotation on its own does nothing of course. It's the AnnotationBeanConfigurerAspect in spring-aspects.jar that acts on the presence of the annotation. In essence the aspect says "after returning from the initialization of a new object of a type with the @Configurable annotation, configure the newly created object using Spring in accordance with the properties of the annotation". For this to work the annotated types must be woven with the AspectJ weaver - you can either use a build-time ant or maven task to do this (see for example the AspectJ Development Environment Guide) or load-time weaving (see Section 6.8.4, "Using AspectJ Load-time weaving (LTW) with Spring applications").

The AnnotationBeanConfigurerAspect itself needs configuring by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). The Spring AOP namespace defines a convenient tag for doing this. Simply include the following in your application context configuration:

```
<aop:spring-configured/>
```

If you are using the DTD instead of schema, the equivalent definition is:

```
<bean
  class="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"
  factory-method="aspectOf"/>
```

Instances of @Configurable objects created before the aspect has been configured will result in a warning being issued to the log and no configuration of the object taking place. An example might be a bean in the Spring configuration that creates domain objects when it is initialized by Spring. In this case you can use the "depends-on" bean attribute to manually specify that the bean depends on the configuration aspect.

```
<bean id="myService"
  class="com.xzy.myapp.service.MyService"
  depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">
  ...
</bean>
```

6.8.1.1. Unit testing @Configurable objects

One of the goals of the @Configurable support is to enable independent unit testing of domain

objects without the difficulties associated with hard-coded lookups. If @Configurable types have not been woven by AspectJ then the annotation has no affect during unit testing, and you can simply set mock or stub property references in the object under test and proceed as normal. If @Configurable types have been woven by AspectJ then you can still unit test outside of the container as normal, but you will see a warning message each time that you construct an @Configurable object indicating that it has not been configured by Spring.

6.8.1.2. Working with multiple application contexts

The AnnotationBeanConfigurerAspect used to implement the @Configurable support is an AspectJ singleton aspect. The scope of a singleton aspect is the same as the scope of static members, i.e. there is one aspect instance per classloader that defines the type. This means that if you define multiple application contexts within the same classloader hierarchy you need to consider where to define the <aop:spring-configured/> bean and where to place spring-aspects.jar on the classpath.

Consider a typical Spring web-app configuration with a shared parent application context defining common business services and everything needed to support them, and one child application context per servlet containing definitions particular to that servlet. All of these contexts will co-exist within the same classloader hierarchy, and so the AnnotationBeanConfigurerAspect can only hold a reference to one of them. In this case we recommend defining the <aop:spring-configured/> bean in the shared (parent) application context: this defines the services that you are likely to want to inject into domain objects. A consequence is that you cannot configure domain objects with references to beans defined in the child (servlet-specific) contexts using the @Configurable mechanism (probably not something you want to do anyway!).

When deploying multiple web-apps within the same container, ensure that each web-application loads the types in spring-aspects.jar using its own classloader (for example, by placing spring-aspects.jar in WEB-INF/lib). If spring-aspects.jar is only added to the container wide classpath (and hence loaded by the shared parent classloader), all web applications will share the same aspect instance which is probably not what you want.

6.8.2. Other Spring aspects for AspectJ

In addition to the @Configurable support, spring-aspects.jar contains an AspectJ aspect that can be used to drive Spring's transaction management for types and methods annotated with the @Transactional annotation. This is primarily intended for users who want to use Spring's transaction support outside of the Spring container.

The aspect that interprets @Transactional annotations is the AnnotationTransactionAspect. When using this aspect, you must annotate the implementation class (and/or methods within that class), not the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are not inherited.

A @Transactional annotation on a class specifies the default transaction semantics for the execution of any public operation in the class.

A @Transactional annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Methods with public, protected, and

default visibility may all be annotated. Annotating protected and default visibility methods directly is the only way to get transaction demarcation for the execution of such operations.

For AspectJ programmers that want to use the Spring configuration and transaction management support but don't want to (or can't) use annotations, spring-aspects.jar also contains abstract aspects you can extend to provide your own pointcut definitions. See the Javadocs for AbstractBeanConfigurerAspect and AbstractTransactionAspect for more information. As an example, the following excerpt shows how you could write an aspect to configure all instances of objects defined in the domain model using prototypical bean definitions that match the fully-qualified class names:

```
public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {

  public DomainObjectConfiguration() {
    setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());
  }

  // the creation of a new bean (any object in the domain model)
  protected pointcut beanCreation(Object beanInstance) :
    initialization(new(..)) &&
    SystemArchitecture.inDomainModel() &&
    this(beanInstance);

}
```

6.8.3. Configuring AspectJ aspects using Spring IoC

When using AspectJ aspects with Spring applications, it's natural to want to configure such aspects using Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ created aspects via Spring depends on the AspectJ instantiation model (per-clause) used by the aspect.

The majority of AspectJ aspects are singleton aspects. Configuration of these aspects is very easy, simply create a bean definition referencing the aspect type as normal, and include the bean attribute 'factory-method="aspectOf"'. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. For example:

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
      factory-method="aspectOf">
  <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
</bean>
```

For non-singleton aspects the easiest way to configure them is to create prototypical bean definitions and annotate use the @Configurable support from spring-aspects.jar to configure the aspect instances once they have bean created by the AspectJ runtime.

If you have some @AspectJ aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other @AspectJ aspects that you want to use with Spring AOP, and these aspects are all configured using Spring then you'll need to tell the Spring AOP @AspectJ autoproxying support which subset of the @AspectJ aspects defined in the configuration should be used for autoproxying. You can do this by using one or more <include/> elements inside the <aop:aspectj-autoproxy/> declaration. Each include element specifies a

name pattern, and only beans with names matched by at least one of the patterns will be used for Spring AOP autoproxy configuration:

```
<aop:aspectj-autoproxy>
  <include name="thisBean"/>
  <include name="thatBean"/>
</aop:aspectj-autoproxy>
```

6.8.4. Using AspectJ Load-time weaving (LTW) with Spring applications

Load-time weaving (or LTW) refers to the process of weaving AspectJ aspects with an application's class files as they are loaded into the VM. For full details on configuring load-time weaving with AspectJ, see the LTW section of the AspectJ Development Environment Guide . We will focus here on the essentials of configuring load-time weaving for Spring applications running on Java 5.

Load-time weaving is controlled by defining a file 'aop.xml' in the META-INF directory. AspectJ automatically looks for all 'META-INF/aop.xml' files visible on the classpath and configures itself based on the aggregation of their content.

A basic META-INF/aop.xml for your application should look like this:

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN"     "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
   <weaver>
         <include within="com.xyz.myapp..*"/>
   </weaver>
</aspectj>
```

The <include/>' element tells AspectJ what set of types should be included in the weaving process. Use the package prefix for your application followed by "..*" (meaning '... and any type defined in a subpackage of this') as a good default. Using the include element is important as otherwise AspectJ will look at every type loaded in support of your application (including all the Spring library classes and many more besides). Normally you don't want to weave these types and don't want to pay the overhead of AspectJ attempting to match against them.

To get informational messages in your log file regarding the activity of the load-time weaver, add the following options to the weaver element:

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN"     "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
   <weaver
         options="-showWeaveInfo
               -XmessageHandlerClass:org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
         <include within="com.xyz.myapp..*"/>
   </weaver>
</aspectj>
```

Finally, to control exactly which aspects are used, you can use the aspects element. By default

all defined aspects are used for weaving (spring-aspects.jar contains a META-INF/aop.xml file that defines the configuration and transaction aspects). If you were using spring-aspects.jar, but only want the configuration support and not the transaction support you could specify this as follows:

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN"     "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
  <aspects>
    <include within="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"/>
  </aspects>
  <weaver
    options="-showWeaveInfo -XmessageHandlerClass:org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
    <include within="com.xyz.myapp..*"/>
  </weaver>
</aspectj>
```

On the Java 5 platform, load-time weaving is enabled by specifying the following VM argument when launching the Java virtual machine:

```
-javaagent:<path-to-ajlibs>/aspectjweaver.jar
```

6.9. Further Resources

More information on AspectJ can be found at the AspectJ home page.

The book Eclipse AspectJ by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

The excellent AspectJ in Action by Ramnivas Laddad (Manning, 2003) comes highly recommended as an introduction to AOP; the focus of the book is on AspectJ, but a lot of general AOP themes are explored in some depth.

Chapter 7. Spring AOP APIs

7.1. 소개

이전 장에서 @AspectJ를 사용하는 AOP에 관한 Spring 2.0 지원과 스키마 기반 aspect 정의를 설명하였다. 이번 장에서 우리는 저수준 Spring AOP APIs와 Spring 1.2 애플리케이션에서 사용된 AOP 지원을 논의 할 것이다. 새로운 애플리케이션에 대하여, 우리는 이전 장에서 설명한 Spring 2.0 AOP 지원의 사용을 권장한다. 그러나 기존 애플리케이션을 이용하여 작업을 하거나 도서와 기사를 읽을 때 여러분은 Spring 1.2 스타일 예제를 교차하여 참고해야 할 지도 모른다. Spring 2.0은 Spring 1.2 이전 버전 호환을 완벽히 하고 있다. 이전 장에서 설명한 모든것은 Spring 2.0에서 전부 지원한다.

7.2. Spring에서의 Pointcut API

어떻게 Spring이 결정적인 pointcut(교차점)개념을 처리하는지 보도록 하자.

7.2.1. 개념

Spring의 pointcut(교차점) 모델은 advice(충고) 유형의 pointcut 재사용 독립을 가능하게 한다. 동일한 pointcut(교차점)을 사용하여 다른 advice(충고)를 설정할 수 있다.

org.springframework.aop.Pointcut interface는 중심이 되는 인터페이스 이며, 특정한 클래스와 메서드에 advice들을 설정 하고는 했다. 아래에서 완성된 인터페이스를 보자:

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

두개 부분으로 나눠진 Pointcut 인터페이는 클래스와 매치되는 부분을 갖는 메서드의 재사용과 매끄럽게 구성된 기능들(예를 들어 다른 메서드 matcher를 이용하여 "union(병합)" 수행을 허용한다.

주어진 타켓 클래스들의 세트에 pointcut(교차점)을 제한하기 위해 ClassFilter 인터페이스를 사용했다. matches() 메소드가 항상 true를 반환 한다면, 모든 타켓 클래스들을 매치 시킬 것이다:

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

일반 적으로 MethodMatcher 인터페이스는 좀 더 중요한다. 아래에서 완성된 인터페이스를 보자:

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

```

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

이 pointcut(교차점)이 타켓 클래스상에서 주어진 메서드를 매치하여 포함할 것인지 검사하기 위해 matches(Method, Class)을 사용한다. 모든 메서드 호출에 대하여 검사의 요구를 피하기 위해서 AOP 프록시를 생성할때 이 평가를 수행한다. 만일 인자 2개를 가진 matches 메서드가 주어진 메서드에 대해 true를 반환 한다면, 그리고 MethodMatcher에 대하여 isRuntime()메서드가 true를 반환하게 된다면, 모든 메서드 호출에 대하여 인자 3개를 가진 matches 메서드을 호출 할 것이다. 이것으로 pointcut(교차점)은 타켓 advice(충고)가 수행하기 위한 시점 바로 전에 메소드 호출에 전달된 인자들을 조사하는게 가능하게 한다.

대부분 MethodMatcher들은 정적이며, 그것들의 isRuntime() 메소드는 fasle를 반환함을 의미한다. 이 경우에, 인자 3개를 가진 matches 메서드를 결코 호출하지 않을 것이다.

Tip

가능하다면, AOP프록시가 생성되었을때 AOP프레임워크가 pointcut평가의 결과를 캐시하는 것을 허용하도록 pointcut를 정적으로 만들어라.

7.2.2. pointcut에서의 기능

Spring은 pointcut에서의 기능을 지원한다. 특히 조합(union) 과 교차(intersection).

☒ 조합(Union)은 pointcut이 일치하는 메소드를 의미한다.

☒ 교차(Intersection)는 두개의 pointcut이 서로 일치하는 메소드를 의미한다.

☒ 조합(Union)은 대개 좀더 유용하다.

☒ pointcut은 org.springframework.aop.support.Pointcuts 클래스내 정적 메소드나 같은 패키지내 ComposablePointcut 클래스를 사용하여 이루어질수 있다. 어쨌든, AspectJ pointcut표현을 사용하는 것이 대개 좀더 단순한 접근법이다.

7.2.3. AspectJ 표현 pointcuts

2.0 이후, Spring에 의해 사용되는 pointcut의 가장 중요한 타입은 org.springframework.aop.aspectj.AspectJExpressionPointcut이다. 이것은 AspectJ pointcut 표현 문자열을 파싱하기 위한 AspectJ 제공 라이브러리를 사용하는 pointcut이다.

제공되는 AspectJ pointcut에 대한 논의를 위해 앞장을 보라.

7.2.4. 편리한 pointcut 구현물

Spring은 다양하고 편리한 pointcut 구현물을 제공한다. 몇가지는 특별히 사용될수 있다. 다른 것들은 애플리케이션의 특성을 가지는 pointcut으로 하위클래스화되는 경향이 있다.

7.2.4.1. 정적인 pointcuts

정적인 pointcut은 메소드와 대상 클래스에 기초하고 메소드의 인자를 고려할수 없다. 정적인 pointcut은

충분하다. 그리고 대부분의 사용을 위해 제일좋다. 메소드가 처음으로 호출될때 Spring이 정적인 pointcut을 오직 한번 평가하는 것이 가능하다. 그리고 나서, 각각의 메소드 호출로 pointcut을 다시 평가할 필요는 없다.

Spring에 포함된 몇가지 정적인 pointcut구현물을 고려하자.

7.2.4.1.1. 정규 표현식 pointcut

정적인 pointcut을 명시하는 알기쉬운 방법은 정규표현식이다. Spring에 더하여 다양한 AOP프레임워크는 이 가능성을 만든다. org.springframework.aop.support.Perl5RegexpMethodPointcut은 Perl5 정규표현식 문법을 사용하는 일반적인 정규표현식 pointcut이다. Perl5RegexpMethodPointcut 클래스는 정규표현식을 일치시키기 위한 Jakarta ORO에 의존한다. Spring또한 JDK 1.4이상에서 지원되는 정규 표현식을 사용하는 JdkRegexpMethodPointcut 클래스를 제공한다.

Perl5RegexpMethodPointcut 클래스를 사용하여, 당신은 패턴 문자열의 목록을 제공할수 있다. 이러한 것들이 일치한다면, pointcut이 true로 평가될것이다(그래서 결과는 이러한 pointcut의 효과적인 조합이다.)

아래에서 사용법을 보여준다.

```
<bean id="settersAndAbsquatulatePointcut"
    class="org.springframework.aop.support.Perl5RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring은 advice를 참조하는 것을 허용하는 RegexpMethodPointcutAdvisor이라는 편리한 클래스를 제공한다(advice는 인터셉터, before advice, throws advice등이 될수 있다). Spring은 J2SE 1.4나 그 이상의 버전에서 JdkRegexpMethodPointcut를 사용할것이다. 그리고 이전 VM에서는 Perl5RegexpMethodPointcut를 사용할것이다. Perl5RegexpMethodPointcut은 perl5 프라퍼티를 true로 셋팅하여 강제로 사용할수 있다. 아래에서 보여주는 것과 같이 하나의 bean이 pointcut과 advice 모두를 캡슐화하는 것처럼 RegexpMethodPointcutAdvisor를 사용하는 것은 묶기(wiring)를 단순화한다.

```
<bean id="settersAndAbsquatulateAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

RegexpMethodPointcutAdvisor는 어떤 advice타입과도 사용될수 있다.

7.2.4.1.2. 속성-지향 pointcuts

정적인 pointcut의 중요한 타입은 메타데이터-지향 pointcut이다. 이것은 대개 소스-레벨의 메타데이터인 메타데이터 속성의 값을 사용한다.

7.2.4.2. 동적 pointcuts

동적인 pointcut은 정적인 pointcut보다 평가하는데 좀더 가치가 있다. 그것들은 정적인 정보만큼 메소드 인자를 고려한다. 이것은 모든 메소드 호출시 평가되어야만 한다는 것을 의미한다. 결과는 인자가 다양한 만큼, 캐시될수 없다.

중요한 예제는 control flow pointcut이다.

7.2.4.2.1. 제어흐름(Control flow) pointcuts

Spring 제어흐름(control flow) pointcuts은 다소 덜 강력하지만 개념적으로 AspectJ cflow pointcuts과 유사하다(pointcut이 다른 pointcut에 의해 일치되는 join pointcut아래에서 수행되는 것을 명시하는 방법은 없다). 제어흐름(control flow) pointcut은 현재 호출 스택과 일치한다. 예를 들어, join point가 com.mycompany.web 패키지내 메소드나 SomeCaller 클래스에 의해 호출된다면 수행된다. 제어흐름(control flow) pointcut은 org.springframework.aop.support.ControlFlowPointcut 클래스를 사용하여 명시된다.

Note

제어흐름(control flow) pointcut은 다른 동적인 pointcut보다 런타임시 평가하는 것이 명백히 좀더 비싸다. Java 1.4에서, 비용은 대략 다른 동적인 pointcut의 다섯배이다. Java 1.3에서는 10배 이상이다.

7.2.5. Pointcut 슈퍼클래스

Spring은 자체적인 pointcut을 구현하는 것을 돕기 위한 유용한 pointcut 슈퍼클래스를 제공한다.

정적인 pointcut이 가장 유용하기 때문에, 당신은 아마도 아래에서 보여주는 것처럼 StaticMethodMatcherPointcut의 하위클래스를 만들것이다. 이것은 하나의 추상 메소드를 구현하는 것을 요구한다(비록 행위자를 사용자정의하기 위해 다른 메소드를 오버라이드하는 가능성이 있더라도).

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

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

여기에는 동적인 pointcut을 위한 슈퍼클래스가 있다.

당신은 Spring 1.0 RC2와 그 이상에서 advice타입을 가진 사용자정의 pointcut을 사용할수 있다.

7.2.6. 사용자정의 pointcuts

Spring AOP내 pointcut이 Java클래스이기 때문에, 정적이거나 동적이거나 사용자정의 pointcut을 선언하는 것이 가능하다. Spring내 사용자정의 pointcut은 임의로 복잡할수 있다. 어쨌든, AspectJ pointcut표현 언어를 사용하는 것은 가능하다면 추천된다.

Note

Spring의 차후 버전은 JAC에 의해 제공되는 것처럼 "의미론적인(semantic) pointcut"을 위한 지원을 제공한다. 예를 들어, "모든 메소드는 대상 객체내 인스턴스 변수를 변경한다."

7.3. Spring내 Advice API

Spring AOP가 advice를 다루는 방법을 보자.

7.3.1. Advice 생명주기

각각의 advice는 Spring bean이다. advice인스턴스는 모든 advice된 객체를 통해서나 각각의 advice된 객체에 유일하게 공유될수 있다. 이것은 클래스별 이나 인스턴스별 advice에 일치한다.

클래스별 advice는 가장 종종 사용된다. 이것은 트랜잭션 advisor와 같이 대개의 advice를 위해 적절하다. 이것은 프록시된 객체의 상태에 의존하지 않거나 새로운 상태를 추가한다. 그것들은 단순히 메소드와 인자에서 작동한다.

인스턴스별 advice는 믹스인(mixins)을 지원하기 위한 소개(introduction)를 위해 적절하다. 이 경우, advice는 프록시된 객체를 위한 상태를 추가한다.

같은 AOP프록시에서 공유되고 인스턴스별 advice를 혼합하여 사용하는 것은 가능하다.

7.3.2. Spring내 Advice 타입

Spring은 특별히 다양한 advice타입을 제공하고 임의의 advice타입을 지원하기 위해 확장가능하다. 기본적인 개념과 표준 advice타입을 보자.

7.3.2.1. Interception around advice

Spring내 가장 기본적인 advice타입은 interception around advice이다.

Spring은 메소드 인터셉션을 사용하여 around advice를 위한 AOP제휴 인터페이스와 호환된다. MethodInterceptors 구현 around advice는 다음의 인터페이스를 구현해야만 한다.

```
public interface MethodInterceptor extends Interceptor {

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

invoke() 메소드를 위한 MethodInvocation 인자는 호출되는 메소드, 대상 join point, AOP프록시 그리고 메소드를 위한 인자를 나타낸다. invoke() 메소드는 호출의 결과(join point의 반환값)를 반환해야만 한다.

간단한 MethodInterceptor 구현물은 다음처럼 보일것이다.

```
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 호출하는 것을 노트하라. 이것은 join point를 향한 인터셉터 연쇄가 발생한다. 대부분의 인터셉터는 이 메소드를 호출하고 반환값을 반환할것이다. 어쨌든, around advice와 같은 MethodInterceptor는 proceed메소드를 호출하는 것보다 다른 값을 던지거나 예외를 던질수 있다. 어쨌든, 당신은 좋은 이유없이는 이것을 하길 원하지 않는다.

Note

MethodInterceptor는 다른 AOP 제휴-호환 AOP구현물과의 상호운용을 제공한다. 이 장의 나머지에서 언급되는 다른 advice타입은 Spring특성을 가지는 방법으로 공통적인 AOP개념을 구현한다. 대부분의 특정 advice타입을 사용하는 장점이 있는 반면에, 당신이 또다른 AOP프레임워크내 aspect를 수행하고자 한다면 MethodInterceptor around advice에 충실하라. pointcut은 프레임워크간의 상호작용하지 않고 AOP제휴은 pointcut인터페이스를 정의하지 않는다는 것을 노트하라.

7.3.2.2. Before advice

좀더 간단한 advice타입은 before advice이다. 이것은 메소드에 들어가기 전에만 호출되기 때문에 MethodInvocation객체가 필요하지 않다.

before advice의 중요한 장점은 proceed() 메소드를 호출할 필요가 없다는 것이다. 그러므로 인터셉터 연쇄를 처리하는 것을 실패하는 우연한 가능성이 없다.

아래에서 MethodBeforeAdvice 인터페이스를 보여준다. (비록 대개의 객체는 필드 인터셉션에 적용하고 Spring은 이것을 구현할것 같지 않더라도 Spring API 디자인은 필드 before advice를 허용할것이다.)

```
public interface MethodBeforeAdvice extends BeforeAdvice {

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

반환타입이 void라는 것을 노트하라. before advice는 join point가 수행되기 전에 사용자정의 행위자를 추가할수 있다. 하지만 반환값을 변경할수는 없다. before advice가 예외를 던진다면, 이것은 인터셉터 연쇄의 수행을 취소할것이다. 예외는 인터셉터 연쇄를 위임할것이다. 체크되지 않았거나 호출된 메소드의 시그너처위라면, 클라이언트로 직접 전달될것이다. 그렇지 않다면, AOP프록시에 의해 체크되지 않은 예외로 포장될것이다.

모든 메소드 호출을 세는 Spring내 before advice의 예제이다.

```
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;
    }
}
```

Tip

Before advice는 어떤 pointcut과도 사용될수 있다.

7.3.2.3. Throws advice

Throws advice는 join point가 예외를 던질때, join point의 반환후에 호출된다. Spring은 타입화된 throws advice를 제공한다. 이것은 org.springframework.aop.ThrowsAdvice 인터페이스가 어떠한 메소드도 포함하지 않는다는 것을 의미한다. 이것은 주어진 객체가 하나 이상의 타입화된 throws advice메소드를 구현하는 태그 인터페이스 구분이다. 이것은 다음의 형태가 될것이다.

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

오직 마지막 인자만 필수이다. 메소드 시그너처는 advice메소드가 메소드나 인자에 관심을 가지는지에 대해 의존하는 하나의 인자에서 4개의 인자를 가지는 형태로 다양하다. 다음은 throws advice의 예제이다.

아래의 advice는 RemoteException(하위클래스를 포함해서)이 던져진다면 호출된다.

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

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

다음의 advice는 ServletException가 던져진다면 호출된다. 위 advice와는 달리, 이것은 4개의 인자를 선언한다. 그래서 호출된 메소드, 메소드 인자와 대상 객체에 접근한다.

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

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

마지막 예제는 이러한 두개의 메소드가 RemoteException 과 ServletException를 다루는 한개의 클래스에서 사용되는 방법을 보여준다. 상당수의 throws advice메소드는 하나의 클래스에서 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
    }
}
```

Tip

Throws advice는 어떤 pointcut과도 사용될수 있다.

7.3.2.4. After Returning advice

Spring내 after returning advice는 아래에서 보여주는 것처럼 org.springframework.aop.AfterReturningAdvice 인터페이스를 구현해야만 한다.

```
public interface AfterReturningAdvice extends Advice {

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

after returning advice는 반환값(변경할수 없는), 호출된 메소드, 메소드 인자와 대상에 접근한다.

다음의 after returning advice는 예외를 던지지 않는 모든 성공적인 메소드 호출을 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 수행경로를 변경하지 않는다. 예외를 던진다면, 이것은 반환값대신에 인터셉터 연쇄를 던질것이다.

Tip

After returning advice는 어떠한 pointcut과도 사용될수 있다.

7.3.2.5. Introduction advice

Spring은 특별한 종류의 인터셉션 advice처럼 introduction advice를 처리한다.

Introduction은 다음의 인터페이스를 구현하는 IntroductionAdvisor 와 IntroductionInterceptor를 요구한다.

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

AOP제휴 MethodInterceptor 인터페이스로부터 상속된 invoke() 메소드는 introduction을 구현해야만 한다. 소개된(introduced) 인터페이스에서 메소드가 호출된다면, 소개(introduction) 인터셉터가 메소드 호출을 다루는 책임을 진다. 이것은 proceed()를 호출할수 없다.

소개(Introduction) advice는 메소드 레벨보다는 클래스 레벨에서만 적용되는것처럼 어떠한 pointcut과도 사용될수 없다. 당신은 다음의 메소드를 가지는 IntroductionAdvisor를 가진 소개(introduction) advice만을 사용할수 있다.

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {
```

```
        ClassFilter getClassFilter();

        void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

        Class[] getInterfaces();
}
```

여기에는 소개(introduction) advice와 관련된 MethodMatcher가 없다. 나아가 Pointcut도 없다. 오직 클래스 필터링이 논리적이다.

getInterfaces() 메소드는 이 advisor에 의해 소개된(introduced) 인터페이스를 반환한다.
validateInterfaces()메소드는 소개된(introduced) 인터페이스가 설정된 IntroductionInterceptor에 의해 구현될수 있는 있는지를 보기 위해 내부적으로 사용된다.

Spring 테스트 묶음으로부터 간단한 예제를 보자. 하나 이상의 객체를 위해 다음 인터페이스를 소개(introduce)하길 원한다고 가정해보자.

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

이것은 mixin을 보여준다. 우리는 타입이 무엇이든, lock과 unlock메소드를 호출하든 충고된(advised) 객체를 Lockable로 형변환할수 있도록 원한다. lock() 메소드를 호출한다면, LockedException를 던지기 위한 모든 setter메소드를 원한다. 게다가 우리는 객체를 불변으로 만들기 위한 능력을 제공하는 aspect를 추가할수 있다. 이것은 AOP의 좋은 예제이다.

첫째로, 우리는 IntroductionInterceptor가 필요할것이다. 이 경우, 편리한org.springframework.aop.support.DelegatingIntroductionInterceptor 클래스를 확장한다. 우리는 IntroductionInterceptor를 직접 구현할수 있다. 하지만 DelegatingIntroductionInterceptor를 사용하는 것이 대부분의 경우를 위해 가장 좋다.

DelegatingIntroductionInterceptor는 가로채기(interception)의 사용을 숨기고 소개된(introduced) 인터페이스의 실제 구현물을 위한 소개(introduction)를 위임하기 위해 디자인되었다. 위임은 생성자의 인자를 사용하여 어떠한 객체에도 셋팅될수 있다. 디폴트 위임(인자가 없는 생성자가 사용될때)은 이것이다. 게다가 아래의 예제에서, 위임은 DelegatingIntroductionInterceptor의 LockMixin 하위클래스이다. 위임이 주어졌을때, DelegatingIntroductionInterceptor 인스턴스는 위임에 의해 구현된 모든 인터페이스를 검색하고 인터페이스에 대한 소개(introduction)을 지원할것이다. LockMixin과 같은 하위클래스가 표시되지 않는 인터페이스를 억누르기 위한 suppressInterface(Class intf) 메소드를 호출하는 것이 가능하다. 어쨌든, IntroductionInterceptor가 지원하기 위해 준비되는 인터페이스가 얼마나 많은지는 문제가 되지 않는다. IntroductionAdvisor는 인터페이스가 실제로 나타나는 것을 제어할것이다. 소개된(introduced) 인터페이스는 대상에 의해 같은 인터페이스의 구현물을 숨길것이다.

게다가 LockMixin은 DelegatingIntroductionInterceptor의 하위클래스를 만들고 Lockable 자체를 구현한다. 슈퍼클래스는 Lockable이 소개(introduction)를 위해 제공될수 있는 것을 자동적으로 집어올린다. 그래서 우리는 명시할 필요가 없다. 우리는 이 방법으로 많은 인터페이스를 소개(introduce)할수 있다.

locked 인스턴스 변수의 사용을 노트하라. 이것은 대상 객체를 고정하기 위해 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 오버라이드할 필요가 없다. DelegatingIntroductionInterceptor 구현물은 메소드가 소개(introduced)된다면 위임메소드를 호출하지만 그렇지 않다면 join point로 처리된다. 이것은 언제나 충분하다. 현재 상황에서, 우리는 체크를 추가할 필요가 있다. 잠김모드라면 호출될수 있는 setter메소드는 없다.

요구되는 소개(introduction) advisor는 단순하다. 해야할 필요가 있는 모든것은 구분되는 LockMixin 인스턴스를 가지고 소개된(introduced) 인터페이스(이 경우 Lockable)를 명시한다. 좀더 복잡한 예제는 소개(introduction) 인터셉터(프로토타입으로 정의될)에 대한 참조를 가진다. 이 경우, LockMixin에 관련된 설정은 없다. 그래서 우리는 new를 사용해서 이것을 간단히 생성한다.

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

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

우리는 이 advisor를 매우 간단히 적용할수 있다. 이것은 설정을 요구하지 않는다(어쨌든 이것은 필요하다. IntroductionAdvisor없이 IntroductionInterceptor를 사용하는 것은 불가능하다). advisor는 상태유지인것처럼 인스턴스별이어야만 한다. 우리는 LockMixinAdvisor의 다른 인스턴스를 필요로 하고 나아가 각각의 advicsed객체를 위해 LockMixin을 필요로 한다. advisor는 advised된 객체의 상태의 일부로 이루어져있다.

우리는 다른 advisor처럼 Advised.addAdvisor() 메소드나 XML설정(추천방법이다)을 사용하여 이 advisor를 프로그램으로 적용할수 있다. 모든 프록시 생성의 선택은 소개(introduction)와 상태유지 믹스인(mixin)을 정확하게 다루는 "자동 프록시 생성자(auto proxy creator)"을 포함해서 아래에서 언급된다.

7.4. Spring내 Advisor API

Spring에서, Advisor는 pointcut표현에 관련된 하나의 advice객체를 포함하는 aspect이다.

소개(introduction)의 특별한 경우와 개별로, 어떤 advisor는 어떤 advice와도 사용될수 있다. org.springframework.aop.support.DefaultPointcutAdvisor는 가장 공통적으로 사용된 advisor클래스이다. 예를 들어, 이것은 MethodInterceptor, BeforeAdvice 나 ThrowsAdvice와 사용될수 있다.

이것은 Spring내 advisor와 advice타입을 같은 AOP프록시로 혼합하는 것이 가능하다. 예를 들어, 당신은 하나의 프록시 설정내 interception around advice, throws advice 와 before advice를 사용할수 있다. Spring은 필요한 생성 인터셉터 연쇄를 자동으로 생성할것이다.

7.5. AOP프록시를 생성하기 위한 ProxyFactoryBean사용하기

만약 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.)

Note

Spring 2.0 AOP 지원도 factory bean을 사용한다.

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.

7.5.1. 기본

다른 Spring 구현물과 같은 ProxyFactoryBean은 indirection의 레벨을 소개한다. 만약 당신이 foo이름을 가진 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.

7.5.2. JavaBean 프라퍼티

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

☒ 프록시 되기를 원하는 대상을 명시

☒ CGLIB를 사용할지에 대한 명시(아래를 보고 Section 7.5.3, “JDK 와 CGLIB 기반의 프록시” 부분도 보라)

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 이 프라퍼티를 true로 셋팅한다면, CGLIB 프록시가 생성될것이다(Section 7.5.3, “JDK 와 CGLIB 기반의 프록시” 를 보라)

☒ optimize: 의욕적인 최적화가 CGLIB를 통해 생성된 프록시에 적용될지를 제어. 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 허용되지 않을지에 대한 값. 디폴트는 false이다(이를테면, 프록시가 설정된후 프록시 설정에 대한 변경이 허용되지 않는다.).

☒ exposeProxy: 현재 프록시가 대상에 의해 접근될수 있는 ThreadLocal으로 나타날수 있는지에 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.

ProxyFactory을 위해 명시하는 다른 프라퍼티는 다음을 포함한다.

☒ proxyInterfaces: Spring인터페이스 이름의 배열. 만약 이것을 제공하지 않는다면 대상 클래스를 위한 CGLIB프록시는 사용될것이다(하지만 Section 7.5.3, “JDK 와 CGLIB 기반의 프록시” 를 보라).

☒ interceptorNames: Advisor의 문자열 배열, 적용하는 인터셉터나 다른 advice명. 정렬은 명백하다. 처음 들어온것을 처음 제공한다. 이것은 리스트내 첫번째 인터셉터가 처음으로 호출을 인터셉터할것이라는 것을 말한다.

  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 인터셉터 이름에 덧붙일수 있다. 이것은 적용될 별표(*) 앞에 부분으로 시작하는 이름으로 모든 advisor 빈즈의 애플리케이션으로 결과를 생성한다. 이 특징을 사용하는 예제는 Section 7.5.6, “‘global’ advisor 사용하기” 에서 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.

7.5.3. JDK 와 CGLIB 기반의 프록시

이 부분은 ProxyFactoryBean이 특정 대상 객체를 위해 JDK 와 CGLIB 기반의 프록시중 하나를 생성하는 것을 선택하는 방법에 대해 명확한 문서를 제공한다.

Note

JDK와 CGLIB기반의 프록시를 생성하는 것과 관련하여 ProxyFactoryBean의 행위는 Spring의 1.2.x와 2.0사이에 변경되었다. ProxyFactoryBean는 현재 TransactionProxyFactoryBean 클래스의 어구처럼 자동-감지 인터페이스와 관련한 유사한 어구를 제시한다.

프록시가 되는 대상 객체의 클래스가 어떠한 인터페이스도 구현하지 않는다면, CGLIB기반의 프록시는 생성될것이다. JDK프록시는 인터페이스 기반이고 인터페이스가 없다는 것은 JDK프록시가 불가능하기 때문에 이것은 가장 쉬운 시나리오이다. 하나는 간단히 대상 bean으로 삽입되고 interceptorNames 프라퍼티를 통해 인터셉터의 목록을 명시한다. ProxyFactoryBean의 proxyTargetClass 프라퍼티가 false로 셋팅된다면 CGLIB기반 프록시가 생성될것이다.(명백하게도 이것은 타당하지 않고 매우 장황하기 때문에 bean정의로부터 제거하는 것이 가장 좋다.)

대상 클래스가 하나(또는 그 이상)의 인터페이스를 구현한다면, 생성된 프록시의 타입은 ProxyFactoryBean의 설정에 의존한다.

ProxyFactoryBean의 proxyTargetClass 프라퍼티가 true로 셋팅된다면, CGLIB기반의 프록시는 생성될것이다. 이것은 타당하고 최소한의 뜻밖의 상황이라는 개념을 유지한다. ProxyFactoryBean의 proxyInterfaces 프라퍼티가 하나 이상의 전체 경로를 포함하는 인터페이스명으로 셋팅된다면, 사실 proxyTargetClass 프라퍼티는 true로 셋팅된다.

ProxyFactoryBean의 proxyInterfaces 프라퍼티가 하나 이상의 전체 경로를 포함한 인터페이스명으로 셋팅된다면, JDK기반의 프록시가 생성될것이다. 생성된 프록시는 proxyInterfaces프라퍼티에 명시된 모든 인터페이스를 구현할것이다. 대상 클래스가 proxyInterfaces 프라퍼티에 명시된것보다 더 많은 인터페이스를 구현한다면, 이것은 잘되고 좋지만 이러한 추가적인 인터페이스는 반환된 프록시에 의해 구현되지 않을것이다.

ProxyFactoryBean의 proxyInterfaces 프라퍼티가 셋팅되지 않지만 대상 클래스가 하나(또는 그 이상) 인터페이스를 구현한다면, ProxyFactoryBean은 대상 클래스가 적어도 하나의 인터페이스를 구현하고 JDK기반의 프록시가 생성된다는 사실을 자동-감지할것이다. 실제로 프록시가 될 인터페이스는 대상 클래스가 구현하는 모든 인터페이스가 될것이다. 사실, 이것은 대상 클래스가 proxyInterfaces 프라퍼티를 위해 구현하는 각각 그리고 모든 인터페이스의 목록을 제공한다. 어쨌든, 이것은 명백히 적은 작업을 수행하고 타이핑도 줄인다.

7.5.4. 프록시 인터페이스

액션내 ProxyFactoryBean의 간단한 예제를 보자. 이 예제는 다음을 포함한다.

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

☒ advisor와 advice를 제공하기 위해 사용되는 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.

Note

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 factory로부터 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 bean은 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 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.

7.5.5. 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.

7.5.6. '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="service"/>
  <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"/>
```

7.6. 간결한 프록시 정의

특별히 트랜잭션 성질을 가지는 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="transactionManager"/>
  <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.

7.7. ProxyFactory로 프로그램으로 AOP프록시를 생성하기.

Spring을 사용해서 프로그램으로 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.

Tip

IoC프레임워크를 사용해서 AOP프록시 생성을 통합하는 것은 대부분의 애플리케이션에서 최상의 선택이다. 우리는 당신이 일반적인 것처럼 AOP를 사용해서 자바코드로 부터 설정을 구체화하는것을 추천한다.

7.8. 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);
```

Note

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.

7.9. "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.

7.9.1. autoproxy bean정의

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

7.9.1.1. BeanNameAutoProxyCreator

BeanNameAutoProxyCreator는 이름과 문자값또는 와일드카드가 들어맞는 bean을 위해 자동적으로 AOP프록시를 생성한다.

```
<bean 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.

7.9.1.2. 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 class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

```
<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
  <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

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.

7.9.1.3. AbstractAdvisorAutoProxyCreator

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

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

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
```

```
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
      <property name="attributes" ref="attributes"/>
    </bean>
  </property>
</bean>

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

DefaultAdvisorAutoProxyCreator bean정의(이름은 명백하지 않다. 나아가 이것은 생략될수 있다.)는 현재 애플리케이션 컨텍스트내 모든 적절한 pointcut을 가져올것이다. 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.)

JPetStore샘플 애플리케이션의 /annotation디렉토리는 JDK 1.5+ annotation이 다루는 자동프록시(auto-proxying)를 위한 유사한 예제를 포함한다. 다음 설정은 bean이 annotation을 포함하기 위한 프록시를 무조건 이끌어내는 Spring의 Transactional annotation의 자동 감지를 가능하게 한다.

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

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
  </property>
</bean>
```

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"/>
```

Tip

> 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 그것과 유사하다.

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

☒ 당신의 사용자 지정 속성 명시하기.

☒ 클래스나 메소드의 사용자정의 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"
    scope="prototype"/>

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

7.10. 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.

Tip

사용자 지정 대상 소스를 사용할때 당신의 대상은 싱글톤 bean정의보다 프로토타입이 될 필요가 있을것이다. 이것은 요구될때 Spring이 새로운 대상 인스턴스를 생성하는것을 허용한다.

7.10.1. 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 id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
  <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="swapper"/>
</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.

7.10.2. 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 1.3을 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"
    scope="prototype">
  ... properties omitted
</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
  <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="poolTargetSource"/>
  <property name="interceptorNames" value="myInterceptor"/>
</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="poolTargetSource"/>
  <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

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

형변환은 다음처럼 보일것이다.

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

Note

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.

7.10.3. 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" ref="businessObjectTarget"/>
</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.

7.10.4. 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"/>
</bean>
```

Note

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.

7.11. 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.

7.12. 추가적인 자원

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.

Chapter 8. Testing

8.1. 소개

Spring팀은 전사적 소프트웨어 개발에서 완전히 필수부분이 되는 테스트를 생각한다.

이 장은 테스트에 관련된 부분이다. 전사적 소프트웨어 영역에서 테스트 처리를 통하는것은 이 장의 범위를 넘어선다. 간단하게는 IoC개념의 채택이 단위 테스트를 위해 가져올수 있는 값 추가에 집중하고 핵심적으로는 Spring프레임워크가 통합 테스트 영역에서 제공하는 명백한 이득에 집중한다.

8.2. 단위 테스팅

의존성 삽입(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 new연산자 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 장점을 따른다면, 당신의 코드기반의 명백한 레이어화와 컴포넌트화가 테스팅을 좀더 쉽게 해준다는 사실을 알게될 것이다. 예를 들어, 당신은 단위 테스트 중에 퍼시스턴트 데이터에 접근할 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. 이것의 결과는 당신이 Spring기반의 애플리케이션에 효과적인 단위 테스트를 작성하도록 도와주는 테스팅 장에서 이 부분은 필요하지 않다는 것이다.

8.3. 통합 테스팅

그러나, 당신의 어플리케이션 서버에 대한 배치가 없거나 전사적 통합 시스템에 대한 실제 연결이 없이 몇몇 통합 테스팅을 실행할 수 있는 것은 매우 중요하다. 이것은 다음과 같은 테스트를 가능하게 할것이다.

☒ 당신의 Spring IoC컨테이너 컨텍스트들을 올바르게 연결하기

☒ JDBC 혹은 ORM 툴을 사용한 데이터 접근. 이것은 정확한 SQL문이나 Hibernate XML맵핑 파일 설정과 같은 것들을 포함할것이다.

Spring은 통합 테스트를 위해 첫번째 클래스 지원을 제공한다. 첫번째 클래스 지원은 Spring배포판에 포함되어 있는 spring-mock.jar 내 많은 수의 클래스로 획득한다. 이 라이브러리내 클래스들은 Cactus와 같은 도구를 사용한 컨테이너 테스팅에 명백한 대안처럼 생각될수 있다.

Note

이 장의 나머지에서 언급된 test클래스의 모든것은 JUnit특성을 지닌다.

org.springframework.test 패키지는 애플리케이션 서버나 다른 배치된 환경을 의지하지 않는 같은 시간동안

Spring 컨테이너를 사용하는 통합 테스팅을 위해 가치있는 수퍼클래스를 제공한다. 그런 테스트들은 다른 특별한 배포단계 없이 JUnit-심지어 IDE 내에서도--에서 실행될 수 있다. 그것들은 단위 테스트보다는 늦게 실행되겠지만, Cacus 테스트 또는 어플리케이션 서버 배포판에 의존적인 원격 테스트보다는 훨씬 빠를 것이다.

이 패키지내 다양한 추상 클래스는 다음의 기능들을 제공한다:

☒ Spring IoC컨테이너는 테스트 케이스 수행간에 캐싱한다.

☒ 테스트 시설 자체의 의존성 삽입.

☒ 통합 테스팅에 적절한 트랜잭션 관리.

☒ 테스팅을 위해 유용한 상속된 인스턴스 변수.

2004년 말이래 많은 수의 Interface21와 다른 프로젝트들은 이러한 접근방식의 강력함과 유용함을 보여왔다. 기능에 있어서의 몇가지 중요한 부분들을 상세하게 살펴보도록 하자.

8.3.1. 컨텍스트 관리와 캐싱

org.springframework.test 패키지는 Spring 컨텍스트의 일관된 로딩과 로딩된 컨텍스트의 캐싱을 제공한다. 후자는 매우 중요한데, 왜냐하면 만약 당신이 큰 프로젝트에서 일을하고 있다면, 시작 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 Hibernate 매핑 파일들을 가진 프로젝트는 언급한 맵핑파일들을 로드하는데 10-20초 가량 소요되고, 모든 한개 단위의 테스트 기법내 모든 한개 단위의 테스트 케이스가 전체 테스트보다 더 늦게 되는 것은 굉장한 생산성 감소를 초래하게 될 것이다.

이 이슈를 할당하기 위해, AbstractDependencyInjectionSpringContextTests는 컨텍스트들의 위치를 제공하기 위해 하위 클래스들이 반드시 구현해야 하는 abstract protected 메소드를 가진다.

```
protected abstract String[] getConfigLocations();
```

이 메소드의 구현물은 애플리케이션을 설정하기 위해 사용되는 XML설정 메타데이타의 자원 위치-대개 클래스패스인-를 가지는 배열을 제공해야만 한다. 이것은 web.xml이나 다른 배치 설정내 명시되는 설정 위치의 목록처럼 같거나 거의 같을것이다.

디폴트로, 한번 로드되면, 설정세트는 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 애플리케이션 객체의 상태를 수정 등으로 인해-- 리로딩을 요청하게 될 때, 당신은 AbstractDependencyInjectionSpringContextTests에 있는 setDirty() 메소드를 호출할 수 있다. 이것은 다음 테스트 케이스를 실행하기 전에 설정을 리로드하고 어플리케이션 컨텍스트를 재생성하게 한다.

8.3.2. 테스트 기능의 의존성 삽입

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() 메소드에서 기술된 설정 파일들의 세트에서 일치되는 객체들을 자동으로 위치시킬 것이다.

액션내 강력한 기능의 간단한 예제를 보자. Title도메인 객체를 말하기 위한 데이터 접근 로직을 수행하는 HibernateTitleDao 클래스를 가지는 시나리오를 생각해보자. 우리는 다음 영역모두를 테스트하는 통합 테스트를 작성하길 원한다.

☒ Spring설정 : 이를테면, HibernateTitleDao에 관련된 모든것이 정확하고 존재하는가..?

☒ Hibernate맵핑 파일 설정 : 이를테면 모든것이 정확하게 맵핑되고 대신 의미를 늦게 정확하게 로드하는가.?

☒ HibernateTitleDao의 로직 : 이를테면 이 클래스가 예상하는것처럼 수행되는가.?

test클래스 자체를 보자.(우리는 부수적인 설정을 볼것이다.)

```
public class HibernateTitleDaoTests extends AbstractDependencyInjectionSpringContextTests  {

    // this instance will be (automatically) dependency injected
    private HibernateTitleDao titleDao;


    // a setter method to enable DI of the 'titleDao' instance variable
    public void setTitleDao(HibernateTitleDao titleDao) {
        this.titleDao = titleDao;
    }


    public void testLoadTitle() throws Exception {
        Title title = this.titleDao.loadTitle(new Long(10));
        assertNotNull(title);
    }

    // specifies the Spring configuration to load for this fixture
    protected String[] getConfigLocations() {
        return new String[] { "classpath:com/foo/daos.xml" };
    }

}
```

getConfigLocations()메소드에 의해 참조되는 부수적인 파일('classpath:com/foo/daos.xml')은 다음처럼 보일것이다.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <!-- this bean will be injected into the HibernateTitleDaoTests class -->
    <bean id="titleDao" class="com/foo/dao/hibernate/HibernateTitleDao">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- dependencies elided for clarity -->
    </bean>

</beans>
```

AbstractDependencyInjectionSpringContextTests는 타입에 의한 autowire를 사용한다(autowiring에 대한 정보는 Section 3.3.6, “Autowiring 협력자” 에서 보라). 그래서 만약 당신이 동일한 타입의 빈 정의들을 여러개 가진다면, 당신은 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 의존성 삽입을 원하지 않는다면, 간단히 setter들을 선언하지 말라. org.springframework.test 패키지의 클래스 구조의 가장 상위인, AbstractSpringContextTests를 확장할수 있다. 이 클래스는 단지 Spring 컨텍스트들을 로드하기 위해 편리한 메소드만을 가지고 있으며 테스트 기능의 어떤 의존성 삽입도 수행하지 않는다.

8.3.3. 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.AbstractTransactionalDataSourceSpringContextTests상위 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 메소드들을 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() 메소드를 호출할 수 있다. 이것은 롤백 대신에 트랜잭션이 커밋되도록 할 것이다.

또한 테스트 케이스가 끝나기 전에 트랜잭션을 종료시키는 편리한 기능이 있는데, endTransaction() 메소드를 호출하면 된다. 이것은 기본적으로 트랜잭션을 롤백시키며, 오로지 이전에 setComplete() 가 호출되었을 때만 커밋시킨다. 이 기능은 만약 당신이 이를테면, 웹 혹은 트랜잭션 외부의 원격티어에서 사용될 Hibernate 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.

8.3.4. 편리한 변수들

당신이 AbstractTransactionalDataSourceSpringContextTests 클래스를 확장할때, 당신은 다음의 protected

인스턴스 변수들에 접근할것이다.

☒ 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, Hibernate Session 인터페이스의 flush() 메소드를 사용해서 말이다.

종종 당신은 통합 테스트를 위해 많은 테스트들에서 사용되는 보다 유용한 변수들을 제공하는 어플리케이션-포괄 상위 클래스를 제공할 것이다.

8.3.5. 예제

Spring 배포판에 포함된 PetClinic 샘플 어플리케이션은 이러한 테스트 상위 클래스들의 사용을 설명해준다.(Spring 1.1.5 이상 버전). 대부분의 테스트 기능은AbstractClinicTests에 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 삽입과 트랜잭션적인 동작을 위해 org.springframework.AbstractTransactionalDataSourceSpringContextTests 를 상속받았다.

☒ clinic 인스턴스 변수--테스트될 어플리케이션 객체--는 setClinic(..) 메소드를 통해 의존성 삽입에 의해

세팅된다.

☒ testGetVets() 메소드는 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 4가지 데이터 접근 기술들을 제공한다.--JDBC, Hibernate, TopLink, 그리고 아파치 OJB. 그래서 AbstractClinicTests 는 컨텍스트 위치들을 기술하지 않는다. 이것은 AbstractDependencyInjectionSpringContextTests의 필수적인 protected abstract 메소드를 구현하는 하위 클래스들에 따라 다르다..

예를 들어, PetClinic 테스트의 Hibernate 구현은 다음의 메소드를 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 삽입에 의해 제공되는--Hibernate를 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.

8.3.6. 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.

8.4. 더 많은 자원

이 부분은 일반적으로 테스팅에 대한 더 많은 자원을 위한 링크를 포함한다.

☒ JUnit 홈페이지.

☒ DbUnit 홈페이지.

☒ Grinder 홈페이지 (로드테스팅 프레임워크).

Part II. Middle Tier Data Access

This part of the reference documentation is concerned with the middle tier, and specifically the data access responsibilities of said tier.

Spring's comprehensive transaction management support is covered in some detail, followed by thorough coverage of the various middle tier data access frameworks and technologies that the Spring Framework integrates with.

☒hapter 9, 트랜잭션 관리

☒hapter 10, DAO support

☒hapter 11, JDBC를 사용한 데이터 접근

☒hapter 12, 객체 관계 맵핑(ORM) 데이터 접근

Chapter 9. 트랜잭션 관리

9.1. 소개

Spring 프레임워크을 사용하기 위한 가장 강제적인 이유중 하나는 트랜잭션 지원이다. Spring 프레임워크는 다음의 이득을 부여하는 트랜잭션 관리를 위한 일관적인 추상화를 제공한다.

☒ JTA, JDBC, Hibernate, JPA, 그리고 JDO와 같은 서로 다른 트랜잭션 API간의 일관적인 프로그래밍 모델을 제공한다.

☒ 선언적인 트랜잭션 관리를 제공한다.

☒ JTA와 같은 많은 개별 트랜잭션 API보다 좀더 간단하고, 사용하기 좀더 쉬운 프로그램으로 처리하는 트랜잭션 관리를 위한 API를 제공한다.

☒ Spring 프레임워크의 다양한 데이터 접근 추상화와 통합한다.

이 장은 많은 부분으로 나뉜다. 각각은 Spring 프레임워크의 트랜잭션 지원의 추가된 값이나 기술중 하나를 언급한다. 이 장은 트랜잭션 관리에 대한 가장 좋은 상황에 대한 몇가지를 모은다(예를 들어, 선언적이고 프로그램으로 처리하는 트랜잭션 관리간의 선택).

☒ 첫번째 부분, 동기은 EJB CMT나 Hibernate와 같은 적절한 API를 통해 트랜잭션을 다루는 것과 대비되는 Spring 프레임워크의 트랜잭션 추상화를 사용하길 원하는 이유를 언급한다.

☒ 두번째 부분, Key 추상화는 다양한 소스로부터 DataSource 인스턴스를 설정하고 얻는 방법만큼 Spring 프레임워크의 트랜잭션 지원내 핵심 클래스의 개요를 말한다.

☒ 세번째 부분, 선언적인 트랜잭션 관리는 선언적인 트랜잭션 관리를 위한 Spring 프레임워크의 지원을 다룬다.

☒ 네번째 부분, 프로그램으로 처리하는 트랜잭션 관리는 프로그램으로 처리(명시적으로 코딩하는)하는 트랜잭션 관리를 위한 Spring 프레임워크의 지원을 다룬다.

9.2. 동기

애플리케이션 서버가 트랜잭션 관리를 위해 필요한가.?

Spring 프레임워크의 트랜잭션 관리 지원은 J2EE애플리케이션이 애플리케이션 서버를 요구할때처럼 전통적인 생각을 명백하게 바꾼다.

특히, 당신은 EJB를 통해 선언적인 트랜잭션을 하기 위해 애플리케이션 서버가 필요하지 않다. 사실, 강력한 JTA기능을 가진 애플리케이션 서버를 가지고 있다면, Spring 프레임워크의 선언적인 트랜잭션을 좀더 강력하게 EJB CMT보다 좀더 생산적인 프로그래밍 모델을 결정할것이다.

다중 트랜잭션 자원을을 지원하고 많은 애플리케이션이 다중 자원이 필수가 아닌 트랜잭션을 다룰때 애플리케이션 서버의 JTA기능만이 필요할 것이다. 예를들어, 많은 높은 수준의 애플리케이션은 하나이고 크게 확장가능한 데이터베이스(Oracle 9i RAC와 같은)를 사용한다.

(물론 당신은 JMS와 JCA와 같은 다른 애플리케이션 서버의 기능이 필요할지로 모른다.)

가장 중요한 것은 당신의 애플리케이션을 가장 큰 형태의 애플리케이션 서버로 확장할때 당신이 선택할수 있는Spring 프레임워크을 사용하는 것이다. EJB CMT나 JTA를 사용하는 것에 대한 대안이 JDBC connection에서와 같이 로컬 트랜잭션을 사용하여 코드를 작성하고 컨테이너-관리 트랜잭션인 전역에서 수행할 코드가 필요하다면 재작업을 해야하는 시기는 지났다. Spring 프레임워크을 사용하여, 당신의 코드를 수정할 필요없이 설정만을 변경할 필요가 있다.

전통적으로, J2EE 개발자들은 트랜잭션 관리에 있어 두 가지 선택사항(전역 또는 로컬 트랜잭션)을 가진다. 전역 트랜잭션은 Java트랜잭션 API(JTA)를 사용하여 애플리케이션 서버에 의해 관리된다. 로컬 트랜잭션은, 예를 들어 JDBC 커넥션과 연관된 트랜잭션처럼, 자원 특성을 따른다. 이 선택은 심오한 의미를 가진다. 예를 들어, 전역 트랜잭션은 다중 트랜잭션 자원(대개 관계형 데이터베이스와 메시지 큐)들을 가지고 동작할 수 있게 해준다. 로컬 트랜잭션을 사용한다면, 애플리케이션 서버는 트랜잭션 관리에 관여하지 않으며, 다중 자원에 걸쳐 정확함을 보증해주지 않는다(이것은 대부분의 애플리케이션들이 하나의 트랜잭션 자원를 사용하기 때문에 그다지 가치가 없다).

전역 트랜잭션. 전역 트랜잭션은 JTA를 사용하기 위해 필요한 코드에서 명백히 불리한 면을 가진다. 그리고 JTA는 사용하기에 번거로운(부분적으로 예외모델 때문에) API이다. 게다가, JTA UserTransaction은 일반적으로 JNDI를 통해 얻어야만 한다. 이것은 우리가 JTA를 사용하기 위해서는 JNDI와 JTA 모두 사용해야만 한다는 것을 의미한다. 명백하게 전역 트랜잭션을 사용하는 것은 JTA가 일반적으로 오로지 애플리케이션 서버 환경에서만 가능한 것처럼, 애플리케이션 코드의 재사용성을 제약할 것이다.이전에, 전역 트랜잭션을 사용하기 위해 선호된 방법은 EJB CMT(Container Managed Transaction)를 통하는 것이었다. CMT는 선언적인 트랜잭션 관리(프로그램으로 처리하는 트랜잭션 관리와 구별되는)의 형태이다. EJB CMT는 물론 JNDI의 사용을 요청하는 EJB자체의 사용에도 불구하고 트랜잭션-관련 JNDT룩업을 위한 필요를 제거한다. 이것은 트랜잭션을 제거하기 위한 Java코드를 작성할 필요를 대부분(비록 전체는 아니다) 제거한다. 명백히 불리한 면은 CMT가 JTA와 애플리케이션 서버 환경에 묶인다는 것이다. EJB나 적어도 EJB 트랜잭션성격을 지니는 EJB외관(facade) 뒤에서 비지니스 로직을 구현하는것을 선택한다면 오직 사용가능하다. 대개 EJB에 관련된 부정적인 면은 특히 선언적인 트랜잭션 관리를 위한 대안이라는 측면에서는 매력적인 계획이 아니다.

로컬 트랜잭션. 로컬 트랜잭션은 사용하기가 더 쉽다. 하지만 명백한 단점을 가진다. 그것들은 다중 트랜잭션 성격을 지니는 자원에 대해서 작동할수 없다. 예를 들어, JDBC connection을 사용하여 트랜잭션을 관리하는 코드는 전역 JTA트랜잭션내 작동할수 없다. 다른 부정적인 면은 로컬 트랜잭션이 프로그래밍 모델의 침해적인 경향이라는 것이다.

Spring 프레임워크는 이러한 문제점들을 해결해준다. Spring 프레임워크는 애플리케이션 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. Spring 프레임워크는 선언적/프로그래밍적 트랜잭션 관리 모두 제공한다. 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.

9.3. 핵심(key) 추상화

Spring 트랜잭션 추상화의 핵심은 트랜잭션 전략(transaction strategy)에 대한 개념이다. 트랜잭션 전략은 아래에서 보이는 org.springframework.transaction.PlatformTransactionManager인터페이스에 의해 정의된다.

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

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

비록 이것이 프로그램으로 처리되어 사용될수 있다고 하더라도 기본적으로 SPI(Service Provider Interface) 인터페이스이다. Spring의 철학으로 유지하여, PlatformTransactionManager는 인터페이스이고 나아가 필요하면 쉽게 mock되거나 stub될수 있다. 더군다나 이것은 JNDI처럼 룩업 전략에 묶이지도 않는다 : PlatformTransactionManager 구현물은 Spring IoC컨테이너에서 다른 객체(또는 bean)처럼 정의된다. 이러한 이득은 심지어 JTA로 작업할때조차도 추상화할 보람이 있도록 해준다는 것이다: 트랜잭션 코드는 직접 JTA를 사용하는 것보다 훨씬 더 쉽게 테스트될 수 있다.

다시 Spring의 철학에서처럼, PlatformTransactionManager인터페이스의 메소드에 의해 던져질수 있는 TransactionException은 체크되지 않았다(unchecked)(이를테면, 이것은 java.lang.RuntimeException 클래스를 확장한다.). 트랜잭션 하부구조의 실패는 대부분 늘 치명적이다. 애플리케이션 코드가 트랜잭션 실패로부터 복구될 수 있는 아주 드문 경우에는, 애플리케이션 개발자는 여전히 TransactionException을 잡고 핸들링하는 것을 선택할 수 있다. 두드러진 점은 개발자가 그렇게 하도록 강요하지 않는다는 것이다.

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

최근 호출 스택내 트랜잭션과 일치한다면, (J2EE 트랜잭션 컨텍스트처럼) TransactionStatus는 실행의 쓰레드와 연관된다.

TransactionDefinition 인터페이스는 명시한다 :

☒ 격리성: 이 트랜잭션을 격리하는 정도는 다른 트랜잭션들의 작업으로부터 가진다. 예를 들어, 이 트랜잭션이 다른 트랜잭션들로부터 커밋되지 않은 쓰기작업 내용을 볼 수 있는가?

☒ 전파(propagation): 일반적으로 트랜잭션 영역 내에서 실행되는 모든 코드는 그 트랜잭션 내에서 실행될 것이다. 그러나, 만약 트랜잭션 컨텍스트가 이미 존재하는 상황에서 트랜잭션적인 메소드가 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. Spring 프레임워크는 EJB CMT로부터 익숙한 트랜잭션 전달 옵션을 제공한다.

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

☒ 읽기전용 상태: 읽기전용 트랜잭션은 어떠한 데이터도 수정하지 않는다. 읽기전용 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();
}
```

Spring에서 선언적이거나 프로그램으로 트랜잭션을 처리하든지, PlatformTransactionManager 구현물을 정의하는 것은 필수이다. 좋은 Spring 형태에서는, 중요한 정의는 의존성삽입을 사용하여 만들어진다.

PlatformTransactionManager 구현물은 대개 작업환경이 JDBC인지, JTA인지, Hibernate인지 등에 대한 지식을 필요로 한다. Spring jPetStore 샘플 애플리케이션의 dataAccessContext-local.xml에서 추출한 다음의 예제는 로컬 PlatformTransactionManager 구현이 정의되는 방법을 보여준다. (이것은 JDBC 환경에서 작동하는 것이다).

우리는 JDBC DataSource를 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="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

PlatformTransactionManager bean 정의는 다음과 같을 것이다 :

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

동일한 샘플 애플리케이션에 있는 ''dataAccessContext-jta.xml'' 파일에서처럼, 만약 우리가 JTA를 사용한다면 JNDI를 경유해 얻어진 우리는 컨테이너 DataSource를 사용할 필요가 있으며, JtaTransactionManager를 구현해야 한다. JtaTransactionManager는 컨테이너의 전역 트랜잭션 관리 구조를 사용할 것이기 때문에, DataSource 혹은 어떤 다른 자원들에 대해 에 대해 알 필요가 없다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

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

```
  <!-- other <bean/> definitions here -->

</beans>
```

Note

'dataSource' bean의 위 정의는 'jee' 명명공간으로 부터 <jndi-lookup/> 태그를 사용한다. 스키마-기반 설정에서 좀더 많은 정보를 위해, Appendix A, XML 스키마-기반 설정를 보라. 그리고 <jee/> 태그에 대해 좀더 많은 정보를 위해, Section A.2.3, "jee 스키마" 를 보라.

우리는 Spring PetClinic 샘플 애플리케이션으로부터 다음의 예제를 보는것처럼 Hibernate 로컬 트랜잭션을 쉽게 사용할수 있다. 이 경우, 우리는 애플리케이션 코드가 Hibernate Session 인스턴스를 얻기 위해 사용할 Hibernate LocalSessionFactoryBean을 정의할 필요가 있다.

DataSource bean정의는 위 예제중 하나와 유사할것이다.(이것이 컨테이너 DataSource라면, 컨테이너보다는 Spring처럼 비-트랜잭션 성격을 지닐것이다.)

이 경우 'txManager' bean은 HibernateTransactionManager 클래스이다. 같은 방법으로 DataSourceTransactionManager는 DataSource에 대한 참조를 필요로 한다. HibernateTransactionManager는 SessionFactory에 대한 참조를 필요로 한다.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
          hibernate.dialect=${hibernate.dialect}
        </value>
  </property>
</bean>

<bean id="txManager" class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Hibernate와 JTA 트랜잭션을 사용하여, 우리는 JDBC 혹은 어떤 다른 자원 전략들처럼 JtaTransactionManager를 그냥 사용하면 된다.

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

이것은 어떤 트랜잭션 자원에 참여하는 전역 트랜잭션처럼, 어떤 자원을 위한 JTA 설정과 동일하다는 점을 명심하라.

이런 모든 경우에서, 애플리케이션 코드는 아무것도 변경될 필요가 없을 것이다. 우리는 로컬에서 전역 트랜잭션으로 혹은 그 반대의 경우 역시 단지 설정을 바꾸는 것만으로 트랜잭션이 관리되는 방법을 변경할 수 있다.

9.4. 트랜잭션으로 resource 동기화하기

다른 트랜잭션 관리자가 생성되는 방법과 트랜잭션에 동기화될 필요가 있는 관련 resource에 연결할 방법이 명백해야만 한다(이를테면, JDBC DataSource에 DataSourceTransactionManager, Hibernate SessionFactory에 HibernateTransactionManager 등등). 여기엔 생성/재사용/cleanup과 관련 PlatformTransactionManager를 통한 트랜잭션 동기화의 개념에서 이러한 resource가 임의로 얻어지고 다루는 것을 확실히 하기 위한 애플리케이션이 영속성(persistence) API(JDBC, Hibernate, JDO 등등)를 직접적이나 우회적으로 사용하는 방법에 대한 질문이 남아있다.

9.4.1. 높은 레벨의 접근법

선호하는 접근법은 Spring의 가장 높은 레벨의 영속성(persistence) 통합 API를 사용하는 것이다. 고유(native) API를 대체할뿐 아니라, resource 생성/재사용, cleanup, resource의 선택적인 트랜잭션 동기화 그리고 예외 맵핑을 내부적으로 다룬다. 그래서 사용자 데이터 접근코드는 이러한 사항에 대한 걱정을 전혀 하지 않는다. 하지만 영속성 로직에 전적으로 집중할수 있다. 대개 같은 템플릿 접근법은 다음(JdbcTemplate, HibernateTemplate, JdoTemplate과 같은 클래스)의 모든 영속성 API이다. 이러한 통합 클래스는 이 메뉴얼의 다음 장에서 상세화된다.

9.4.2. 낮은 레벨의 접근법

낮은 레벨에는 DataSourceUtils (JDBC를 위한), SessionFactoryUtils (Hibernate를 위한), PersistenceManagerFactoryUtils (JDO를 위한)와 같은 클래스가 존재한다. 고유 영속성(persistence) API의 resource타입을 가지고 직접적으로 다루기 위한 애플리케이션 코드를 선호할때, 이러한 클래스는 Spring 프레임워크-관리 인스턴스가 얻어지고, 트랜잭션이 동기화되고, 영속성 API에 맵핑되는 프로세스에서 발생하는 예외를 확인한다.

예를 들어, JDBC를 위해, DataSource에서 getConnection()메소드를 호출하는 전통적인 JDBC접근법 대신에, 당신은 다음처럼 Spring의 org.springframework.jdbc.datasource.DataSourceUtils를 대신 사용할것이다.

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

만약 트랜잭션이 존재하고 동기화된 connection을 가진다면, 인스턴스는 반환될것이다. 반면에, 메소드 호출은 존재하는 트랜잭션에 (선택적으로)동기화되고 같은 트랜잭션내 재사용하기 위해 사용가능한 새로운 connection의 생성을 처리할것이다. 언급된것처럼, 이것은 SQLException이 체크되지 않은 DataAccessExceptions의 Spring구조중 하나인 CannotGetJdbcConnectionException로 포장되는 추가된 장점을 가진다. 이것은 당신에게 SQLException으로 부터 쉽게 얻고 데이터베이스간의(다른 영속성 기술간의) 이식성을 확실시할수 있는 좀더 많은 정보를 줄것이다.

이것은 Spring트랜잭션 관리 없이도 잘 작동한다. 그래서 당신은 트랜잭션 관리를 위해 Spring을 사용하든지 말든지 이것을 사용할수 있다.

물론, 당신이 Spring의 JDBC지원이나 Hibernate지원을 사용했을때, 관련 API에 직접적으로 작업하기 보다는 Spring추상화를 통해 좀더 좋게 작업할수 있기 때문에, 당신은 대개 DataSourceUtils나 다른 헬퍼 클래스를 사용하지 않는것을 선호할것이다. 예를 들어, 당신이 JDBC사용을 단순화하기 위해 Spring JdbcTemplate이나 jdbc.object패키지를 사용한다면, 정확한 connection 복구(retrieval)가 발생하고 당신은 어떤 특별한 코드를 작성할 필요가 없을것이다.

9.4.3. TransactionAwareDataSourceProxy

매우 낮은 레벨에 TransactionAwareDataSourceProxy클래스가 존재한다. 이것은 Spring관리 트랜잭션의 인지를

추가하는 목표 DataSource를 위한 프록시이다. 이 점에서 J2EE서버가 제공하는 전통적인 JNDI DataSource와 유사하다.

호출되어야만 하는 코드가 존재하고 표준 JDBC DataSource인터페이스 구현물이 전달될때를 제외하고 이 클래스를 사용하는 것이 결코 필요하거나 바람직하지는 않다. 이 경우, 재사용될수 있는 코드를 가지는 것은 가능하다. 하지만 Spring관리 트랜잭션내 포함된다. 위에서 언급된 더 높은 레벨의 추상화를 사용하여 당신 자신의 새로운 코드를 작성하는 것이 선호된다.

9.5. 선언적인 트랜잭션 관리

대부분의 Spring사용자는 선언적인 트랜잭션 관리를 선택한다. 이것은 애플리케이션 코드의 가장 최소의 영향을 가지는 것은 선택사항이고 나아가 비-침략적 경량 컨테이너의 목표를 가지고 일관적이다.

Spring의 선언적인 트랜잭션 관리는 Spring AOP로 가능하다. 비록, 트랜잭션 성격을 가지는 aspect가 Spring에서 나오고 진부한 반복형태로 사용된다고 하더라도, AOP개념은 이 코드의 효과적인 사용을 위해 대개 이해되지 않는다.

EJB CMT를 검토하여 시작하고 Spring 선언적인 트랜잭션 관리의 유사함과 차이점을 설명하는 것이 도움이 된다. 기본적인 접근법은 유사하다. 트랜잭션 행위를 개별 메소드 레벨로 명시하는 것은 가능하다. 필요하다면 트랜잭션 컨텍스트내 setRollbackOnly()를 호출하는 것이 가능하다. 차이점은 :

☒ JTA에 묶이는 EJB CMT와는 달리, Spring 선언적인 트랜잭션 관리는 어떠한 환경에서도 작동한다. 이것은 설정상의 변경만을 하면서 JDBC, JDO, Hibernate 또는 다른 트랜잭션과 작동할수 있다.

☒ Spring 프레임워크는 EJB와 같이 특정 클래스에만이 아닌 선언적인 트랜잭션 관리가 어느 클래스에 적용되는것이 가능하다.

☒ Spring 프레임워크는 선언적인 롤백 규칙을 제공한다. EJB가 없는 기능은 우리가 아래에서 언급할것이다. 롤백은 프로그램으로가 아닌 선언적으로 제어될수 있다.

☒ Spring 프레임워크는 AOP를 사용하여, 트랜잭션 성격을 지니는 행위를 사용자정의하는 기회를 준다. 예를 들어, 트랜잭션 롤백의 경우 사용자정의 행위를 추가하길 원한다면, 당신은 할수 있다. 당신은 트랜잭션 성격을 지니는 advice에 따라 임의의 advice를 추가할수 있다. EJB CMT로, 당신은 setRollbackOnly()보다 컨테이너의 트랜잭션 관리에 영향을 줄 방법을 가지지 않는다.

☒ Spring 프레임워크는 높은 수준의 애플리케이션 서버가 하는 것처럼 원격 호출에 대해 트랜잭션 컨텍스트의 위임을 지원하지 않는다. 이 기능이 필요하다면, 우리는 EJB를 사용하도록 추천한다. 어쨌든, 이러한 기능을 사용하기전에 주의깊게 검토하라. 대개, 우리는 원격 호출을 확장하기 위한 트랜잭션을 원하지 않는다.

> TransactionProxyFactoryBean는 어디인가?
>
> Spring 2.0와 그 상위에서 선언적인 트랜잭션 설정은 Spring의 이전버전과 다르다. 중요한 차이점은 TransactionProxyFactoryBean bean을 더이상 설정할 필요가 없다는 것이다.
>
> 예전, Spring 2.0의 설정 스타일은 자체적으로 TransactionProxyFactoryBean bean을 간단히 정의하는 것처럼 새로운 <tx:tags/>의 생각을 다루는 100% 유효한 설정이다.

롤백 규칙의 개념은 중요하다. 그것들은 우리에게 예외가 자동 롤백을 야기하는 것을 명시하는 것을 가능하게 한다. 우리는 이것을 Java코드가 아닌 설정에서 선언적으로 명시한다. 그래서 우리가 현재

트랜잭션을 프로그램으로 롤백하기 위해 TransactionStatus객체에서 여전히 setRollbackOnly() 를 호출할수 있다. 우리는 MyApplicationException이 결과적으로 언제나 롤백하는 규칙을 명시할수 있다. 이것은 비지니스 객체가 트랜잭션 하위구조에 의존할 필요가 없는 명백한 장점을 가진다. 예를 들어, 그것들은 Spring API를 import할 필요가 없다.

EJB 디폴트 행위는 시스템 예외(언제나 런타임 예외인)에서 트랜잭션을 자동적으로 롤백하기 위한 EJB컨테이너를 위하는 동안, EJB CMT는 애플리케이션 예외(이를테면, java.rmi.RemoteException보다는 체크된 예외)에서 자동적으로 트랜잭션을 롤백하지 않는다. 선언적인 트랜잭션 관리를 위한 Spring 디폴트 행위가 EJB규칙(체크되지 않은 예외에서만 자동적으로 롤백이 된다)을 따르는 동안, 이것은 종종 사용자정의하는 것이 유용하다.

9.5.1. Spring의 선언적인 트랜잭션 구현물을 이해하기

이 부분의 목적은 때때로 선언적인 트랜잭션의 사용과 관련된 신비함을 풀어내는 것이다. @Transactional 어노테이션을 가지고 당신의 클래스를 추석처리하기 위한 참조문서이다. 당신의 설정에 ('<tx:annotation-driven/>')를 추가하고 이것이 작동하는 방법을 이해하길 바란다. This section will explain the inner workings of Spring's declarative transaction infrastructure to help you navigate your way back upstream to calmer waters in the event of transaction-related issues.

Tip

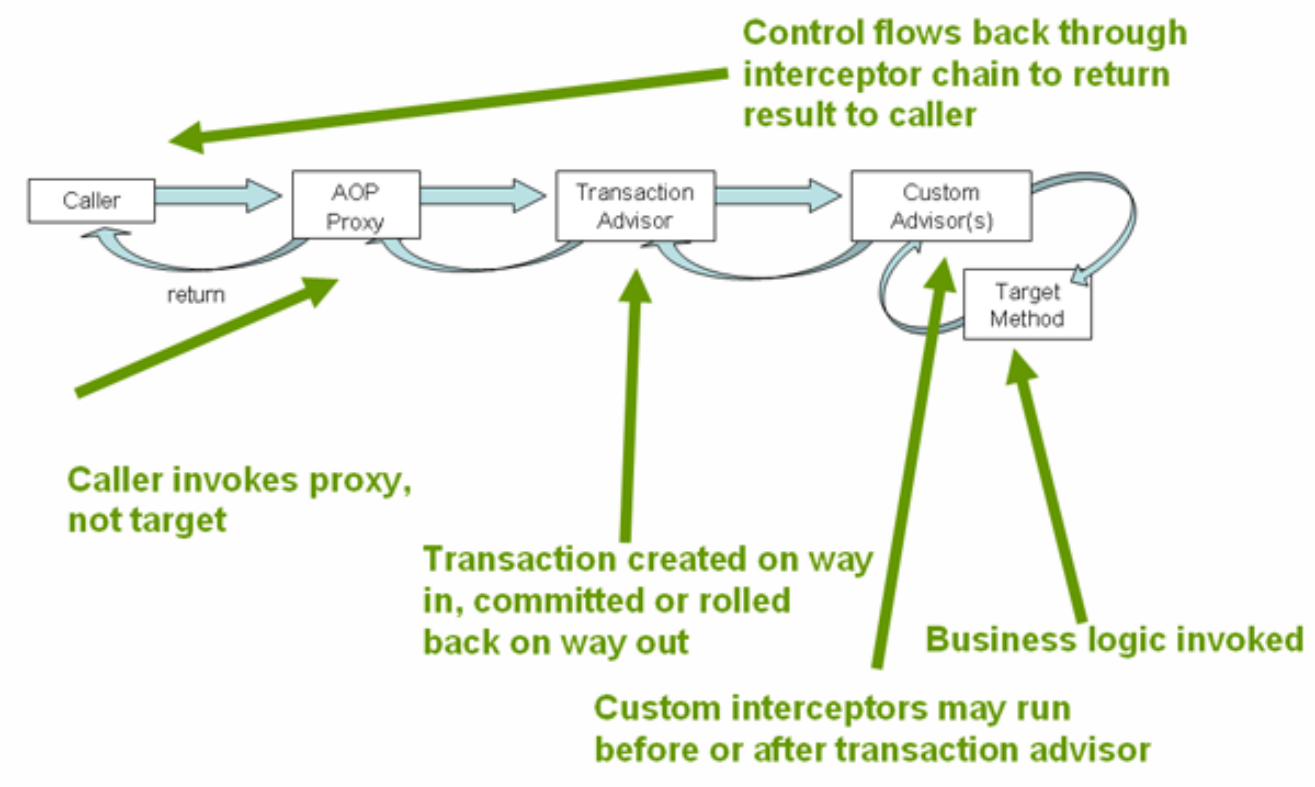
Spring 소스코드를 보는 것은 Spring이 트랜잭션 지원에 대해 실제로 이해하기 위한 좋은 방법이다. 당신은 Javadoc정보를 완벽하게 찾을수 있다. 우리는 개발기간동안 어떻게 이루어지는지 좀더 제대로 보기 위해 Spring-가능한 애플리케이션에서 로깅레벨을 'DEBUG'로 두기를 제안한다.

Spring의 선언적인 트랜잭션 지원에 관련하여 가장 중요한 개념은 이 지원이 AOP 프록시를 통해서만 가능하고, 트랜잭션 성격을 가지는 advice는 메타데이타(XML- 이나 어노테이션-기반의)에 의해 다루어진다는 것이다. 트랜잭션 성격을 가지는 메타데이타를 가진 프록시의 조합은 메소드 호출에 대해 트랜잭션을 다루는 적절한 PlatformTransactionManager 구현물과 함께 TransactionInterceptor를 사용하는 AOP프록시를 만들어낸다.

Note

비록 AOP의 지식은 Spring의 선언적인 트랜잭션 지원을 사용하기 위해 필수는 아니다. 이것을 도울수 있을 뿐이다. Spring AOP는 Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍에서 전반적으로 다루어진다.

개념적으로, 트랜잭션 성격을 가지는 프록시에서 메소드를 호출하는 것은 다음과 같을것이다.



9.5.2. 첫번째 예제

다음의 인터페이스와 관련 구현물을 보라. 의도는 개념을 전달한다. Foo 와 Bar를 사용하는 것은 당신이 트랜잭션 사용에 집중하고 도메인 모델에 대해 걱정할 필요가 없다는 것을 의미한다.

```
<!-- the service interface that we want to make transactional -->

package x.y.service;

public interface FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);

}
```

```
<!-- an implementation of the above interface -->

package x.y.service;

public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }

    public Foo getFoo(String fooName, String barName) {
        throw new UnsupportedOperationException();
    }

    public void insertFoo(Foo foo) {
```

```
        throw new UnsupportedOperationException();
    }

    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }
}
```

(이 예제의 목적을 위해, 사실 구현물 클래스 (DefaultFooService)는 각각의 구현된 메소드의 몸체에서 UnsupportedOperationException을 던진다. 이것은 생성된 트랜잭션을 보도록 허용하고 던져지는 UnsupportedOperationException 인스턴스에 대한 응답으로 롤백한다.

FooService 인터페이스의 첫번째 두개의 메소드(getFoo(String) 와 getFoo(String, String))가 읽기전용으로 트랜잭션내에서 수행된다고 가정해보자. 그리고 다른 메소드(insertFoo(Foo) 와 updateFoo(Foo))는 읽고 쓰기가 가능한 트랜잭션내 수행된다고 가정한다.

XML-기반의 메타데이타를 사용하여 선언적인 형태로 이 상황을 설정하기 위해, 당신은 다음의 설정을 작성할 것이다.

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

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

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

  <!-- don't forget the DataSource -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
  </bean>

  <!-- similarly, don't forget the (particular) PlatformTransactionManager -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
  </bean>
```

```
  <!-- other <bean/> definitions here -->

</beans>
```

위 설정을 신중하게 보자. 우리는 트랜잭션 성격을 가지길 원하는 서비스 객체('fooService' bean)를 가진다. 우리가 적용하길 원하는 트랜잭션 의미는 <tx:advice/>정의에 캡슐화된다. <tx:advice/>정의는 "'get'으로 시작하는 모든 메소드는 읽기전용 트랜잭션에서 수행하고 다른 모든 메소드는 디폴트 트랜잭션으로 수행된다." 라고 읽는다. <tx:advice/> 태그의 'transaction-manager' 속성은 트랜잭션을 실제로 다루는(이 경우 'txManager' bean) PlatformTransactionManager bean의 이름으로 셋팅된다.

Tip

PlatformTransactionManager의 bean이름이 'transactionManager' 이름을 가져서 묶이길 원한다면 당신은 실제로 트랜잭션 성격을 가지는 advice(<tx:advice/>) 내 'transaction-manager' 속성을 생략할수 있다. PlatformTransactionManager bean이 다른 이름을 가져서 묶이길 원한다면, 명시적으로 가지고 위 예제처럼, 'transaction-manager' 속성을 사용한다.

설정의 마지막은 <aop:config/> 정의이다. 이것은 'txAdvice' bean에 의해 정의되는 트랜잭션 성격을 가지는 advice가 애플리케이션내 적절한 지점에서 수행된다는 것을 확신한다. 첫번째 우리는 FooService 인터페이스에 정의된 어떤 작업의 수행에 일치하는 pointcut(우리는 이 pointcut 'fooServiceOperation'을 말한다.)를 정의한다. 그리고나서 우리는 advisor를 사용하여 'txAdvice'를 가진 pointcut을 조합한다. 결과는 'fooServiceOperation' 수행에 표시한다. 'txAdvice'에 의해 정의된 advice는 작동할것이다.

<aop:pointcut/>요소내 정의된 표현은 AspectJ pointcut표현이다. Spring 2.0내 pointcut표현에 대한 좀더 상세한 정보를 위해서는 Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍를 보라.

공통의 요구사항은 전체 서비스 레이어를 트랜잭션 성격을 가지도록 만드는 것이다. 이것을 하기 위해 가장 좋은 방법은 서비스 레이어내 어떤 작업(operation)에 일치하는 pointcut표현을 간단히 변경하는 것이다. 예를 들면 :

```
<aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
  </aop:config>
```

(이 예제는 당신의 모든 서비스 인터페이스가 'x.y.service' 패키지내 정의된다고 가정한다. 다시 말해. 좀더 상세한 정보를 위해서는 Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍를 보라.)

지금 우리는 설정을 꼼꼼히 살펴보고 있다. 당신은 스스로에게 물어볼지도 모른다. "좋아.. 하지만 이 설정 모두가 실제로 무엇을 하는가.?" .

위 설정이 하는것은 'fooService' bean 정의로부터 생성된 객체에 대해 트랜잭션 성격을 가지는 프록시를 생성하는것이다. 프록시는 트랜잭션 성격을 가지는 advice로 설정될것이다. 그래서 적절한 메소드가 프록시에서 호출될때, 트랜잭션은 메소드에 관련된 트랜잭션에 의존하여 아마도 시작되고, 일시정지하고, 읽기전용으로 표시된다.

위 설정을 테스트하기 위한 다음의 드라이버 프로그램을 보자.

```
public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);
        FooService fooService = (FooService) ctx.getBean("fooService");
```

```
        fooService.insertFoo (new Foo());
    }
}
```

위 프로그램을 실행하여 얻은 출력은 다음과 같을것이다. (Log4J출력과 DefaultFooService 클래스의 insertFoo(..) 메소드에 의해 던져진 UnsupportedOperationException으로부터 관련 스택추적 메시지는 명백하게 짤린다.)

```
    <!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy
        for bean 'fooService' with 0 common interceptors and 1 specific interceptors
    <!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]

    <!-- ... the insertFoo(..) method is now being invoked on the proxy -->

[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo
    <!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name [x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction

    <!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should
        rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo
        due to throwable [java.lang.UnsupportedOperationException]

   <!-- and the transaction is rolled back (by default, RuntimeException instances cause rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException
        at x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)
   <!-- AOP infrastructure stack trace elements removed for clarity -->
        at $Proxy0.insertFoo(Unknown Source)
        at Boot.main(Boot.java:11)
```

9.5.3. 롤백

The previous section outlined the basics of how to specify the transactional settings for the classes, typically service layer classes, in your application in a declarative fashion. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The easiest (and indeed recommended) way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an Exception from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code will catch any unhandled Exception as it bubbles up the call stack, and will mark the transaction for rollback.

However, please note that the Spring Framework's transaction infrastructure code will, by default, only mark a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception an instance or subclass of RuntimeException. (Errors will also - by default - result in a rollback.) Checked exceptions that are thrown from a transactional

method will not result in the transaction being rolled back.

So much for the default settings; these settings, namely exactly which Exception types mark a transaction for rollback, can of course be changed. Find below a snippet of XML configuration that demonstrates how one would configure rollback for a checked, application-specific Exception type.

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
        <tx:method name="get*" read-only="false" rollback-for="NoProductInStockException"/>
        <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

The second way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to do so programmatically Although very simple, this way is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure. Find below a snippet of code that does programmatic rollback of a Spring Framework-managed transaction:

```
public void resolvePosition() {
    try {
        // some business logic...
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

You are strongly encouraged to use the declarative approach to rollback if at all possible. The programmtic means of rolling back is, as you can see, available to you should you need it, but it's usage flies in the face of achieving a clean POJO-based model in your application.

9.5.4. 다른 bean에 다른 트랜잭션 성격을 가지는 의미를 설정하기

많은 서비스 레이어 객체를 가지고 이러한 객체의 각각의 그룹에 전체적으로 다른 트랜잭션의 의미를 적용하길 원하는 상황을 생각해보자. 당신은 'pointcut' 와 'advice-ref' 속성값을 가지는 많은 수의 다른 <aop:advisor/>요소를 정의하여 Spring에 영향을 끼칠수 있다.

예제의 방법으로, 모든 서비스 레이어 클래스가 가장 상위 'x.y.service' 패키지내 정의된다고 가정하자. 클래스의 인스턴스인 모든 bean을 만들기 위해 언급된 패키지내 정의하고 디폴트 트랜잭션의 의미를 가지는 'Service'로 끝나는 이름을 가진 bean을 만들기 위해 당신은 다음의 설정을 작성할수 있다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <aop:config>

        <aop:pointcut id="serviceOperation"
```

```
                expression="execution(* x.y.service..*Service.*(..))"/>

        <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>

    </aop:config>

    <!-- these two beans will be transactional... -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>
    <bean id="barService" class="x.y.service.extras.SimpleBarService"/>

    <!-- ... and these two beans won't -->
    <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right package) -->
    <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in 'Service') -->

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

아래는 전체적으로 다른 트랜잭션 셋팅을 가지고 두개의 구별되는 bean을 설정하는 예제이다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <aop:config>

        <aop:pointcut id="defaultServiceOperation"
                expression="execution(* x.y.service.*Service.*(..))"/>
        <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>

        <aop:pointcut id="noTxServiceOperation"
                expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>
        <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

    </aop:config>

    <!-- this bean will be transactional (c.f. the 'defaultServiceOperation' pointcut) -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this bean will also be transactional, but with totally different transactional settings -->
    <bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

    <tx:advice id="defaultTxAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <tx:advice id="noTxAdvice">
        <tx:attributes>
            <tx:method name="*" propagation="NEVER"/>
```

```
        </tx:attributes>
    </tx:advice>

    <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

9.5.5. <tx:advice/> settings

This section summarises the various transactional settings that can be specified using the <tx:advice/> tag. The default <tx:advice/> settings are:

☒ The propagation setting is REQUIRED

☒ The isolation level is DEFAULT

☒ The transaction is read/write

☒ The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported

☒ Any RuntimeException will trigger rollback, and any checked Exception will not

These default settings can, of course, be changed; the various attributes of the <tx:method/> tags that are nested within <tx:advice/> and <tx:attributes/> tags are summarized below:

Table 9.1. <tx:method/> settings

Attribute	Required?	Default	Description
name	Yes		The method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, 'get*', 'handle*', 'on*Event', etc.
propagation	No	REQUIRED	The transaction propagation behavior
isolation	No	DEFAULT	The transaction

Attribute	Required?	Default	Description
			isolation level
timeout	No	-1	The transaction timeout value (in seconds)
read-only	No	false	Is this transaction read-only?
rollback-for	No		The Exception(s) that will trigger rollback; comma-delimited. For example, 'com.foo.MyBusinessException,ServletExc
no-rollback-for	No		The Exception(s) that will not trigger rollback; comma-delimited. For example, 'com.foo.MyBusinessException,ServletExc

9.5.6. @Transactional 사용하기

Note

@Transactional어노테이션에 의해 제공되는 기능과 관련된 지원 클래스는 당신이 적어도 Java 5를 사용할때만 사용가능하다.

트랜잭션 설정에 대해 XML기반의 선언적인 접근법에 추가적으로, 당신은 트랜잭션 설정에 대해 어노테이션-기반의 선언적인 접근법을 사용할수 있다.

Java소스코드내 직접적으로 트랜잭션 구문을 선언하는 것은 선언을 영향을 끼치는 코드에 좀더 근접하게 두는것이다. 불필요한 커플링의 위험이 더 많지 않다. 트랜잭션 성질을 가지면서 배치되는 코드는 언제나 이러한 방법으로 배치된다.

@Transactional 어노테이션의 사용으로 사용하기 쉬운것은 예제에서 가장 잘 표현되었다. 다음의 인터페이스 정의를 보라.

```
<!-- the service class that we want to make transactional -->
@Transactional
public class DefaultFooService implements FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);
}
```

Spring 컨테이너내 bean처럼 정의된 위 POJO는 XML설정에서 한줄을 추가하여 트랜잭션 성질을 가질수 있다.

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- enable the configuration of transactional behavior based on annotations -->
  <tx:annotation-driven transaction-manager="txManager"/>

  <!-- a PlatformTransactionManager is still required -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- (this dependency is defined somewhere else) -->
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

Tip

당신이 묶기를 원하는 PlatformTransactionManager의 bean이름이 'transactionManager'이름을 가진다면 <tx:annotation-driven/> 태그내 'transaction-manager' 속성을 생략할수 있다. 당신이 묶길 원하는 PlatformTransactionManager bean이 다른 이름을 가진다면, 명시적이거나 위 예제에서처럼 'transaction-manager' 속성을 사용해야만 한다.

Method visibility and @Transactional

The @Transactional annotation should only be applied to methods with public visibility. If you do annotate protected, private or package-visible methods with the @Transactional annotation, no error will be raised, but the annotated method will not exhibit the configured transactional settings.

@Transactional 어노테이션은 인터페이스 정의, 인터페이스의 메소드, 클래스 정의, 또는 클래스의 public 메소드 앞에 둘것이다. 단지 @Transactional 어노테이션의 존재로는 트랜잭션 성격을 가지는 행위를 활성화하기에는 충분하지 않다. @Transactional 어노테이션은 @Transactional을 인식하는 어떤것에 의해 소비될수 있고 트랜잭션 성격을 가지는 행위를 적용하기 위한 트랜잭션 성격을 가지는 메타데이타를 사용할수 있는 간단한 메타데이타이다. 위 예제의 경우, 이것은 트랜잭션 성격을 가지는 행위를 교체하는 <tx:annotation-driven/> 요소의 존재이다.

The Spring team's recommendation is that you place the @Transactional annotation on the concrete class (or method of a concrete class), as opposed to on any interface(s) that the class may implement. You certainly can place the @Transactional annotation on an interface, but

this will only work as you would expect it to if you are using interface-based proxies. Because annotations are not inherited it means that if you are using class-based proxying then the transaction settings will not be recognised by the class-based proxying infrastructure and the object will not be wrapped in a transaction proxy (which would be decidedly bad). So please do take the Spring teams's advice and use the @Transactional annotation on concrete classes.

Note

When using the @Transactional style of declarative transaction demarcation you can control whether or not interface- or class-based proxies are created via the presence and value of the "proxy-target-class" attribute on the attendant <tx:annotation-driven/> element. If the value of the boolean-style value of the "proxy-target-class" attribute is set to "true", then class-based proxying will be in effect (and this currently requires the presence of the CGLIB library (cglib.jar) on the classpath). If the value of the "proxy-target-class" attribute is set to "false" or if the attribute is omitted, then standard JDK interface-based proxying will be in effect.

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the DefaultFooService class is annotated with the settings for a read-only transaction, but the @Transactional annotation on the updateFoo(Foo) method in the same class takes precedence over the transactional settings defined in the class level annotation.
아래와 비교..??

Spring 프레임워크의 중심적인 주의(tenets)를 유지한체, 상속에 관련하여 Spring의 @Transactional 어노테이션의 처리는 이해된다. @Transactional 어노테이션을 가지고 클래스 레벨에서 인터페이스를 주석처리한다면, 인터페이스의 모든 구현물은 인터페이스에 적용되는 트랜잭션 셋팅을 상속할 것이다. 이것은 결코 상속되지 않는 인터페이스와 메소드의 어노테이션에 적용하는 대개의 구문에 대한 직접적인 반대개념이다. Spring으로, 당신은 자체적인 @Transactional 값을 명시하여 수퍼클래스나 인터페이스로부터 상속된 디폴트 트랜잭션 셋팅을 오버라이드할수 있다. 기본적으로, 대개의 유래된 위치는 메소드의 트랜잭션 의미를 평가할때 우선권을 가진다. 다음의 예제의 경우, FooService 인터페이스는 디폴트 트랜잭션 셋팅을 가지고 주석처리되지만, DefaultFooService 클래스의 updateFoo(Foo)에 있는 @Transactional 어노테이션은 FooService 인터페이스로부터 상속된 트랜잭션 성격을 가지는 셋팅에 대해 우선권을 가진다.

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // do something
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
}
```

9.5.6.1. @Transactional 셋팅

@Transactional 어노테이션은 트랜잭션 성격을 가지는 인터페이스, 클래스 나 메소드를 명시하는 메타데이터이다. 예를 들어, "이 메소드가 호출되고 존재하는 트랜잭션을 다시 실행할때 새로운 읽기전용

트랜잭션을 새기는 것을 시작한다.” . 디폴트 @Transactional 셋팅은

☒ 위임 셋팅은 PROPAGATION_REQUIRED이다.

☒ 격리 레벨은 ISOLATION_DEFAULT이다.

☒ 트랜잭션은 읽기/쓰기이다.

☒ 트랜잭션 타임아웃은 참조하는 트랜잭션 시스템의 디폴트 타임아웃에 디폴트하거나 타임아웃이 지원되지 않는다면 없다.

☒ 어떤 RuntimeException 이 롤백하고 어떤 체크된 Exception은 그렇지 않다.

이러한 디폴트 셋팅은 물론 변경될수 있다. @Transactional 어노테이션의 다양한 프라퍼티는 다음 테이블에서 개요화된다.

트랜잭션 셋팅을 변경하는 어노테이션의 선택적인 프라퍼티

Table 9.2. @Transactional 프라퍼티

프라퍼티	타입	상세설명
위임(propagation)	enum: Propagation	선택적인 위임 셋팅
격리(isolation)	enum: Isolation	선택적인 위임 레벨
읽기전용(readOnly)	boolean	읽기/쓰기 대 읽기전용 트랜잭션
타임아웃(timeout)	int (초단위)	트랜잭션 타임아웃
rollbackFor	Throwable로부터 유래되는 Class 객체의 배열	롤백을 야기해야하는 예외 클래스의 선택적인 배열.
rollbackForClassname	클래스명의 배열. Throwable로부터 유래되는 클래스	롤백을 야기해야하는 예외 클래스 이름의 선택적인 배열
noRollbackFor	Throwable로부터 유래된 Class 객체의 배열	롤백을 야기하지 않는 예외 클래스의 선택적인 배열.
noRollbackForClassname	Throwable로부터 유래되는 String 클래스 명의 배열	롤백을 야기하지 않는 예외 클래스의 선택적인 배열.

우리는 위 프라퍼티와 관련값을 좀더 상세하게 언급하는 @Transactional 어노테이션을 위한 Javadoc을 보길 권한다.

9.5.7. 트랜잭션 성격을 가지는 작업에 충고하기(advise)

클래스의 인스턴스를 가지는 상황이라고 생각해보자. 그리고 당신은 트랜잭션 성격을 가지고 몇가지 기본적인 프로파일 advice를 수행하길 원할것이다. <tx:annotation-driven/>를 사용하는 면에서 어떻게 영향을 주는가.?

우리가 updateFoo(Foo) 메소드를 호출할때 보길 원하는 것은

☒ 설정된 프로파일링 aspect가 시작되고,

☒ 트랜잭션 성격을 가지는 advice가 수행되고,

☒ advised객체의 메소드가 수행되고

☒ 트랜잭션이 커밋되고,

☒ 프로파일링 aspect가 전체 메소드 호출이 소요하는 시각이 정확이 어떻게 되는지 리포팅한다

Note

이 장은 설명된 AOP와 연관되지 않는다(트랜잭션에 적용되는것을 제외하고). 다음의 AOP설정의 일부를 상세히 다루기 위해 Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍를 보라.

이것은 간단한 프로파일링 aspect를 위한 코드이다. (advice의 순서는 Ordered 인터페이스를 통해 제어된다. advice순서에 대한 완전한 상세설명을 위해, Section 6.2.4.7, “Advice ordering” 를 보라.)

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

    private int order;

    // allows us to control the ordering of advice
    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    // this method is the around advice
    public Object profile(ProceedingJoinPoint call) throws Throwable {
        Object returnValue;
        StopWatch clock = new StopWatch(getClass().getName());
        try {
            clock.start(call.toShortString());
            returnValue = call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
        return returnValue;
    }
}
```

이것은 우리가 원하는 영향을 줄 관련 설정이다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
```

```
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

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

    <!-- this is the aspect -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <tx:annotation-driven transaction-manager="txManager"/>

    <aop:config>
        <!-- this advice will execute around the transactional advice -->
        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                    expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>
    </aop:config>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
    </bean>

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

</beans>
```

위 설정의 결과는 프로파일링된 'fooService' bean와 순서대로 이것에 적용된 트랜잭션 성격을 가지는 aspect일것이다. 추가적인 aspect의 설정은 유사한 형태로 영향을 끼친다.

마지막으로, XML 선언 접근법을 사용하지만 위와 같은 셋업에 영향을 끼치기 위한 예제 설정을 아래에서 보라.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

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

    <!-- the profiling advice -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <aop:config>

        <aop:pointcut id="entryPointMethod" expression="execution(* x.y..*Service.*(..))"/>
```

```
        <!-- will execute after the profiling advice (c.f. the order attribute) -->
        <aop:advisor
                advice-ref="txAdvice"
                pointcut-ref="entryPointMethod"
                order="2"/> <!-- order value is higher than the profiling aspect -->

        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                    expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>

    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other <bean/> definitions such as a DataSource and a PlatformTransactionManager here -->

</beans>
```

위 설정의 결과는 프로파일링된 'fooService' bean와 순서대로 이것에 적용된 트랜잭션 성격을 가지는 aspect일것이다. 추가적인 aspect의 설정은 유사한 형태로 영향을 끼친다. 입구에서 트랜잭션 성격을 가지는 advice 뒤에서와 출구에서 트랜잭션 성격을 가지는 advice 이전에 수행할 프로파일링 advice를 원한다면, 트랜잭션 성격을 가지는 advice의 order값보다 더 큰 값과 같이 프로파일링 aspect bean의 'order' 프라퍼티를 간단히 교체할것이다.

많은 수의 추가적인 aspect의 설정은 유사한 형태로 영향을 끼친다.

9.5.8. AspectJ와 @Transactional 를 사용하기

AspectJ aspect의 방법에 의해 Spring 컨테이너의 외부에서 Spring의 @Transactional 지원을 사용하는 것은 가능하다. 이 지원을 사용하기 위해 당신은 먼저 타입과 @Transactional 어노테이션을 가진 메소드를 주석처리하고 spring-aspects.jar파일내 정의된 org.springframework.transaction.aspectj.AnnotationTransactionAspect를 가진 애플리케이션을 연결해야만 한다. aspect는 트랜잭션 관리자로 설정되어야만 한다. 당신은 aspect를 의존성 삽입하기 위해 Spring을 사용할수 있지만 우리는 Spring컨테이너 외부에서 구동중인 애플리케이션을 여기서 집중할것이다. 우리는 당신에게 프로그램으로 처리하는 방법을 보여줄것이다.

Note

계속하기 전에, 당신은 Section 9.5.6, "@Transactional 사용하기" 와 Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍를 각각 보길 원할것이다.

```
// construct an appropriate transaction manager
DataSourceTransactionManager txManager = new DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect to use it; this must be done before executing any transactional methods
AnnotationTransactionAspect.aspectOf().setTransactionManager (txManager);
```

이 aspect를 사용할때, 당신은 클래스가 구현하는 인터페이스가 아닌, 구현 클래스(그리고/또는 클래스내 메소드)를 주석처리(annotate)해야만 한다. AspectJ는 인터페이스의 주석이 상속되지 않는 Java의 규칙을 따른다.

클래스의 @Transactional어노테이션은 클래스내 어떠한 public 작업의 수행을 위한 디폴트 트랜잭션 구문을 명시한다.

클래스내 메소드의 @Transactional 어노테이션은 클래스 어노테이션에 의해 주어진 디폴트 트랜잭션 구문을 오버라이드한다. public, protected, 그리고 default 가시성(visibility)을 가진 메소드는 어노테이션처리되어야 할것이다. protected와 default 가시성을 가진 메소드를 직접 어노테이션 처리하는 것은 이러한 작업의 수행을 위한 트랜잭션 구문을 얻기 위한 유일한 방법이다.

이 aspect를 사용할때 @Transactional 어노테이션은 트랜잭션 성격을 가지도록 타입이나 인터페이스를 주석처리하기 위해 사용될수 있다. 그리고 순차적으로 그것에 의해 정의된 어떤 public작업의 수행은 트랜잭션 성격의 의미를 가질것이다. 개별 public 메소드를 위한 트랜잭션 성격을 가지는 의미는 관련 메소드 정의에 주석처리를 함으로써 정의될수 있다. 인터페이스 멤버가 주석처리가 된다면(인터페이스를 구현하는 메소드에 반대되는), 인터페이스 자체는 @Transactional 처럼 주석처리될것이다.

AnnotationTransactionAspect로 애플리케이션을 묶거(weave)나 로드시 묶는것을 사용하기 위해 당신은 AspectJ(AspectJ 개발 가이드를 보라)로 애플리케이션을 빌드해야만 한다. AspectJ를 사용한 로드시 묶기에 대해서 Section 6.8.4, “Using AspectJ Load-time weaving (LTW) with Spring applications” 를 보라.

9.6. 프로그램으로 처리하는 트랜잭션 관리

Spring 프로그래밍적인 트랜잭션 관리에 있어 두 가지 방법을 제시한다.

☒ TransactionTemplate을 사용하기

☒ 직접 PlatformTransactionManager 구현물을 사용하기

당신이 프로그램으로 처리하는 트랜잭션 관리를 사용한다면, 우리는 일반적으로 전자의 접근방법(이를테면, TransactionTemplate를 사용하여)을 추천한다. 두 번째 접근법은 (비록 예외처리는 덜 성가시지만) JTA UserTransaction API의 사용과 비슷하다.

9.6.1. TransactionTemplate 사용하기

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

트랜잭션 컨텍스트 내에서 실행되어야 하는 애플리케이션 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 익명 클래스를 통해 TransactionCallbackWithoutResult를 사용하라.

```
tt.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }

});
```

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

TransactionTemplate를 사용하려는 애플리케이션 클래스들은 반드시 PlatformTransactionManager에 접근해야만 한다(의존성 삽입을 통해 클래스에 제공될). 이것은 mock 혹은 stub PlatformTransactionManager와 같은 클래스를 단위 테스트 하기는 쉽다. 여기에는 JNDI 룩업 혹은 정적인 마법이 존재하지 않는다 : 이것은 단순한 인터페이스이다. 대개, 당신은 유닛 테스트를 간단하게 만들기 위해 Spring을 사용할 수 있다.

9.6.2. PlatformTransactionManager 사용하기

당신은 트랜잭션을 직접 관리하기 위해 org.springframework.transaction.PlatformTransactionManager도 역시 사용할 수 있다. bean참조를 통해 bean을 사용하여 PlatformTransactionManager의 구현물을 간단히 전달하라.그리고나서, TransactionDefinition와 TransactionStatus 객체를 사용함으로써, 당신은 트랜잭션을 초기화하고, 롤백, 커밋할 수 있다.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

9.7. 프로그램으로 처리하는 것과 선언적인 트랜잭션 관리간의 선택하기

프로그램으로 처리하는 트랜잭션 관리는 당신이 적은 수의 트랜잭션 성격을 가지는 작업을 가진다면 언제나 좋은 생각이다. 예를 들어, 업데이트 작업을 위한 트랜잭션을 요구하는 웹 애플리케이션을 가진다면, 당신은 Spring이나 다른 기술을 사용하여 트랜잭션 성격을 가지는 프록시를 셋업하길 원하지는 않을것이다. 이 경우, TransactionTemplate를 사용하는 것이 아마 좋은 접근법일것이다.

반면에, 애플리케이션이 많은 트랜잭션 성격을 가지는 작업을 가진다면, 선언적인 트랜잭션 관리가 가치가 있다. 이것은 비지니스 로직외부에서 트랜잭션 관리를 유지하고 Spring에서는 설정하기가 어렵지 않다. EJB CMT보다 Spring을 사용하는 것이 선언적인 트랜잭션 관리의 설정 비용이 크게 감소한다.

9.8. 특정 애플리케이션 서버에 대한 통합

Spring의 트랜잭션 추상화는 대개 애플리케이션 서버에 대해 관용적이다. 추가적으로, JTA UserTransaction과 TransactionManager객체를 위한 JNDI룩업을 선택적으로 수행할수 있는 Spring의 JtaTransactionManager클래스는 애플리케이션 서버에 따라 다양한 후자의 객체의 위치를 자동감지하기 위해 셋팅될수 있다. TransactionManager인스턴스에 대한 접근을 가지는 것은 고급 트랜잭션 성질을 허용한다. 좀더 상세한 정보를 위해서는 JtaTransactionManager Javadocs을 보라.

9.8.1. BEA 웹로직

웹로직 7.0, 8.1 또는 그 이상의 환경에서, 당신은 JtaTransactionManager클래스 대신에 WebLogicJtaTransactionManager를 사용하는 것을 선호할것이다. 특정 웹로직은 일반적인 JtaTransactionManager의 하위클래스를 명시한다. 이것은 표준 JTA 성질(트랜잭션 명, 트랜잭션마다의 격리 레벨, 모든 경우내 트랜잭션의 재개를 포함한 기능)을 넘어서 웹로직 관리 트랜잭션 환경에서 강력한 Spring의 트랜잭션 정의의 지원한다.

9.8.2. IBM 웹스피어

웹스피어 5.1, 5.0 과 4 환경에서, 당신은 Spring의 WebSphereTransactionManagerFactoryBean클래스를 사용하고자 할것이다. 이것은 웹스피어의 정적 접근 메소드를 통해 수행하는 웹스피어 환경에서 JTA TransactionManager를 가져오는 factory bean이다. 이 메소드는 웹스피어의 각각의 버전마다 다르다. JTA TransactionManager 인스턴스가 이 factory bean을 통해 획득되었을때, JTA UserTransaction객체의 사용을 넘어 고급 트랜잭션 성질을 위해 Spring의 JtaTransactionManager는 이것에 대한 참조를 가지고 설정된다.

9.9. 공통적인 문제에 대한 해결법

9.9.1. 특정 DataSource를 위한 잘못된 트랜잭션 관리자 사용하기

개발자들은 그들의 요구사항들에 맞는 적절한 PlatformTransactionManager 구현을 사용하는데 주의를 기울여야 한다. Spring 트랜잭션 추상화가 JTA 전역 트랜잭션과 동작하는 방식을 이해하는 것은 중요한 일이다. 적절하게 사용되었을 때, 여기엔 아무런 문제가 없다. Spring 프레임워크는 단지 간소화하고 이식가능한 추상화만을 제공한다.

만약 당신이 전역 트랜잭션을 사용한다면, 당신은 모든 트랜잭션적인 동작들에 대해 Spring의 org.springframework.transaction.jta.JtaTransactionManager 클래스(또는 특정 애플리케이션 서버에 대한 하위클래스)를 반드시 사용해야만 한다. 만약 그렇지 않으면 Spring 프레임워크는 컨테이너 DataSource 인스턴스와 같은 자원들에서 로컬 트랜잭션을 수행하고자 할 것이다. 그런 로컬트랜잭션들은 말이 안되며, 좋은 애플리케이션 서버라면 그것들을 에러로 간주할 것이다.

9.10. 더 많은 자원

Spring 프레임워크의 트랜잭션 지원에 대한 더 많은 자원을 위해 아래의 링크를 보라.

☒ InfoQ에서 출판된 Java트랜잭션 디자인 전략이라는 책은 좋다. Java의 트랜잭션에 대해 잘 소개했다. 이것또한 Spring 프레임워크와 EJB3모두에서 트랜잭션을 설정하고 사용하는 방법에 대한 예제를

포함한다.

Chapter 10. DAO support

10.1. 소개

Spring에서 데이터 접근 객체(DAO) 지원은 JDBC, Hibernate또는 JDO를 표준화된 방법으로의 데이터 접근 기술을 가지고 작업하는것을 쉽게 하자는데 가장 큰 목적이 있다. 이것은 이미 언급한 퍼시스턴스 기술간의 교체를 쉽게 하도록 하고 각각의 기술로 명시한 예외에 처리하는 것에 대한 걱정없이 코딩하도록 허락한다.

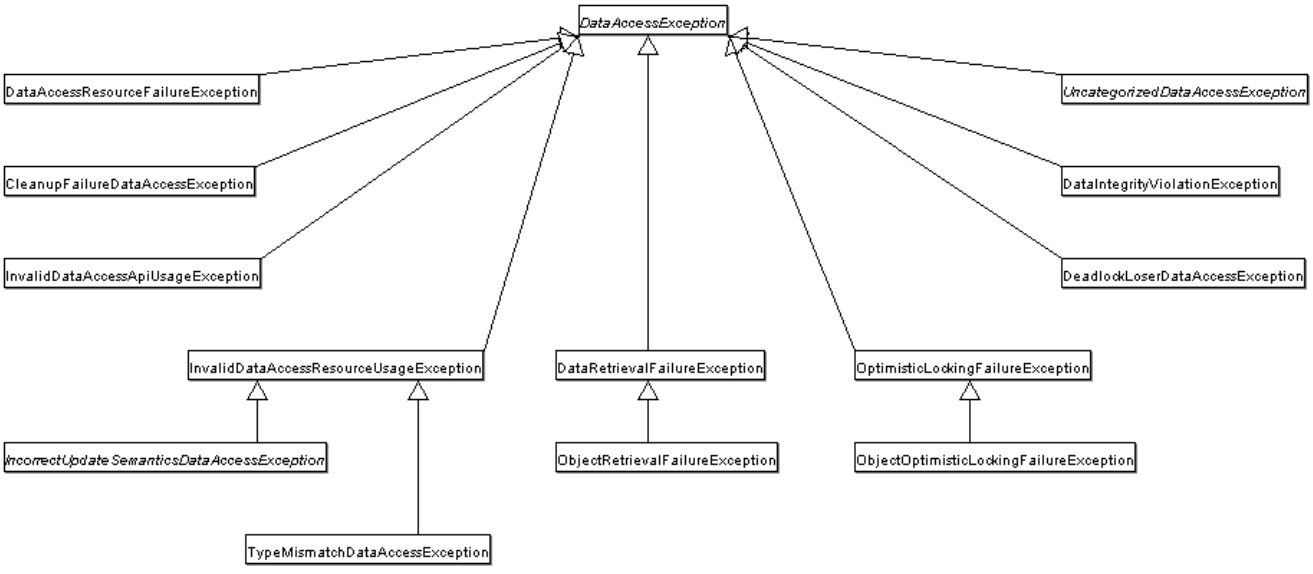
10.2. 일관된 예외 구조

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

JDBC 예외에 추가적으로 Spring은 추상화된 런타임 예외의 세트를 위해 Hibernate 특유의 예외를 포장하고 개개인의 것으로 부터 그것들을 변환하고, 예외를 체크할수 있다(이것은 JDO예외에서도 같다). 이것은 괴로운 반복적 catches/throws구문과 예외선언 없이 적절한 레이어에서만 회복될수 없는 대부분의 퍼시스턴스 예외를 다루도록 한다(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 다양한 Template-기반 버전을 위해서 참이다. 인터셉터-기반의 클래스를 사용한다면, 애플리케이션은 SessionFactoryUtils의 convertHibernateAccessException 이나 convertJdoAccessException 메소드로 각각 위임되는 HibernateException 와 JDOException를 다루어야만 한다. 이 메소드는 예외를 org.springframework.dao 예외 구조와 호환이 되는 것으로 변환한다. JDOException이 체크되지 않은것 처럼, 그것들은 간단히 던져질수 있다. 예외의 개념에서 일반적인 DAO추상화를 희생한다.

Spring이 사용하는 예외 구조는 다음 그래프내에서 윤곽이 그려진다.



(위 이미지내 상세하게 설명된 클래스 구조가 일부만을 보여주고 있다는 것에 주의하라.)

10.3. DAO지원을 위한 일관된 추상클래스

JDBC, JDO그리고 Hibernate처럼 일관적인 방법으로 다양한 데이터 접근 기술로 좀더 쉽게 작업을 수행하기 위해, Spring은 당신이 확장할수 있는 추상화된 DAO클래스의 세트를 제공한다. 이런 추상화된 클래스들은 데이터 소스와 현재 사용중인 기술을 명시하는 다른 설정상의 셋팅을 제공하는 메소드를 가진다.

DAO지원 클래스:

- JdbcDaoSupport - JDBC데이터 접근 객체를 위한 슈퍼클래스(super class), DataSource를 필요로 한다. 이 클래스는 하위 클래스를 위해 제공된 DataSource로부터 초기화된 JdbcTemplate 인스턴스를 제공한다.
- HibernateDaoSupport - Hibernate데이터 접근 객체를 위한 슈퍼클래스(super class), SessionFactory 를 필요로 한다. 이 클래스는 하위 클래스를 위해 제공되는 SessionFactory로부터 초기화된 HibernateTemplate 인스턴스를 제공한다. SessionFactory, flush mode, 예외 번역과 같이 나중에 설정된 셋팅을 다시 사용하기 위해, HibernateTemplate을 통해 직접 초기화될수 있다.
- JdoDaoSupport - JDO데이터 접근 객체를 위한 슈퍼클래스(super class), PersistenceManagerFactory 를 필요로 한다. 이 클래스는 하위 클래스를 위해 제공된 PersistenceManagerFactory로부터 초기화된 JdoTemplate 인스턴스를 제공한다.
- JpaDaoSupport - JPA 데이터 접근 객체를 위한 슈퍼클래스(super class). EntityManagerFactory를 필요로 한다. 이 클래스는 하위 클래스를 위해 제공된 EntityManagerFactory로부터 초기화된 JpaTemplate 인스턴스를 제공한다.

Chapter 11. JDBC를 사용한 데이터 접근

11.1. 소개

Spring의 JDBC추상화 프레임워크에 의해 제공되는 값-추가는 다음의 목록에 의해 가장 잘 보여진다(기울임꼴로 되어 있는 줄은 Spring의 JDBC추상화 프레임워크를 사용할때 애플리케이션 개발자에 의해 코딩될 필요가 있다는데 노트하라.).

1. connection 파라미터 정의하기
2. connection 열기
3. 구문(statement) 명시하기
4. 구문을 준비하고 수행하기
5. 결과를 통해 반복(iterate)을 반복하기
6. 각각의 반복에 대해 작업하기
7. 예외 처리하기
8. 트랜잭션 다루기
9. connection 닫기

Spring은 사용하기 어렵고 개발을 반복하는 복잡한 API와 같은 JDBC를 만드는 하위레벨의 상세한 사항 모두를 다룬다.

11.1.1. 패키지 구조

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

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

org.springframework.jdbc.datasource패키지는 쉬운 DataSource 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.object패키지는 쓰레드에 안전하고 재사용가능한 객체처럼 RDBMS 쿼리, update 그리고 저장 프로시저를 표현하는 클래스를 포함한다. 이 접근법은 JDO에 의해 형상화 되었다. 쿼리에 의해 반환된 객체는 데이터베이스로 부터 “disconnected” 된다. JDBC추상화의 높은 레벨은 org.springframework.jdbc.core패키지내에서 하위 레벨에 의존한다.

마지막으로 org.springframework.jdbc.support패키지는 SQLException번역 기능과 몇개의 유틸리티 클래스를

찾을수 있는 곳이다.

JDBC처리중에 던져진 예외는 org.springframework.dao패키지내에서 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.

11.2. 기본적인 JDBC처리와 에러 처리를 위한 JDBC Core클래스 사용하기

11.2.1. JdbcTemplate

JdbcTemplate은 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, ResultSet를 넘어서 순환을 모방하고 반환된 인자값을 보여주는 작업을 수행한다. 이것은 또한 JDBC예외를 잡고 일반적인 것으로 그것들을 번역하고 좀더 다양한 정보를 제공하도록 하고 org.springframework.dao패키지내에 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 DataSource 참조또는 애플리케이션 컨텍스트내에서 준비되고 빈(bean)참조처럼 서비스하기 위해 직접적인 초기화를 통해 서비스구현내에서 사용될수 있다. 주의: DataSource는 애플리케이션 컨텍스트내에서 언제나 빈처럼 설정되어야 한다. 이 클래스는 callback인터페이스와 SQLExceptionTranslator인터페이스에 의해 인자화 되기 때문에 이것을 하위클래스화 할 필요가 없다.

마지막으로, 이 클래스에 의해 이슈화되는 모든 SQL은 템플릿 인스턴스의 전체 경로를 포함한 클래스명에 관련된 카테고리에서 'DEBUG' 레벨에서 로그화된다(대개 JdbcTemplate이지만, JdbcTemplate 클래스의 사용자정의 하위클래스가 사용된다면 다를것이다.).

11.2.2. NamedParameterJdbcTemplate

NamedParameterJdbcTemplate 클래스는 명명 파라미터(오직 전통적인 위치고정자('?') 인자를 사용하여 JDBC구문을 프로그램하는 것과는 대립되는)를 사용하여 JDBC구문을 프로그램하기 위한 지원을 추가한다. NamedParameterJdbcTemplate 클래스는 평범한 JdbcTemplate를 포장하고 포장된 JdbcTemplate로 위임한다. 이 부분은 NamedParameterJdbcTemplate 클래스가 JdbcTemplate 자체와 다른 영역만을 언급할것이다.

```
// some JDBC-backed DAO class...
public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(0) from T_ACTOR where first_name = :first_name";

    NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(this.getDataSource());
    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);
```

```
    return template.queryForInt(sql, namedParameters);
}
```

value내 명명 파라미터 표기의 사용은 'sql' 변수로 할당되고 관련된 값은 'namedParameters' 변수(MapSqlParameterSource 타입인)로 플러그인 된다.

당신이 원한다면, 명명 파라미터(그리고 관련된 값들)를 (아마도 좀더 친숙한)Map-기반 스타일을 사용하여 NamedParameterJdbcTemplate 인스턴스로 전달할수 있다. (나머지 메소드는 NamedParameterJdbcOperations에 의해 드러나고 NamedParameterJdbcTemplate 클래스에 의해 구현된다)

```
// some JDBC-backed DAO class...
public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(0) from T_ACTOR where first_name = :first_name";

    NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(this.getDataSource());
    Map namedParameters = new HashMap();
    namedParameters.put("first_name", firstName);

    return template.queryForInt(sql, namedParameters);
}
```

NamedParameterJdbcTemplate에 관련된 다른 좋은 기능은 SqlParameterSource(같은 패키지내 존재하는) 인터페이스이다. 이미 코드조각(MapSqlParameterSource 클래스)을 처리하는 것중 하나로 이 인터페이스의 구현물 예제를 보았다. SqlParameterSource 전체는 NamedParameterJdbcTemplate를 위한 명명 파라미터 값의 근원으로 제공하는 것이다. MapSqlParameterSource 클래스는 가장 간단한 구현물이고 java.util.Map에 대한 어댑터이고 자체적으로 명백하게 사용한다.

다른, 좀더 흥미로운, SqlParameterSource 인터페이스의 구현물은 BeanPropertySqlParameterSource 클래스이다. 이 클래스는 임의의 JavaBean과 같은 객체를 포장하고 명명 파라미터 값의 근원처럼 포장된 객체의 프라퍼티를 사용한다. 예제는 이것을 좀더 명확하게 해줄것이다.

```
// some JavaBean-like class...
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }

    // setters omitted...

}
```

```
// some JDBC-backed DAO class...
public int countOfActors(Actor exampleActor) {

        // notice how the named parameters match the properties of the above 'Actor' class
```

```
    String sql = "select count(0) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(this.getDataSource());
    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return template.queryForInt(sql, namedParameters);
}
```

NamedParameterJdbcTemplate 클래스가 전통적인 JdbcTemplate 템플릿을 포장한다는 것을 기억하라. 포장된 JdbcTemplate 인스턴스에 접근할 필요가 있다면(JdbcTemplate 클래스에만 존재하는 몇가지 기능에 접근하는), JdbcOperations 인터페이스를 통해 포장된 JdbcTemplate에 접근하는 getJdbcOperations() 메소드를 사용할수 있다.

NamedParameterJdbcTemplate 클래스는 쓰레드에 안전하고 기대되는 사용 패턴은 작업마다 새로운 NamedParameterJdbcTemplate 인스턴스를 인스턴스화할 뿐 아니라 DataSource(Spring의 IoC기술을 사용한다면 Spring IoC컨테이너를 통해) 마다 하나의 NamedParameterJdbcTemplate 인스턴스를 간단히 설정하고 필요한 DAO에 대해 같은 인스턴스를 공유한다.

11.2.3. SimpleJdbcTemplate

Note

이 클래스에 의해 제공되는 기능이 Java 5를 사용할때만 사용가능하다는 것을 알라.

SimpleJdbcTemplate 클래스는 Java 5언어의 기능적인 장점을 가지는 전통적인 Spring JdbcTemplate에 대한 래퍼이다. SimpleJdbcTemplate 클래스가 Java 5의 구문상 유연성과 같은 기능을 위한 선물이다. 하지만 Java 5에서 개발하는 사람이 JDK이전 버전에서 개발하는 것으로 돌아간다. 이러한 구문상 유연성과 같은 기능이 멋지다.

구문상 유연성 영역에서 SimpleJdbcTemplate 클래스의 값-추가는 'before and after' 예제로 가장 잘 보여진다. 다음의 코드 조각은 전통적인 Spring JdbcTemplate 을 사용하여 몇가지 데이터 접근 코드를 보여준다.

```
// classic JdbcTemplate-style...
public Actor findActor(long id) {
    String sql = "select id, first_name, last_name from T_ACTOR where id = ?";

    RowMapper mapper = new RowMapper() {

        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong(Long.valueOf(rs.getLong("id"))));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    };

    // normally this would be dependency injected of course...
    JdbcTemplate jdbcTemplate = new JdbcTemplate(this.getDataSource());

    // notice the cast, and the wrapping up of the 'id' argument
    // in an array, and the boxing of the 'id' argument as a reference type
    return (Actor) jdbcTemplate.queryForObject(sql, mapper, new Object[] {Long.valueOf(id)});
}
```

이것은 SimpleJdbcTemplate를 사용할때만 같은 메소드이다. '좀더 명백한' 코드를 어떻게 하는지 알라.

```
// SimpleJdbcTemplate-style...
public Actor findActor(long id) {
    String sql = "select id, first_name, last_name from T_ACTOR where id = ?";

    ParameterizedRowMapper<Actor> mapper = new ParameterizedRowMapper<Actor>() {

        // notice the return type with respect to Java 5 covariant return types
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong("id"));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    };

    // again, normally this would be dependency injected of course...
    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(this.getDataSource());

    return simpleJdbcTemplate.queryForObject(sql, mapper, id);
}
```

11.2.4. DataSource

데이터베이스로부터 데이터작업을 수행하기 위해서 우리는 데이터베이스로 부터 Connection을 얻을 필요가 있다. Spring은 DataSource을 통해서 이것을 수행한다. DataSource는 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 DriverManager가 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("");
```

11.2.5. SQLExceptionTranslator

SQLExceptionTranslator은 SQLException과 Spring의 데이터접근 전략에 얽매이지 않는 org.springframework.dao.DataAccessException사이에 해석할수 있는 클래스에 의해 구현될수 있는

인터페이스이다.

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

SQLErrorCodeSQLExceptionTranslator는 초기설정에 의해서 사용이 되는 SQLExceptionTranslator의 구현이다. 이 구현은 업체코드를 명시하는데 사용한다. SQLState 구현보다 좀더 정확하지만 업체에 종속적이다. 에러코드 해석은 SQLErrorCodes이라고 불리는 자바빈 타입의 코드에 기초를 둔다. 이 클래스는 "sql-error-codes.xml"라는 이름의 설정파일의 내용에 기초를 두는 SQLErrorCodes를 생성하기 위한 공장같은 이름의 SQLErrorCodesFactory에 의해서 생성되고 활성화된다. 이 파일은 업체코드에 의해 활성화되고 DatabaseMetaData로 부터 얻어진 DatabaseProductName에 기초를 둔다.

SQLErrorCodeSQLExceptionTranslator는 다음의 일치규칙(matching rules)을 적용한다.

☒ 어느 하위 클래스에 의해서 구현되는 사용자정의해석(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.

SQLErrorCodeSQLExceptionTranslator는 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 사용자정의 해석자(custom translator)를 사용하기 위해서 setExceptionTranslator 메소드를 사용하고 이 해석자가 필요한 데이터 접근 처리를 위해 JdbcTemplate를 사용하기 위한 JdbcTemplate 으로 값을 넘길필요가 있다. 여기에 사용자정의 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 사용자정의 해석자는 sql-error-codes.xml내의 에러코드를 찾기위해 디폴트 해석자를 원하기 때문에 데이터소스를 전달했다.

11.2.6. 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;
    }
}
```

11.2.7. 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 InvalidDataAccessApiUsageException를 던질것이다. 여기에 int를 위한것과 String을 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", 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 List를 반환하는 queryForList이다. 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 이것처럼 보일것이다.

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

11.2.8. 데이터베이스 수정하기

당신이 사용할수 있는 많은 update메소드가 있다. 나는 어떠한 기본키를 위한 칼럼을 수정하는 예제를 아래에서 보라. 이 예제에서 row 파라미터를 위한 위치고정자(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;
    }
}
```

11.3. 데이터베이스 연결을 제어하기

11.3.1. DataSourceUtils

DataSourceUtils 클래스는 필요하다면 JNDI로 부터 connection을 얻거나 connection을 닫기 위한 static메소드를 제공하는 편리하고 강력한 헬퍼 클래스이다. 이것은 쓰레드 범위의 connection을 위한 지원 예를 들면 DataSourceTransactionManager를 사용한다.

주의 : getDataSourceFromJndi메소드는 bean factory나 애플리케이션 컨텍스트를 사용하지 않는 애플리케이션을 목표로 한다. 이것은 factory내 당신의 빈즈나 JdbcTemplate인스턴스를 먼저 설정하는것을 선호한다. JndiObjectFactoryBean는 JNDI로 부터 DataSource를 꺼내고 다른 빈즈에게 DataSource bean참조를 주는데 사용될수 있다. 다른 DataSource로의 교체는 설정의 문제이다. 당신은 non-JNDI의 DataSource를

가진 FactoryBean의 정의를 교체할수 있다.

11.3.2. SmartDataSource

SmartDataSource 인터페이스는 관계형 데이터베이스를 위한 connection을 제공할수 있는 클래스에 의해 구현되는 것이다. DataSource 인터페이스를 확장하는것은 주어진 작업후에 닫혀야만하는 connection인지 아닌지 쿼리하기 위해 사용하도록 허용한다. 이것은 때때로 우리가 connection을 재사용하길 원한다는 것을 안다면 효율성을 위해 유용할수 있다.

11.3.3. AbstractDataSource

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

11.3.4. SingleConnectionDataSource

SingleConnectionDataSource클래스는 사용후에 닫히질 않는 connection을 포장한 SmartDataSource 의 구현물이다. 분명히 이것은 다중 쓰레드 성능은 아니다.

만약 퍼시스턴스 툴을 사용할때처럼, 라이언트코드가 풀링된 connection의 소비내 close를 호출한다면, suppressClose 을 true로 설정하라. 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 서버 외부에서 코드를 쉽게 테스팅 가능하도록 한다. DriverManagerDataSource와는 대조적으로 이것은 언제나 물리적 connection생성 초과를 피하고 같은 connection을 재사용한다.

11.3.5. DriverManagerDataSource

DriverManagerDataSource 클래스는 빈 프라퍼티를 통해 명백한 예전 JDBC드라이버를 설정하고 언제나 새로운 connection을 반환하는 SmartDataSource 인터페이스의 구현물이다.

이것은 각각의 ApplicationContext내 DataSource bean이나 간단한 JNDI환경과의 결합처럼 테스트나 J2EE컨테이너 외부의 독립적인 환경을 위해 잠재적으로 유용하다. 풀 성격의 Connection.close()호출은 간단하게 connection을 닫을 것이다. 그래서 어떠한 DataSource-인식 퍼시스턴스코드도 작동할것이다. 어쨌든, commons-dbcp와 같은 자바빈 스타일의 connection pool을 사용하는 것은 테스트 환경에서조차 쉽다. 이것은 DriverManagerDataSource 에 비해 connection pool을 사용하는 것을 언제나 선호한다.

11.3.6. TransactionAwareDataSourceProxy

TransactionAwareDataSourceProxy는 Spring-관리 트랜잭션의 인지을 추가하기 위한 대상 DataSource를 포장하는 대상 DataSource를 위한 프록시이다. 이 점에서, 이것은 J2EE서버가 제공하는 전통적인 JNDI DataSource와 유사하다.

Note

호출해야만 하는 코드가 존재하고 표준 JDBC DataSource인터페이스 구현물을 전달하는 것을

제외하고 이 클래스를 사용하는 것은 결코 필요하거나 바람직하지 않다. 이 경우, 이 코드가 여전히 사용가능하지만 Spring관리 트랜잭션에 관계한다. JdbcTemplate 이나 DataSourceUtils와 같은 resource관리를 위한 더 높은 레벨의 추상화를 사용하여 당신 자신만의 새로운 코드를 작성하는 것이 대개 선호된다.

(좀더 상세한 정보를 위해 TransactionAwareDataSourceProxy Javadoc을 보라.)

11.3.7. DataSourceTransactionManager

DataSourceTransactionManager 클래스는 하나의 JDBC데이터 소스를 위한 PlatformTransactionManager구현물이다. 이것은 명시된 데이터 소스의 JDBC connection을 최근에 수행된 쓰레드에 바인드한다. 잠재적으로 데이터 소스당 하나의 쓰레드 connection을 허용한다.

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.

DataSourceTransactionManager 클래스는 사용자정의 격리 레벨, 적절한 JDBC 구문 쿼리 타임아웃처럼 적용되는 타임아웃을 지원한다. 후자를 지원하기 위해 애플리케이션 코드는 JdbcTemplate 나 각각의 생성된 구문를 위한 DataSourceUtils.applyTransactionTimeout 메소드 호출을 사용해야만 한다.

이 구현물은 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.!

11.4. 자바 객체처럼 JDBC작업을 모델링 하기.

org.springframework.jdbc.object패키지는 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.

Note

밑에서 언급된(StoredProcedure 클래스의 예외를 가진) 다양한 RDBMS작업 클래스가 종종 JdbcTemplate 호출에 대체될수 있는 몇몇 Spring개발자들간에 view가 있다. 종종 이것은 직접 JdbcTemplate의 메소드를 간단히 호출하는 DAO메소드를 사용하는 것과 유사하다.

아무리 view 더라도 스트레스가 될것이다. RDBMS 작업 클래스를 사용하여 측정가능한 값을 얻는데 자유롭다면, 언급된 클래스를 사용하는것도 자유롭게 느낄것이다.

11.4.1. SqlQuery

SqlQuery는 SQL쿼리를 캡슐화하는 쓰레드에 안전한 객체를 재사용가능하다. 하위 클래스는 ResultSet를 반복하는 동안 결과를 저장할수 있는 객체를 제공하기 위해 newResultReader()메소드를 구현해야만 한다. SqlQuery는 MappingSqlQuery 하위클래스가 Java클래스에 대한 맵핑 row를 위한 좀더 많은 편리한 구현물을

제공하기 때문에 드물게 직접적으로 사용된다. SqlQuery을 확장하는 다른 구현물은 MappingSqlQueryWithParameters 와 UpdatableSqlQuery이다.

11.4.2. MappingSqlQuery

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

모든 다양한 SqlQuery 구현물인, MappingSqlQuery는 매우 종종 사용되고 이것은 사용하기 가장 쉬운 것중 하나이다.

customer테이블의 데이터를 Customer의 인스턴스로 맵핑하는 사용자정의 쿼리의 예제를 아래에서 보라.

```
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 DataSource를 가지는 사용자정의 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 PreparedStatement를 생성하기 위해 사용될 것이다. 각각의 파라미터는 SqlParameter내에 전달될 declareParameter메소드를 사용해서 선언되어야 한다. SqlParameter 는 이름과 java.sql.Types내에 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 Integer로 전달된다. 지금 우리는 파라미터의 배열을 사용해서 쿼리를 수행할 준비가 되었고 우리의 쿼리를 통해 반환되는 각각의 레코드를 위한 Customer 객체를

포함하는 List를 얻게된다. 이 경우에 적합한 경우 하나의 항목이 될것이다.

11.4.3. SqlUpdate

SqlUpdate 클래스는 SQL update를 캡슐화한다. 쿼리처럼, update객체는 재사용가능하다. 모든 RdbmsOperation클래스 처럼, update는 파라미터를 가지고 SQL내 정의된다.

이 클래스는 쿼리 객체의 execute()메소드와 유사한 많은 update()메소드를 제공한다.

이 클래스는 명백하다. 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);
    }
}
```

11.4.4. StoredProcedure

StoredProcedure클래스는 RDBMS 저장 프로시저의 객체 추상화를 위한 수퍼클래스이다. 이 클래스는 추상적이고 execute(..)메소드들은 protected상태이다. 좀더 단단하게 타이핑된 하위 클래스를 통해 다른것보다 좀더 사용이 제한적이다.

상속된 sql프라퍼티는 RDBMS의 저장프로시저의 이름이 될것이다. JDBC 3.0은 명명 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 StoredProcedure를 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.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:mydb");
        ds.setUsername("scott");
        ds.setPassword("tiger");

        MyStoredProcedure sproc = new MyStoredProcedure(ds);
        Map results = sproc.execute();
        printMap(results);
    }

    private class MyStoredProcedure extends StoredProcedure {

        private 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() {
            // the 'sysdate' sproc has no input parameters, so an empty Map is supplied...
            return execute(new HashMap());
        }
    }

    private static void printMap(Map results) {
        for (Iterator it = results.entrySet().iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
    }
}
```

두개의 출력 파라미터(Oracle 커서의 경우)를 가지는 StoredProcedure의 예제를 아래에서 보라.

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;
```

```
public class TitlesAndGenresStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "AllTitlesAndGenres";

    public TitlesAndGenresStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new GenreMapper()));
        compile();
    }

    public Map execute() {
        // again, this sproc has no input parameters, so an empty Map is supplied...
        return super.execute(new HashMap());
    }
}
```

TitlesAndGenresStoredProcedure 생성자에서 사용된 declareParameter(..) 메소드의 오버로드된 형태가 RowMapper 구현물 인스턴스에 전달되는 방법을 노트하라. 이것은 존재하는 기능을 재사용하기 위해 편리하고 강력한 방법이다(두개의 RowMapper구현물을 위한 코드는 아래에서 제공된다).

첫번째로, ResultSet을 제공되는 ResultSet내 각각의 row를 위한 Title도메인 객체로 간단히 맵핑하는 TitleMapper클래스이다.

```
import com.foo.sprocs.domain.Title;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public final class TitleMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Title title = new Title();
        title.setId(rs.getLong("id"));
        title.setName(rs.getString("name"));
        return title;
    }
}
```

두번째로, ResultSet을 제공되는 ResultSet내 각각의 row를 위한 Genre도메인 객체로 간단히 맵핑하는 GenreMapper클래스이다.

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Genre;

public final class GenreMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

저장 프로시저(이를테면, 저장 프로시저는 RDBMS의 정의내 한개 또는 그 이상의 입력 파라미터를 가지는 것처럼 명시된다.)에 파라미터를 전달할 필요가 있다면, 수퍼클래스(타입화되지 않은)의 execute(Map

parameters)(protected 접근을 하는)에 위임할 강력하게 타입화된 execute(..) 메소드를 코딩할것이다.

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate";

    public TitlesAfterDateStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declaraParameter(new SqlParameter(CUTOFF_DATE_PARAM, Types.DATE);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        compile();
    }

    public Map execute(Date cutoffDate) {
        Map inputs = new HashMap();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

11.4.5. SqlFunction

SqlFunction RDBMS작업 클래스는 결과의 하나의 레코드를 반환하는 쿼리를 위한 SQL "함수" 랩퍼를 캡슐화한다. 디폴트 행위는 int를 반환하는 것이지만 추가적인 반환 타입 파라미터를 가진 메소드를 사용해서 오버라이드 할수 있다. 이것은 JdbcTemplate의 queryForXxx 메소드를 사용하는것이 유사하다. SqlFunction이 가진 장점은 JdbcTemplate을 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 CallableStatement를 사용하는 경향은 아니다. 이러한 타입의 처리를 위해 StoredProcedure 나 SqlCall을 사용하라.

SqlFunction은 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 12. 객체 관계 맵핑(ORM) 데이터 접근

12.1. 소개

Spring은 자원관리측면에서 Hibernate, JDO, Oracle TopLink, Apache OJB, iBATIS SQL Map 그리고 JPA : DAO구현 지원, 트랜잭션 전략등의 통합을 제공한다. Hibernate를 예로 들어볼때, 많은 Hibernate통합 이슈를 할당하는 편리한 IoC 특성을 가진 가장 중요한 클래스 지원이 있다. O/R매퍼을 위한 이러한 모든 지원 패키지는 Spring의 일반적 트랜잭션과 DAO예외 구조의 요구에 응한다. 여기엔 두가지 통합 스타일(Spring의 DAO 'template' 이나 명백한 Hibernate/JDO/TopLink/등등 API에 대한 DAO를 코딩하는)이 있다. 이러한 두 경우, DAO는 의존성삽입을 통해 설정되고 Spring의 자원및 트랜잭션 관리에 참여할수 있다.

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. Spring IoC 컨테이너내부에서의 사용은 설정과 배치의 쉬움이라는 면에서 추가적인 이득을 제공한다. 이 섹션의 예제들은 Spring Applicationcontext내에서 설정이 됨을 보여준다.

다음은 O/R맵핑 DAO를 생성하기 위해 Spring을 사용함으로써 발생하는 몇몇 장점들이다.

☒ 테스트의 용이함(ease) Spring의 IoC접근은 Hibernate SessionFactory인스턴스의 구현과 설정 위치, JDBC DataSource, 트랜잭션 관리자 그리고 매퍼 객체 구현물을 쉽게 교체할수 있게 만든다. 이것은 격리수준내에서 각각의 영속성 관련 코드를 쉽게 격리하고 테스트 할수 있게 만든다.

☒ 공통적인 데이터 접근에 관련된 예외 Spring은 자체적인 예외(잠재적으로는 체크된)를 공통적인 런타임 DataAccessException구조로 변환하도록 당신이 선택한 O/R맵핑툴로부터 예외를 포장할수 있다. 이것은 당신에게 적절한 레이어에서 짜증나는 반복적인 catches/throws와 예외 선언없이 회복할수 없는 대부분의 영속성 예외를 다루도록 해준다. 당신은 필요한 어느 지점에서 예외를 추출하고 다룰수 있다. JDBC예외는 일관적인 프로그래밍 모델내에서 JDBC로 몇가지 작업을 수행할수 있다는 것을 의미하면서 같은 구조로 변환되는 것을 기억하라.

☒ 일반적인 자원 관리 Spring애플리케이션 컨텍스트는 Hibernate SessionFactory 인스턴스, 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 영속성 자원을 다룰수 있는 기능을 제공한다. 예를 들어, Hibernate를 사용하는 관련코드는 효율적이고 적합한 트랜잭션 관리를 위해 Hibernate Session객체를 사용할 필요가 있다. Spring은 Java코드 레벨에서 'template' 래퍼 클래스를 명시적으로 사용하거나 Hibernate SessionFactory(명백한 Hibernate3 API에 기반한 DAO를 위한)를 통해 현재 Session을 나타내서 현재 쓰레드에 Session을 투명하게 생성하고 연결하는(bind) 것을 쉽게 해준다. 게다가 Spring은 어떤 트랜잭션 환경(local 또는 JTA)을 위해 전형적인 Hibernate사용으로부터 반복적으로 발생하는 많은 이슈를 해결한다.

☒ 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 Hibernate/JDO관련 코드에 영향이 없이 다양한 트랜잭션 관리자를 사용하거나 교체할수 있게 되는 장점또한 가지게 된다. 예를 들어, local트랜잭션과 JTA사이에서, 같은 서비스(선언적인 트랜잭션같은)는 두개의 시나리오에서 모두 사용가능하다. 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. O/R맵핑 작업으로 공통적인 트랜잭션을 공유하는 것이 여전히 필요한 배치처리나 BLOB의 스트리밍처럼 O/R맵핑을 위해 적합하지 않은 데이터 접근에 유용하다.

☒ 업체에 종속적인 락(lock-in)방식을 피할수 있고 mix-and-match구현방식을 허용한다. 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 Hibernate Transaction과 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.

Spring배포 패키지내 PetClinic샘플은 DAO를 대체하는 구현물과 JDBC, Hibernate, Oracle TopLink, 그리고 JPA를 위한 애플리케이션 컨텍스트 설정을 제공한다. PetClinic는 Spring웹 애플리케이션내 Hibernate, TopLink 그리고 JPA를 사용하는 샘플 애플리케이션처럼 제공된다. 이것은 다른 트랜잭션 전략을 가진 선언적인 트랜잭션 구분에 영향을 끼친다.

JPetStore 샘플은 Spring환경내 iBATIS SQL Maps의 사용을 보여준다. 이것은 또한 두가지 웹 티어 버전의 특징을 가진다. 하나는 Spring MVC이고 다른 하나는 Struts에 기초를 둔다.

Spring에 포함된 샘플을 넘어서, 여기엔 특정업체가 제공하는 다양한 Spring기반 O/R맵핑이 있다. 예를 들어, JDO구현물인 JPOX(http://www.jpox.org/) and Kodo (http://www.bea.com/kodo).

12.2. Hibernate

우리는 Spring이 O/R매퍼를 통합하는 방법을 보여주기 위해 Spring환경에서의 Hibernate(http://www.hibernate.org)에 대해 다루기 시작할것이다. 이 섹션은 많은 이슈를 상세하게 다루고 다양한 DAO구현물과 트랜잭션 구분의 차이점을 보여줄것이다. 대부분의 패턴은 다른 지원되는 O/R맵핑툴로 직접 전환될수 있다. 이 장의 다음 섹션은 다른 O/R매퍼를 다루고, 여기서 예제를 보여준다.

다음의 언급은 Hibernate 3에 집중한다. 이것은 Hibernate의 주요 제품버전이다. Hibernate 2.x는 지원된 이후 Spring에서 계속적으로 지원되고 있다. 다음의 예제는 Hibernate 3클래스와 설정을 모두 사용한다. 이것들 모두는 유사한 Hibernate 2.x지원 패키지인 org.springframework.orm.hibernate를 사용하여 Hibernate 2.x에서 적용될수 있다. Hibernate 3은 org.springframework.orm.hibernate3를 사용한다. org.hibernate패키지에 대한 참조는 Hibernate 3에서의 가장 상위 패키지의 변경에 따라 net.sf.hibernate를 대체할 필요가 있다. 간단히 패키지명(예제에서 사용된)을 따르라.

12.2.1. 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 서비스의 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 서비스, 웹의 화면 해설자(resolvers), 비지니스 서비스를 사용하는 웹 컨트롤러 등등)는 자바빈이다.

12.2.2. Spring 애플리케이션 컨텍스트내에서 SessionFactory 셋업

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.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
      </value>
    </property>
  </bean>
```

```
  ...
</beans>
```

local Jakarta Commons DBCP BasicDataSource를 JNDI를 사용하는 DataSource로 전환하는 것은 설정상의 문제라는 것을 기억하라.

```
<beans>

  <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/myds"/>
  </bean>

  ...
</beans>
```

당신은 이것을 가져오고 나타내기 위한 Spring의 JndiObjectFactoryBean를 사용하여 JNDI를 사용하는 SessionFactory를 에 접근할수도 있다. 어쨌든, EJB컨텍스트외부에서는 대개 필요한 것이 아니다.

12.2.3. HibernateTemplate

탬플릿팅을 위한 기본적인 프로그래밍 모델은 어떤 데이터접근 객체나 비지니스 서비스의 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="mySessionFactory"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        HibernateTemplate ht = new HibernateTemplate(this.sessionFactory);
        return (Collection) ht.execute(new HibernateCallback() {
            public Object doInHibernate(Session session) throws HibernateException {
                Query query = session.createQuery(
                    "from test.Product product where product.category=?");
                query.setString(0, category);
                return query.list();
            }
        });
    }
}
```

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 Collection loadProductsByCategory(String category) throws DataAccessException {
        return getHibernateTemplate().find(
            "from test.Product product where product.category=?", category);
    }
}
```

12.2.4. 콜백없이 Spring기반의 DAO를 구현하기

DAO를 구현하기 위해 Spring의 HibernateTemplate을 사용하는 것에 대한 대안처럼, Spring의 일반적인 DataAccessException 구조를 따르는 동안 데이터 접근 코드는 콜백내 Hibernate접근 코드를 포장하지 않는 좀더 전통적인 방법으로 작성될수 있다. Spring의 HibernateDaoSupport base클래스는 현재의 트랜잭션 성질을 가지는 Session에 접근하고 시나리오내 예외로 변환하기 위한 메소드를 제공한다. 유사한 메소드는 SessionFactoryUtils 클래스의 정적 헬퍼처럼 사용가능하다. 이러한 코드는 트랜잭션내에서 강제로 수행하기 위해(생명주기가 트랜잭션에 의해 관리되는 것처럼, 반환된 Session을 닫을 필요성을 제거하는) getSession의 "allowCreate" 플래그를 언제나 "false"로 전달할것이다.

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category)
            throws DataAccessException, MyException {

        Session session = 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 convertHibernateAccessException(ex);
        }
    }
}
```

이러한 직접적인 Hibernate 접근코드의 가장 큰 장점은 HibernateTemplate가 콜백내에서 체크되지 않은 예외를 제한하는 동안 데이터 접근 코드내에서 던져지는 체크된 애플리케이션 예외를 허용하는 것이다. HibernateTemplate와 작동하는 동안 관련 체크와 콜백후 애플리케이션 예외를 던지는 것을 종종 미룬다. 대개 HibernateTemplate의 편리한 메소드는 더욱 간단하고 많은 시나리오를 위해 좀더 편리하다.

12.2.5. 명백한 Hibernate 3 API에 기초한 DAO구현하기

Hibernate 3.0.1은 Hibernate자체가 하나의 Session을 트랜잭션마다 관리하는 "컨텍스트상의 Session(contextual Sessions)"이라고 불리는 기능을 소개했다. 이것은 트랜잭션마다 하나의 Hibernate Session에 대한 Spring의 동기화와 극히 일치한다. 관련 DAO구현물은 명백한 Hibernate API에 기초를 두고 다음처럼 볼수 있다.

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

이 Hibernate 접근 스타일은 인스턴스 변수내 SessionFactory를 유지하는 것을 제외하고 당신이 Hibernate문서와 예제에서 찾을것과 매우 유사하다. 우리는 Hibernate의 CaveatEmptor 샘플 애플리케이션에서 보수적인 정적(static) HibernateUtil 클래스 곳곳에서 인스턴스-기반 셋업같은것을 강력하게 추천한다. (대개, 무조건적으로 필요하지 않는한, 정적(static) 변수내 어떠한 자원도 유지하지 말라.)

위 DAO는 의존성삽입 패턴을 따른다. 이것이 Spring의 HibernateTemplate에 대해 코딩된것처럼 이것은 여전히 Spring 애플리케이션 컨텍스트로 잘 끼워진다(fit). 명백히, 이것은 Setter삽입을 사용한다. 원한다면, 이것은 대신 생성자 삽입을 사용할수 있다. 물론 이러한 DAO는 명백한 Java(예를 들어, 단위 테스트내)로 셋업될수 있다. 이것을 간단히 인스턴스화하고 요구되는 factory참조를 가진 setSessionFactory를 호출한다. Spring bean정의는 다음과 같을것이다.

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

이 DAO스타일의 가장 큰 장점은 Hibernate API에만 의존한다(import를 요구하는 Spring클래스는 없다.)는 것이다. 이것은 물론 침략하지 않는(non-invasiveness) 관점을 요구하고 Hibernate개발자에게는 좀더 자연스럽게 느껴질것이다.

어쨌든, DAO는 비록 Hibernate 자체의 예외 구조에 의존하기를 원하지 않더라도 호출자가 치명적인 것처럼 예외를 처리할수 있다는 것을 의미하는 명백한 HibernateException(체크되지 않았고, 그래서 선언되거나 catch되어야만 한다.)를 던진다. 최적화된 락 실패와 같은 것의 발생을 잡는것은 구현 전략에 호출자를 묶는 것을 시도하지 않고서는 가능하지 않다. 이 교환은 강력한 Hibernate-기반 그리고/또는 특별한 예외 처리가 필요하지 않은 애플리케이션에 받아들여질수 있을것이다.

운이 좋게도, Spring의 LocalSessionFactoryBean은 HibernateTransactionManager를 가진 현재의 Spring-기반 트랜잭션 성질의 Session을 반환하는 Spring트랜잭션 전략을 위해 Hibernate의 SessionFactory.getCurrentSession()메소드를 지원한다. 물론 이 메소드의 표준 행위가 남아있다. 만약 그렇다면(Spring의 JtaTransactionManager, EJB CMT 또는 명백한 JTA에 의해 다루어지는지는 상관없이)

진행중인 JTA트랜잭션과 관련된 현재 Session을 반환한다.

요약하면, Spring관리 트랜잭션에 참여할수 있는 동안 DAO는 명백한 Hibernate 3 API에 기초를 두고 구현될수 있다. 이것은 이미 Hibernate에 친숙한 사람에게 매력이 될것이다. 어쨌든, 이러한 DAO는 명백한 HibernateException를 던질것이다. Spring의 DataAccessException로의 변환은 명시적으로 발생해야만 한다.

12.2.6. 프로그램의 트랜잭션 구분(Demarcation)

하위 레벨의 데이터 접근 서비스의 가장 상위에서 트랜잭션은 애플리케이션의 더 높은 레벨내에서 구분될수 있다. 여기서는 또한 비지니스 서비스의 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 서비스 정의를 보여준다. 그리고 비지니스 메소드의 구현을 위한 예제를 보여준다.

```
<beans>
  ...

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="productDao" ref="myProductDao"/>
  </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.execute(
            new TransactionCallbackWithoutResult() {
                public void doInTransactionWithoutResult(TransactionStatus status) {
                    List productsToChange = productDAO.loadProductsByCategory(category);
                    // do the price increase...
                }
            }
        );
    }
}
```

12.2.7. 선언적인 트랜잭션 구분

대신에, Spring 컨테이너 XML파일내 설정된 AOP트랜잭션 인터셉터를 가지는 Java코드에서 명시적 트랜잭션 선언 API호출을 대체하는 것을 가능하게 하는 Spring의 선언적인 트랜잭션 지원을 사용할수 있다. 이것은 당신에게 반복적인 트랜잭션 선언 코드의 비지니스 서비스를 유지하는 것을 허용하고 비지니스 로직을 추가하는 것에 집중하도록 허용한다. 게다가 전달행위와 격리레벨같은 트랜잭션 구문은 설정파일내에서 변경될수도 있다. 그리고 비지니스 서비스 구현에는 어떠한 영향도 끼치지 않는다.

```
<beans>
    ...

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="product.ProductService"/>
    <property name="target">
        <bean class="product.DefaultProductService">
            <property name="productDao" ref="myProductDao"/>
        </bean>
    </property>
    <property name="interceptorNames">
      <list>
        <value>myTxInterceptor</value> <!-- the transaction interceptor (configured elsewhere) -->
      </list>
    </property>
  </bean>

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    // notice the absence of transaction demarcation code in this method
    // Spring's declarative transaction infrastructure will be demarcating transactions on your behalf
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDAO.loadProductsByCategory(category);
        ...
    }

    ...
}
```

Spring의 TransactionInterceptor는 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 트랜잭션을 위한 다음의 더 높은 레벨의 접근법은 ProxyFactoryBean를 사용하지 않고 트랜잭션형태를 가지도록 만들길 바라는 많은 서비스 객체를 가진다면 사용하기 더 쉬울것이다.

Note

당신이 이전의 것을 지속하지 않는다면 Section 9.5, “선언적인 트랜잭션 관리” 부분을 읽도록 강력히 장려한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- SessionFactory, DataSource, etc. omitted -->

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <bean id="myProductService" class="product.SimpleProductService">
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

12.2.8. 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 서비스는 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"/>
  </bean>

  <bean id="myDataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/myds2"/>
  </bean>

  <bean id="mySessionFactory1" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource1"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
        hibernate.show_sql=true
      </value>
    </property>
  </bean>

  <bean id="mySessionFactory2" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource2"/>
    <property name="mappingResources">
      <list>
        <value>inventory.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.OracleDialect
      </value>
    </property>
  </bean>

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

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory1"/>
  </bean>

  <bean id="myInventoryDao" class="product.InventoryDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory2"/>
  </bean>

  <!-- this shows the Spring 1.x style of declarative transaction configuration -->
  <!-- it is totally supported, 100% legal in Spring 2.x, but see also above for the sleeker, Spring 2.0 style -->
  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target">
      <bean class="product.ProductServiceImpl">
        <property name="productDao" ref="myProductDao"/>
        <property name="inventoryDao" ref="myInventoryDao"/>
      </bean>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
        <prop key="someOtherBusinessMethod">PROPAGATION_REQUIRES_NEW</prop>
        <prop key="*">PROPAGATION_SUPPORTS,readOnly</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.

HibernateTransactionManager는 평범한 특정 DataSource를 위한 JDBC접근 코드를 위해 Hibernate의해 사용되는 JDBC Connection을 추출할수 있다. 이것은 하나의 데이터베이스에 접근하는 한 JTA없이 완벽한 Hibernate/JDBC혼합 접근으로 높은 레벨의 트랜잭션 구분을 허락한다. 전달된 SessionFactory가 DataSource(LocalSessionFactoryBean의 "dataSource"를 통해)로 셋업된다면 HibernateTransactionManager는 JDBC트랜잭션처럼 Hibernate트랜잭션을 자동적으로 나타낼것이다. 대신, HibernateTransactionManager의 "dataSource" 프라퍼티를 통해 트랜잭션을 제공하는 DataSource는 명시적으로 선언될수 있다.

12.2.9. 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 JCA connector를 통해 SessionFactory를 배치하는 것은 J2EE서버의 관리구조내 추가된 값을 제공한다. 하지만 실질적인 값을 추가하지 않는다.

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.

Spring지향 트랜잭션이 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.

12.2.10. 트랜잭션이나 DataSource에 대한 애플리케이션 서버의 가짜(spurious) 경고는 더이상 활성화되지 않는다.

매우 엄격한 XADataSource구현물을 가지는 몇몇 JTA환경(현재 웹로직과 웹스피어의 몇몇 버전에서만 가능한)에서 JTA PlatformTransactionManager의 인지가 없이 설정된 Hibernate를 사용할때, 애플리케이션 서버 로그를 보여주기 위한 가까 경고나 예외가 가능하다. 이러한 경고와 예외는 트랜잭션이 더이상 활성화되지 않기 때문에 접속되는 connection이나 JDBC접근이 더이상 유효하지 않도록 영향을 주는 것들에 대해 말해줄것이다. 예제에서, 이것은 웹로직에서의 실질적인 예외이다.

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'.
   No further JDBC access is allowed within this transaction.
```

이 경고는 동기화(Spring과 함께)하기 위해 Hibernate가 JTA PlatformTransactionManager 인스턴스를 인지하도록 하여 분석하기 쉽다. 이것은 두가지 방법으로 수행될것이다.

☒ 만약 당신의 애플리케이션 컨텍스트에서 당신이 Spring의 JtaTransactionManager 예제를 위해 JTA PlatformTransactionManager를 직접적으로 얻고 이것을 공급한다면, 가장 쉬운 방법은 LocalSessionFactoryBean의 jtaTransactionManager프라퍼티의 값으로 이것에 대한 참조를 간단히 명시한다. Spring은 Hibernate를 위해 사용가능한 객체를 만들것이다.

☒ 당신이 JTA PlatformTransactionManager인스턴스를 이미 가지지 않는다(Spring의 JtaTransactionManager는 자체적으로 이것을 찾을수 있기 때문에). 그래서 당신은 이것을 직접적으로 찾기 위해 Hibernate를 설정하는 것을 대신할 필요가 있다. 이것은 Hibernate메뉴얼에서 언급된것처럼, Hibernate설정내 애플리케이션 서버 특정의 TransactionManagerLookup클래스를 설정하여 수행된다.

이것은 임의의 사용법을 위해 좀더 많은 것을 읽는것이 필요할 뿐 아니라 JTA PlatformTransactionManager를 인지하는 Hibernate를 가지거나 가지지 않는 일련의 이벤트가 언급되었다.

Hibernate가 JTA PlatformTransactionManager를 알면서 설정되지 않을때, JTA트랜잭션이 수행하는 일련의 이벤트는 다음과 같다.

☒ JTA 트랜잭션이 커밋된다.

☒ Spring의 JtaTransactionManager는 JTA트랜잭션에 동기화된다. 그래서 이것은 JTA트랜잭션 관리자에 의해 afterCompletion 콜백 메소드를 통해 콜백된다.

☒ 활동간에, 이것은 Hibernate의 afterTransactionCompletion 콜백(Hibernate캐시를 지우기 위해 사용되는)을 통해, Hibernate Session에서 명시적으로 Hibernate가 JDBC Connection을 close()하도록 시도하는 결과를 만드는 close() 메소드를 호출하여 Hibernate를 위해 Spring에 의해 콜백을 처리할수 있다.

☒ 몇몇 환경에서, 이 Connection.close()호출은 트랜잭션이 이미 커밋된후 애플리케이션 서버가 Connection사용을 더이상 고려하지 않는 것과 같이 경고나 에러를 처리한다.

Hibernate가 JTA PlatformTransactionManager를 알고 설정될때, JTA트랜잭션이 수행하는 일련의 이벤트는 다음과 같다.

☒ JTA 트랜잭션은 커밋할 준비가 된다.

☒ Spring의 JtaTransactionManager는 JTA트랜잭션에 동기화된다. 그래서 이것은 JTA트랜잭션 관리자에 의한 beforeCompletion 콜백 메소드를 통해 콜백된다.

☒ Spring은 Hibernate자체가 JTA트랜잭션으로 동기화되고 이전 시나리오와는 다르게 작동하는 것을 알아차린다. Hibernate Session이 전혀 닫힐 필요가 없다고 가정할때, Spring은 이것을 지금 닫을것이다.

☒ JTA 트랜잭션이 커밋된다.

☒ Hibernate는 JTA트랜잭션에 동기화된다. 그래서 이것은 JTA트랜잭션 관리자에 의한 afterCompletion 콜백메소드를 통해 콜백하고 캐시를 지울수 있다.

12.3. JDO

Spring은 Hibernate지원처럼 같은 스타일을 따르는 데이터 접근 전략으로 표준 JDO 1.0/2.0 API를 지원한다. 관련 통합 클래스는 org.springframework.orm.jdo 패키지에 존재한다.

12.3.1. PersistenceManagerFactory 셋업

Spring은 Spring 애플리케이션 컨텍스트내 local JDO PersistenceManagerFactory를 정의하는 것을 허용하는 LocalPersistenceManagerFactoryBean 클래스를 제공한다.

```
<beans>

  <bean id="myPmf" class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
    <property name="configLocation" value="classpath:kodo.properties"/>
  </bean>

  ...
</beans>
```

대신, PersistenceManagerFactory는 PersistenceManagerFactory 구현물 클래스를 직접 인스턴스화하는 것을 통해 셋업될수 있다. JDO PersistenceManagerFactory 구현물 클래스는 Spring bean정의에 자연스럽게 맞아 떨어지는 JDBC DataSource 구현물 클래스처럼 자바빈 패턴을 따르기 위해 제공된다. 이 셋업 스타일은 "connectionFactory" 프라퍼티로 전달되는 Spring-정의 JDBC DataSource를 언제나 지원한다. 예를 들어, 오픈 소스 JDO구현물인 JPOX(http://www.jpox.org):

```
<beans>

 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
   <property name="driverClassName" value="${jdbc.driverClassName}"/>
   <property name="url" value="${jdbc.url}"/>
   <property name="username" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>
 </bean>

 <bean id="myPmf" class="org.jpox.PersistenceManagerFactoryImpl" destroy-method="close">
   <property name="connectionFactory" ref="dataSource"/>
   <property name="nontransactionalRead" value="true"/>
 </bean>

 ...
</beans>
```

JDO PersistenceManagerFactory는 대개 특정 JDO구현물이 제공하는 JCA connector를 통해 J2EE애플리케이션 서버의 JNDI환경으로 셋업될수 있다. Spring의 표준 JndiObjectFactoryBean은 PersistenceManagerFactory와 같은 것을 가져오고 나타내기 위해 사용될수 있다. 어쨌든, EJB컨텍스트 외부에는, JNDI내 PersistenceManagerFactory를 유지하는 이득을 종종 앞도하지 않는다. 검토를 위해 Hibernate섹션내 "컨테이너 resource 대 local resource"를 보라. 인자들은 JDO에도 잘 적용된다.

12.3.2. JdoTemplate 과 JdoDaoSupport

각각의 JDO기반 DAO는 의존성 삽입(이를테면, bean프라퍼티 setter나 생성자의 인자를 통해서)을 통해 PersistenceManagerFactory를 가져올것이다. DAO는 주어진 PersistenceManagerFactory와 작동하는 명백한 JDO API에 대해 코딩될수 있다. 하지만 Spring의 JdoTemplate과 사용될것이다. :

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        JdoTemplate jdoTemplate = new JdoTemplate(this.persistenceManagerFactory);
        return (Collection) jdoTemplate.execute(new JdoCallback() {
            public Object doInJdo(PersistenceManager pm) throws JDOException {
                Query query = pm.newQuery(Product.class, "category = pCategory");
                query.declareParameters("String pCategory");
                List result = query.execute(category);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

콜백 구현물은 JDO데이터 접근을 위해 효과적으로 사용될수 있다. JdoTemplate은 PersistenceManager가 열렸는지 닫혔는지를 책임지고 트랜잭션내 자동으로 포함될것이다. 템플릿 인스턴스는 쓰레드에 안전하고(thread-safe) 재사용가능하다. 게다가 클래스 전역의 인스턴스 변수처럼 유지될수 있다. find, load, makePersistent, 또는 delete 호출중 하나와 같은 간단히 한개의 단계를 가진 실행을 위해, JdoTemplate은 한줄의 콜백 구현물처럼 대체될수 있는 대안인 편리한 메소드를 제공한다. 게다가, Spring은 PersistenceManagerFactory를 가져오기 위한 setPersistenceManagerFactory메소드와 하위 클래스에 의해 사용하기 위한 getPersistenceManagerFactory 과 getJdoTemplate을 가진 편리한 JdoDaoSupport base클래스를 제공한다. 복합적으로, 이것은 대개의 요구사항을 위한 간단한 DAO구현물을 허용한다.

```
public class ProductDaoImpl extends JdoDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return getJdoTemplate().find(
            Product.class, "category = pCategory", "String category", new Object[] {category});
```

```
    }
}
```

Spring의 JdoTemplate과 작동하는 대신에, 당신은 명시적으로 PersistenceManager를 열고 닫는 JDO API레벨에서 Spring기반의 DAO를 코딩할수 있다. 관련 Hibernate섹션에서 상세히 설명된것처럼, 이 접근법의 가장 중요한 장점은 당신의 데이터 접근코드가 체크된 예외를 던질수 있다는 것이다. JdoDaoSupport는 예외를 변환하는 것만큼 트랜잭션 성질을 가진 PersistenceManager을 가져오고 해제하기 위한 이러한 시나리오는 위해 다양한 지원 메소드를 제공한다.

12.3.3. 명백한 JDO API에 기반한 DAO 구현하기

DAO는 Spring의존성 없이 삽입된 PersistenceManagerFactory를 직접 사용하여 명백한 JDO API에 대해 작성될수 있다. 관련 DAO 구현물은 다음과 같을것이다.

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        try {
            Query query = pm.newQuery(Product.class, "category = pCategory");
            query.declareParameters("String pCategory");
            return query.execute(category);
        }
        finally {
          pm.close();
        }
    }
}
```

의존성 삽입 패턴을 따르는 위 DAO처럼, 이것은 Spring의 JdoTemplate에 대해 코딩된다면 Spring 애플리케이션 컨텍스트에 잘 맞다.

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

이런 DAO에 대한 중요한 이슈는 DAO들이 factory로 부터 언제나 새로운 PersistenceManager를 가진다는 것이다. Spring-관리 트랜잭션 성질을 가진 PersistenceManager에 접근하기 위해, 당신의 목표 PersistenceManagerFactory앞에서 당신의 DAO로 프록시를 전달하는 TransactionAwarePersistenceManagerFactoryProxy(Spring에 포함된것처럼)를 정의하는 것을 고려해보라.

```
<beans>
  ...

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
```

```
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

  ...
</beans>
```

당신의 데이터 접근 코드는 PersistenceManagerFactory.getPersistenceManager()메소드로 부터 트랜잭션 성질을 가지는 PersistenceManager를 가져올것이다. 후자의 메소드는 factory로부터 새로운 것을 얻기 전에 현재의 트랜잭션 성질을 가지는 PersistenceManager를 먼저 체크할 프록시를 통해 호출한다. PersistenceManager에서의 close()호출은 트랜잭션 PersistenceManager의 경우 무시될것이다.

만약 당신의 데이터 접근 코드가 활성화된 트랜잭션내에서 수행될것이라면, 당신의 DAO구현물이 간결하도록 유지하기 위해 선호할 PersistenceManager.close()호출과 전체 finally블럭을 생략하는 것이 안전하다.

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        Query query = pm.newQuery(Product.class, "category = pCategory");
        query.declareParameters("String pCategory");
        return query.execute(category);
    }
}
```

활성화된 트랜잭션을 의존하는 DAO와 함께, TransactionAwarePersistenceManagerFactoryProxy의 "allowCreate" 플래그를 꺼서 활성화된 트랜잭션을 강요하는 것이 추천된다.

```
<beans>
  ...

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
    <property name="allowCreate" value="false"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

  ...
</beans>
```

이러한 DAO스타일의 가장 큰 장점은 오직 JDO API에 의존적인 것이다. 요구되는 Spring클래스 import는 없다. 이것은 물론 침략적이지 않은(non-invasiveness) 관점을 요구하고 JDO개발자에게 좀더 자연스럽게 느껴질것이다.

어쨌든, DAO는 호출자가 비록 JDO자체의 예외 구조에 의존하길 원하지 않더라도 치명적인 것처럼 예외를 처리할수 있다는 것을 의미하는 명백한 JDOException(체크되지 않은, 그래서 선언되지 않거나 catch되지 않는)을 던진다. 최적화 락 실패와 같은 것을 야기하는 것을 잡는것은 구현 전략에 호출자를 묶지 않고서는 불가능하다. 이 교환(tradeoff)은 강력하게 JDO기반이고/이거나 어느 특별한 예외 처리가 필요하지 않은 애플리케이션에 받아들일수 있을것이다.

요약해서, DAO는 Spring관리 트랜잭션에서 작동할수 있는 반면에 명백한 JDO API에 기초를 두고 구현될수 있다. 이것은 JDO에 이미 친숙한 사람에게 특히 매력적이다. 어쨌든, 이러한 DAO는 명백한 JDOException를 던질것이다. Sprnig의 DataAccessException로의 변환은 명시적으로 발생할것이다(물론 원한다면).

12.3.4. 트랜잭션 관리

트랜잭션내에서 서비스 작업을 수행하기 위해, 당신은 Spring의 공통적인 선언적 트랜잭션 기능을 사용할수 있다. 예를 들어,

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
  ...

  <bean id="myTxManager" class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

</beans>
```

JDO는 영속성 객체를 변경할때 활성화된 트랜잭션을 요구한다. Hibernate와는 반대로, JDO에는 비-트랜잭션 속성의(non-transactional) flush와 같은 개념은 없다. 이러한 이유로, 선택된 JDO구현물은 특정 환경을 위해 셋업될 필요가 있다. 특히, 활성화된 JTA트랜잭션 자체를 감지하는 JTA동기화를 위해 셋업될 필요가 있다. 이것은 Spring의 JdoTransactionManager가 수행하는 것과 같이 local트랜잭션을 위해 필요하지 않다. 하지만 (Spring의 JtaTransactionManager나 EJB CMT / 명백한 JTA에 의해 다루어지는) JTA트랜잭션에 포함될 필요가 있다.

JdoTransactionManager는 같은 JDBC DataSource에 접근하는 JDBC접근 코드에 대해 JDO트랜잭션을 나타낼수 있다. 등록된 JdoDialect는 참조하는 JDBC Connection의 획득을 지원한다. 이것은 디폴트에 의해 JDBC-기반 JDO 2.0구현물을 위한 경우이다. JDO 1.0구현물을 위해, 사용자 정의 JdoDialect는 사용될 필요가 있다. JdoDialect 기법에 대한 상세한 정보를 위해서 다음 섹션을 보라.

12.3.5. JdoDialect

고급 기능으로, JdoTemplate 과 JdoTransactionManager는 "jdoDialect" bean프라퍼티로 전달되는 사용자정의 JdoDialect를 지원한다. 이런 시나리오에서, DAO는 JdoTemplate인스턴스 대신에 PersistenceManagerFactory참조를 받지 않을것이다. JdoDialect구현물은 언제나 업체 종속적인 방법으로 Spring이 제공하는 몇가지 고급기능을 가능하게 할수 있다.

☒ 특정 트랜잭션 성질 적용하기(사용자정의 격리레벨 이나 트랜잭션 타임아웃과 같은)

☒ 트랜잭션 성질을 가진 JDBC Connection 가져오기(JDBC기반 DAO에 나타내기 위해)

☒ 쿼리 타임아웃 적용하기(Spring관리 트랜잭션 타임아웃으로부터 자동으로 계산된)

☒ PersistenceManager를 열심히 flush하기(눈에 보이는 트랜잭션 성질을 JDBC기반 데이터 접근 코드에 변경하기)

☒ Spring DataAccessExceptions를 위한 JDOExceptions의 고급 번역

이러한 기능은 표준 API에 의해 다루어지지 않는 JDO 1.0구현물을 위해 특히 가치있다. JDO 2.0에서, 대부분의 기능은 표준적인 방법으로 지원된다. 나아가 Spring의 DefaultJdoDialect 은 디폴트로 관련 JDO 2.0 API 메소드를 사용한다(Spring 1.2에서). 특별한 트랜잭션 성질과 향상된 예외 분석을 위해, 이것은 업체 종속적인 JdoDialect 하위클래스를 끌어내기 위해 가치있다.

기능과 이 기능이 Spring의 JDO지원내에서 사용되는 방법에 대한 좀더 상세한 정보를 위해서 JdoDialect JavaDoc을 보라.

12.4. Oracle TopLink

Spring 1.2 이후, Spring은 Hibernate지원과 같은 스타일을 따르는 데이터 접근 전략처럼 Oracle TopLink (http://www.oracle.com/technology/products/ias/toplink)를 지원한다. TopLink 9.0.4 (Spring 1.2에서 정식제품(production) 버전) 와 10.1.3(Spring 1.2에서 여전히 베타인) 모두 지원된다. 관련 통합 클래스는 org.springframework.orm.toplink패키지에 위치한다.

Spring의 TopLink 지원은 Oracle TopLink팀과 함께 개발되었다. TopLink팀에게 매우 감사한다. 특히 모든 영역에 상세하게 도와준 Jim Clark에게 특히 감사한다.

12.4.1. SessionFactory 추상화

TopLink 자체는 SessionFactory추상화를 가지지 않는다. 대신, 다중 쓰레드 접근은 하나의 쓰레드 사용을 위해 순서대로 ClientSession를 일으킬수있는 중심적인 ServerSession 개념에 기초한다. 유연한 셋업 옵션을 위해, Spring은 다른 Session 생성 전략간에 전환을 가능하게 하는 TopLink를 위한 SessionFactory 추상화를 정의한다.

한군데에서 모든것이 준비된 가게처럼, Spring은 bean스타일 설정을 가진 TopLink SessionFactory를 정의하는 것을 허용하는 LocalSessionFactoryBean클래스를 제공한다. 이것은 TopLink세션 설정 파일의 위치를 가지고 설정될 필요가 있고 사용하는 Spring-관리 JDBC DataSource를 언제나 가져올것이다.

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="mySessionFactory" class="org.springframework.orm.toplink.LocalSessionFactoryBean">
    <property name="configLocation" value="toplink-sessions.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

  ...
</beans>
```

```
<toplink-configuration>

  <session>
    <name>Session</name>
    <project-xml>toplink-mappings.xml</project-xml>
    <session-type>
      <server-session/>
    </session-type>
    <enable-logging>true</enable-logging>
    <logging-options/>
  </session>

</toplink-configuration>
```

LocalSessionFactoryBean은 다중 쓰레드 TopLink ServerSession 아래에 유지하고 적절한 클라이언트 Session을 생성한다(명백한 Session, 관리되는 ClientSession, 또는 트랜잭션을 인지하는 Session(후자는 Spring의 TopLink지원에 의해 주로 내부적으로 사용된다.)). 이것은 하나의 쓰레드인 TopLink를 유지한다.

12.4.2. TopLinkTemplate 과 TopLinkDaoSupport

각각의 TopLink-기반 DAO는 의존성삽입(이를테면, bean프라퍼티 setter나 생성자의 인자를 통해)을 통해 SessionFactory를 가져올것이다. DAO는 주어진 SessionFactory로부터 Session을 가져오는 명백한 TopLink API에 대해 코딩될수 있지만 Spring의 TopLinkTemplate로 언제나 사용될것이다.

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;
```

```
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        TopLinkTemplate tlTemplate = new TopLinkTemplate(this.sessionFactory);
        return (Collection) tlTemplate.execute(new TopLinkCallback() {
            public Object doInTopLink(Session session) throws TopLinkException {
                ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
                findOwnersQuery.addArgument("Category");
                ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
                findOwnersQuery.setSelectionCriteria(
                    builder.get("category").like(builder.getParameter("Category")));

                Vector args = new Vector();
                args.add(category);
                List result = session.executeQuery(findOwnersQuery, args);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

콜백 구현물은 TopLink데이터 접근을 위해 효과적으로 사용될수 있다. TopLinkTemplate은 Session이 임의로 열렸는지 닫혔는지, 그리고 트랜잭션에 자동으로 포함되는지를 보장할것이다. 템플릿 인스턴스는 쓰레드에 안전하고 재사용가능하다. 인스턴스들은 클래스 전역의 인스턴스 변수처럼 유지될수 있다. executeQuery, readAll, readById, 또는 merge 등의 하나의 호출같은 간단한 한단계의 작동을 위해, JdoTemplate은 한줄의 콜백 구현물과 같은 것을 대체할수 있는 편리한 대체 메소드를 제공한다. 게다가, Spring은 편리한 SessionFactory를 가져오기 위한 setSessionFactory메소드, 하위클래스에 의해 사용하기 위한 getSessionFactory 와 getTopLinkTemplate를 제공하는 TopLinkDaoSupport base클래스를 제공한다. 복합적으로, 이것은 대개의 요구사항을 위한 간단한 DAO구현물을 허용한다.

```
public class ProductDaoImpl extends TopLinkDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));

        return getTopLinkTemplate().executeQuery(findOwnersQuery, new Object[] {category});
    }
}</programlisting>
```

사이드 노트: TopLink query 객체는 쓰레드에 안전하고 DAO내에서 캐시될수 있다. 이를테면, 시작시 생성되고 인스턴스 변수에서 유지된다.

Spring의 TopLinkTemplate과 작동하기 위한 대안으로, 당신은 명시적으로 Session을 열고 닫는 TopLink API에 기초를 둔 TopLink데이터 접근을 코딩할수 있다. Hibernate섹션에서 상세히 설명된것처럼, 이 접근법의 가장 큰 장점은 당신의 데이터 접근 코드가 체크된 예외를 던질수 있다는 것이다. TopLinkDaoSupport는 예외를 변환하는 것만큼 트랜잭션 성질을 가진 Session을 가져오고 해제하는 시나리오를 위한 다양한 지원 메소드를 제공한다.

12.4.3. 명백한 TopLink API에 기반하여 DAO구현하기

DAO는 Spring에 대한 의존성없이 삽입된 Session를 직접 사용하여 TopLink API에 대해서 작성될수 있다. LocalSessionFactoryBean에 의해 정의된 SessionFactory에 기초할 후자는 Spring의 TransactionAwareSessionAdapter를 통해 Session타입의 bean참조를 나타낸다.

TopLink의 Session인터페이스에서 정의된 getActiveSession() 메소드는 이러한 시나리오에서 현재의 트랜잭션 성질을 가지는 Session을 반환할것이다. 만약 여기에 활성화된 트랜잭션이 없다면, 오직 읽기전용 접근만 사용하도록 가정하는 공유 TopLink ServerSession을 반환할것이다. 여기에는 현재 트랜잭션에 속한 TopLink UnitOfWork를 반환하는 유사한 getActiveUnitOfWork() 메소드도 있다.

관련 DAO구현물은 다음과 같다.

```
public class ProductDaoImpl implements ProductDao {

    private Session session;

    public void setSession(Session session) {
        this.session = session;
    }

    public Collection loadProductsByCategory(String category) {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));

        Vector args = new Vector();
        args.add(category);
        return session.getActiveSession().executeQuery(findOwnersQuery, args);
    }
}
```

의존성 삽입 패턴을 따르는 위 DAO처럼, 이것은 Spring의 TopLinkTemplate에 대해 코딩된것과 유사하게 여전히 Spring애플리케이션 컨텍스트에 잘 맞다. Session 타입에 대한 bean참조를 나타내기 위해 사용되는 Spring의 TransactionAwareSessionAdapter는 DAO전달된다.

```
<beans>
  ...

  <bean id="mySessionAdapter"
      class="org.springframework.orm.toplink.support.TransactionAwareSessionAdapter">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="session" ref="mySessionAdapter"/>
  </bean>

  ...
</beans>
```

이 DAO스타일의 가장 큰 장점은 TopLink API에만 의존하는 것이다. import해야할 Spring 클래스는 없다. 물론 다른 API에 대해 의존성을 가지지 않는다는(non-invasiveness) 관점에서 매력적이고, TopLink개발자에게 좀더 자연스럽게 느껴지도록 한다.

어쨌든, DAO는 호출자가 비록 TopLink자체의 예외 구조에 의존하길 원하지 않더라도 치명적인 것처럼 예외를 처리할수 있다는 것을 의미하는 명백한 TopLinkException(체크되지 않은, 그래서 선언되지 않거나 catch되지 않는)을 던진다. 최적화 락 실패와 같은 것을 야기하는 것을 잡는것은 구현 전략에 호출자를

묶지 않고서는 불가능하다. 이 교환(tradeoff)은 강력하게 TopLink기반이고/이거나 어느 특별한 예외처리가 필요하지 않은 애플리케이션에 받아들일수 있을것이다.

DAO스타일의 단점은 TopLink의 표준 getActiveSession()기능이 JTA트랜잭션 내에서 작동한다는 것이다. 이것은 특히 local TopLink트랜잭션과 함께하지 않는 다른 트랜잭션 전략과 작동하지 않는다.

운이 좋게도, Spring의 TransactionAwareSessionAdapter는 TopLinkTransactionManager와 함께 현재 Spring이 관리하는 트랜잭션 성질을 가지는 Session을 반환하는 Spring트랜잭션 전략을 위한 TopLink의 Session.getActiveSession() 과 Session.getActiveUnitOfWork() 지원을 가지는 TopLink ServerSession를 위한 관련 프록시를 나타낸다. 물론, 이 메소드의 표준 행위는 남아있다. 만약 그렇다면(Spring의 JtaTransactionManager, EJB CMT, 또는 명백한 JTA에 의해 다루어지는지는 상관없이) 진행중인 JTA트랜잭션과 관련된 현재 Session을 반환한다.

요약해서, DAO는 Spring관리 트랜잭션에서 작동할수 있는 반면에 명백한 TopLink API에 기초를 두고 구현될수 있다. 이것은 TopLink에 이미 친숙한 사람에게 특히 매력적이다. 어쨌든, 이러한 DAO는 명백한 TopLinkException를 던질것이다. Sprnig의 DataAccessException로의 변환은 명시적으로 발생할것이다(물론 원한다면).

12.4.4. 트랜잭션 관리

트랜잭션내에서 서비스 작업을 수행하기 위해, 당신은 Spring의 공통적인 선언적 트랜잭션 기능을 사용할수 있다. 예를 들어,

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
  ...

  <bean id="myTxManager" class="org.springframework.orm.toplink.TopLinkTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

JDO는 영속성 객체를 변경할때 활성화된 UnitOfWork를 요구한다(당신은 명백한 TopLink Session에 의해 반환되는 객체를 변경하지 말아야 한다. 이러한 객체는 언제나 하위레벨(second-level) 캐시에서 직접 가져오는 읽기전용 객체이다). Hibernate와는 반대로, TopLink에는 비-트랜잭션 속성의(non-transactional) flush와 같은 개념은 없다. 이러한 이유로, 선택된 TopLink구현물은 특정 환경을 위해 셋업될 필요가 있다. 특히, 활성화된 JTA트랜잭션 자체를 감지하고 활성화된 관련 Session 과 UnitOfWork를 나타내는 JTA동기화를 위해 셋업될 필요가 있다. 이것은 Spring의 TopLinkTransactionManager가 수행하는 것과 같이 local트랜잭션을 위해 필요하지 않다. 하지만 (Spring의 JtaTransactionManager나 EJB CMT / 명백한 JTA에 의해 다루어지는) JTA트랜잭션에 포함될 필요가 있다.

당신의 TopLink기반 DAO코드에서, 현재의 UnitOfWork에 접근하기 위한 Session.getActiveUnitOfWork() 메소드를 사용하고 이것을 통해 쓰기(write)작업을 수행하라. 이것은 활성화된 트랜잭션(Spring관리 트랜잭션과 명백한 JTA트랜잭션 모두)내에서만 작동할것이다. 특별한 목적을 위해, 당신은 현재 트랜잭션에는 포함되지 않는 개별적인 UnitOfWork 인스턴스를 가질수 있다. 이것은 거의 필요한 경우가 없다.

TopLinkTransactionManager는 같은 JDBC DataSource에 접근하는 JDBC접근 코드에 대해 TopLink트랜잭션을 나타낼수 있다. 그리고 Top는 백엔드(backend)에서 JDBC와 작동하고 참조하는 JDBC Connection을 나타낼수 있다. 트랜잭션을 나타내기 위한 DataSource는 명시적으로 선언될 필요가 있다. 이것은 자동감지되지 않을것이다.

12.5. iBATIS SQL Maps

org.springframework.orm.ibatis패키지를 통해 Spring은 iBATIS SqlMaps(http://www.ibatis.com) 1.x 과 2.x을 지원한다. iBATIS지원은 JDBC 또는 Hibernate처럼 템플릿 스타일 프로그래밍을 지원하는 면에서 JDBC / Hibernate지원과 많은 공통점을 가진다. iBATIS지원은 Spring의 예외구조와 함께 작동하고 당신은 Spring이 가지는 모든 IoC특징을 즐기자.

트랜잭션 관리는 Spring의 표준기능을 통해 다루어질수 있다. JDBC Connection과 다르게 포함되는 특별한 트랜잭션 성질의 자원이 없기 때문에 iBATIS만을 위한 특별한 트랜잭션 전략은 없다. 나아가, Spring의 표준 JDBC DataSourceTransactionManager 나 JtaTransactionManager 으로도 완벽하게 충분하다.

12.5.1. 1.x and 2.x 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 12.1. 1.3 과 2.0을 위한 iBATIS SqlMaps 지원 클래스

특징	1.x	2.x
SqlMap(클라이언트)의 생성	SqlMapFactoryBean	SqlMapClientFactoryBean
템플릿 스타일의 헬퍼 클래스	SqlMapTemplate	SqlMapClientTemplate
MappedStatement를 사용하기 콜백	SqlMapCallback	SqlMapClientCallback
DAO를 위한 수퍼 클래스	SqlMapDaoSupport	SqlMapClientDaoSupport

12.5.2. iBATIS 1.x

12.5.2.1. SqlMap을 셋업하기

iBATIS SQL Maps를 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 Map를 정의한 다음, 우리는 iBATIS를 위한 설정파일을 생성한다. (sqlmap-config.xml):

```
<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.

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="sqlMap" class="org.springframework.orm.ibatis.SqlMapFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
  </bean>

  ...
</beans>
```

12.5.2.2. SqlMapTemplate 과 SqlMapDaoSupport 사용하기

SqlMapDaoSupport 클래스는 HibernateDaoSupport 과 JdoDaoSupport 과 유사한 지원 클래스를 제공한다. DAO를 구현해보자.

```
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 보는 것처럼, 우리는 쿼리를 수행하기 위해 미리 설정된 SqlMapTemplate를 사용하고 있다. Spring은 SqlMapFactoryBean을 사용하여 SqlMap을 초기화한다. 그리고 다음처럼 SqlMapAccountDao를 셋업할때, 당신은 모든것을 셋팅한것이다. 노트 : iBATIS SQL Maps 1.x를 사용할때, JDBC DataSource는 언제나 DAO에 명시된다.

```
<beans>
  ...

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="dataSource" ref="dataSource"/>
    <property name="sqlMap" ref="sqlMap"/>
  </bean>

</beans>
```

노트 : SqlMapTemplate 인스턴스는 생성자 인자로 DataSource와 SqlMap을 전달해서 직접 생성할수 있다. SqlMapDaoSupport base클래스는 SqlMapTemplate인스턴스를 간단하게 미리 초기화한다.

12.5.3. iBATIS SQL Maps 2.x

12.5.3.1. SqlMapClient 셋업하기

우리는 iBATIS 2.x를 사용해서 앞의 Account를 맵핑하기를 원한다면 우리는 다음의 SQLMaps Account.xml을 생성할 필요가 있다.

```
<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.

```
<sqlMapConfig>

  <sqlMap resource="example/Account.xml"/>

</sqlMapConfig>
```

iBATIS는 classpath로부터 자원을 로드한다는것을 기억하라. 그래서 classpath 어딘가에 Account.xml파일을 추가하는것을 확인하라.

우리는 Spring애플리케이션 컨텍스트내에서 SqlMapClientFactoryBean을 사용할수 있다. iBATIS SQL Maps 2.x에서, JDBC DataSource는 늦은(lazy) 로딩을 가능하게 하는 SqlMapClientFactoryBean에 언제나 명시된다.

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="$ {jdbc.driverClassName}"/>
    <property name="url" value="$ {jdbc.url}"/>
    <property name="username" value="$ {jdbc.username}"/>
    <property name="password" value="$ {jdbc.password}"/>
  </bean>

  <bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

  ...
</beans>
```

12.5.3.2. SqlMapClientTemplate과 SqlMapClientDaoSupport 사용하기

SqlMapClientDaoSupport클래스는 SqlMapDaoSupport와 유사한 지원 클래스를 제공한다. 우리는 DAO를 구현하기 위해 이것을 확장한다.

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        return (Account) getSqlMapClientTemplate().queryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().update("insertAccount", account);
    }
}
```

DAO에서, 우리는 애플리케이션 컨텍스트에서 SqlMapAccountDao를 셋업하고 이것을 SqlMapClient인스턴스에 묶은 후에 쿼리를 수행하기 위해 미리 설정된 SqlMapClientTemplate을 사용한다.

```
<beans>
  ...

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

노트 : SqlMapTemplate 인스턴스는 생성자의 인자로 SqlMapClient를 전달하여 직업 생성할수 있다. SqlMapClientDaoSupport base클래스는 SqlMapClientTemplate인스턴스를 미리 간단히 초기화한다.

SqlMapClientTemplate은 인자로 사용자정의 SqlMapClientCallback 구현물을 가지는 일반적인 execute 메소드를 제공한다. 이것은 예를 들어, 배치를 위해 사용될수 있다.

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {
    ...

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().execute(new SqlMapClientCallback() {
            public Object doInSqlMapClient(SqlMapExecutor executor) throws SQLException {
                executor.startBatch();
                executor.update("insertAccount", account);
                executor.update("insertAddress", account.getAddress());
                executor.executeBatch();
            }
        });
    }
}
```

대개, 본래의 SqlMapExecutor API에 의해 제공되는 이러한 작업의 조합은 콜백에서 사용될수 있다. SQLException은 자동으로 Spring의 포괄적인 DataAccessException구조로 변환될것이다.

12.5.3.3. 명백한 iBATIS API에 기반하여 DAO구현하기

DAO는 Spring에 대한 의존성없이 삽입된 SqlMapClient를 직접 사용하여 명백한 iBATIS API에 대해 작성될수 있다. 관련 DAO구현물은 다음처럼 보일것이다.

```
public class SqlMapAccountDao implements AccountDao {

    private SqlMapClient sqlMapClient;

    public void setSqlMapClient(SqlMapClient sqlMapClient) {
        this.sqlMapClient = sqlMapClient;
```

```
    }

    public Account getAccount(String email) {
        try {
            return (Account) this.sqlMapClient.queryForObject("getAccountByEmail", email);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }

    public void insertAccount(Account account) throws DataAccessException {
        try {
            this.sqlMapClient.update("insertAccount", account);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }
}
```

이러한 시나리오에서, iBATIS API에 의해 던져지는 SQLException은 사용자정의 형태(대개, 이것을 자신만의 애플리케이션 특성을 가지는 DAO예외로 포장하는)로 다루어질 필요가 있다. 애플리케이션 컨텍스트로 묶는 것은 명백한 iBATIS기반의 DAO가 여전히 의존성삽입 패턴을 따른다는 사실때문에 이전과 같을것이다.

```
<beans>
  ...

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

12.6. JPA

Spring JPA(org.springframework.orm.jpa 패키지에서 사용가능한)는 추가적인 기능을 제공하기 위한 근본적인 구현물을 인식하는 동안 Hibernate나 JDO와의 통합과 유사항 방법으로 Java Persistence API를 위한 편한 지원을 제공한다.

12.6.1. Spring환경에서 JPA 셋업하기

Spring JPA는JPA EntityManagerFactory를 셋업하는 두가지 방법을 제공한다.

12.6.1.1. LocalEntityManagerFactoryBean

LocalEntityManagerFactoryBean 는 데이터 접근을 위해 JPA를 사용하는 환경을 위해 적합한 EntityManager를 생성한다. factory bean은 JPA PersistenceProvider 자동감지 기법을 사용하고 대부분의 경우 퍼시스턴스 단위 이름만을 요구한다.

```
<beans>
...
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
 <property name="persistenceUnitName" value="myPersistenceUnit"/>
```

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

JNDI EntityManagerFactory(예를 들어 JTA환경에서)으로 교체하는 것은 XML설정을 변경의 문제이다.

```
<beans>
...
<jndi:lookup id="entityManagerFactory" jndi-name="jpa/myPersistenceUnit"/>
...
</beans>
```

12.6.1.2. LocalContainerEntityManagerFactoryBean

LocalContainerEntityManagerFactoryBean는 JPA EntityManagerFactory 에 대한 완전한 제어를 제공하고 사용자정의가 요구되는 환경에 적절하다. LocalContainerEntityManagerFactoryBean은 'persistence.xml' 파일에 기초하여 PersistenceUnitInfo 를 생성할것이다. dataSourceLookup 전략과 loadTimeWeaver를 제공한다. JNDI외부의 사용자정의 데이터소스와 작동하고 weaving처리에 대한 제어가 가능하다.

```
<beans>
 ...
 <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="someDataSource"/>
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
  </property>
 </bean>
</beans>
```

> 로드타임 weaving이 언제 필요한가.?
>
> 모든 JPA제공자가 JDK 에이전트의 필요성을 부과하는 것은 아니다. 당신의 제공자가 에이전트를 요구하지 않거나 다른 대안(예를 들어 사용자정의 컴파일러나 ant작업을 통한 빌드타임시 향상된 점을 적용한다.)을 가지지 않는다면, 로드타임 직조자(weaver)를 사용할 필요가 없다.

LoadTimeWeaver 인터페이스는 환경(웹 컨테이너/애플리케이션 서버)에 의존하는 특정 방법으로 플러그되는 JPA ClassTransformer 를 허용하는 Spring-기반의 클래스이다. JDK 5.0 agent을 통해 ClassTransformers를 채우는 것은 대개 효과적이지 않다. 에이전트는 entire virtual machine에 대해 작동하고 로드되는 모든 클래스를 조사한다. 때때로 제품 서버환경에서는 적절하지 않다.

Spring은 다양한 환경을 위해 많은 수의 LoadTimeWeaver 구현물을 제공한다. VM별로가 아닌 오직 클래스로더 별로 적용되는 ClassTransformer 를 허용한다.

12.6.1.2.1. Tomcat 셋업

Jakarta Tomcat의 디폴트 클래스로더는 사용되는 사용자정의 클래스로더를 허용하지만 클래스 번형은 지원하지 않는다. Spring은 Tomcat 클래스로더(WebappClassLoader)를 확장하는 TomcatInstrumentableClassLoader(org.springframework.instrument.classloading.tomcat 패키지내)를 제공하고 JPA ClassTransformer 인스턴스를 로드되는 모든 클래스를 '강화하도록' 허용한다. 짧게, JPA 변형자는 오직 웹 애플리케이션(TomcatInstrumentableClassLoader를 사용하는)내부에서만 적용될것이다.

사용자정의 클래스로더를 사용하기 위해

1. spring-tomcat-weaver.jar를 $CATALINA_HOME/server/lib($CATALINA_HOME는 Tomcat 설치 디렉토리를 표시한다)로 복사한다.

2. 웹 애플리케이션 컨텍스트 파일을 편집하여 Tomcat에 사용자정의 클래스로더를 사용하도록 지시한다.

```
<Context path="/myWebApp" docBase="/my/webApp/location" ...>
   <Loader loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
...
</Context>
```

   Tomcat 5.0.x 과 5.5.x 시리즈는 여러개의 컨텍스트 위치(서버 설정파일인($CATALINA_HOME/conf/server.xml), 배치된 모든 웹 애플리케이션에 영향을 끼치는 전역위치($CATALINA_HOME/conf/context.xml)와 서버에 배치된 웹 애플리케이션별 설정($CATALINA_HOME/conf/[enginename]/[hostname]/my-webapp-context.xml) 또는 웹 애플리케이션에 따르는 것(your-webapp.war/META-INF/context.xml))를 지원한다. 효율성을 위해, 웹 애플리케이션 내부의 설정 스타일은 JPA가 사용자정의 클래스로더를 사용할 애플리케이션일때만 추천된다. 사용가능한 컨텍스트 위치에 대한 좀더 상세한 정보를 위해서는 Tomcat 5.x 문서를 보라.

   이 시점(Tomcat 5.5.17)에서, server.xml 내부의 Loader 태그의 사용시 XML설정 파싱에 버그가 있다.

   Tomcat 4.x에서, 같은 context.xml파일을 사용할수 있고 $CATALINA_HOME/webapps 밑에 두거나 디폴트에 의해 사용자 정의 클래스로더를 사용하는 $CATALINA_HOME/conf/server.xml을 변경할수 있다. 좀더 많은 정보를 위해서 Tomcat 4.x documentation를 보라.

3. LocalContainerEntityManagerFactoryBean 를 설정할때 적절한 LoadTimeWeaver를 사용하라.

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  ...
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
  </property>
  ...
</bean>
```

이 기술을 사용하여 에이전트가 필요하지 않은 Tomcat에서 작동할수 있다. 이것은 JPA변형자가 적용된이후 다른 JPA구현물에 의존하는 애플리케이션을 호스팅할때 특히 중요하다. 오직 클래스로더 레벨에서 각각은 격리된다.

12.6.1.2.2. OC4J 셋업 (10.1.3.1+)

Oracle의 OC4J 클래스로더는 원시 바이트코드 번형 지원을 가지는 것처럼, JDK에이전트에서 LoadTimeWeaver로의 전환은 애플리케이션 Spring설정을 통해 수행될수 있다.

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  ...
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.oc4j.OC4JLoadTimeWeaver"/>
  </property>
  ...
```

```
</bean>
```

12.6.1.2.3. 일반적인 LoadTimeWeaver

클래스가 필요하지만 LoadTimeWeaver구현물에 의해 지원되지 않는 환경을 위해, JDK에이전트는 유일한 해결법이 될수 있다. 이러한 경우를 위해, Spring은 Spring특유의 VM에이전트(spring-agent.jar)를 필요로 하는 InstrumentationLoadTimeWeaver를 제공한다.

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
  </property>
</bean>
```

가상머신은 다음의 JVM옵션을 제공하여 Spring에이전트로 시작되어야만 하는 것을 노트하라.

```
-javaagent:/path/to/spring-agent.jar
```

12.6.1.3. 다중 퍼시스턴스 유닛을 다루기.

다중 퍼시스턴스 유닛 위치(예를 들면 classpath내 다양한 jar파일로 저장된)에 의존하는 애플리케이션을 위해, Spring은 중심적인 저장소처럼 작동하고 퍼시스턴스 유닛 복구 처리를 피하기 위한 PersistenceUnitManager를 제공한다. 디폴트 구현물은 파싱되고 퍼시스턴스 유닛 이름을 통해 나중에 가져오는 명시된 다중 위치(디폴트에 의해, classpath는 META-INF/persistence.xml 파일을 위해 검색된다.)를 허용한다.

```
<bean id="persistenceUnitManager" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
  <property name="persistenceXmlLocation">
    <list>
     <value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>
     <value>classpath:/my/package/**/custom-persistence.xml</value>
     <value>classpath*:META-INF/persistence.xml</value>
    </list>
  </property>
  <property>
   <map>
    <entry key="localDataSource" value-ref="local-db"/>
    <entry key="remoteDataSource" value-ref="remote-db"/>
   </map>
  </property>
  <!-- if no datasource is specified, use this one -->
  <property name="defaultDataSource" ref="remoteDataSource"/>

</bean>

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="persistenceUnitManager" ref="persistenceUnitManager"/>
 ...
</bean>
```

디폴트 구현물은 프라퍼티나 프로그램으로 처리하여 선언적으로나 PersistenceUnitPostProcessor를 통해 JPA제공자에게 제공하기 전에 퍼시스턴스 유닛 정보의 사용자정의를 허용한다. persistenceUnitManager가 명시되지 않았다면, LocalContainerEntityManagerFactoryBean에 의해 내부적으로 생성되고 사용될것이다.

12.6.2. JpaTemplate 과 JpaDaoSupport

각각의 JPA기반의 DAO는 의존성 삽입을 통해 EntityManagerFactory를 받을것이다. 이러한 DAO는 명백한 JPA에 대해 코딩되고 주어진 EntityManagerFactory이나 Spring의 JpaTemplate과 작동한다.

```
<beans>
...
  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>
...
</beans>
```

```
public class JpaProductDao implements ProductDao {

    private EntityManagerFactory entityManagerFactory;

    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.entityManagerFactoryeManagerFactory = emf;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        JpaTemplate jpaTemplate = new JpaTemplate(this.entityManagerFactory);

        return (Collection) jpaTemplate.execute(new JpaCallback() {

            public Object doInJpa(EntityManager em) throws PersistenceException {

                Query query = em.createQuery("from Product as p where p.category = :category");
                query.setParameter("category", category);
                List result = query.getResultList();
                // do some further processing with the result list
                return result;
            }
        });
    }
}
```

JpaCallback 구현물은 어떠한 JPA 데이터 접근도 허용한다. JpaTemplate은 EntityManager이 열리고 닫히고 자동으로 트랜잭션에 참가하는 것을 확인할것이다. 게다가 JpaTemplate은 자원을 정리하고 적절한 트랜잭션을 롤백하도록 만드는 예외를 다룬다. 템플릿 인스턴스는 쓰레드에 안전하고 재사용가능하며 둘러싼 클래스의 인스턴스 변수처럼 유지될수 있다. JpaTemplate은 한줄의 콜백 구현물을 대체할수 있는 대안이 되는 편리한 메소드를 가지고 검색, 로드, 병합 등과 같은 한가지 단계로 이루어진 action을 제공한다.

게다가, Spring은 get/setEntityManagerFactory와 하위클래스에 의해 사용되는 getJpaTemplate()을 제공하는 편리한 JpaDaoSupport 기본 클래스를 제공한다.

```
public class ProductDaoImpl extends JpaDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        Map<String, String> params = new HashMap<String, String>();
        params.put("category", category);
        return getJpaTemplate().findByNamedParams("from Product as p where p.category = :category", params);
    }
}
```

Spring의 JpaTemplate과 작동하는 것외에도, 자체적으로 명시적인 EntityManager 핸들링하여 JPA에 대해

Spring-기반의 DAO를 코딩할수 있다. 관련된 Hibernate부분에서 고안된것처럼, 이 접근법의 중요한 장점은 당신의 데이터 접근 코드가 체크된 예외를 던질수 있다는 것이다. JpaDaoSupport는 트랜잭션 EntityManager 를 가져오고 풀어주기 위해, 이 시나리오를 위한 다양한 지원 메소드를 제공한다.

12.6.3. 명백한 JPA에 기초한 DAO를 구현하기

Note

EntityManagerFactory인스턴스가 쓰레드에 안전한데 반해, EntityManager 인스턴스는 그렇지 않다. 삽입된 JPA EntityManager는 JPA스펙에 의해 정의된것처럼, 애플리케이션 서버의 JNDI환경으로부터 얻어낸 EntityManager처럼 작동한다. 이것은 현재 트랜잭션 속성을 가지는 EntityManager에 대한 모든 호출을 위임할것이다. 이것은 작업마다 새롭게 생성된 EntityManager를 되돌리고 쓰레드에 안전하게 만든다.

삽입된 EntityManagerFactory 나 EntityManager를 사용하여, 어떤 Spring 의존성을 사용하는것없이 명백한 JPA에 대해 코드를 작성하는 것이 가능하다. Spring은 PersistenceAnnotationBeanPostProcessor가 가능하다면 필드와 메소드 레벨에서 @PersistenceUnit 과 @PersistenceContext 어노테이션을 이해할수 있다. 관련 DAO구현물은 다음과 같을것이다.

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;

    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.entityManagerFactory.getEntityManager();
        try {
             Query query = em.createQuery("from Product as p where p.category = ?1");
             query.setParameter(1, category);
             return query.getResultList();

        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

Spring의 JpaTemplate에 대해 코딩된다면 위 DAO는 Spring에 대한 의존성을 가지지 않고 Spring 애플리케이션 컨텍스트에 잘 맞는다. 게다가, DAO는 디폴트 EntityManagerFactory의 삽입을 요구하는 어노테이션의 장점을 가진다.

```
<beans>
  ...
  <!-- JPA annotations bean post processor -->
  <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

이러한 DAO를 가진 중요한 이슈는 이것이 언제나 factory로부터 새로운 EntityManager를 얻는다는 것이다. 이것은 factory대신에 삽입되는EntityManager를 요청하여 쉽게 극복될수 있다.

```
public class ProductDaoImpl implements ProductDao {
```

```
    private EntityManager em;

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    public Collection loadProductsByCategory(String category) {
       Query query = em.createQuery("from Product as p where p.category = :category");
       query.setParameter("category", category);
       return query.getResultList();
    }
}
```

메소드와 필드 레벨의 삽입

의존성 삽입을 표시하는 어노테이션(@PersistenceUnit 와 @PersistenceContext 같은)은 클래스 내부의 메소드나 필드에 적용될수 있다. 그러므로 "메소드/필드 레벨의 삽입" 이라고 표현한다. 메소드-레벨이 삽입된 의존성을 처리하는데 반해 필드-레벨의 어노테이션은 간결하고 사용하기가 더 쉽다. 두 경우, 멤버 가시성(public, protected, private)은 문제되지 않는다.

클래스 레벨의 어노테이션은 어떤가.?

JEE 5 플랫폼에서, 자원 삽입을 위해서가 아닌 의존성 선언을 위해 사용된다.

삽입된 EntityManager는 Spring관리(진행중인 트랜잭션을 인지하는)된다. 비록 새로운 구현물이 메소드 레벨의 삽입(EntityManagerFactory대신에 EntityManager)을 선호하더라도, 어노테이션 사용으로 인해 애플리케이션 컨텍스트 XML내 요구되는 변경이 없다는 것을 노트하는 것이 중요하다.

이 DAO스타일의 중요한 장점은 Java 퍼시스턴스 API에 의존하는 것이다. Spring클래스의 import가 요구되는 것은 없다. 게다가, JPA어노테이션이 이해되는것처럼, 삽입은 Spring컨테이너에 의해 자동적으로 적용된다. 이것은 물론 비-침략적인 관점에서 호소되고 JPA개발자에 좀더 자연스럽게 느껴질것이다.

12.6.4. 예외 번역

게다가, DAO는 명백한 PersistenceException 예외 클래스(체크되지 않았고, 그래서 선언되거나 catch되지 않은)를 던지지만 호출자가 JPA 자체적인 예외 구조에 의존하는 것을 원하는 것을 제외하고 대개 치명적인 예외만을 처리할수 있는 것을 의미하는 IllegalArgumentException 와 IllegalStateException도 던진다. 최적화 락 실패와 같은 특정 사유를 잡는(catch)것은 호출자를 구현전략에 묶는것없이 불가능하다. 이 교환은 JPA-기반과/또는 특별한 예외 처리를 필요로 하지 않는 애플리케이션에 수락될수 있다. 어쨌든, Spring은 @Repository 어노테이션을 통해 투명하게 적용되는 예외 번역을 허용하는 솔루션을 제공한다.

```
@Repository
public class ProductDaoImpl implements ProductDao {
 ...
}
```

```
<beans>
  ...
  <!-- Exception translation bean post processor -->
  <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>
```

```
  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

후처리자는 모든 예외 번역자(PersistenceExceptionTranslator 인터페이스의 구현물인)와 모든 bean이 @Repository 어노테이션으로 만드는 advice를 자동으로 찾는다. 그래서 발견된 번역자는 가로챌수 있고 던져진 예외에서 적절한 번역을 적용할수 있다.

개요 : DAO는 Spring의 사용자정의 예외 구조를 위한 Spring-관리 트랜잭션, 의존성 삽입 그리고 투명한 예외 변환으로부터 이득이 되는동안 명백한 Java 퍼시스턴스 API와 어노테이션에 기초하여 구현될수 있다.

12.7. 트랜잭션 관리

트랜잭션내 서비스 작업을 수행하기 위해, 당신은 Spring의 공통 선언적인 트랜잭션 기능을 사용할수 있다. 예를 들면:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
  ...

  <bean id="myTxManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

Spring JPA는 JPA트랜잭션을 같은 JDBC DataSource에 접근하는 JDBC 접근 코드에 나타내기 위해 설정된 JpaTransactionManager를 허용한다. 등록된 JpaDialect가 근본적인 JDBC Connection의 획득을 지원한다. 특히, Spring은 Toplink와 Hibernate JPA구현물을 위한 dialect를 제공한다. JpaDialect 기법에 대한 상세한 설명을 위해서는 다음 부분을 보라.

12.8. JpaDialect

JpaTemplate의 고급기능처럼, JpaTransactionManager과 AbstractEntityManagerFactoryBean의 하위클래스는 "jpaDialect" bean프라퍼티에 전달되는 사용자정의 JpaDialect를 지원한다. 이러한 시나리오에서, DAO는 완전한 JpaTemplate 인스턴스 대신(예를 들어, JpaDaoSupport의 "jpaTemplate" 프라퍼티에 전달되는) 보다는 EntityManagerFactory 참조를 가져오지 않을것이다. JpaDialect 구현물은 언제나 업체에 종속적인 방법으로 Spring에 의해 제공되는 몇가지 고급기능이 가능하다.

☒ 특정 트랜잭션 구문 적용(사용자정의 격리 레벨이나 트랜잭션 타임아웃)

☒ 트랜잭션 성격을 가지는 JDBC Connection 가져오기(JDBC-기반의 DAO를 제시하기 위해)

☒ Spring DataAccessExceptions 을 위한 PersistenceExceptions의 고급 번역

이것은 특별한 트랜잭션 의미와 예외의 고급 번역을 위해 부분적으로 가치가 있다. 사용된 디폴트 구현물(DefaultJpaDialect)이 어떤 특별한 기능을 제공하지 않고 위 기능이 요구된다면 적절한 dialect가 명시되어야만 한다.

이 작업의 좀더 상세한 설명과 Spring의 JPA지원내 사용되는 방법을 위해서는 JpaDialect Javadoc를 보라.

Part III. The Web

This part of the reference documentation covers the Spring Framework's support for the presentation tier (and specifically web-based presentation tiers).

The Spring Framework's own web framework, Spring Web MVC, is covered in the first couple of chapters. A number of the remaining chapters in this part of the reference documentation are concerned with the Spring Framework's integration with other web technologies, such as Struts and JSF (to name but two).

This section concludes with coverage of Spring's MVC portlet framework.

☒hapter 13, 웹 MVC framework

☒hapter 14, 통합 뷰 기술들

☒hapter 15, 다른 웹 프레임워크들과의 통합

☒hapter 16, 포틀릿(Portlet) 통합

Chapter 13. 웹 MVC framework

13.1. 소개

Spring의 웹 MVC framework는 설정가능한 핸들러 맵핑들, view해석, 로케일과 파일 업로드를 위한 지원같은 테마해석과 함께 핸들러에 요청을 할당하는 DispatcherServlet을 기반으로 디자인되었다. 디폴트 핸들러는 ModelAndView handleRequest(request,response)메소드를 제공하는 매우 간단한 Controller 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 폼 컨트롤러가 필요없다. 이것이 Struts와 가장 큰 차이점이다.

> "확장을 위해 열기(Open for extension)..."
>
> Spring MVC(그리고 대개 Spring)내에서 지나치게 구조화된 디자인 규칙중 하나는 "확장을 위해 열고, 변경을 위해 닫는다(Open for extension, closed for modification)" 규칙이다.
>
> 이 규칙이 여기서 언급되는 이유는 Spring MVC내 핵심 클래스의 많은 메소드가 final로 표기되기 때문이다. 이것은 물론 개발자가 자신의 행위를 제공하기 위해 메소드를 오버라이드 할수가 없다는 것을 의미한다. 이것은 디자인에 의한 것이고 성가시도록 임의로 수행되지는 않는다.
>
> Seth Ladd와 다른 사람들에 의해 만들어진 'Expert Spring MVC and Web Flow' 책은 'A Look At Design' 부분에 117페이지에서 이것을 고수하기 위해 이 규칙과 동기를 설명한다.
>
> 위 책을 볼수가 없다면, 다음 글에서 "이 메소드를 왜 오버라이드 할수 없는가.?" 라는 부분에 관심을 가질수 있을것이다.
>
> 1. Bob Martin, The Open-Closed Principle (PDF)

Spring Web MVC는 당신에게 command나 form객체로 어떤 객체를 사용하는 것을 허용한다. 여기서 프레임워크 특유의 인터페이스나 기본 클래스를 구현할 필요는 없다. Spring의 데이터 바인딩은 매우 유연하다. 예를 들면 이것은 시스템 에러가 아닌 애플리케이션에 의해서 평가될수 있는 유효성 에러와 같은 타입 불일치를 처리할수 있다. 이것이 의미하는 모든것은 당신이 비지니스 객체의 프라퍼티를 간단히 중복하거나 form객체내 타입화되지 않은 문자열을 유효하지 않은 서브밋을 다룰수 있도록 또는 문자열을 변환할 필요가 없다는 것이다. 대신, 비지니스 객체에 직접 바인딩하는 것을 종종 선호한다. Struts와의 가장 큰 차이점은 Action 와 ActionForm 같은 필수 기본 클래스를 내장하지 않는다는 것이다.

WebWork와 비교해서 Spring은 좀더 차별적인 객체 역할을 가진다. 이것은 Controller, 선택적으로 command나 form 객체, 그리고 view에 전달되는 모델의 개념을 지원한다. 모델은 command나 form객체를 일반적으로 포함하지만 또한 임의의 참조 데이터도 포함한다. 대신에 WebWork Action은 그러한 모든 역할을 하나의 객체로 조합한다. WebWork는 당신에게 당신의 폼 일부처럼 존재하는 비지니스 객체를 사용하도록 하지만 그것들을 만듦으로써 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 view해석은 매우 유연하다. Controller 구현물은 (ModelAndView를 위해 null을 반환하여)응답에 직접적으로 view를 작성할수 있다. 대개의 경우에 ModelAndView인스턴스는 view이름과 빈 이름과 관련 객체(참조 데이터를 포함하는 command또는 폼 같은)를 포함하는 모델 Map으로 구성된다. View이름 해석은 빈 이름, 프라퍼티 파일, 당신 자신의 ViewResolver구현물을 통해 쉽게 설정가능한 사항이다. 사실 모델(MVC에서 M)은 view기술의 완벽한 추상화를 허용하는 Map 인터페이스에 기반한다. 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.

13.1.1. 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.

13.1.2. Spring MVC의 특징

Spring의 웹 모듈은 다음을 포함하는 유일한 웹 지원의 가치를 제공한다.

☒ 역할의 분명한 분리 - 컨트롤러, 유효성 체커, command 객체, 폼 객체, 모델 객체, DispatcherServlet, 핸들러 맵핑, view해석자, 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, command, 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 command 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 view해석 - 핸들러 맵핑과 view해석 전략은 간단한 URL기반의 설정에서 정교한, 목적이 내포된 해석 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 해석, Spring 태그 라이브러리를 사용하거나 사용하지 않은 JSP를 위한 지원, JSTL을 위한 지원, 추가적인 연결자를 위한 필요성없이 Velocity를 위한 지원

☒ Spring 태그 라이브러리처럼 알려진 간단하지만 강력한 JSP 태그 라이브러리는 데이터 바인딩과 테마와 같은 기능을 제공한다. 사용자 정의 태그는 마크업 코드의 개념에서 최대한의 유연성을 허용한다. 태그 라이브러리 기술자에 대한 정보를 위해, Appendix D, spring.tld를 보라.

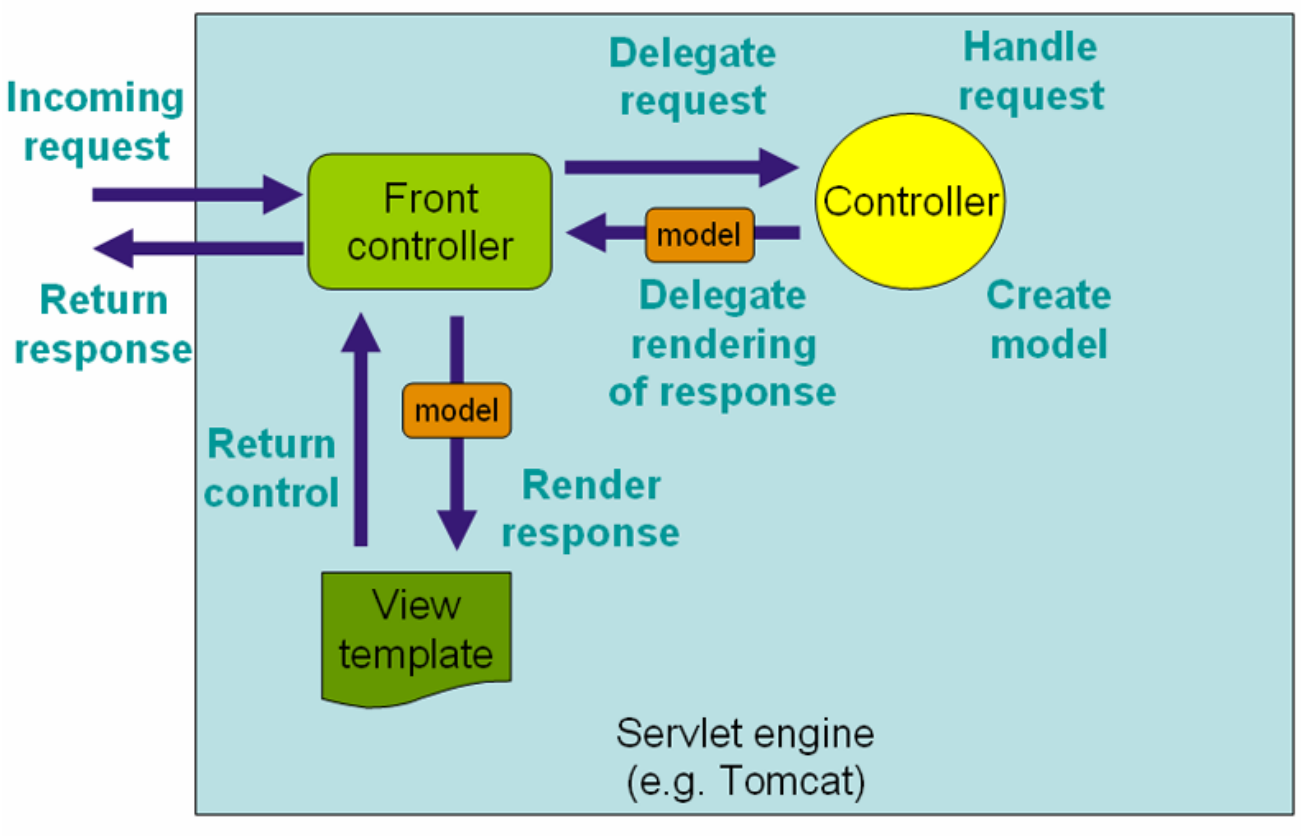
☒ Spring 2.0에서 소개된 JSP 폼 태그 라이브러리는 JSP페이지 작성을 좀더 쉽게 만들어준다. 태그 라이브러리 기술자에 대한 정보를 위해, Appendix E, spring-form.tld를 보라.

☒ 생명주기 bean은 현재 HTTP 요청이나 HTTP Session에 범위화된다. Spring MVC자체의 구체적인 기능이 아닐뿐 아니라 Spring MVC가 사용하는 WebApplicationContext 컨테이너의 기능도 아니다. 이러한 bean범위는 Section 3.5.3, “The other scopes” 에서 상세히 언급되었다.

13.2. 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 IoC컨테이너와 완벽하게 통합되고 당신이 Spring이 가지는 다른 기능을 사용하도록 허락한다.

Spring 웹 MVC DispatcherServlet의 요청 처리 워크플로우는 다음의 그림에서 볼수 있다. 패턴을 이해하는 독자는 DispatcherServlet이 “Front Controller” 디자인 패턴(이것은 Spring 웹 MVC가 다른 많은 웹 프레임워크와 공유하는 패턴이다.)의 표현이라는 것을 인식할것이다.



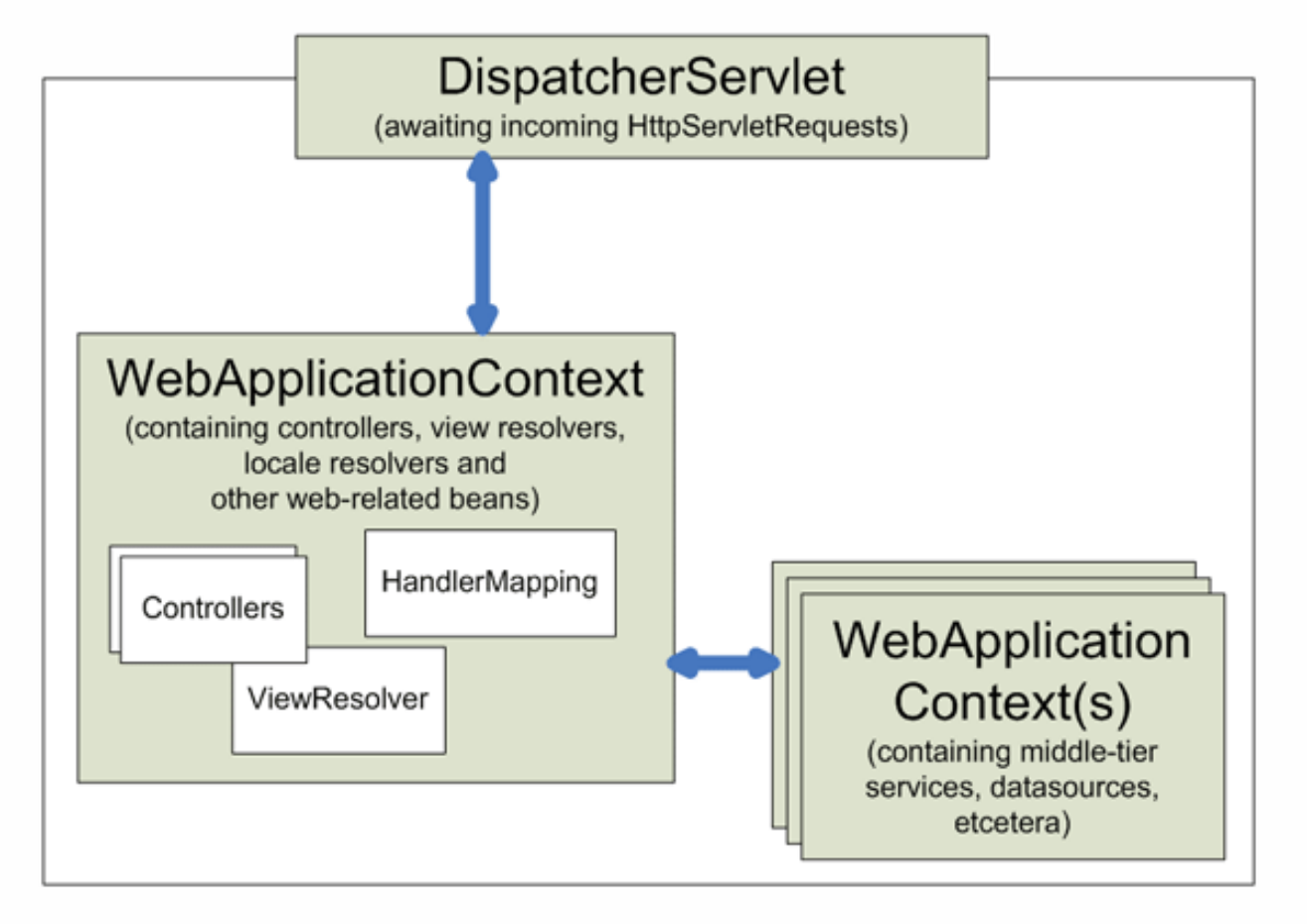
Spring 웹 MVC내 요청 처리 워크플로우

DispatcherServlet은 실제 Servlet(이것은 HttpServlet 기본 클래스로부터 상속된다.) 이고 웹 애플리케이션의 web.xml내 선언된다. 다룰 DispatcherServlet를 원하는 요청은 같은 web.xml 파일내 URL맵핑을 사용하여 맵핑될것이다. 이것은 표준 J2EE서블릿 설정이다. DispatcherServlet 선언과 맵핑의 예제는 아래에서 볼수 있다.

```
<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 'example' DispatcherServlet에 의해 다루어질것이다. 이것은 Spring 웹 MVC를 셋팅하는 첫번째 단계이다. Spring 웹 MVC 프레임워크에 의해 사용되는 다양한 bean(그리고 DispatcherServlet 자체)은 설정될 필요가 있다.

Section 3.9, "ApplicationContext" 에서 설명되는 것처럼, Spring 내 ApplicationContext 인스턴스는 범위화 될수 있다. 웹 MVC 프레임워크내에서 각각의 DispatcherServlet은 가장 상위의 WebApplicationContext에 이미 정의된 모든 bean을 상속하는 자체적인 WebApplicationContext를 가진다. 정의된 상속 bean은 서블릿-특유의 범위에서 오버라이드될수 있다. 그리고 새로운 범위의 특성을 가지는 bean은 주어진 서블릿 인스턴스에 지역적으로 정의될수 있다.



Spring MVC내 컨텍스트 구조

DispatcherServlet의 초기화에서, 프레임워크는 당신의 웹 애플리케이션의 WEB-INF 디렉토리에 있는 [servlet-name]-servlet.xml라고 명명된 파일을 찾고 거기에 정의된 bean을 생성한다(전역 범위에서 같은 이름을 가지고 정의된 bean의 정의를 오버라이드한다.).

다음의 DispatcherServlet 서블릿 설정('web.xml' 파일내)을 보라.

```
<web-app>
    ...
    <servlet>
        <servlet-name>golfing</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>golfing</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

위 서블릿 설정 대신에, 당신은 애플리케이션내 '/WEB-INF/golfing-servlet.xml' 라고 불리는 파일을 가질 필요가 있을것이다. 이 파일은 Spring 웹 MVC 특유의 컴포넌트 모두 포함할것이다. 이 설정파일의 정확한 위치는 서블릿 초기화 파라미터를 통해 변경될수 있다.

WebApplicationContext는 웹 애플리케이션을 위해 필요한 몇몇 추가적인 특징을 가지는 명료한 ApplicationContext의 확장이다. 이것은 테마(Section 13.7, "테마(themes) 사용하기"를 보라)를 해석하는 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 bean을 위해, 실용적인 디폴트는 그것들을 설정할 걱정을 하지 않도록 해준다.

Table 13.1. WebApplicationContext내 특별한 빈즈들

bean타입	설명
컨트롤러	Controllers는 MVC에서 'C' 부분을 나타내는 컴포넌트이다.
핸들러 맵핑	핸들러 맵핑은 그것들이 어떠한 기준(예를 들면 컨트롤러와 함께 명시된 URL에 일치할때)에 일치할때 수행되어야 할 선-처리자, 후-처리자와 컨트롤러의 목록의 수행을 다룬다.
view 해석자	View 해석자는 view이름을 view로 해석하는 능력을 가진 컴포넌트이다.
로케일 해석자	locale 해석자는 국제화된 view를 제공할수 있도록 하기 위해 클라이언트가 사용하는 로케일을 해석하는 능력을 가진 컴포넌트이다.
테마 해석자	테마 해석자는 예를 들어 개인화된 레이아웃을 제공하는 것처럼, 웹 애플리케이션이 사용할수 있는 테마를 해석하는 능력을 가진다.
멀티파트 파일 해석자	멀티파트 파일 해석자는 HTML폼으로 부터 파일 업로드 처리를 위한 기능을 제공한다.
핸들러 예외 해석자	핸들러 예외 해석자는 view에 예외를 맵핑하거나 좀더 복잡한 예외 핸들링 코드를 구현하는 기능을 제공한다.

DispatcherServlet이 사용하기 위해 셋업되고 특정 DispatcherServlet을 위해 요청이 접수되면 언급된 DispatcherServlet이 요청을 처리하길 시작한다. 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 해석자는 요청을 처리할때(view를 표현하고, 데이터를 준비하는 등등) 사용하기 위한 로케일을 프로세스가 해석하도록 요청에 바운드된다. 만약 당신이 해석자를 사용하지 않는다면 이것은 어떠한 영향도 끼치지 않는다. 그래서 당신이 로케일 해석이 필요하지 않다면 그것을 사용하지 않아도 된다.
3. 테마 해석자는 view가 사용하기 위한 테마가 무엇인지 결정하는 요청에 바운드된다. 테마 해석자는 사용하지 않는다면 어떠한 영향도 끼치지 않는다. 그래서 당신이 테마를 사용할 필요가 없다면 당신은 그것을 무시할수 있다.
4. 만약 멀티파트 해석자가 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 13.8.2, “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, 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 핸들러 예외 해석자에 의해 처리된다. 이러한 예외 해석자를 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 LastModified인터페이스의 long getLastModified(request)메소드가 클라이언트에 반환된다.

당신은 web.xml파일이나 서블릿 초기화 파라미터에 컨텍스트 파라미터를 추가함으로써 Spring의 DispatcherServlet을 사용자 정의화 할수 있다. 가능한 사항을 밑에서 보여준다.

Table 13.2. DispatcherServlet 초기화 파라미터

파라미터	설명
contextClass	서블릿에 의해 사용되는 컨텍스트를 초기화하기 위해 사용될 WebApplicationContext을 구현하는 클래스. 만약 이 파라미터가 명시되지 않는다면 XmlWebApplicationContext가 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.

13.3. 컨트롤러

컨트롤러의 개념은 MVC디자인 패턴의 일부이다(좀더 상세하게는 MVC의 'C'이다). 컨트롤러는 서비스 인터페이스에 의해 정의된 애플리케이션 행위에 접근한다. 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 특성의 컨트롤러, command-기반 컨트롤러, 마법사 스타일의 로직을 수행하는 컨트롤러 그리고 더 다양한 방법을 포함한다.

Spring의 컨트롤러 구조의 기본은 아래에 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 하나의 메소드를 정의한다. 이러한 3개의 개념은 Spring MVC구현물(-ModelAndView 와 Controller)을 위한 기본이다. Controller인터페이스가 추상적인 동안 Spring은 필요한 많은 기능을 포함한 많은 Controller 구현물을 제공한다. Controller인터페이스는 모든 컨트롤러의 요구되는 가장 공통적인 기능(-요청을 처리하고 모델및 view를 반환하는)을 정의한다.

13.3.1. AbstractController 와 WebContentGenerator

기본적인 내부구조를 제공하기 위해서 Spring의 모든 Controller는 캐시지원과 예를 들면 mimetype셋팅을 제공하는 클래스인 AbstractController로부터 상속되었다.

Table 13.3. 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 던져서 진척되는) 할것이다.
requiresSession	컨트롤러가 작업을 하기위해 HTTP 세션을 필요로 하는지 하지 않는지를 표시. 이 기능은 모든 컨트롤러에 의해 제공된다. 만일 컨트롤러가 요청을 받을때 세션이 존재하지 않는다면 사용자는 던져지는 ServletException을 사용해서 정보화한다.
synchronizeSession	당신이 사용자의 HTTP 세션에서 동기화되는 컨트롤러에 의해 핸들링하길 원한다면 이것을 사용하라.
cacheSeconds	당신이 HTTP응답에서 캐시를 직업적으로 생성할 컨트롤러를 원한다면 여기에 양수값을 명시하라. 디폴트에 의해 이 프라퍼티의 값은 -1로 셋팅된다. 그래서 생성되는 응답에 어떤 캐싱도 포함되지 않을것이다.
useExpiresHeader	HTTP 1.0호환 "Expires"헤더를 명시하기 위해 당신의 컨트롤러를 부분적으로 개량하라. 디폴트에 의해 이 프라퍼티의 값은 true이다.
useCacheHeader	HTTP 1.1호환 "Cache-Control"헤더를 명시하기 위해 당신의 컨트롤러를 부분적으로 개량하라. 디폴트에 의해 이 프라퍼티의 값은 true이다.

당신의 컨트롤러(당신을 위해 작업을 이미 수행하는 많은 다른 컨트롤러때문에 추천되지는 않는)를 위한 기본 클래스로 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("hello");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}
```

```
<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
</bean>
```

위 클래스와 웹 애플리케이션 컨텍스트내 선언은 매우 간단히 작동중인 컨트롤러를 얻기 위해 당신이 핸들러 맵핑(Section 13.4, “Handler mappings”을 보라)을 셋업할 필요가 있는 모든것이다. 이 컨트롤러는 다시 체크하기 전에 2분동안 캐시하는것을 클라이언트에게 알리는 직접적인 캐시를 생성할것이다. 이 컨트롤러는 직접 작성된(흠.. 별로 좋지 않은) 코드의 view를 반환할것이다.

13.3.2. 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(Java클래스내 viewname을 하드코딩할 필요를 제거하고)는 사실을 제외하면 위 예제와 기본적으로 같다.

UrlFilenameViewController는 URL을 조사하고 요청으로 부터 파일이름을 가져오고 viewname으로 그것을 사용한다. 예를 들어, http://www.springframework.org/index.html요청의 파일명은 index이다.

13.3.3. 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 13.4. MultiActionController에 의해 제공되는 특징

특징	설명
delegate	MultiActionController을 위한 두가지 사용 시나리오가 있다. 당신이 MultiActionController를 세분화하고 하위 클래스에서 MethodNameResolver에 의해 해석될 메소드를 정의하거나 MethodNameResolver에 의해 해석될 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 해석할 필요가 있을것이다. 이 전략은 MethodNameResolver 인터페이스에 의해

특징	설명
	정의된다. MultiActionController는 해석자를 제공할수 있는 프라퍼티를 나타낸다.

다중액션 컨트롤러를 위해 명시한 메소드는 다음의 시그너처에 따를 필요가 있다.

```
// anyMeaningfulName can be replaced by any methodname
public [ModelAndView | Map | void] anyMeaningfulName(HttpServletRequest, HttpServletResponse [, Exception | AnyObject]);
```

메소드 오버로딩은 MultiActionController을 혼동시키기 때문에 허락되지 않는다. 게다가 당신은 명시한 메소드에 의해 던져질 예외 핸들링 능력을 가지는 exception handlers를 정의할수 있다.

(선택적) Exception 인자는 java.lang.Exception 이나 java.lang.RuntimeException의 하위클래스라면 어떤 예외가 될수 있다. (선택적) AnyObject 인자는 어떤 클래스가 될수 있다. 요청 파라미터는 편리한 소모를 위해 이 객체로 바운드 될것이다.

아래에서 유효한 MultiActionController 메소드 시그너처의 몇몇 예제를 보라.

표준 시그너처 (Controller 인터페이스 메소드를 반영한다.)

```
public ModelAndView doRequest(HttpServletRequest, HttpServletResponse)
```

이 시그너처는 요청에서 나오는 파라미터로 활성화(바운드)될 Login 인자를 받아들인다.

```
public ModelAndView doLogin(HttpServletRequest, HttpServletResponse, Login)
```

Exception 핸들링 메소드를 위한 시그너처

```
public ModelAndView processException(HttpServletRequest, HttpServletResponse, IllegalArgumentException)
```

이 시그너처는 void 반환타입을 가진다(아래의 Section 13.11, “설정에 대한 규칙” 를 보라).

```
public void goHome(HttpServletRequest, HttpServletResponse)
```

이 시그너처는 Map 반환타입을 가진다. (아래의 Section 13.11, “설정에 대한 규칙” 를 보라).

```
public Map doRequest(HttpServletRequest, HttpServletResponse)
```

MethodNameResolver는 들어온 요청에 기반하여 메소드 이름을 해석하는 책임을 진다. 아래에서 Spring이 특별히 제공하는 3개의 MethodNameResolver 구현물에 대한 상세를 보라.

- ☒ ParameterMethodNameResolver - 요청 파라미터를 해석하고 메소드이름(http://www.sf.net/index.view?testParam=testIt는 호출될 testIt(HttpServletRequest, HttpServletResponse) 메소드로 결과를 낼것이다.)처럼 그것을 사용하는 능력. paramName 프라퍼티는 조사된 요청 파라미터를 명시한다.
- ☒ InternalPathMethodNameResolver - 요청 경로로 부터 파일이름을 가져오고 메소드 이름처럼 그것을 사용한다. (http://www.sf.net/testing.view는 호출될 testing(HttpServletRequest, HttpServletResponse)메소드내 결과를 낼것이다.)
- ☒ PropertiesMethodNameResolver - 요청 URL을 메소드 이름에 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 해석자는 PathMatcher과 함께 작동한다. 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"/>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
  <property name="methodNameResolver" ref="paramResolver"/>
  <property name="delegate" ref="sampleDelegate"/>
</bean>

<bean id="sampleDelegate" class="samples.SampleDelegate"/>

## together with

public class SampleDelegate {

    public ModelAndView retrieveIndex(HttpServletRequest req, HttpServletResponse resp) {

        return new ModelAndView("index", "date", new Long(System.currentTimeMillis()));
    }
}
```

위에서 보여진 위임을 사용할때, 우리는 정의된 메소드에 URL을 일치시키기 위해 PropertiesMethodNameResolver를 사용할수 있다.

```
<bean id="propsResolver" class="org....mvc.multiaction.PropertiesMethodNameResolver">
  <property name="mappings">
    <value>
        /index/welcome.html=retrieveIndex
        /**/notwelcome.html=retrieveIndex
        /*/user?.html=retrieveIndex
    </value>
  </property>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
    <property name="methodNameResolver" ref="propsResolver"/>
    <property name="delegate" ref="sampleDelegate"/>
</bean>
```

13.3.4. Command Controllers

Spring의 command 컨트롤러는 Spring 웹 MVC패키지의 필수적인 부분이다. command 컨트롤러는 데이터 객체와 상호작동하기 위한 방법을 제공하고 HttpServletRequest에서 명시된 데이터 객체로 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 command 컨트롤러가 어떤것이 있는지 조사해보자.

☒ AbstractCommandController - 당신 자신의 컨트롤러를 생성하기 위해 당신이 사용할수 있는 command 컨트롤러는 당신이 명시하는 데이터 객체로 요청 파라미터를 바인드하는 능력을 가진다. 이 클래스는 폼 기능을 제공하지 않는다. 이것은 어쨌든 유효성체크 기능을 제공하고 요청 파라미터값을 채우는 command 객체가 무엇인지 컨트롤러내 명시하자.

☒ AbstractFormController - 추상 컨트롤러는 폼 서브밋 지원을 제공한다. 이 컨트롤러를 사용하면 당신은 폼을 만들고 당신이 컨트롤러내에서 가져온 command 객체를 사용하여 그것들을 형상화 할수 있다. 사용자가 폼을 채우고 난 뒤 AbstractFormController는 필드를 바인드하고 command객체를 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 command 객체를 가진 폼을 생성할때 명백한 폼 컨트롤러는 좀더 많은 지원을 제공한다. SimpleFormController는 당신이 command 객체, 폼을 위한 view이름, 폼 서브밋이 성공했을때 당신이 사용자에게 보여주기를 원하는 페이지를 위한 view이름을 명시하도록 한다.

☒ AbstractWizardFormController - 클래스 이름이 제시하는것처럼 이것은 당신의 wizard 컨트롤러가 이것을 확장해야만 하는 추상 클래스이다. 이것은 당신이 validatePage(), processFinish() 그리고 processCancel()를 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 command 객체를 참조하기 위해 사용될 문자열처럼 가진다.

AbstractFormController의 어떠한 인스턴스처럼 당신은 당신의 폼으로 부터 데이터를 활성화할 자바빈인 command 객체를 사용하기 위해 요구된다. 당신은 command 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 onBindAndValidate()이다.

마지막으로 이것은 현재 페이지에 유효성 체크가 실패한다면 마법사내에서 뒤로나 앞으로 움직이는 것을 사용자에게 허락하는 getTargetPage()로 부터 호출할수 있는 setAllowDirtyBack() 과 setAllowDirtyForward()를 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.

13.4. 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 13.4.3, "요청 가로채기 - HandlerInterceptors 인터페이스" 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.).

13.4.1. 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 폼 컨트롤러(command와 폼 컨트롤러의 좀더 상세한 정보를 위해서 Section 13.3.4, "Command Controllers" 을 보라)과 폼을 표현하는 JSP view(또는 Velocity템플릿)가 제공된다고 얘기해보자. BeanNameUrlHandlerMapping을 사용할때 우리는 다음처럼 적당한 form Controller를 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"/>
    <property name="successView" value="account-created"/>
    <property name="commandName" value="account"/>
    <property name="commandClass" value="samples.Account"/>
  </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>
```

Note

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.

13.4.2. 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>

    <!-- maps the sample dispatcher to *.html -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

위 web.xml설정의 일부는 같은 디스패처(dispatcher) 서블릿에 의해 다루어지기 위해 .html과 .form으로 끝나는 모든 요청을 할당한다.

```
<beans>

    <!-- no 'id' required, HandlerMapping beans are automatically detected by the DispatcherServlet -->
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /*/account.form=editAccountFormController
                /*/editaccount.form=editAccountFormController
                /ex/view*.html=helpController
                /**/help.html=helpController
            </value>
        </property>
    </bean>

    <bean id="helpController"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

    <bean id="editAccountFormController"
          class="org.springframework.web.servlet.mvc.SimpleFormController">
        <property name="formView" value="account"/>
        <property name="successView" value="account-created"/>
        <property name="commandName" value="Account"/>
        <property name="commandClass" value="samples.Account"/>
    </bean>
<beans>
```

이 핸들러 맵핑은 어떤 디렉토리내 'help.html'을 UrlFilenameViewController(Section 13.3, "컨트롤러"에서 찾을수 있는 컨트롤러에 대해서 추가적인)인 'helpController'로 요청을 보낸다. 'ex'디렉토리내에서 'view'로 시작하고 '.html'로 끝나는 자원을 위한 요청은 'helpController'로 경로가 지정될것이다. 둘 이상의 맵핑은 'editAccountFormController'을 위해 명시된다.

13.4.3. 요청 가로채기 - HandlerInterceptors 인터페이스

Spring의 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">
            <value>
                /*.form=editAccountFormController
                /*.view=editAccountFormController
            </value>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
          class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime" value="9"/>
        <property name="closingTime" value="18"/>
    </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) 클래스(HandlerInterceptorAdapter라는)를 가진다.

13.5. 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 14, 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.

13.5.1. view를 결정하기 - ViewResolver 인터페이스

Section 13.3, "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 13.5. 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.
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/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

view이름으로 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"/>
    <property name="defaultParentView" value="parentView"/>
</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.

캐싱부분 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.

13.5.2. 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(체인내에서 마지막 결정자(resolver)처럼 자동으로 위치가 정해지는)와 엑셀 view(InternalResourceViewResolver에 의해 지원되지 않는)를 정의하기 위한 XmlViewResolver로 이루어진다.

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

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in 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 Exception을 던질것이다.

당신은 염두해 두고 있는것을 유지해야한다. view결정자의 규칙은 view결정자가 찾을수 없는 view를 표시하기 위해 null을 반환할수 있다는 것을 말한다. 어쨌든 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.

13.5.3. view로 redirect하기

이미 언급된것처럼, 컨트롤러는 대개 view결정자가 특정 view기술을 결정하는 논리적인 view명을 반환한다. Servlet/JSP 엔진을 통해 실질적으로 처리되는 JSP와 같은 view기술을 위해, 이것은 대개 Servlet API의 RequestDispatcher.forward() 나 RequestDispatcher.include()를 통해 내부 forward나 include를 이슈화하는것을 끝낼 InternalResourceViewResolver/InternalResourceView을 통해 다루어진다. Velocity, XSLT, 등등과 같은 다른 view기술을 위해, view자체는 응답 스트림에 컨텐츠를 생성한다.

때때로 view가 표현되기 전에 클라이언트로 HTTP redirect해서 돌아가는 것을 이슈화하는 것이 바람직하다. 이것은 하나의 컨트롤러가 POST된 데이터를 가지고 호출되고 응답이 다른 컨트롤러(폼서브밋으로부터 성공한 경우에)로 위임되었을때의 예제를 위해 바람직하다. 이 경우, 대개의 내부 forward는 다른 컨트롤러가 POST된 같은 데이터를 볼것이라는 것을 의미한다. 결과가 사용자가 폼 데이터의 이중 서브밋을 수행하는 것의 가능성을 제거할 것을 표시하기 전에 redirect하기 위한 다른 이유를 기대하는 것을 혼동할수 있다면 잠재적으로 문제의 소지가 있다. 브라우저는 초기 POST를 보낼것이다. 현재 페이지는 GET보다는 POST의 결과를 반영하지 않는다. 그래서 refresh를 통해 같은 데이터를 다시 POST할수 있는 방법은 없다. refresh는 결과페이지의 GET을 강제로 수행할뿐 아니라 초기 POST데이터를 다시 보내지는 않는다.

13.5.3.1. RedirectView

컨트롤러 응답의 결과처럼 redirect를 강제로 수행하기 위한 하나의 방법은 생성하기 위한 컨트롤러와 Spring RedirectView의 인스턴스를 반환하는 것이다. 이 경우, DispatcherServlet은 대개의 view결정 기법을 사용하지 않을것이지만 이미 주어진 view처럼 작동하기 위해 요청할것이다.

RedirectView는 HTTP redirect로 클라이언트 브라우저로 돌아갈 HttpServletResponse.sendRedirect()호출을 이슈화하는것을 간단히 끝낸다. 모든 모델 속성은 HTTP쿼리 파라미터처럼 간단히 나타난다. 이것은 모델이 오직 문자열 형태 HTTP쿼리 파라미터로 변환될수 있는 객체만(대개 String이나 String으로 변환가능한)을 포함해야하는것을 의미한다.

만약 RedirectView을 사용하고 view가 컨트롤러 자체에 의해서 생성된다면, 최소한 redirect URL이 컨트롤러로 삽입되는것을 선호할것이다. 그래서 이것은 컨트롤러로 태울(baked into)뿐 아니라 view명과 함께 컨텍스트내 설정된다.

13.5.3.2. redirect: 접두사

RedirectView사용이 잘 작동하는 동안, 컨트롤러자체가 RedirectView를 생성한다면, 컨트롤러는 redirection이 발생하는것을 감지한다. 이것은 실제로 차선책이고 너무 타이트하게 커플링한다. 컨트롤러는 응답이 다루어지는 방법에 대해 다루지는 않을것이다. 이것은 대개 삽입되는 view명의 개념에서만 생각한다.

특수한 redirect: 접두사는 이것이 달성되도록 허용한다. 만약 view명이 접두사 redirect를 가지고 반환된다면, UrlBasedViewResolver(과 모든 하위클래스)는 redirect가 필요하다는것을 특별히 표시하는것처럼 이것을 인식할것이다. view명의 나머지는 redirect URL처럼 처리될것이다.

net 효과는 컨트롤러가 RedirectView를 반환하는 것과 같다. 하지만 컨트롤러 자체는 논리적인 view명의 개념에서 다루어진다. redirect:/my/response/controller.html과 같은 논리적인 view명은 현재 서블릿 컨텍스트에 상대적으로 redirect 될것이다. 반면에 redirect:http://myhost.com/some/arbitrary/path.html와 같은 이름은 절대경로의 URL로 redirect할것이다. 중요한 것은 redirect view명이 다른 논리적인 view명처럼 컨트롤러내로 삽입되는것이다. 컨트롤러는 redirection이 발생하는 것을 감지하지 않는다.

13.5.3.3. forward: 접두사

UrlBasedViewResolver와 하위클래스에 의해 해석될 view명을 위한 접두사인 특별한 forward:를 사용하는 것은 가능하다. 이 모든것은 URL을 고려하는 view명의 나머지에 InternalResourceView(궁극적으로는 RequestDispatcher.forward() 하는)를 생성한다. 그러므로, InternalResourceViewResolver/InternalResourceView를 사용할때 이 접두사를 결코 사용하지 않는다. 하지만 당신이 다른 view기술을 사용할때 잠재적으로 사용한다. 몇몇 경우 Servlet/JSP엔진에 의해 다루어지는 자원에 대해 발생하는 forward를 강요한다. 만약 당신이 많은 것을 할 필요가 있다면, 당신은 다중 view결정자를 묶을수 있다.

redirect: 접두사를 사용할때, 접두사를 가진 view명이 컨트롤러로 삽입된다면, 어떤 특별한 것을 감지하지 않는 컨트롤러는 응답을 다루는 개념에서 발생한다.

13.6. 로케일 사용하기.

Spring구조의 대부분은 Spring 웹 MVC프레임워크가 하는것처럼 국제화를 지원한다. DispatcherServlet은 당신에게 클라이언트 로케일을 사용하여 메시지를 자동적으로 결정하도록 한다. 이것은LocaleResolver객체를 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 13.4.3, “요청 가로채기 - HandlerInterceptors 인터페이스” 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.

13.6.1. AcceptHeaderLocaleResolver

이 로케일 결정자는 클라이언트의 브라우저에 의해 보내어진 요청내 accept-language헤더를 조사한다. 언제나 이 헤더 필드는 클라이언트의 OS시스템의 로케일을 포함한다.

13.6.2. CookieLocaleResolver

이 로케일 결정자는 로케일이 정의되었다면 보기 위해 클라이언트내 존재하는 Cookie를 조사한다. 만약 쿠키가 존재한다면 그 특정 로케일을 사용한다. 로케일 결정자의 프라퍼티를 사용하여 당신은 최대생명(maximum age)만큼 쿠키의 이름을 명시할수 있다. 아래에서 CookieLocaleResolver를 정의하는 예제를 보라.

```
<bean id="localeResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
    <property name="cookieMaxAge" value="100000">

</bean>
```

Table 13.6. CookieLocaleResolver 프라퍼티

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.

13.6.3. SessionLocaleResolver

SessionLocaleResolver는 당신에게 사용자의 요청이 속한 세션으로 부터 로케일을 가져오도록 허락한다.

13.6.4. LocaleChangeInterceptor

당신은 LocaleChangeInterceptor을 사용해서 로케일을 변경할수 있다. 이 인터셉터는 하나의 핸들러 맵핑(Section 13.4, “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"/>
</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 bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

siteLanguage라는 이름의 파라미터를 포함하는 모든 *.view형태의 자원 호출은 지금 로케일을 변경할것이다. 그래서 다음 URL인 http://www.sf.net/home.view?siteLanguage=nl을 위한 요청은 사이트 언어를 네델란드어로 변경할것이다.

13.7. 테마(themes) 사용하기

13.7.1. 소개

Spring 웹 MVC프레임워크가 제공되는 테마(theme) 지원은 테마가 적용된 애플리케이션의 룩앤필을 좀더 향상되게 해준다. 테마는 기본적으로 애플리케이션의 시작적인 스타일에 영향을 주는 정적 자원(대개 스타일시트와 이미지)의 집합이다.

13.7.2. 테마 정의하기

당신이 웹 애플리케이션내 테마를 사용하고자 한다면, 당신은 org.springframework.ui.context.ThemeSource을 셋업헤야할것이다. WebApplicationContext인터페이스는 ThemeSource를 확장하지만 전용 구현물에 대한 응답을 위임한다. 디폴트로 위임은 classpath의 가장 상위의 프라퍼티파일로부터 로드하는 org.springframework.ui.context.support.ResourceBundleThemeSource가 될것이다. 사용자정의 ThemeSource구현물을 사용하길 원하거나 ResourceBundleThemeSource의 basename접두사를 설정할 필요가 있다면, 예약된 이름은 "themeSource"을 가지고 애플리케이션 컨텍스트내 bean을 등록할수 있다. 웹 애플리케이션 컨텍스트는 bean을 자동으로 감지하고 시작한다.

ResourceBundleThemeSource를 사용할때, 테마는 간단한 프라퍼티 파일내 정의된다. 프라퍼티 파일은 테마를 만드는 자원을 목록화한다. 다음은 프라퍼티 파일의 예제이다.

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

프라퍼티 파일의 key들은 view코드로부터 테마의 요소를 참조하기 위해 사용되는 이름들이다. JSP를 위해서는 대개 spring:message태그와 매우 유사한 spring:theme 사용자정의 태그를 사용하여 수행된다. 다음의 JSP 일부분은 룩앤필을 커스터마이징하기 위해 위에서 정의된 테마를 사용한다.

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
    <head>
        <link rel="stylesheet" href="<spring:theme code="styleSheet"/>" type="text/css"/>
    </head>
    <body background="<spring:theme code="background"/>">
        ...
    </body>
</html>
```

디폴트에 의해, ResourceBundleThemeSource는 빈(empty) basename접두사를 사용한다. 결과처럼, 프라퍼티 파일은 classpath의 가장 상위에서 로드될것이다. 그래서 우리는 classpath의 가장 상위 디렉토리(이를테면, /WEB-INF/classes)에 cool.properties 테마 정의를 둘것이다. ResourceBundleThemeSource는 테마의 완벽한 국제화를 허용하는 표준 Java자원 번들 로딩 기법을 사용한다. 예를 들어, 우리는 특별한 배경이미지(이를테면, 네델란드어의 텍스트를 가진)를 참조하는 /WEB-INF/classes/cool_nl.properties를 가질수 있다.

13.7.3. 테마 결정자(resolver)

지금 우리는 정의된 테마를 가진다. 해야할 것중에 남은것은 사용할 테마를 결정하는 것이다. DispatcherServlet은 사용할 ThemeResolver구현물을 찾기 위해 "themeResolver"라는 이름의 bean을 검색할것이다. 테마 결정자는 LocalResolver와 같은 방법으로 작동한다. 이것은 특정 요청을 위해 사용되는 테마를 감지하고 요청의 테마를 변경할수 있다. 다음의 테마 결정자는 Spring에 의해 제공된다.

Table 13.7. ThemeResolver 구현물

클래스	상세설명
FixedThemeResolver	"defaultThemeName" 프라퍼티를 사용하여 셋팅된 고정된 테마를 선택한다.
SessionThemeResolver	테마는 사용자 HTTP세션에서 유지된다. 이것은 단지 각각의 세션을 위해 한번만 셋팅될 필요가 있을뿐 아니라 세션간에 지속되지 않는다.
CookieThemeResolver	선택된 테마는 user-agent 머신의 쿠키에 저장된다.

Spring은 또한 간단한 요청 파라미터를 포함하는 모든 요청에 테마를 변경하도록 해주는 ThemeChangeInterceptor도 제공한다.

13.8. Spring의 multipart (파일업로드) 지원

13.8.1. 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.

13.8.2. 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"/>
</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"/>
</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.

13.8.3. 폼에서 파일업로드를 다루기.

MultipartResolver가 그 작업을 마친후 요청은 다른것처럼 처리될것이다. 이것을 사용하기 위해 당신은 파일 업로드 필드를 가진 폼을 생성하고 Spring이 폼의 필드에 바인드 하도록 하자. 사용자가 실제로 파일을 업로드하기 위해, 우리는 (HTML)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 있는것처럼, bean의 프라퍼티가 byte[]를 가지는 데 이어 명명된 필드를 생성한다. 게다가 우리는 multipart필드을 인코딩하는 방법을 브라우저에 알려주는 인코딩 속성(enctype="multipart/form-data")을 추가했다(이것을 잊지말라.).

다른 프라퍼티처럼 그것은 자동적으로 당신이 ServletRequestDatabinder로 사용자 정의 편집기(editor)를 등록하기 위해 객체내 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>
        <!-- lets use the Commons-based implementation of the MultipartResolver interface -->
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /upload.form=fileUploadController
            </value>
        </property>
    </bean>

    <bean id="fileUploadController" class="examples.FileUploadController">
        <property name="commandClass" value="examples.FileUploadBean"/>
        <property name="formView" value="fileuploadform"/>
        <property name="successView" value="confirmation"/>
    </bean>

</beans>
```

그후 컨트롤러와 파일 프라퍼티를 가지는 실질적인 빈을 생성하라.

```
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
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

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.

파일이 (폼 지원)객체의 문자열타입 프라퍼티에 일관적으로 바운드되는 예제는 다음과 같을것이다.

```
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
        String 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 a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class, new StringMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

물론, 이것은 일반적인 텍스트 파일을 업로드하는 개념을 설명하는 마지막 예제이다(이것은 이미지 파일을

업로드하는 경우에는 잘 작동하지 않을것이다.)

세번째(그리고 마지막) 옵션은 (폼 지원)객체의 클래스에 선언된 MultipartFile 프라퍼티에 직접적으로 바인딩하는 것이다. 이 경우, 수행될 형변환이 없기 때문에 사용자 정의 PropertyEditor를 등록할 필요가 없다.

```
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
        MultipartFile 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);
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
}
```

13.9. Spring의 폼 태그 라이브러리 사용하기.

2.0버전에서, Spring은 JSP와 Spring 웹 MVC를 사용할때 폼 요소를 다루기 위해 포괄적인 데이터 바인딩 태그를 제공한다. 각각의 태그는 사용하기에 친숙하고 직관적이도록 만들기 위해 관련 HTML태그의 속성을 지원한다. 태그로 생성된 HTML은 HTML 4.01/XHTML 1.0 과 호환된다.

다른 폼/input 태그 라이브러리와는 달리, Spring의 폼 태그 라이브러리는 당신의 컨트롤러가 다루는 command객체와 참조 데이터에 접근하여 Spring MVC와 통합된다. 다음의 예제에서 보는것처럼, 폼 태그는 JSP개발, 읽기및 유지보수를 좀더 쉽게 만든다.

폼 태그를 통해 각각의 태그가 사용된 예제를 보자. 우리는 해설이 필요한 태그에 대한 생성된 HTML부분을 포함한다.

13.9.1. 설정

폼 태그 라이브러리는 spring.jar내 포함되어 있다. 라이브러리 서술자는 spring-form.tld 이다.

이 라이브러리로부터 태그를 사용하기 위해, JSP페이지의 가장 상위에 다음의 지시자를 추가한다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

form 이 태그명의 접두사일때, 당신은 이 라이브러리로부터 이 태그를 사용한다.

13.9.2. form 태그

이 태그는 HTML 'form' 태그를 표시하고 바인딩을 위해 바인딩 경로를 내부 태그로 나타낸다. PageContext내 command객체에 두어서, command객체는 내부 태그에 의해 접근할수 있다. 이 라이브러리내 다른 모든 태그는 form 태그의 내포된다.

User 라고 불리는 도메인 객체가 있다고 가정하자. 이것은 firstName 과 lastName과 같은 프라퍼티를 가지는 JavaBean이다. 우리는 form.jsp를 반환하는 폼 컨트롤러의 폼지원객체처럼 이것을 사용할것이다. 아래는 form.jsp의 예제이다.

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

firstName 과 lastName 값들은 페이지 컨트롤러에 의해 PageContext에 위치하는 command객체로부터 가져온다. 내부 태그가 form태그와 함께 사용되는 방법의 좀더 복잡한 예제를 보기 위해 계속 보라.

생성된 HTML은 표준적인 형태처럼 보인다.

```
<form method="POST">
    <table>
      <tr>
          <td>First Name:</td>
          <td><input name="firstName" type="text" value="Harry"/></td>
          <td></td>
      </tr>
      <tr>
          <td>Last Name:</td>
          <td><input name="lastName" type="text" value="Potter"/></td>
          <td></td>
      </tr>
      <tr>
          <td colspan="3">
            <input type="submit" value="Save Changes" />
          </td>
      </tr>
    </table>
```

```
</form>
```

선행 JSP는 폼 지원 객체의 변수명이 'command'라고 가정한다. 다른 이름으로 폼 지원객체를 모델에 넣는다면, 명명된 변수에 폼을 바인딩할수 있다.

```
<form:form commandName="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

13.9.3. input 태그

이 태그는 바운드(bound)값을 사용하여 'text' 타입으로 HTML 'input' 태그를 표시한다. 이 태그의 예제를 위해. Section 13.9.2, "form 태그" 를 보라.

13.9.4. checkbox 태그

이 태그는 'checkbox' 타입의 HTML 'input' 태그를 표시한다.

User가 신문 예약구독자와 취미목록과 같은 선택물을 가진다고 가정해보자. 아래는 Preferences 클래스의 예제이다.

```
public class Preferences {

    private boolean receiveNewsletter;

    private String[] interests;

    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }
```

```
    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

form.jsp 는 다음과 같을것이다.

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
            <td></td>
        </tr>

        <tr>
            <td>Interests:</td>
            <td>
                <%-- Approach 2: Property is of an array or of type java.util.Collection --%>
                Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox path="preferences.interests"
                    value="Defence Against the Dark Arts"/>
            </td>
            <td></td>
        </tr>
        <tr>
            <td>Favourite Word:</td>
            <td>
                <%-- Approach 3: Property is of type java.lang.Object --%>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
            <td></td>
        </tr>
    </table>
</form:form>
```

checkbox 태그에 대해 3가지 접근법이 있다.

☒ 접근법 1 - java.lang.Boolean 타입의 값에 할당할때, input(checkbox)는 할당된 값이 true라면 'checked' 처럼 표시된다. value 속성은 setValue(Object) 값 프라퍼티의 값을 결정한다.

☒ 접근법 2 - array 이나 java.util.Collection 타입의 값을 할당할때, input(checkbox)은 설정된 setValue(Object)값이 할당된 Collection에 존재한다면 'checked' 처럼 표시된다.

☒ 접근법 3 - 다른 타입의 값을 할당하기 위해, input(checkbox)은 설정된 setValue(Object) 이 할당된 값과 일치한다면, 'checked' 로 표시된다.

이러한 접근법에도 불구하고, 같은 HTML구조가 생성된다. 아래는 몇가지 checkbox를 가지는 HTML이다.

```
<tr>
    <td>Interests:</td>
    <td>
```

```
        Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox"
            value="Defence Against the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
    <td></td>
</tr>
```

당신이 보기를 기대하지 않는 것은 각각의 checkbox뒤 추가적인 hidden필드이다. HTML페이지내 checkbox가 체크되지 않았을때, 폼이 서브릿되었을때 HTTP요청 파라미터의 일부로 서버에 보내지지는 않을것이다. 그래서 우리는 Spring 폼 데이터 바인딩이 작업하도록 하기 위해 HTML내 이러한 흠을 위한 대안이 필요하다. checkbox 태그는 각각의 checkbox를 위한 밑줄("_") 에 의해 접두사 처리된 hidden파라미터를 포함하는 Spring규칙을 따른다. 이렇게 함으로써, 당신은 “checkbox가 폼에서 보여지고 폼 데이터가 checkbox의 상태를 반영하도록 바운드할 객체를 원한다고” Spring에 효과적으로 알린다.

13.9.5. radiobutton 태그

이 태그는 ‘radio’ 타입을 가진 HTML ‘input’ 태그를 표현한다.

전형적인 사용법 패턴은 값은 다르지만 같은 프라퍼티에 대해 바운드되는 다중 태그 인스턴스를 포함할것이다.

```
<tr>
    <td>Sex:</td>
    <td>Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/> </td>
    <td></td>
</tr>
```

13.9.6. password 태그

이 태그는 바운드 값을 사용하는 ‘password’ 타입을 가진 HTML ‘input’ 태그를 표현한다.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" />
    </td>
</tr>
```

13.9.7. select 태그

이 태그는 HTML ‘select’ 요소를 표현한다. 이것은 내포된 option 과 options 태그의 사용처럼 선택된 옵션에 대한 데이터 바인딩을 지원한다.

User가 많은 skill 을 가진다고 가정해보자.

```
<tr>
    <td>Skills:</td>
```

```
    <td><form:select path="skills" items="${skills}"/></td>
    <td></td>
</tr>
```

User의 skill이 Herbology라면, 'Skills' 의 HTML소스는 다음과 같을것이다.

```
<tr>
    <td>Skills:</td>
    <td><select name="skills" multiple="true">
        <option value="Potions">Potions</option>
        <option value="Herbology" selected="true">Herbology</option>
        <option value="Quidditch">Quidditch</option></select></td>
    <td></td>
</tr>
```

13.9.8. option 태그

이 태그는 HTML 'option'을 표현한다. 이것은 바운드된 값에 기초하여 'selected'를 셋팅한다.

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

User의 house가 Gryffindor라면, 'House'의 HTML소스는 다음과 같을것이다.

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor" selected="true">Gryffindor</option>
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
 </tr>
```

13.9.9. options 태그

이 태그는 HTML 'option' 태그의 목록을 표현한다. 이것은 바운드 값에 기초하여 'selected' 속성을 셋팅한다.

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code" itemLabel="name"/>
        </form:select>
```

```
    </td>
    <td></td>
</tr>
```

User 가 UK에서 살고있다면, 'Country'의 HTML소스는 다음과 같을것이다.

```
 <tr>
    <td>Country:</td>
    <tr>
        <td>Country:</td>
        <td>
            <select name="country">
                <option value="-">--Please Select</option>
                <option value="AT">Austria</option>
                <option value="UK" selected="true">United Kingdom</option>
                <option value="US">United States</option>
            </select>
        </td>
        <td></td>
    </tr>
    <td></td>
</tr>
```

예제가 보여주는 것처럼, options태그와 option태그의 조합의 사용법은 같은 표준 HTML을 생성한다. 하지만 예제 "-- Please Select" 의 디폴트 문자열처럼 보여주기 위한 JSP내 값을 명시하는것을 허용한다.

13.9.10. textarea 태그

이 태그는 HTML 'textarea'를 표현한다.

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20" /></td>
    <td><form:errors path="notes" /></td>
</tr>
```

13.9.11. hidden 태그

이 태그는 바운드 값을 사용하여 'hidden' 타입을 가진 HTML 'input' 를 표현한다. 바운드되지 않은 hidden값을 서브밋하기 위해, 'hidden' 타입을 가진 HTML input 태그를 사용하라.

```
<form:hidden path="house" />
```

우리가 hidden값으로 'house' 값을 서브밋하도록 선택한다면, HTML은 다음과 같을것이다.

```
<input name="house" type="hidden" value="Gryffindor"/>
```

13.9.12. errors 태그

이 태그는 HTML 'span' 태그내 필드 에러를 표현한다. 이것은 컨트롤러내 생성된 에러와 컨트롤러에 관련된 유효성체커(validators)에 의해 생성에러에 접근한다.

우리가 폼을 서브밋할때 firstName 과 lastName 필드를 위한 모든 에러 메시지를 보여주길 원한다고 가정하자. 우리는 UserValidator 라고 불리는 User 클래스의 인스턴스를 위한 유효성체커(validator)를 가진다.

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required", "Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field is required.");
    }
}
```

form.jsp 은 다음과 같을것이다.

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName" /></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

우리가 firstHame 과 lastName 필드내 빈 값을 가진 폼을 서브밋한다면, HTML은 다음과 같을것이다.

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
```

```
        </tr>
    </table>
</form>
```

주어진 페이지를 위한 에러의 전체 목록은 보여주길 원하는가..? 아래의 예제는 errors 태그가 몇가지 기본적인 와이들카드 기능을 제공한다는 것을 보여준다.

☒ path="*" - 모든 에러를 보여준다.

☒ path="lastName*" - lastName 필드와 관련된 모든 에러를 보여준다.

아래의 예제는 필드뒤에 필드에 따른 에러를 보여주면서 페이지의 가장 상위에서 에러의 목록을 보여줄것이다.

```
<form:form>
    <form:errors path="*" cssClass="errorBox" />
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <td><form:errors path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

HTML은 다음과 같이 보일것이다.

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br/>Field is required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
</form>
```

13.10. 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.

13.11. 설정에 대한 규칙

많은 프로젝트를 위해, 규칙을 만들고 가능한 디폴트를 가지는 것은 필요하다. 설정에 대한 규칙의 테마는 현재 Spring 웹 MVC에서 명시적으로 지원된다. 이것이 의미하는 것은 당신이 명명규칙과 같은 것을 만들때, 핸들러 맵핑, view 결정자(resolver), ModelAndView 인스턴스를 셋업하기 위해 필요한 설정의 양을 충분히 줄일수 있다. 이것은 빠른 프로토타이핑을 위해 큰 이득이고 일관성의 정도를 더해준다.

Tip

Spring배포판은 이 부분에서 언급된 설정에 대한 규칙지원을 보여주는 웹 애플리케이션을 포함한다. 애플리케이션은 'samples/showcases/mvc-convention' 디렉토리에서 찾을수 있다.

설정에 대한 규칙은 MVC의 3가지 핵심 영역(모델, 뷰, 컨트롤러)을 할당한다.

13.11.1. 컨트롤러 - ControllerClassNameHandlerMapping

ControllerClassNameHandlerMapping 클래스는 요청 URL과 이러한 요청을 다루기 위한 Controller 인스턴스간에 맵핑을 판단하는 규칙을 사용하는 HandlerMapping 구현물이다.

예제 : 다음 Controller 구현물을 보라. 클래스의 name의 특별한 인지를 가진다.

```
public class ViewShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // the implementation is not hugely important for this example...
    }
}
```

이것은 관련 Spring 웹 MVC설정 파일의 일부이다.

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">
    <!-- inject dependencies as required... -->
</bean>
```

ControllerClassNameHandlerMapping은 애플리케이션 컨텍스트내 정의된 다양한 핸들러(또는 Controller) bean 모두를 찾고

몇가지 더 많은 예제를 보라. 그러면 중심적인 생각이 즉시 친숙해질것이다.

☒ WelcomeController는 '/welcome*' 요청 URL을 맵핑한다.

☒ HomeController는 '/home*' 요청 URL을 맵핑한다.

☒ IndexController는 '/index*' 요청 URL을 맵핑한다.

☒ RegisterController는 '/register*' 요청 URL을 맴핑한다.

☒ DisplayShoppingCartController 는 '/displayshoppingcart*' 요청 URL을 맵핑한다.

(대소문자에 대한 주의 - camel-case성질을 가지는 Controller 클래스명의 경우. 모두 소문자)

MultiActionController 핸들러 클래스의 경우, 생성된 맵핑은 좀더 복잡하다. 하지만 다소 덜 이해가 될수 있다. 다음의 모든 Controller 이름은 MultiActionController 구현물이 되는것으로 가정된다.)

☒ AdminController는 '/admin/*' 요청 URL을 맵핑한다.

☒ CatalogController는 '/catalog/*' 요청 URL을 맵핑한다.

xxxController 처럼 Controller 구현물 명명의 표준적인 규칙을 따른다면, ControllerClassNameHandlerMapping은 처음으로 정의하는 것의 지루함을 줄이고 잠재적으로 SimpleUrlHandlerMapping을 관리한다.

ControllerClassNameHandlerMapping 클래스는 AbstractHandlerMapping 기본 클래스를 확장해서 당신은 HandlerInterceptor 인스턴스를 정의할수있다.

13.11.2. 모델 - ModelMap (ModelAndView)

ModelMap 클래스는 기본적으로 공통적인 명명 규칙에 충실한 View에 표시되는 객체를 추가할수 있는 실제보다 좋게 보이는 Map이다. 그 생각은 상세하게 설명하는 것보다는 간단하다. 나는 당신에게 몇가지 예제를 보여줄것이다.

다음의 Controller 구현물을 보자. 객체가 명시된 이름없이 ModelAndView에 추가되는 것에 주의하라.

```
public class DisplayShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

        List cartItems = // get a List of CartItem objects
        User user = // get the User doing the shopping

        ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- the logical view name

        mav.addObject(cartItems); <-- look ma, no name, just the object
        mav.addObject(user); <-- and again ma!

        return mav;
    }
}
```

ModelAndView 클래스는 객체가 추가될때 객체를 위한 key를 자동적으로 생성하는 사용자정의 Map 구현물인 ModelMap 클래스를 사용한다. 추가된 객체를 위한 이름을 판단하기 위한 전략은 User와 같은 단계적 객체의 경우, 객체 클래스의 짧은 클래스명을 사용하는 것이다. 단계적 객체를 위해 생성되는 이름의 몇가지 예제가 ModelMap 인스턴스를 두는 것을 아래에서보라.

☒ 추가된 x.y.User 인스턴스는 생성된 'user' 이름을 가질것이다.

☒ x.y.Registration 인스턴스는 생성된 'registration' 이름을 가질것이다.

☒ x.y.Foo 인스턴스는 생성된 'foo' 이름을 가질것이다.

☒ 추가된 java.util.HashMap 인스턴스는 생성된 'hashMap' 이름을 가질것이다. (당신은 아마도 'hashMap'이 다소 덜 직관적이기 때문에 이 경우 이름에 대해 명시적이길 원할것이다.)

☒ null을 추가하는 것은 IllegalArgumentException이 던져지는 결과를 만든다. 잠재적으로 객체가 null로 추가된다면, 이름에 대해 명시적이길 원할것이다.).

> 자동 복수화(pluralisation)는 없다..?
>
> Spring 웹 MVC의 설정에 대한 규칙 지원은 자동적인 복수화(pluralisation)을 지원하지 않는다. 당신은 Person 객체의 List를 ModelAndView에 추가할수 없고 'people' 가 되는 생성된 이름을 가진다.
>
> 이 결정은 결국에 "최소한의 놀라움의 법칙(Principle of Least Surprise)" 를 가지고 몇가지 논쟁후 얻었다.

Set, List 또는 배열 객체를 추가한 후 이름을 생성하기 위한 전략은 collection내 첫번째 객체의 짧은 클래스명을 가지고 collection을 검색한다. 그리고 name에 추가된 'List'를 가지고 사용한다. 몇가지 예제는 collection을 좀더 명백하게 하기 위해 이름 생성을 할것이다.

☒ 추가된 하나 이상의 x.y.User요소를 가진 x.y.User[] 배열은 생성된 'userList' 이름을 가질것이다.

☒ 추가된 하나 이상의 x.y.User요소를 가진 x.y.Foo[] 배열은 생성된 'fooList' 이름을 가질것이다.

☒ 추가된 하나 이상의 x.y.User요소를 가진 java.util.ArrayList는 생성된 'userList' 이름을 가질것이다.

☒ 추가된 하나 이상의 x.y.Foo 요소를 가진 java.util.HashSet은 생성된 'fooList' 이름을 가질것이다.

☒ 빈 java.util.ArrayList는 전혀 추가되지 않을것이다.(이를테면, addObject(..) 호출은 기본적으로 무연산 명령이 될것이다.).

13.11.3. 뷰(view) - RequestToViewNameTranslator

RequestToViewNameTranslator 인터페이스는 논리적인 view명이 명시적으로 제공되지 않았을때 논리적인 View 명을 판단하는 책임을 가진다. 이것은 하나의 구현물인 다소 교활하게 명명된 DefaultRequestToViewNameTranslator 클래스를 제외한다.

DefaultRequestToViewNameTranslator는 이러한 형태로 요청 URL을 논리적인 view명으로 맵핑한다. 다음의 Controller 구현물과 관련된 Spring 웹 MVC XML 설정을 보라.

```
public class RegistrationController implements Controller {
```

```
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // process the request...
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <!-- this bean with the well known name generates view names for us -->
    <bean id="viewNameTranslator" class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator"/>

    <bean class="x.y.RegistrationControllerController">
        <!-- inject dependencies as necessary -->
    </bean>

    <!-- maps request URLs to Controller names -->
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

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

</beans>
```

반환된 ModelAndView에 셋팅되는 View 나 논리적인 view이름은 없다. 이것은 요청 URL로부터 논리적인 view명를 생성할 DefaultRequestToViewNameTranslator이다. ControllerClassNameHandlerMapping와 함께 사용되는 위 RegistrationControllerController의 경우, 'http://localhost/registration.html' 요청 URL은 DefaultRequestToViewNameTranslator에 의해 생성된 'registration'의 논리적인 view명의 결과가 된다. 논리적인 view명은 InternalResourceViewResolver bean에 의해 '/WEB-INF/jsp/registration.jsp' view로 해석될것이다.

Tip

당신은 DefaultRequestToViewNameTranslator bean을 명시적으로 정의할 필요가 없다. 당신이 DefaultRequestToViewNameTranslator의 디폴트 셋팅을 승낙한다면, 명시적으로 설정되지 않았을때 Spring 웹 MVC DispatcherServlet이 클래스의 인스턴스를 인스턴스화할것이라는 사실에 의존할수 있다.

물론, 당신이 디폴트 셋팅을 변경할 필요가 있다면, 당신은 자체적인 DefaultRequestToViewNameTranslator bean을 명시적으로 설정할 필요가 있다. 설정될수 있는 다양한 프라퍼티의 상세한 설명을 위해서 DefaultRequestToViewNameTranslator 클래스를 위한 편리한 Javadoc을 보라.

13.12. 더많은 자원

Spring 웹 MVC에 대한 더 많은 자원을 위한 링크와 위치를 아래에서 보라.

☒ Spring 배포판은 단계별 접근법을 사용하여 완전한 Spring 웹 MVC-기반의 애플리케이션을 빌드하여

독자를 가이드하는 Spring 웹 MVC 튜토리얼을 포함한다. 이 튜토리얼은 Spring 배포판에서 'docs' 디렉토리에 있다. 온라인 버전은 Spring 프레임워크 웹사이트에서 찾을수 있다..

☒ Seth Ladd와 다른 사람들에 의해 만들어진 "Expert Spring Web MVC and WebFlow" 책은 Spring 웹 MVC 의 완벽한 하드카피 소스이다.

Chapter 14. 통합 뷰 기술들

14.1. 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 13.5, “view와 view결정하기” 와 유사할 것이다.

14.2. 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.

14.2.1. 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"/>
</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"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</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.

14.2.2. 'Plain-old' JSPs 대(versus) JSTL

자바 표준 테그 라이브러리를 사용할때, 특별한 뷰 클래스, JstlView을 사용해야하고, JSTL는 i18N기능들을 작업하기전에 몇몇의 표현이 필요하다.

14.2.3. 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 정보는 Appendix D, spring.tld에서 찾을 수 있다:

14.3. Tiles

Spring을 사용하는 웹 애플리케이션 안에서 -다른 뷰 기술들과 같이- Tiles는 통합 가능하다. 다음은 대체로 타일을 사용하는 방법을 설명한다.

14.3.1. 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 또는 그 이상
- ☒ Commons BeanUtils
- ☒ Commons Digester
- ☒ Commons Lang
- ☒ Commons Logging

의존물들은 Spring 구분(distribution)안에 모두 이용할 수 있다.

14.3.2. 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"/>
  <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-설정에 의해 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.

14.3.2.1. InternalResourceViewResolver

InternalResourceViewResolver는 각 뷰가 가지고있는 것을 분석하기위해(resolve) viewClass에서 주어진 것을 증명한다. InternalResourceViewResolver는 해석하기 위한 각각의 view를 위한 주어진 viewClass를 초기화한다.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="requestContextAttribute" value="requestContext"/>
  <property name="viewClass" value="org.springframework.web.servlet.view.tiles.TilesView"/>
</bean>
```

14.3.2.2. 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"/>
</bean>
```

```
    ...
welcomeView.class=org.springframework.web.servlet.view.tiles.TilesView
welcomeView.url=welcome (&lt;b&gt;this is the name of a definition&lt;/b&gt;)

vetsView.class=org.springframework.web.servlet.view.tiles.TilesView
vetsView.url=vetsView (again, this is the name of a definition)

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.

14.4. Velocity & FreeMarker

Velocity 과 FreeMarker는 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 웹사이트에서 보아라.

14.4.1. 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.

14.4.2. 컨텍스트 설정(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/"/>
</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"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".vm"/>

  <!-- if you want to use the Spring Velocity macros, set this property to true -->
  <property name="exposeSpringMacroHelpers" value="true"/>

</bean>
```

```
<!-- freemarker config -->
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</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"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".ftl"/>

  <!-- if you want to use the Spring FreeMarker macros, set this property to true -->
  <property name="exposeSpringMacroHelpers" value="true"/>

</bean>
```

Note

NB: 애플리케이션 컨텍스트 정의 파일에 VelocityConfigurationFactoryBean 또는 FreeMarkerConfigurationFactoryBean을 web-apps에 추가시키지 말아라.

14.4.3. 생성 템플릿들(Creating templates)

템플릿들은 위에서 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.vm의 적합한템플릿을 찾을 것이다.

14.4.4. 진보한 구성(Advanced configuration)

위의 중요한 기본 구성들은 대부분 애플리케이션 요구사항에 적합할 것이다. 그러나 추가적인 구성선택들은 색다르거나 진보한 요구사항을 지시할때 이용할 수 있다.

14.4.4.1. 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"/>
</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 문서을 참조하거나, velocity.properties 파일 자체의 예들과 정의들을 위한 Velocity 문서를 참조하라.

14.4.4.2. 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/"/>
  <property name="freemarkerVariables">
    <map>
      <entry key="xml_escape" value-ref="fmXmlEscape"/>
    </map>
  </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

Configuration객체에 적용할 셋팅과 변수의 상세사항을 위해서 FreeMarker문서를 보라.

14.4.5. 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.

14.4.5.1. 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 VelocityView / FreeMarkerViewbean에서 bean프라퍼티인 exposeSpringMacroHelpers를 'true'로 셋팅될때만 사용될수 있다. 간단히 얘기하면 당신의 view가 값을 상속할 모든 경우에 VelocityViewResolver 나 FreeMarkerViewResolver에서 이 프라퍼티를 설정할수 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
```

모든 (Velocity) view에 Velocity매크로를 나타내는 완전한 Spring 웹 MVC설정파일의 예제를 아래에서 보라.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

    <bean name="helloController" class="info.wilhelms.springwebapp.SampleController">
        <property name="commandName" value="command"/>
        <property name="commandClass" value="info.wilhelms.springwebapp.Message"/>
        <property name="formView" value="foo"/>
        <property name="successView" value="banjo"/>
```

```
        <property name="bindOnNewForm" value="true"/>
        <property name="sessionForm" value="true"/>
    </bean>

    <bean id="velocityConfig"
          class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
        <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
    </bean>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
        <property name="cache" value="false"/>
        <property name="prefix" value=""/>
        <property name="suffix" value=".vm"/>
        <property name="exposeSpringMacroHelpers" value="true"/>
    </bean>

    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                **/hello.htm=helloController
            </value>
        </property>
    </bean>

</beans>
```

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.

14.4.5.2. 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.

14.4.5.3. 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 14.1. 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 $text)	<@spring.messageText code, text/>
url (애플리케이션 컨텍스트 root를 가진 상대적인 URL을 접두사로 붙이는)	#springUrl($relativeUrl)	<@spring.url relativeUrl/>
formInput (사용자 입력을	#springFormInput($path $attributes)	<@spring.formInput path, attributes,

매크로	VTL 정의	FTL 정의
모으기 위한 표준적인 input필드)		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 $attributes)	<@spring.formMultiSelect path, options, attributes/>
formRadioButtons (사용가능한 선택사항으로 부터 만들수 있는 하나의 selection을 허용하는 radio버튼의 세트)	#springFormRadioButtons($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.

14.4.5.3.1. 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.

14.4.5.3.2. 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
```

14.4.5.4. 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 -->
```

14.5. 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 Spring 웹 애플리케이션내 XSLT로 변형한다.

14.5.1. 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 13.3, “컨트롤러” 을 보라. XSLT view는 단어의 목록에서 변형될 준비가 된 간단한 XML문서로 바뀔것이다.

14.5.1.1. 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.

14.5.1.2. 표준적인 MVC 컨트롤러 코드

컨트롤러 로직은 정의된 핸들러 메소드와 함께 AbstractController의 하위클래스로 캡슐화된다.

```
protected ModelAndView handleRequestInternal(
    HttpServletRequest request,
    HttpServletResponse response) 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 추가하여 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.

14.5.1.3. 모델 데이터를 XML로 변환하기

우리의 단어 목록이나 다른 모델 데이터로부터 DOM문서를 생성하기 위해 우리는 org.springframework.web.servlet.view.xslt.AbstractXsltView의 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 Source createXsltSource(Map model, String rootName, HttpServletRequest
        request, HttpServletResponse response) throws Exception {

        Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element root = document.createElement(rootName);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(nextWord);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }
        return new DOMSource(root);
    }

}
```

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.

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.

14.5.1.4. 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 createXsltSource(..) 메소드를 위한 두번째 파라미터로 위 HomePage 클래스에 전달된다.

14.5.1.5. 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" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>
                <h1>My First Words</h1>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="word">
        <xsl:value-of select="."/><br/>
    </xsl:template>

</xsl:stylesheet>
```

14.5.2. 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.

14.6. 문서 views (PDF/Excel)

14.6.1. 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.

14.6.2. 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.

14.6.2.1. 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.

14.6.2.2. 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.

14.6.2.3. 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.AbstractExcelView(POI에 의해 생성되는 엑셀 파일을 위해)나 org.springframework.web.servlet.view.document.AbstractJExcelView(JExcelApi-에 의해 생선된 엑셀 파일)의 하위클래스를 생성하고 buildExcelDocument를 구현한다.

새로운 스프레드시트의 첫번째 칼럼의 연속적인 row내 모델 map으로 부터 단어목록을 보여주는 POI 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));

        }
    }
}
```

그리고 이것은 지금 JExcelApi를 사용하여 같은 엑셀 파일을 생성한다.

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(Map model,
        WritableWorkbook wb,
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {

        WritableSheet sheet = wb.createSheet("Spring");

        sheet.addCell(new Label(0, 0, "Spring-Excel test");

        List words  = (List)model.get("wordList");
        for (int i = -; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String)words.get(i));
        }
    }
}
```

API들간의 차이점을 알아두라. 우리는 JExcelApi가 좀더 직관적이고 잘 이끌어낸다는 것을 알았다. JExcelApi는 더 나은 이미지-핸들링 기능을 가진다. JExcelApi를 사용할때 큰 엑셀을 가지는 메모리 문제가 있다.

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.

14.6.2.4. 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.

14.7. 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.

14.7.1. 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.

14.7.2. 설정

ApplicationContext에서 JasperReports view를 설정하기 위해 당신의 리포트가 표시되길 원하는 포맷에 의존하는 적당한 view클래스를 위한 view이름을 맵핑하는 ViewResolver를 정의해야만 한다.

14.7.2.1. 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"/>
</bean>
```

우리는 기본이름인 views을 가진 자원번들내 view맵핑을 찾을 ResourceBundleViewResolver의 인스턴스를 설정한다. 이 파일의 정확한 내용은 다음 부분에서 언급된다.

14.7.2.2. View 설정하기

Spring은 JasperReports에 의해 지원되는 4가지의 출력 포맷중 하나에 관련된 것중 4가지의 JasperReports를 위한 5가지의 다른 View구현물을 포함하고 실행시 결정되는 포맷을 허용한다.

Table 14.2. JasperReports View 클래스

클래스명	표시(Render) 형태
JasperReportsCsvView	CSV
JasperReportsHtmlView	HTML
JasperReportsPdfView	PDF
JasperReportsXlsView	Microsoft Excel
JasperReportsMultiFormatView	실행시 결정됨(Section 14.7.2.4, “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.

14.7.2.3. 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.

14.7.2.4. 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 14.3. 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.

14.7.3. 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.

14.7.4. 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.

14.7.4.1. 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"/>
    </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.

14.7.4.2. 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"/>
```

여기서 당신이 제공하는 키는 ModelAndView내 사용되는 키와 리포트 디자인 파일내 사용되는 키 모두와 일치해야만 한다.

14.7.5. 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"/>
  <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 15. 다른 웹 프레임워크들과의 통합

15.1. 소개

이 장은 Spring과 Struts, JSF, Tapestry, 그리고 WebWork와 같은 다른 웹 프레임워크와의 통합을 상세히 다룬다.

Spring프레임워크의 핵심적인 가치를 가지는 유혹(proposition)중 하나는 각각에 대해 선택이 가능하도록 한다는 것이다. 대개 Spring은 특정 구조, 기술 또는 방법론을 사용하거나 구매하도록 강요하지는 않는다(비록 다른 누군가에게 명백하게 추천하더라도). 대부분의 개발자와 개발팀에게 관련된 구조, 기술 또는 방법론을 끄집어내고 선택하기 위한 자유는 같은 시점에 다른 많은 웹 프레임워크와의 통합을 제공하는 Spring이 자체적인 웹 프레임워크(SpringMVC)을 제공하는 웹 영역에서 대부분 명백하다. 이것은 같은 시점에 데이터 접근, 선언적인 트랜잭션 관리 그리고 유연한 설정및 애플리케이션 조합과 같은 다른 영역에서 Spring에 의해 달성되는 이득을 즐길수 있는 반면에 Struts와 같은 유명한 웹 프레임워크를 습득하는 모든 스킬이나 일부에 영향을 지소적으로 끼친다.

이 장의 나머지는 선호하는 웹 프레임워크와 Spring을 통합하는 충실하고 상세한 내용에 집중할것이다. 다른 개발언어에서 Java로 전향하는 개발자들에 의해 종종 남겨지는 글의 내용은 Java로 사용가능한 웹 프레임워크의 풍부함이다. Java영역에는 정말 많은 웹프레임워크가 있다. 사실 하나의 장에서 외형적인 상세함을 너무 많이 다룬다. 이 장은 웹 프레임워크를 모두 지원하는 공통적인 Spring설정을 시작으로 각각의 웹 프레임워크의 개별적인 통합 옵션을 다루면서 Java에서 좀더 유명한 4개의 웹 프레임워크를 다룬다.

이 장에서 지원되는 웹 프레임워크를 사용하는 방법을 설명하고자 하지는 않는다. 예를 들면, 웹 애플리케이션의 표현 레이어를 위해 Struts를 사용하길 원한다면, 이미 우리는 당신이 Struts에 친숙하다고 가정한다. 만약 지원되는 각각의 프레임워크에 대한 상세정보를 얻기를 원한다면, 이 장의 마지막의 Section 15.7, “추가적인 자원” 부분을 보라.

15.2. 공통 설정

지원되는 각각의 웹 프레임워크에 특정한 통합으로 나누기 전에, 하나의 웹 프레임워크에 종속적이지않은 Spring설정을 보자(이 부분은 Spring자체의 웹 프레임워크인 SpringMVC에도 적용가능하다.).

(Spring의) 가벼운 애플리케이션 모델에 의해 채택되는 개념중 하나는 레이어화된(layered) 구조이다. ‘고전적(classic)’ 레이어 구조에서 생각해보자. 웹 레이어는 많은 레이어중 하나는 아니다. 이것은 서버측 애플리케이션에 대한 항목지점(entry point)중 하나로 제공한다. 그리고 이것은 비지니스 종속적인(그리고 표현-기술에 관용적인) 유즈케이스를 만족하기 위한 서비스 레이어에 정의된 서비스 객체(facades)에 위임한다. Spring에서, 서비스 객체, 다른 비지니스-종속 객체, 데이터 접근 객체 등등. 다른 ‘비지니스 컨텍스트’에 존재하는 웹이나 표현레이어 객체(다른 ‘표현 컨텍스트(presentation context)’에 대개 설정되는 Spring MVC컨트롤러와 같은 표현 객체)를 포함하지 않는다. 이 부분은 하나의 애플리케이션내 모든 ‘비지니스 bean’을 포함하는 Spring컨테이너(WebApplicationContext)를 설정하는 방법을 설명한다.

할 필요가 있는 모든것은 ContextLoaderListener 를 표준 J2EE서블릿 web.xml 파일에 선언하고, 어떤 Spring XML설정파일을 로드할것인지 정의하는 contextConfigLocation <context-param>를 사용하기만 하면 된다.

아래 <listener> 설정을 보라.

```
<listener>
```

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

Note

Listeners는 Servlet API 2.3 버전에 추가되었다. 만약 당신이 Servlet 2.2 컨테이너를 사용한다면, 당신은 동일한 기능을 얻기 위해 ContextLoaderServlet를 사용할 수 있다.

Find below the <context-param/> configuration:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

만약 당신이 contextConfigLocation 컨텍스트 파라미터를 명시하지 않는다면, ContextLoaderListener는 로드할 /WEB-INF/applicationContext.xml 파일을 찾을 것이다. 일단 컨텍스트 파일이 로드되면, Spring은 bean 정의에 기반하여 WebApplicationContext 객체를 생성하고 이것을 웹 애플리케이션의 ServletContext에 저장한다.

모든 자바 웹 프레임워크들은 Servlet API에 기반하여 만들어졌기 때문에, ContextLoaderListener에 의해 생성된 '비지니스 컨텍스트(business context)' ApplicationContext를 얻기 위해 다음의 코드를 사용할 수 있다.

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

WebApplicationContextUtils클래스는 편의를 위해 만들어진 것인데, 때문에 당신은 ServletContext 속성의 이름을 기억할 필요가 없다. 그것의 getWebApplicationContext() 메소드는 만약 (ApplicationContext) 객체가 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE 키로 존재하지 않는다면 null을 반환할 것이다. 애플리케이션에서 NullPointerExceptions를 받는 위험을 감수하는 것보다는 getRequiredWebApplicationContext() 메소드를 사용하는 것이 낫다. 이 메소드는 ApplicationContext를 찾지 못할 경우, 예외를 던진다.

일단 당신이 WebApplicationContext를 참조하게 되면, 당신은 이름 혹은 타입으로 bean들을 가져올 수 있다. 대부분의 개발자들은 bean들을 이름으로 가져와서 그것의 구현된 인터페이스들 중 하나로 캐스팅한다.

운 좋게도 이번 장에서 다루는 대부분의 프레임워크들은 bean들을 룩업하는 방식이 매우 간단하다. bean들을 Spring 컨테이너로부터 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.

15.3. JavaServer Faces

JavaServer Faces(JSF)는 점점 유명해지는 컴포넌트 기반, 이벤트-주도(driven) 웹 프레임워크이다. Spring의 JSF통합내 핵심 클래스는 DelegatingVariableResolver 클래스이다.

15.3.1. DelegatingVariableResolver

당신의 Spring 미들티어를 JSF 웹 레이어와 통합하는 가장 쉬운 방법은 DelegatingVariableResolver 클래스를 사용하는 것이다. 이 변수 처리자(variable resolver)를 당신의 애플리케이션에 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>
</faces-config>
```

DelegatingVariableResolver는 처음엔 값을 룩업하는 것을 기반하는 JSF 구현의 디폴트 처리자에 위임한다. 그리고나서 Spring의 '비지니스 컨텍스트(business context)' WebApplicationContext에 위임한다. 이것은 당신이 JSF에 의해 관리되는 bean들에 의존성을 쉽게 주입할 수 있도록 해준다.

관리되는 bean들은 faces-config.xml 파일에 정의된다. 아래는 Spring '비지니스 컨텍스트(business context)' 로부터 가져온 bean인 #{userManager} 예제이다.

```
<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>
```

15.3.2. FacesContextUtils

커스텀 VariableResolver는 프라퍼티들을 faces-config.xml 내의 bean들과 매핑할 때 매우 잘 동작한다. 그러나, 종종 당신은 bean을 명시적으로 가로챌 필요가 있을 것이다. FacesContextUtils 클래스는 그것을 쉽게 해준다. 이것은 WebApplicationContextUtils와 비슷한데, ServletContext 파라미터 보다는 FacesContext 파라미터를 가진다는 점만 다르다.

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

DelegatingVariableResolver는 JSF와 Spring 통합을 위해 추천되는 전략이다. 만약 좀더 견고한 통합 기능을 찾는다면, JSF-Spring 프로젝트를 찾길 원할지도 모른다.

15.4. Struts

Struts 는 자바 애플리케이션을 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 애플리케이션을 Spring과 통합하는 데는 두 가지 방법이 있다.

☒ ContextLoaderPlugin를 사용하여 Spring이 Action들을 bean들로 관리하도록 설정하고 Action들의 의존성을 Spring 컨텍스트 파일에 세팅하는 방법

☒ Spring의 ActionSupport 클래스를 상속해서 getWebApplicationContext() 메소드를 사용하여 Spring 관리되는 bean들을 명시적으로 가로채는 방법

15.4.1. ContextLoaderPlugin

ContextLoaderPlugin 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 정리를 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 type 속성에 DelegatingActionProxy 클래스를 사용한다.

이 메소드들 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 bean의 "name"으로 연결된다. 당신이 struts-config.xml 파일에 다음과 같은 설정을 가진다고 가정하자.

```
<action path="/users" .../>
```

그러면, 당신은 "/users"라는 이름을 가진 Action bean을 action-servlet.xml 내에 정의해야만 한다.

```
<bean name="/users" .../>
```

15.4.1.1. DelegatingRequestProcessor

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 bean 이름들은 반드시 모듈 접두어를 포함해야만 한다. 예를 들어, 모듈 접두어 "admin"을 가진 <action path="/user"/>로 정의된 action은 <bean name="/admin/user"/>라는 bean 이름을 가져야 한다.

Note

만약 당신이 Struts 애플리케이션에서 Tiles를 사용한다면, 당신은 DelegatingTilesRequestProcessor 로 <controller>를 대신 설정해야만 한다.

15.4.1.2. DelegatingActionProxy

만약 사용자정의 RequestProcessor를 가지고 있고 DelegatingRequestProcessor와 DelegatingTilesRequestProcessor를 사용할 수 없다면, DelegatingActionProxy 를 action-mapping 내 타입으로 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 bean 정의는 사용자정의 RequestProcessor나 DelegatingActionProxy를 사용하는 것에 상관없이 동일하다.

만약 당신이 컨텍스트 파일에 Action을 정의한다면, 그 Action에 대해 Spring bean 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 bean 정의에 scope="prototype"를 추가해주어야 한다.

```
<bean name="/user" scope="prototype" autowire="byName" class="org.example.web.UserAction"/>
```

15.4.2. 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 클래스를 상속할 수 있다.

ActionSupport 클래스는 getWebApplicationContext()처럼 부가적인 편의 메소드들을 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,
- ☒ DispatchActionSupport,
- ☒ LookupDispatchActionSupport and
- ☒ MappingDispatchActionSupport.

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.

15.5. Tapestry

Tapestry 홈페이지로 부터...
" Tapestry는 Java로 동적이고 견고하며 높은 확장성을 가진 웹 애플리케이션을 만들기 위한 오픈소스 프레임워크이다. Tapestry 는 표준 Java Servlet API를 보완하고 빌드한다. 그래서 서블릿 컨테이너나 애플리케이션 서버에서 작동한다. "

Spring이 자체의 강력한 웹 레이어를 가지는 반면, 웹 유저 인터페이스를 위한 Tapestry와 낮은 레이어를 위한 Spring컨테이너의 조합을 사용하여 J2EE애플리케이션을 빌드하기 위한 유일한 장점을 많이 있다. 웹 통합부분은 이러한 두개의 프레임워크를 조합하기 위한 몇몇 가장 좋은 상황을 설명하는 것을 시도한다.

Tapestry와 Spring으로 빌드된 전형적으로 계층화된 J2EE애플리케이션은 하나 이상의 Spring컨테이너에 의해 묶인 Tapestry로 빌드된 가장 상위의 유저 인터페이스(UI) 레이어, 많은 수의 하위 레이어를 구성할것이다. Tapestry 자체의 참조문서 는 가장 좋은 상황에 대한 다음의 충고를 포함한다(내가 작성한

텍스트는 []내 포함된다.) .
“ Tapestry에서 가장 성공한 디자인 패턴은 페이지를 유지하고 컴포넌트를 매우 간단한 상태로 유지한다. 그리고 가능한한 HiveMind [또는 Spring, 이나 어떤것들] 서비스로 많은 로직을 위임한다. 리스너 메소드는 정확한 정보와 함께 직렬화되는것보다 조금 작동하고 서비스로 이것을 전달한다. ”

핵심 질문은 협력 서비스를 가진 Tapestry페이지를 어떻게 제공하는가 ? 대답은 이러한 서비스를 Tapestry페이지로 직접 의존성 삽입하는 것을원하는 것이다. Tapestry에서, 더양한 도구에 의해 의존성 삽입에 영향을 줄수 있다. 이 부분은 Spring에 의해 발생할수 있는 의존성삽입을 설명할것이다. Spring-Tapestry 통합에서 실제로 멋진 것은 Tapestry의 멋지고 유연한 디자인 자체가 Spring관리 bean의 의존성 삽입을 달성한다는것이다.(다른 좋은 것은 Tapestry생성자인 Howard M. Lewis Ship에 의해 Spring-Tapestry통합 코드가 작성되고 유지가 지속된다는 것이다. 그래서 유연한 통합이 된다.)

15.5.1. Injecting Spring-managed beans

Assume we have the following simple Spring container definition (in the ubiquitous XML format):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    <!-- the DataSource -->
    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:DefaultDS"/>
    </bean>

    <bean id="hibSessionFactory"
          class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
    </bean>

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

    <bean id="mapper"
          class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
        <property name="sessionFactory" ref="hibSessionFactory"/>
    </bean>

    <!-- (transactional) AuthenticationService -->
    <bean id="authenticationService"
          class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="target">
            <bean class="com.whatever.services.service.user.AuthenticationServiceImpl">
                <property name="mapper" ref="mapper"/>
            </bean>
        </property>
        <property name="proxyInterfacesOnly" value="true"/>
        <property name="transactionAttributes">
            <value>
                *=PROPAGATION_REQUIRED
            </value>
        </property>
    </bean>

    <!-- (transactional) UserService -->
    <bean id="userService"
          class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager"/>
```

```
        <property name="target">
            <bean class="com.whatever.services.service.user.UserServiceImpl">
                <property name="mapper" ref="mapper"/>
            </bean>
        </property>
        <property name="proxyInterfacesOnly" value="true"/>
        <property name="transactionAttributes">
            <value>
                *=PROPAGATION_REQUIRED
            </value>
        </property>
    </bean>

</beans>
```

Tapestry 애플리케이션 내부에서, 위 bean정의는 Spring컨테이너에 로드될 필요가 있고 관련 Tapestry페이지는 각각 AuthenticationService 인터페이스와 UserService 인터페이스를 구현하는 authenticationService bean과 userService bean과 함께 제공(삽입)될 필요가 있다.

이 지점에서, 애플리케이션 컨텍스트는 Spring의 정적 유틸리티 함수인 WebApplicationContextUtils.getApplicationContext(servletContext)(servletContext가 J2EE서블릿 스펙의 표준 ServletContext인)를 호출하여 웹 애플리케이션에 사용가능하다. UserService 인스턴스를 얻기 위한 페이지를 위한 간단한 기법은 예를 들면, 다음과 같을것이다.

```
WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
    getRequestCycle().getRequestContext().getServlet().getServletContext());
UserService userService = (UserService) appContext.getBean("userService");
... some code which uses UserService
```

이 기법은 언급한것처럼 작동한다. 이것은 페이지나 컴포넌트를 위한 기본(base) 클래스내 메소드내 대부분의 기능을 캡슐화하여 다서 덜 장황하다. 어쨌든, 이 애플리케이션의 다른 레이어에서 사용되는 몇가지 관점에서, Spring이 지원하는 IoC접근법에 대해 반대로 작동한다. 이론적으로 당신은 이름(name)에 의해 종속 bean을 위한 컨텍스트를 요청하지 않기 위한 페이지를 좋아할것이다. 그리고 사실, 페이지는 컨텍스트에 대해 알지 않는다.

운좋게, 여기에 이것을 허용하는 기법이 있다. Tapestry는 프라퍼티를 페이지에 명시적으로 추가하기 위한 기법을 이미 가지고 있다. 그리고 이것은 선언적인 방식으로 페이지의 모든 프라퍼티를 관리하기 위한 접근법을 선호한다. 그래서 Tapestry는 아마도 페이지와 컴포넌트 생명주기의 일부처럼 자체의 생명주기를 관리할수 있다.

Note

다음 부분은 Tapestry 4.0이전 버전에 사용가능하다. 만약 Tapestry 4.0+을 사용한다면, Section 15.5.1.4, “Tapestry 페이지에 Spring Beans 의존성 삽입하기 - Tapestry 4.0이상의 버전 스타일” 를 보라.

15.5.1.1. Tapestry페이지에 대한 Spring Bean 의존성 삽입

먼저 우리는 ServletContext를 가지지 않고 Tapestry페이지나 컴포넌트에 사용가능한 ApplicationContext를 만들필요가 있다. 이것 때문에, 우리가 ApplicationContext에 접근할때 페이지/컴포넌트의 생명주기에서, ServletContext를 위해 페이지에 쉽게 사용가능하지 않을것이다. 그래서 우리는 직접 WebApplicationContextUtils.getApplicationContext(servletContext)를 사용할수 없다. 하나의 방법은 Tapestry IEngine의 사용자정의를 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 Tapestry애플리케이션의 '전역(Global)' 객체내 'appContext' 라고 불리는 속성처럼 Spring 애플리케이션 컨텍스트를 둔다. 특별한 IEngine인스턴스가 Tapestry애플리케이션 정의 파일내 항목과 함께 Tapestry애플리케이션을 위해 사용되어야만 하는 것을 등록하라. 예를 들면,

```
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>
```

15.5.1.2. 컴포넌트 정의 파일

우리의 페이지나 컴포넌트 정의 파일(*.page 나 *.jwc)에서, 우리는 ApplicationContext를 필요로 하고 페이지나 컴포넌트 프라퍼티를 생성하는 bean을 가로채기 위해 프라퍼티 성격의 요소를 간단히 추가한다. 예를 들면:

```
    <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>
```

프라퍼티 특성 내부의 OGNL표현은 컨텍스트로부터 얻어지는 bean처럼 프라퍼티를 위한 초기값을 명시한다. 전체 페이지 정의는 다음과 같을것이다.

```
<?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>
```

15.5.1.3. 추상 접근자(accessor) 추가하기

지금 페이지나 컴포넌트 자체를 위한 Java클래스 정의내에서, 우리가 해야할 필요가 있는 모든것은 우리가 정의(언급된 프라퍼티에 접근하기 위해)한 프라퍼티를 위한 추상 getter메소드를 추가하는 것이다.

```
// our UserService implementation; will come from page definition
public abstract UserService getUserService();
// our AuthenticationService implementation; will come from page definition
public abstract AuthenticationService getAuthenticationService();
```

완성을 위해, 이 예제내 로그인 페이지를 위한 전체 Java클래스는 다음과 같을 것이다.

```
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 the cookie that identifies a user **/
private static final String COOKIE_NAME = Login.class.getName() + ".username";
private final static int ONE_WEEK = 7 * 24 * 60 * 60;

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();

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);
        }

        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));
        }
    }
}
```

15.5.1.4. Tapestry 페이지에 Spring Beans 의존성 삽입하기 - Tapestry 4.0이상의 버전 스타일

Tapestry 4.0이상의 버전에서 Tapestry 페이지에 Spring관리 bean의 의존성 삽입의 영향은 좀더 단순하다. 해야 할 필요가 있는 것은 하나의 애드온 라이브러리, 와 몇몇 설정, 다른 웹 프레임워크에 의해 요구되는 다른 라이브러리와 함께(대개 WEB-INF/lib에서) 간단히 패키징하고 배치하는 것이다.

그리고 나서 이전에 이미 언급된 메소드를 사용하여 Spring컨테이너를 생성하고 나타낼 필요가 있다. Tapestry에 Spring 관리 bean을 매우 간단히 삽입할수 있다. 만약 Java 5를 사용한다면, 위 Login 페이지를 보라. 우리는 Spring관리 userService와 authenticationService객체의 의존성 삽입을 위한 적절한 getter메소드를 주석처리할 필요가 있다.

```
package com.whatever.web.xportal.pages;

public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    @InjectObject("spring:userService")
    public abstract UserService getUserService();

    @InjectObject("spring:authenticationService")
    public abstract AuthenticationService getAuthenticationService();

}
```

지금 시점에서는 많은 작업을 수행했다. 남은 것은 HiveMind서비스로 ServletContext내 저장된 Spring 컨테이너를 나타내는 HiveMind설정이다.

```
<?xml version="1.0"?>
<module id="com.javaforge.tapestry.spring" version="0.1.1">

    <service-point id="SpringApplicationInitializer"
        interface="org.apache.tapestry.services.ApplicationInitializer"
        visibility="private">
        <invoke-factory>
            <construct class="com.javaforge.tapestry.spring.SpringApplicationInitializer">
                <set-object property="beanFactoryHolder"
                    value="service:hivemind.lib.DefaultSpringBeanFactoryHolder" />
            </construct>
        </invoke-factory>
    </service-point>

    <!-- Hook the Spring setup into the overall application initialization. -->
    <contribution
        configuration-id="tapestry.init.ApplicationInitializers">
        <command id="spring-context"
            object="service:SpringApplicationInitializer" />
    </contribution>

</module>
```

만약 Java 5를 사용한다면(게다가 주석(annotations)에 접근한다면), 이것은 다 된셈이다.

만약 Java 5를 사용하지 않는다면, 주석을 가진 Tapestry페이지 클래스를 주석처리하지 않는다. 대신 의존성 삽입을 명시하기 위해 좋은 예전 형태의 XML을 사용한다. Login 페이지(나 컴포넌트)를 위한 .page 나 .jwc 파일내부는 ..

```
<inject property="userService" object="spring:userService"/>
<inject property="authenticationService" object="spring:authenticationService"/>
```

이 예제에서, 우리는 선언적인 형태로 Tapestry페이지를 위해 제공되는 Spring컨터이너내 정의된 서비스 bean을 허용하기 위해 관리된다. 페이지 클래스는 서비스 구현물이 어디서 발생하는지 알지 않는다. 사실, 예를 들어, 테스팅하는 동안 다른 구현물을 빠뜨리기 쉽다. 이 IoC는 Spring프레임워크의 목표와 이득중 하나이다. 그리고 우리는 이 Tapestry애플리케이션내 J2EE스택을 모두 확장하기 위해 관리된다.

15.6. WebWork

WebWork 홈페이지로 부터...
" WebWork는 Java 웹 애플리케이션 개발 프레임워크이다. 이것은 폼 제어, UI테마, 국제화, JavaBean에 대한 동적 폼 파라미터 맵핑, 견고한 클라이언트및 서버측 유효성 체크 그리고 다른 많은 것들과 같은 재사용가능한 UI템플릿을 빌드하기 위한 견고한 지원을 제공하여 개발자 생산성과 코드 단순화를 가진채 명백하게 빌드되었다. "

WebWork는 매우 깔끔하고, 정밀한 웹 프레임워크이다(저자의 의견을 따르면). 이 구조와 key개념은 이해하기 매우 쉬울뿐 아니라 풍부한 태그 라이브러리, 제대로 디-커플링된 유효성 체크(validation)를 가지고 있다. 그리고 다음번에 생산되기 쉽도록 되어 있다.

WebWork의 기술 스택내 핵심 가능자(enabler)중 하나는 Webwork Action을 관리하고 비지니스 객체의 "wiring"을 다루는 IoC 컨테이너이다. WebWork버전 2.2이전에는, WebWork가 자체적으로 적절한

IoC컨테이너를 사용했다(그리고 제공된 통합 지점은 Spring처럼 IoC컨테이너와 통합할수 있다). 어쨌든, WebWork 2.2처럼, WebWork내 사용되는 디폴트 IoC컨테이너는 Spring이다. Spring개발자라면 이러한 사항은 굉장히 멋진 소식이다. 왜냐하면 이것은 IoC설정의 기본에 이미 친숙하다는 것을 의미한다.

지금 DRY(Dont Repeat Yourself - 당신 스스로 되풀이 하지말라) 법칙을 따른다면, WebWork팀이 이미 작성해둔 Spring-WebWork통합을 다시 작성하는 것을 행하지 말라. WebWork 위키에서 Spring-WebWork 통합 페이지를 보라.

Spring-WebWork통합 코드는 WebWork개발자들에 의해 개발되었다. WebWork사이트에서 참조하라. 관련글은 Spring 지원 포럼에서 볼수 있다.

15.7. 추가적인 자원

아래는 이 장에서 언급한 다양한 웹 프레임워크에 대한 추가적인 자원에 대한 링크이다.

☒ JSF 홈페이지

☒ Tapestry 홈페이지

☒ WebWork 홈페이지

Chapter 16. 포틀릿(Portlet) 통합

16.1. 소개

JSR-168 Java 포틀릿 스펙

포틀릿 개발에 대한 좀더 다양한 정보를 위해, "Introduction to JSR 168"라는 Sun의 백서와 JSR-168 Specification 를 다시 보라.

편리한 웹 개발을 지원하기 위해 추가적으로, Spring은 JSR-168 포틀릿 개발을 지원한다. 가능한한 많이, 포틀릿 MVC 프레임워크는 웹 MVC 프레임워크를 대표하는 이미지이고 같이 참조하는 view추상화와 통합 기술을 사용한다. 이 장을 계속 보기 전에 Chapter 13, 웹 MVC framework 와 Chapter 14, 통합 뷰 기술들를 먼저보라.

Note

Spring MVC의 개념이 Spring 포틀릿 MVc와 같을때, 여기엔 JSR-168 포틀릿의 유일한 워크플로우에 의해 생성되는 몇가지 눈에 띄는 차이점이 있다는 것을 명심하라.

서블릿 워크플로우와는 다른 포트릿 워크플로우내 가장 핵심적인 방법은 포틀릿에 대한 요청이 두가지 다른 단계(action과 render단계)를 가진다. action단계는 데이터베이스내 변경사항처럼 오직 한번만 실행되고 "backend" 변경이나 action이 발생하는 곳에서 실행된다. render단계는 표시가 갱신되는 매번 사용자에게 표시되는 것을 만든다. 여기서의 중대한 점은 하나의 종합적인 요청을 위한 것이다. action단계는 오직 한번만 실행되지만 render단계는 여러번 실행된다. 이것은 시스템의 영속적인 상태와 사용자에게 보여주는 것을 생성하는 활동을 변경하는 활동(activities)간의 명백한 구분을 제공(요구)한다.

포틀릿 요청의 두 단계는 JSR-168 개발의 실제 강력함중의 하나이다. 예를 들어, 동적인 검색 결과는 사용자가 검색을 명시적으로 다시 수행하지 않고 대개 업데이트할수 있다. 대부분의 다른 포틀릿 MVC 프레임워크는 개발자로부터 두 단계를 완벽하게 숨기도록 시도한다. 그리고 가능한한 전통적인 서블릿 개발처럼 보이도록 한다. ☒ 우리는 이 접근법이 포틀릿을 사용하는 중요한 이익중 하나를 제거한다고 생각한다. 그래서 두 단계의 분리는 Spring 포틀릿 MVC 프레임워크도처에 유지했다. 이 접근법의 중요한 표현은 MVC 클래스의 서블릿 버전이 요청을 다루는 두개의 메소드(action단계를 위한 메소드와 render단계를 위한 메소드)를 가지는 것이다. 예를 들어, AbstractController 의 서블릿 버전은 handleRequestInternal(..) 메소드를 가진다. AbstractController의 포틀릿 버전은 handleActionRequestInternal(..) 메소드와 handleRenderRequestInternal(..) 메소드를 가진다.

프레임워크는 웹 프레임워크내 DispatcherServlet가 하는것처럼 설정가능한 핸들러 맵핑과 view해석(resolution)을 가지고 요청을 핸들러에 분배하는 DispatcherPortlet에 대해 디자인되었다. 파일 업로드는 같은 방법으로 제공된다.

로케일 해석과 테마 해석은 포틀릿 MVC 에서 제공되지 않는다. - 이러한 영역은 포틀릿/포틀릿-컨테이너의 범위에 있고 Spring레벨에서는 적절하지 않다. 어쨌든, 로케일(메시지의 국제화와 같은)에 의존하는 Spring내 모든 기법은 DispatcherPortlet이 DispatcherServlet와 같은 방법으로 현재 로케일을 나타내기 때문에 적당하게 기능화될것이다.

16.1.1. Controllers - MVC내 C

디폴트 핸들러는 두개의 메소드만을 제공하는 여전히 매우 간단한 Controller 인터페이스이다.

☒ void handleActionRequest(request,response)

☒ ModelAndView handleRenderRequest(request,response)

프레임워크 AbstractController, SimpleFormController 등등과 같이 대부분 같은 컨트롤러 구현물 구조를 포함한다. 데이타 바인딩, command 객체 사용법, 모델 핸들링, 그리고 view해석은 서블릿 프레임워크에서 모두 같다.

16.1.2. Views - MVC내 V

서블릿 프레임워크의 모든 view표현 기능은 ViewRendererServlet라고 명명된 특별한 다리 역활의 서블릿을 통해 직접 사용된다. 이 서블릿을 사용하여, 포틀릿 요청은 서블릿 요청으로 변환되고 view는 전체적으로 대개의 서블릿 내부구조를 사용하여 표시될수 있다. 이것은 JSP, Velocity등등 존재하는 모든 표시자(renderer)가 포틀릿내 사용될수 있다는 것을 의미한다.

16.1.3. web 범위의 bean

Spring 포틀릿 MVC는 생명주기가 현재 HTTP요청또는 HTTP Session에 범위화되는 bean을 지원한다. 이것은 Spring 포틀릿 MVC자체의 특별한 기능일뿐 아니라 Spring 포틀릿 MVC가 사용하는 WebApplicationContext 컨테이너이다. bean범위는 Section 3.5.3, “The other scopes” 에서 상세히 언급된다.

Note

Spring배포판은 Spring 포틀릿 MVC프레임워크의 모든 기능과 함수를 보여주는 완벽한 Spring 포틀릿 MVC샘플 애플리케이션을 가진다.

‘petportal’ 애플리케이션은 Spring 배포판의 samples/petportal’ 디렉토리에서 찾을수 있다.

16.2. DispatcherPortlet

포틀릿 MVC는 요청-지향의 웹 MVC프레임워크이다. 요청을 컨트롤러에 분배하는 포틀릿에 대해 디자인되었고 포틀릿 애플리케이션의 개발 기능을 제공한다. Spring의 DispatcherPortlet는 어쨌든, 그것보다 더 많은 것을 한다. 이것은 Spring의 ApplicationContext와 완전하게 통합되었고 Spring이 가지는 다른 모든 기능을 사용할수 있도록 해준다.

보통의 포틀릿처럼, DispatcherPortlet이 웹 애플리케이션의 portlet.xml내 선언되었다.

```
<portlet>
        <portlet-name>sample</portlet-name>
        <portlet-class>org.springframework.web.portlet.DispatcherPortlet</portlet-class>
        <supports>
                <mime-type>text/html</mime-type>
                <portlet-mode>view</portlet-mode>
        </supports>
        <portlet-info>
                <title>Sample Portlet</title>
        </portlet-info>
</portlet>
```

DispatcherPortlet이 현재 설정될 필요가 있다.

포틀릿 MVC프레임워크에서, 각각의 DispatcherPortlet은 가장 상위의 WebApplicationContext내 이미 정의된 모든 bean을 상속하는 자체적인 WebApplicationContext를 가진다. 이렇게 상속된 bean은 포틀릿 특성의 scope에서 오버라이드될수 있고 새로운 scope 특성을 가진 bean은 주어진 포틀릿 인스턴스에 로컬로 정의될수 있다.

DispatcherPortlet 의 초기화중 프레임워크는 웹 애플리케이션내 WEB-INF 디렉토리내 [portlet-name]-portlet.xml 라는 이름의 파일을 찾을것이고 거기에 정의된 bean을 생성할것이다(전역 scope에서 같은 이름을 가지고 정의된 bean의 정의를 오버라이드한다.).

DispatcherPortlet 에 의해 사용된 설정 위치는 포틀릿 초기화 파라미터를 통해 변경될수 있다(상세한 사항은 아래에서 보라).

Spring DispatcherPortlet은 요청을 처리하고 적절한 view로 표시하는 것을 가능하게 하기 위해 몇가지 특별한 bean을 가진다. 여기엔 Spring 프레임워크에 포함된 bean이 있고 WebApplicationContext 에서 설정될수 있다. 각각의 bean은 아래에서 좀더 상세하게 언급된다. 우리는 지금 그것들을 언급할것이다. DispatcherPortlet에 대해 좀더 얘기해보자. 대부분이 bean을 위해, 디폴트가 제공되기 때문에 당신은 설정에 대해 걱정할 필요가 없다.

Table 16.1. WebApplicationContext 내 특별한 bean

표시	설명
핸들러 팹핑	(Section 16.5, “핸들러 맵핑”) 어떤 규칙에 일치한다면 수행될 전처리(pre-) 와 후처리(post-) 프로세서와 컨트롤러의 목록(예를 들어, 일치되는 포틀릿은 컨트롤러와 함께 명시된다.)
컨트롤러	(Section 16.4, “컨트롤러”) MVC 의 일부처럼 실질적인 기능(적어도 기능에 접근하는)을 제공하는 bean
view 해석자(resolver)	(Section 16.6, “View와 View해석하기”) view이름을 view정의에 해석하는 능력
multipart 해석자(resolver)	(Section 16.7, “Multipart (파일 업로드) 지원”) HTML폼으로부터 파일 업로드를 처리하는 기능을 제공
핸들러 예외 해석자(resolver)	(Section 16.8, “예외 다루기”) view에 대한 예외를 맵핑하거나 좀더 복잡한 예외 핸들링 코드를 구현하는 기능을 제공

DispatcherPortlet이 사용하기 위해 셋업되고 요청이 특정 DispatcherPortlet를 위해 들어올때, 이것은 요청을 처리하기 시작한다. 아래 목록은 DispatcherPortlet에 의해 다루어진다는 것을 통해 요청을 완벽하게 처리하는 것을 언급한다.

1. PortletRequest.getLocale()에 의해 반환되는 로케일은 요청을 처리(view를 표시하고 데이터를 준비하는 등등)할때 사용할 로케일을 프로세스가 해석하여 요청에 반영한다.

2. multipart 해석자가 명시되고 이것이 ActionRequest 라면, 요청은 multipart를 위해 검사되고 만약 발견된다면, 프로세스내 다른 요소에 의해 처리되기 위해 MultipartActionRequest에 포장된다(multipart 핸들링에 대한 더 많은 정보를 위해 Section 16.7, “Multipart (파일 업로드) 지원” 를 보라)

3. 적절한 핸들러가 검색되었다. 핸들러가 발견된다면, 핸들러에 관련된 수행체인(execution chain - 전처리 프로세서, 후처리 프로세서, 컨트롤러)은 모델을 준비하기 위해 수행될것이다.

4. 모델이 반환되면, view를 표시된다, view해석자를 사용하는 것은 WebApplicationContext로 설정된다. 만약 모델이 반환되지 않는다면(예를 들어, 보안적인 이유로, 전처리 또는 후처리 프로세서가 요청을 가로챌수 있다), 요청이 이미 완수된 이래 어떠한 view도 표시되지 않는다.

요청의 처리중 던져지는 예외는 WebApplicationContext내 선언된 예외 해석자 핸들러를 가진다. 이러한 예외 해석자를 사용하여 당신은 예외가 던져지는 것과 같은 경우 사용자정의 행위를 정의할수 있다.

당신은 portlet.xml파일내 컨텍스트 파라미터나 포틀릿 init 파라미터를 추가하여 Spring의 DispatcherPortlet를 사용자정의 할수 있다. 가능한 값은 아래와 같다.

Table 16.2. DispatcherPortlet 초기화 파라미터

파라미터	설명
contextClass	WebApplicationContext를 구현한 클래스는 포틀릿에 의해 사용되는 컨텍스트를 인스턴스화하기 위해 사용될것이다. 파라미터가 명시되지 않는다면, XmlPortletApplicationContext가 사용될것이다.
contextConfigLocation	컨텍스트가 발견되는 위치를 표시하기 위해 컨텍스트 인스턴스에 전달(contextClass에 의해 명시되는)되는 문자열. 문자열은 잠재적으로 다중 컨텍스트(이 경우 다중 컨텍스트 위치, 두개로 정의된 bean, 마지막것이 우선권을 가진다)를 지원하기 위해 다중 문자열로 나누어진다(구분자로 콤마를 사용).
namespace	WebApplicationContext의 명명공간. 디폴트는 [portlet-name]-portlet 이다.
viewRendererUrl	DispatcherPortlet의 URL은 ViewRendererServlet(Section 16.3, “ViewRendererServlet” 를 보라) 인스턴스에 접근할수 있다.

16.3. ViewRendererServlet

포틀릿 MVC내 랜더링 처리는 웹 MVC보다는 좀더 복잡하다. Spring 웹 MVC로부터 모든 View기술을 다시 사용하기 위해, PortletRequest / PortletResponse를 HttpServletRequest / HttpServletResponse로 변환하고 View의 render 메소드를 호출한다. 이것을 하기 위해, DispatcherPortlet은 이러한 목적을 위해 존재하는 특별한 서블릿인 ViewRendererServlet을 사용한다.

DispatcherPortlet 랜더링을 위해, 다음처럼 웹 애플리케이션을 위한 web.xml 파일내 ViewRendererServlet 인스턴스를 명시해야만 한다.

```
<servlet>
    <servlet-name>ViewRendererServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.ViewRendererServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>ViewRendererServlet</servlet-name>
    <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

실제 랜더링을 수행하기 위해, DispatcherPortlet은 다음을 수행한다.

1. DispatcherServlet이 사용하는 WEB_APPLICATION_CONTEXT_ATTRIBUTE key와 같은 속성처럼 WebApplicationContext을 요청으로 바인드한다.
2. Model 과 View 객체를 ViewRendererServlet에 사용가능하도록 만들기 위해 요청에 바인드한다.
3. PortletRequestDispatcher를 생성하고 ViewRendererServlet에 맵핑되는 /WEB- INF/servlet/view URL을 포함한다.

ViewRendererServlet 은 적절한 인자를 가진 View의 render 메소드를 호출할수 있다.

ViewRendererServlet 을 위한 실제 URL은 DispatcherPortlet의 viewRendererUrl 설정 파라미터를 사용하여 변경될수 있다.

16.4. 컨트롤러

포틀릿 MVC내 컨트롤러는 웹 MVC컨트롤러와 매우 유사하다. 포팅 코드는 간단할것이다.

포틀릿 MVC 컨트롤러 구조를 위한 기본은 아래에 목록화된 org.springframework.web.portlet.mvc.Controller 인터페이스이다.

```
public interface Controller {

    /**
     * Process the render request and return a ModelAndView object which the
     * DispatcherPortlet will render.
     */
    ModelAndView handleRenderRequest(RenderRequest request, RenderResponse response)
        throws Exception;

    /**
     * Process the action request. There is nothing to return.
     */
    void handleActionRequest(ActionRequest request, ActionResponse response)
        throws Exception;

}
```

당신이 볼수 있는것처럼, 포틀릿 Controller 인터페이스는 포틀릿 요청(action요청과 render요청)의 두 단계를 다루는 두개의 메소드를 요구한다. action단계는 action요청을 다룰수 있고 render단계는 render단계를 다루고 적절한 모델과 view를 반환한다. Controller인터페이스가 추상적인 반면에, Spring 포틀릿 MVC는 당신이 필요한 많은 기능을 포함하는 많은 컨트롤러를 제공하고 Spring 웹 MVC의 컨트롤러와 매우 유사하다. Controller 인터페이스는 모든 컨트롤러에서 요구되는 가장 공통적인 기능(- action요청을 다루고 render요청을 다루고 모델과 view를 반환하는)을 정의한다.

16.4.1. AbstractController 와 PortletContentGenerator

물론, Controller 인터페이스는 충분하지 않다. 기초적인 구조를 제공하기 위해, 모든 Spring 포틀릿 MVC의 Controller는 Spring의 ApplicationContext에 접근하고 캐시를 제어하는 클래스인 AbstractController로부터 상속된다.

Table 16.3. AbstractController에 의해 제공되는 기능

파라미터	설명
requireSession	이 Controller 가 작동하기 위한 session을 필요로 할지에 대한 표시. 이 기능은 모든 컨트롤러에 제공된다. 컨트롤러가 요청을 받을때 session이 존재하지 않는다면, 사용자는 SessionRequiredException을 사용하여 정보를 볼수 있다.
synchronizeSession	사용자 session에서 동기화되는 컨트롤러에 의해 다루어지길 원한다면 이 파라미터를 사용하라. 좀더 다양하게 설정하기 위해, 컨트롤러를 확장하는 것은 이 변수를 명시한다면 사용자의 session에서 동기화될 handleRenderRequestInternal(..)과 handleActionRequestInternal(..) 메소드를 오버라이드할것이다.
renderWhenMinimized	포틀릿이 최소화된 상태에서 view을 실제로 표시하기 위한 컨트롤러를 원한다면, 이 값을 true로 셋팅하라. 디폴트에 의해, 이것은 false로 셋팅되기 때문에 최소화된 상태의 포틀릿은 어떠한 컨텐츠로 표시하지 않는다.
cacheSeconds	컨트롤러가 포틀릿을 위해 정의된 디폴트 캐시 만료를 오버라이드하기 원한다면, 여기에 양수의 값을 명시하라. 디폴트에 의해 이것은 디폴트 캐시를 변경하지 않는 -1로 셋팅된다. 0으로 셋팅하면 결코 캐시되지 않을것이다.

requireSession 과 cacheSeconds 프라퍼티는 실제로 PortletContentGenerator(AbstractController의 수퍼클래그인)의 일부이지만 완전한 설명을 위해 여기에 포함시켰다.

컨트롤러를 위한 기본클래스처럼 AbstractController를 사용(다른 많은 컨트롤러가 당신을 위해 많은 작업을 이미 구현하고 있기 때문에, 이를 추천하지는 않는다)할때, 당신은 로직을 구현하고 ModelAndView 객체(이 경우 handleRenderRequestInternal)를 반환하는 handleActionRequestInternal(ActionRequest, ActionResponse) 메소드나 handleRenderRequestInternal(RenderRequest, RenderResponse) 메소드를 오버라이드해야만 한다.

handleActionRequestInternal(..) 과 handleRenderRequestInternal(..)의 디폴트 구현물은 PortletException을 던진다. 이것은 JSR-168 스펙 API의 GenericPortlet의 행위와 일치한다. 그래서 당신의 컨트롤러가 다루고자 하는 메소드를 오버라이드할 필요가 있다.

이것은 웹 애플리케이션 컨텍스트에서 클래스와 선언으로 구성된 간단한 예제이다.

```
package samples;

import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.springframework.web.portlet.mvc.AbstractController;
import org.springframework.web.portlet.ModelAndView;

public class SampleController extends AbstractController {

    public ModelAndView handleRenderRequestInternal(
        RenderRequest request,
        RenderResponse response) throws Exception {
        ModelAndView mav = new ModelAndView("foo");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}

<bean id="sampleController" class="samples.SampleController">
```

```
    <property name="cacheSeconds" value="120"/>
</bean>
```

위 클래스와 웹 애플리케이션 컨텍스트내 선언은 간단한 컨트롤러 작업을 하기 위해 핸들러 맵핑을 셋업(Section 16.5, “핸들러 맵핑” 를 보라.)하는 것외에 필요한 모든것이다.

16.4.2. 다른 간단한 컨트롤러

비록 AbstractController를 확장할수 있다고 하나, Spring 포틀릿 MVC는 간단한 MVC애플리케이션내에서 공통적으로 사용되는 기능을 제공하는 많은 수의 구현물을 제공한다.

ParameterizableViewController 는 웹 애플리케이션 컨텍스트(하드코딩된 view명에 필요하지 않은)내 반환될 view명을 명시할수 있다는 것을 제외하고 위 예제와 기본적으로 같다.

PortletModeNameViewController 는 view명으로 포틀릿의 현재 모드를 사용한다. 그래서 당신의 포틀릿이 View모드(이를테면, PortletMode.VIEW)라면, view명으로 "view"를 사용한다.

16.4.3. Command 컨트롤러

Spring 포틀릿 MVC는 Spring 웹 MVC처럼 command 컨트롤러의 구조와 정확하게 같다. 컨트롤러는 데이터 객체와 상호작용하기 위한 방법을 제공하고 PortletRequest로부터 명시된 데이터 객체에 파라미터를 동적으로 바인드한다. 당신의 데이터 객체는 프레임워크 특유의 인터페이스를 구현하지 않아서 당신이 원한다면 퍼시스턴트 객체를 직접 다룰수 있다. 컨트롤러로 할수 있는 것의 개요를 알기 위해 command컨트롤러가 사용가능한 것인지 시험해보자.

☒ AbstractCommandController - 자체적인 컨트롤러를 생성하기 위해 사용할수 있는 command컨트롤러. 요청 파라미터를 당신이 명시하는 데이터 객체에 바인딩하는 능력이 있다. 이 클래스는 폼 기능을 제공하지 않는다. 이것은 어쨌든 유효성 체크 기능을 제공하고 요청으로부터 파라미터를 다루는 command객체로 하는 것을 컨트롤러에 명시하도록 한다.

☒ AbstractFormController - 폼 서브밋 지원을 제공하는 추상 컨트롤러. 이 컨트롤러를 사용하여 당신은 폼을 모델화할수 있고 컨트롤러내 가져오는 command객체를 사용하여 그것들을 활성화한다. 사용자가 폼을 채운후에, AbstractFormController 는 필드를 바인드(bind)하고 유효성체크하고 적절한 action을 가지는 컨트롤러에 객체를 넘긴다. 지원되는 기능은 유효하지 않은 폼 서브밋(resubmission), 유효성 체크, 일반적인 폼 워크플로우. 당신은 폼 표현을 위해 사용되는 view와 성공여부를 판단하기 위한 메소드를 구현한다. 폼이 필요하다면 이 컨트롤러를 사용하라. 하지만 view를 명시하기를 원하지 않는다면 당신은 애플리케이션 컨텍스트내 사용자를 보여줄것이다.

☒ SimpleFormController - 관련 command객체로 폼을 생성할때 좀더 많은 지원을 제공하는 AbstractFormController. SimpleFormController는 command객체, 폼을 위한 viewname, 폼 서브밋이 성공했을때 사용자에게 보여주길 원하는 페이지를 위한 viewname 그리고 기타등등을 명시하도록 한다.

☒ AbstractWizardFormController ☒ 여러개의 페이지를 통해 command객체의 내용을 편집하기 위한 마법사-스타일의 인터페이스를 제공하는 AbstractFormController. 다중 사용자 action인 종료(finish), 취소(cancel), 또는 페이지 변경, view로부터 요청 파라미터로 쉽게 명시하는 모든것을 지원한다.

이러한 command 컨트롤러는 굉장히 강력하다. 하지만 그것들을 효과적으로 사용하기 위해서 작동하는 방법을 상세하게 이해해야만 한다. 이 전체적인 구조를 위해 JavaDoc를 주의깊게 보고 사용하기 전에

몇가지 샘플 구현물을 보라.

16.4.4. PortletWrappingController

새로운 컨트롤러를 개발하는 대신에, 존재하는 포틀릿을 사용하고 DispatcherPortlet으로부터 요청을 맵핑하는 것이 가능하다. PortletWrappingController을 사용하여, 다음의 Controller처럼 Portlet을 인스턴스화할수 있다.

```
<bean id="wrappingController"
      class="org.springframework.web.portlet.mvc.PortletWrappingController">
    <property name="portletClass" value="sample.MyPortlet"/>
    <property name="portletName" value="my-portlet"/>
    <property name="initParameters">
        <value>
            config=/WEB-INF/my-portlet-config.xml
        </value>
    </property>
</bean>
```

당신이 이러한 포틀릿으로 가는 전처리-프로세스와 후처리-프로세스를 위한 인터셉터를 사용할수 있기 때문에 매우 가치있을수 있다. JSR-168이 어떠한 종류의 필터 기법도 지원하지 않으므로, 다루기 쉽다. 예를들어, 이것은 MyFaces JSF 포틀릿에서 Hibernate OpenSessionInViewInterceptor를 포장하기 위해 사용될수 있다.

16.5. 핸들러 맵핑

핸들러 맵핑을 사용하여 당신은 포틀릿 요청을 적절한 핸들러로 맵핑할수 있다. 여기엔 예를 들어, PortletModeHandlerMapping과 같이 당신이 특별히 사용할수 있는 몇가지 핸들러 맵핑이 있다. 하지만 HandlerMapping의 일반적인 개념부터 살펴보자.

노트 : 우리는 "컨트롤러" 대신에 여기서 "핸들러"의 개념을 사용한다. DispatcherPortlet은 Spring 포틀릿 MVC 자체의 컨트롤러보다는 요청을 처리하기 위한 방법으로 사용되도록 디자인되었다. 핸들러는 포틀릿 요청을 다룰수 있는 객체이다. 컨트롤러는 핸들러의 예이고, 물론 디폴트이다. DispatcherPortlet로 다른 프레임워크를 사용하기 위해, HandlerAdapter의 관련 구현물이 필요한 모든것이다.

기본 HandlerMapping이 제공하는 기능은 요청에 일치하는 핸들러를 포함해야만 하고 요청에 적용되는 핸들러 인터셉터의 목록을 포함하는 HandlerExecutionChain의 전달이다. 요청이 들어오면, DispatcherPortlet은 요청을 조사하고 적절한 HandlerExecutionChain을 가져오도록 핸들러 맵핑을 다룰것이다. DispatcherPortlet은 체인내 핸들러와 인터셉터를 수행할것이다. 이러한 개념은 Spring 웹 MVC와 정확하게 같다.

선택적으로 인터셉터(실제 핸들러가 수행되지 전이나 후에 또는 전후 모두 실행되는)를 포함할수 있는 설정가능한 핸들러 맵핑의 개념은 굉장히 강력하다. 지원되는 많은 기능은 사용자 정의된 HandlerMapping에 포함될수 있다. 핸들러를 선택하는 사용자 정의 핸들러는 들어오는 요청의 포틀릿 모드에 기초할 뿐 아니라 요청에 관련된 session의 명시된 상태에도 기초를 둔다.

Spring 웹 MVC에서, 핸들러 맵핑은 URL에 공통적으로 기반한다. 포틀릿내 URL과 같은 것이 없다면, 맵핑을 제어하기 위한 다른 기법을 사용해야만 한다. 두개의 가장 공통적인 것은 포틀릿 모드와 요청 파라미터이다. 하지만 포틀릿 요청에 대해 사용가능한 것은 사용자 정의 핸들러 맵핑에서 사용될수 있다.

이 부분의 나머지는 Spring 포틀릿 MVC의 가장 공통적으로 사용되는 핸들러 맵핑중 세가지를 언급한다. 그것들은 모두 AbstractHandlerMapping를 확장하고 다음의 프라퍼티를 공유한다.

☒ interceptors: 사용하는 인터셉터의 목록. HandlerInterceptor는 Section 16.5.4, “HandlerInterceptor 추가하기” 에서 다루어진다.

☒ defaultHandler: 핸들러 맵핑이 결과적으로 이루어지지 않았을때 사용하기 위한 디폴트 핸들러.

☒ order: order 프라퍼티(org.springframework.core.Ordered 인터페이스를 보라)의 값에 기초를 둠. Spring은 컨텍스트내 사용가능한 핸들러 맵핑을 모두 정렬하고 첫번째 일치되는 핸들러에 적용할것이다.

☒ lazyInitHandlers: 싱글톤 핸들러의 늦은(lazy) 초기화를 허용(prototype 핸들러는 언제나 늦게 초기화된다). 디폴트 값은 false. 이 프라퍼티는 세가지의 구체적인(concrete) 핸들러내 직접 구현된다.

16.5.1. PortletModeHandlerMapping

포틀릿의 현재 모드(이를테면, ‘view’, ‘edit’, ‘help’)에 기초하여 들어오는 요청을 맵핑하는 간단한 핸들러 맵핑. 예를 들면:

```
<bean id="portletModeHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="viewHandler"/>
            <entry key="edit" value-ref="editHandler"/>
            <entry key="help" value-ref="helpHandler"/>
        </map>
    </property>
</bean>
```

16.5.2. ParameterHandlerMapping

포틀릿 모드의 변경없이 여러개의 컨트롤러를 탐색할 필요가 있다면, 가장 간단한 방법은 맵핑을 제어하는 key처럼 사용되는 요청 파라미터를 가지는 것이다.

ParameterHandlerMapping 은 맵핑을 제어하기 위해 명시된 요청 파라미터의 값을 사용. 파라미터의 디폴트명은 ‘action’이지만, ‘parameterName’ 프라퍼티를 사용하여 변경될수 있다.

이 맵핑을 위한 bean설정은 다음과 비슷할것이다.

```
<bean id="parameterHandlerMapping"
      class="org.springframework.web.portlet.handler.ParameterHandlerMapping" >
    <property name="parameterMap">
        <map>
            <entry key="add" value-ref="addItemHandler"/>
            <entry key="edit" value-ref="editItemHandler"/>
            <entry key="delete" value-ref="deleteItemHandler"/>
        </map>
    </property>
</bean>
```

16.5.3. PortletModeParameterHandlerMapping

가장 강력하게 내장된 핸들러 맵핑. PortletModeParameterHandlerMapping은 이전의 두개와 각각의 포틀릿

모드에서 다른 탐색기법을 허용하는 기능을 모두 가진다.

파라미터의 디폴트명은 "action" 이지만 parameterName 프라퍼티를 사용하여 변경될수 있다.

디폴트에 의해, 같은 파라미터 값은 두개의 다른 포틀릿 모드에서 사용되지 않는다. 포털자체가 포틀릿 모드를 변경한다면, 요청은 맵핑내에서 더이상 유효하지 않을것이다. 이 행위는 allowDupParameters 프라퍼티를 true로 셋팅하여 변경될수 있다. 어쨌든 이것이 추천되지는 않는다.

이 맵핑을 위한 bean설정은 다음과 비슷할것이다.

```
<bean id="portletModeParameterHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeParameterHandlerMapping">
    <property name="portletModeParameterMap">
        <map>
            <entry key="view"> <!-- 'view' portlet mode -->
                <map>
                    <entry key="add" value-ref="addItemHandler"/>
                    <entry key="edit" value-ref="editItemHandler"/>
                    <entry key="delete" value-ref="deleteItemHandler"/>
                </map>
            </entry>
            <entry key="edit"> <!-- 'edit' portlet mode -->
                <map>
                    <entry key="prefs" value-ref="prefsHandler"/>
                    <entry key="resetPrefs" value-ref="resetPrefsHandler"/>
                </map>
            </entry>
        </map>
    </property>
</bean>
```

이 맵핑은 각각의 모드를 위한 디폴트와 전체 디폴트를 제공할수 있는 PortletModeHandlerMapping앞에서 연결될수 있다.

16.5.4. HandlerInterceptor 추가하기

Spring의 핸들러 맵핑 기법은 당신이 어떤 요청에 명시된 기능을 적용하길 원할때 굉장히 유용한 핸들러 인터셉터의 개념을 가진다. 다시 말해 Spring 포틀릿 MVC는 웹 MVC와 같은 방법으로 이러한 개념을 구현한다.

핸들러 맵핑내 위치한 인터셉터는 org.springframework.web.portlet 패키지의 HandlerInterceptor를 구현해야만 한다. 서블릿 버전처럼, 이 인터페이스는 세개의 메소드를 정의한다. 하나는 실제 핸들러가 수행되기 전에 호출되고(preHandle), 하나는 핸들러가 수행된 후에 호출될것이고(postHandle), 다른 하나는 요청이 종료된후에 호출된다(afterCompletion). 이 세개의 메소드는 모든 전- 그리고 후- 처리를 하기 위해 충분한 유연성을 제공해야만 한다.

preHandle 메소드는 boolean값을 반환한다. 당신은 수행체인의 처리를 중지하거나 지속하기 위해 이 메소드를 사용할수 있다. 이 메소드가 true를 반환할때, 핸들러 수행 체인은 지속될것이다. false를 반환할때, DispatcherPortlet은 인터셉터 자체가 요청을 다루고(이를테면, 적절한 view를 표시하는) 수행 체인내 다른 인터셉터와 실제 핸들러의 수행을 지속하지 않는다.

postHandle 메소드는 오직 RenderRequest에서 호출된다. preHandle 와 afterCompletion 메소드는 ActionRequest 와 RenderRequest 모두에서 호출된다. 하나의 요청 타입을 위한 메소드에서 로직을 수행할 필요가 있다면, 처리하기 전에 어떤 요청인지 체크해야만 한다.

16.5.5. HandlerInterceptorAdapter

서블릿 패키지처럼, 포틀릿 패키지는 HandlerInterceptorAdapter라고 불리는 구체적인 HandlerInterceptor 구현물을 가진다. 이 클래스는 모든 메소드의 빈 버전을 가져서 당신은 이 클래스로부터 상속할수 있고 당신이 필요할때 하나 또는 두개의 메소드를 구현할수 있다.

16.5.6. ParameterMappingInterceptor

포틀릿 패키지는 ParameterHandlerMapping 과 PortletModeParameterHandlerMapping으로 직접 사용될수 있는 ParameterMappingInterceptor 라는 이름의 구체적인 인터셉터를 가진다. 이 인터셉터는 ActionRequest에서 직후의 RenderRequest로 포워드되기 위한 맵핑을 제어하기 위해 사용되는 파라미터를 야기할것이다. 이것은 RenderRequest가 ActionRequest와 같은 핸들러로 맵핑되는 것을 도울것이다. 이것은 인터셉터의 preHandle 메소드에서 수행된다. 그래서 RenderRequest가 맵핑될 위치를 변경하기 위한 핸들러내 파라미터 값을 변경할수 있다.

이 인터셉터가 ActionResponse의 setRenderParameter를 호출하는 것은 이 인터셉터를 사용할때 당신의 핸들러내 sendRedirect 를 호출할수 없다는 것을 의미하는 것을 알라. 외부로 리다이렉트할 필요가 있다면, 당신은 맵핑 파라미터를 수동으로 포워드하거나 이것을 다루기 위해 다른 인터셉터를 작성할 필요가 있을것이다.

16.6. View와 View해석하기

이전에 언급된것처럼, Spring 포틀릿 MVC는 Spring 웹 MVC의 모든 view기술을 직접 재사용한다. 이것은 다양한 View 구현물 자체 뿐 아니라 ViewResolver 구현물도 포함한다. 좀더 많은 정보를 위해, Chapter 14, 통합 뷰 기술들 와 Section 13.5, “view와 view결정하기” 를 참조하라.

View 와 ViewResolver 구현물을 사용하는 몇가지 항목은 언급할 가치가 있다.

☒ 대부분의 포탈은 HTML조각이 되는 포틀릿을 표시하는 결과를 기대한다. JSP/JSTL, Velocity, FreeMarker, 그리고 XSLT 는 모두 공통적이다. 하지만 다른 문서 타입을 반환하는 view와는 달리 포틀릿 컨텍스트내에서는 같다.

☒ 포틀릿에서 HTTP 리다이렉트와 같은 것은 없다(ActionResponse의 sendRedirect(..)메소드는 포털내 존재하지 위해 사용될수 없다). 그래서 RedirectView와 ‘redirect:’ 접두사의 사용은 포틀릿 MVC에서 정확하게 작동하지 않을것이다.

☒ 포틀릿 MVC에서 ‘forward:’ 접두사의 사용은 가능하다. 웹 애플리케이션내 다른 자원에 접근하는 상대적인 URL을 사용할수 없고 절대적인 URL을 사용해야만 할 것이라는 것을 의미한다.

또한, JSP개발을 위해, 새로운 Spring Taglib와 새로운 Spring 폼 Taglib 모두 서블릿 view내 작동하는 것과 같은 방법으로 포틀릿 view에서 작동한다.

16.7. Multipart (파일 업로드) 지원

Spring 포틀릿 MVC는 웹 MVC가 하는것처럼 포틀릿 애플리케이션내 파일 업로드를 다루기 위한 내장된 multipart지원을 가진다. multipart지원을 위한 디자인은 org.springframework.web.portlet.multipart 패키지내 정의된 플러그인 성격의 PortletMultipartResolver 객체로 작동한다. Spring은 Commons FileUpload을

사용하기 위한 PortletMultipartResolver를 제공한다.

디폴트에 의해, Spring 포틀릿 MVC에 의해 수행되는 multipart핸들링은 없다. 몇몇 개발자는 multipart를 자체적으로 다루길 원할것이다. 당신은 웹 애플리케이션 컨텍스트에 multipart해석자를 추가하여 자체적으로 가능할것이다. DispatcherPortlet은 multipart를 포함하는지 보기 위해 각각의 요청을 조사할것이다. multipart가 발견되지 않는다면, 요청은 기대하는데로 계속될것이다. 어쨌든, 요청내 multipart가 발견된다면, 컨텍스트에 선언된 PortletMultipartResolver 가 사용될것이다. 그리고 난후, 요청내 multipart속성은 다른 속성처럼 처리될것이다.

16.7.1. PortletMultipartResolver 사용하기

다음의 예제는 CommonsPortletMultipartResolver를 사용하는 방법을 보여준다.

```
<bean id="portletMultipartResolver"
    class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver">

    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

물론, 당신은 multipart해석자가 작동하도록 하기 위해 classpath에 적절한 jar를 둘 필요가 있다. CommonsMultipartResolver의 경우, 당신은 commons-fileupload.jar 를 사용할 필요가 있다. Commons FileUpload는 적어도 1.1버전을 사용해야 JSR-168 포틀릿 애플리케이션을 지원한다.

당신은 multipart 요청을 다루기 위해 포틀릿 MVC를 셋팅하는 방법을 보았다. 이것을 실제 사용하는 방법을 보자. DispatcherPortlet가 multipart요청을 감지했을때, 당신의 컨텍스트내 선언된 해석자를 활성화하고 요청을 다룬다. 해석자가 그리고 나서 하는 것은 현재 ActionRequest를 multipart파일 업로드를 지원하는 MultipartActionRequest로 포장하는 것이다. MultipartActionRequest를 사용하여, 당신은 요청에 포함된 multipart에 대한 정보를 얻고 컨트롤러내 multipart파일 자체에 접근할수 있다.

당신은 ActionRequest의 일부처럼 multipart 파일 업로드을 받을뿐 아니라 RenderRequest 의 일부처럼 받을수도 있다.

16.7.2. 폼내 파일 업로드 다루기

PortletMultipartResolver가 작업을 마친후에, 요청은 다른것처럼 처리될것이다. 이것을 사용하기 위해, 당신은 업로드 필드를 가진 폼을 생성한다. 그리고 나서 Spring에 파일을 당신의 폼에 바인드하도록 한다. 실제로 사용자가 파일을 업로드 하기 위해, 우리는 (JSP/HTML)폼을생성해야만 한다.

```
<h1>Please upload a file</h1>
<form method="post" action="<portlet:actionURL/>" enctype="multipart/form-data">
        <input type="file" name="file"/>
        <input type="submit"/>
</form>
```

당신이 볼수 있는것처럼, 우리는 bean의 프라퍼티가 byte[]를 고정한후 “file” 이라는 이름의 필드를 생성한다. 게다가 multipart필드를 인코딩하는 방법을 브라우저에 알려주기 위해 필요한 인코딩 속성(enctype="multipart/form-data")을 추가했다(이것을 잊지말라..!!).

string이나 원시타입으로 자동 변환할수 없는 다른 프라퍼티처럼, 당신 객체에 바이너리 데이터를 두는것이 가능하도록 하기 위해, 당신은 PortletRequestDataBinder를 가진 사용자 정의 편집기를 등록해야만 한다. 여기엔 파일을 다루고 객체의 결과를 셋팅하기 위해 사용가능한 두개의 편집기가 있다. 파일을 문자열로

변환하는 능력을 가진 StringMultipartFileEditor와 파일을 byte배열로 변환하는 ByteArrayMultipartFileEditor가 있다. 이것은 CustomDateEditor가 하는것처럼 작동한다.

폼을 사용하여 파일을 업로드하기 위해, 해석자, bean을 처리할 컨트롤러를 위한 맵핑, 그리고 컨트롤러 자체를 선언하라.

```
<bean id="portletMultipartResolver"
      class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver"/>

<bean id="portletModeHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="fileUploadController"/>
        </map>
    </property>
</bean>

<bean id="fileUploadController" class="examples.FileUploadController">
    <property name="commandClass" value="examples.FileUploadBean"/>
    <property name="formView" value="fileuploadform"/>
    <property name="successView" value="confirmation"/>
</bean>
```

컨트롤러와 file 프라퍼티를 가지는 실제 클래스를 생성하라.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors)
        throws Exception {

        // 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
        }

        // do something with the file here
    }

    protected void initBinder(
            PortletRequest request, PortletRequestDataBinder binder)
        throws Exception {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert
    }
}

public class FileUploadBean {

    private byte[] file;

    public void setFile(byte[] file) {
        this.file = file;
    }
```

```
    public byte[] getFile() {
        return file;
    }
}
```

당신이 볼수 있는것처럼, FileUploadBean은 파일을 가지는 byte[]타입의 프라퍼티를 가진다. 컨트롤러는 해석자가 bean에 의해 명시된 프라퍼티를 찾는 multipart객체로 변환하는 방법을 알리기 위해 사용자 정의 편집기를 등록한다. 이 예제에서, 어떤것도 bean자체의 byte[] 프라퍼티로 하지 않는다. 하지만 실제로 당신은 원하는 것(데이터베이스내 저장하거나 누군가에게 메일을 보내거나, 기타등등)이 무엇이든 할수 있다.

파일이 (폼 지원)객체의 String타입의 프라퍼티에 직접 바운드하는 동일한 예제는 다음과 비슷할것이다.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        String file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }

    protected void initBinder(
        PortletRequest request, PortletRequestDataBinder binder) throws Exception {

        // to actually be able to convert Multipart instance to a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class,
            new StringMultipartFileEditor());
        // now Spring knows how to handle multipart objects and convert
    }
}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

물론, 마지막 예제는 일반적인 텍스트 파일(이미지 파일의 업로드의 경우 잘 작동하지 않을것이다.)의 업로드이다.

세번째(그리고 마지막) 옵션은 (폼 지원) 객체의 클래스에 선언된 MultipartFile 프라퍼티에 직접 바인드하는 것이다. 이 경우 수행될 타입 변환이 없기 때문에 어떠한 사용자 정의 프라퍼티 편집기를 등록할 필요가 없다.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        MultipartFile file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
}
```

16.8. 예외 다루기

웹 MVC처럼, 포틀릿 M/VC는 당신의 요청이 요청에 일치하는 핸들러에 의해 처리되는 동안 발생하는 기대되지 않은 예외를 쉽게 처리하기 위한 HandlerExceptionResolver를 제공한다. 포틀릿 MVC는 던져지는 예외의 클래스명을 가지고 view명에 이것을 맵핑하는 것을 가능하게 하는 구체적인 SimpleMappingExceptionResolver를 제공한다.

16.9. 포틀릿 애플리케이션 개발

Spring 포틀릿 MVC애플리케이션을 디플로이하는 처리는 JSR-168 포틀릿 애플리케이션을 디플로이하는것과 다르지 않다.

대개, 포털/포틀릿-컨테이너는 서블릿-컨테이너내 웹 애플리케이션으로 수행되고 다른 포틀릿은 서블릿-컨테이너내 다른 웹 애플리케이션으로 수행된다. 포틀릿 웹 애플리케이션에 대한 호출을 만드는 포틀릿-컨테이너 웹 애플리케이션을 위해, portlet.xml파일에 정의된 포틀릿 서비스에 접근하는 잘 알려진 서블릿에 대해 호출을 만들어야만 한다.

JSR-168스펙은 이 방법을 정확하게 명시하지 않았다. 그래서 각각의 포틀릿-컨테이너는 포틀릿 웹 애플리케이션 자체에 변경된 몇가지 “deployment process” 를 포함하고 포틀릿-컨테이너내 포틀릿을 등록하는 자체적인 기법을 가진다.

적어도, 포틀릿 웹 애플리케이션내 web.xml 파일은 포틀릿-컨테이너가 호출할 서블릿을 삽입하기 위해 변경되었다. 몇가지 경우 하나의 서블릿은 웹 애플리케이션내 모든 포틀릿을 서비스할것이다. 다른 경우 각각의 포틀릿을 위한 서블릿 인스턴스가 있을것이다.

몇가지 포틀릿-컨테이너는 라이브러리와(또는) 설정파일을 웹 애플리케이션으로 삽입할것이다. 포틀릿-컨테이너는 웹 애플리케이션에서 사용가능한 포틀릿 JSP 태그 라이브러리의 구현물을 만들어야만 한다.

마지막 라인은 목표 포털에 필요한 배치를 이해하고 다음의 제공하는 자동 배치 처리를 확인하라.

당신의 포틀릿을 배치했을때, web.xml 파일을 다시 보라. 몇가지 예전 포털은 ViewRendererServlet의 정의에 문제를 발생시키는 것으로 알려져있고 포틀릿을 표시하는데 문제가 있다.

Part IV. Integration

This final part of the reference documentation covers the Spring Framework's integration with a number of J2EE (and related) technologies.

Chapter 17, Spring을 사용한 원격(Remoting)및 웹서비스

Chapter 18, Enterprise Java Bean (EJB) 통합

Chapter 19, JMS

Chapter 20, JMX

Chapter 21, JCA CCI

Chapter 22, Spring 메일 추상 계층

Chapter 23, Spring을 사용한 스케줄링과 쓰레드 풀링

Chapter 24, 동적 언어 지원

Chapter 25, 어노테이션(annotation)과 소스 레벨 메타데이타 지원

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"/>
        <property name="service" ref="accountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
```

```
        <!-- defaults to 1099 -->
        <property name="registryPort" value="1199"/>
</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="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</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="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</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="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
        <property name="serviceUrl" value="http://remotehost:8080/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</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/httpclient을 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.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</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.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl" value="http://remotehost:8080/AccountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</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.org를 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"/>
        <property name="wsdlDocumentUrl" value="http://localhost:8080/account/services/accountService?WSDL"/>
        <property name="namespaceUri" value="http://localhost:8080/account/services/accountService"/>
        <property name="serviceName" value="AccountService"/>
        <property name="portName" value="AccountPort"/>
    </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="accountWebService"/>
</bean>
```

그리고 클라이언트 코드로 부터 우리는 이것이
<exception>RemoteException</exception>
을 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 ex) {
           // 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="echo">
    <property name="serviceInterface" value="org.codehaus.xfire.spring.Echo"/>
    <property name="serviceBuilder" ref="xfire.serviceBuilder"/>
    <!-- the XFire bean is wired up in the xfire.xml file you've linked in earlier -->
    <property name="xfire" ref="xfire"/>
  </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/Spring를 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. Enterprise Java Bean (EJB) 통합

18.1. 소개

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.

18.2. EJB에 접근하기

18.2.1. 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 컨테이너내 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.

18.2.2. 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 컨테이너밖에서 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"/>
  <property name="businessInterface" value="com.mycom.MyComponent"/>
</bean>

<bean id="myController" class="com.mycom.myController">
  <property name="myComponent" ref="myComponent"/>
</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 myComponent 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.

18.2.3. 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.

18.3. 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 also 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 컨테이너 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);
}
```

당신은 beanRefContext.xml라는 이름의 bean정의 파일을 생성할 필요가 있을 것이다. 이 파일은 EJB내 사용되는 모든 bean factory(대개 애플리케이션 컨텍스트의 형태로)를 정의한다. 많은 경우, 이 파일은 다음과 같은 하나의 bean정의를 포함할것이다(모든 비지니스 서비스 POJO를 위한 bean정의를 포함하는 businessApplicationContext.xml가 있는).

```
<beans>
    <bean id="businessBeanFactory" class="org.springframework.context.support.ClassPathXmlApplicationContext">
                                <constructor-arg value="businessApplicationContext.xml" />
                        </bean>
</beans>
```

위 예제에서, ServicesConstants.PRIMARY_CONTEXT_ID는 다음처럼 정의될 것이다.

```
public static final String ServicesConstants.PRIMARY_CONTEXT_ID = "businessBeanFactory";
```

그것들의 사용법에 대한 좀더 다양한 정보를 위해서 BeanFactoryLocator 와 ContextSingletonBeanFactoryLocatore를 위한 관련 JavaDoc를 보라.

Chapter 19. JMS

19.1. 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. JmsTemplate 클래스는 메시지 생산과 동기적인 메시지 수령을 위해 사용된다. 비동기적인 수령은 Java EE의 메시지-기반 bean스타일과 유사하다. Spring은 메시지-기반 POJO(MDPs)를 생성하기 위해 사용된 많은 수의 메시지 리스너 컨테이너를 제공한다.

> 도메인 단일화
>
> JMS스펙, 1.0.2와 1.1의 두가지 릴리즈가 있다.
>
> JMS 1.0.2 는 메시지 도메인의 두가지 타입인 point-to-point(Queues) 과 publish/subscribe (Topics)이 정의되었다. 1.0.2 API는 각각의 도메인을 위한 병렬 클래스 구조를 제공하여 이 두개의 메시지 도메인을 반영한다. 결과적으로, 클라이언트 애플리케이션이 JMS API를 사용하여 도메인 스펙이 된다. JMS 1.1은 두개의 도메인 간의 함수적인 차이점과 클라이언트 API 차이점을 최소화하는 도메인 단일화의 개념을 소개한다. 제거된 함수적인 차이점의 예제처럼, 당신이 JMS 1.1제공자를 사용한다면, 당신은 하나의 도메인으로부터 메시지를 소비하고 같은 Session을 사용하여 메시지를 생산할수 있다.
>
> JMS 1.1스펙은 2002년 4월 릴리즈되었고 2003년 11월 Java EE 1.4의 일부로 구체화되었다. 결과적으로, 대부분의 애플리케이션 서버는 JMS 1.0.2를 지원하는 것만 요구되어 사용중이다.

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. 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.

패키지org.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 구현물(JmsTransactionManager라는 이름의)을 포함한다. 이것은 트랜잭션적인 자원에서 Spring의 트랜잭션 관리 기법까지처럼 JMS의 통합을 허용한다.

19.2. Using Spring JMS

19.2.1. 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 SessionCallback은 JMS 세션을 가진 사용자를 제공하고 콜백 ProducerCallback 은 Session과 MessageProducer 쌍을 드러낸다.

JMS API는 send메소드의 두가지 타입을 나타낸다. 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.

19.2.2. Connection Factory

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 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.

19.2.3. 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 의존성 삽입을 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.

19.2.4. 메시지 리스너 컨테이너

EJB에서 JMS메시지의 가장 공통적인 사용중 하나는 메시지-기반 bean(MDB)을 다루는 것이다. Spring은 사용자를 EJB컨테이너에 묶지 않는 방법으로 메시지-기반 POJO(MDP)를 생성하는 해결법을 제공한다(Spring의 MDP지원을 상세히 다루기 위해 Section 19.4.2, “비동기적인 수령 - 메시지-기반 POJO” 부분을 보라.)

AbstractMessageListenerContainer의 하위클래스는 JMS메시지 큐로부터 메시지를 받고 큐에 삽입될 MDP를 다루기 위해 사용된다. AbstractMessageListenerContainer는 메시지 수령의 모든 쓰레드를 다루고 처리하기 위한 MDP로 배치한다. 메시지 리스너 컨테이너는 MDP와 메시지 제공자간에 매개수단이고 메시지를 받기 위한 등록, 트랜잭션내 들어가기, 자원 획득과 제거, 예외 처리와 같은 것들을 다룬다. 이것은 당신에게 애플리케이션 개발자가 메시지를 받는것과 관련있는 비지니스 로직을 작성하는 것과 프레임워크에 관련된 JMS내부구조를 위임하는 것을 가능하게 한다.

여기에는 각각의 특별한 기능을 가지는 Spring에 패키지된 AbstractMessageListenerContainer의 3개의 하위클래스가 있다.

19.2.4.1. SimpleMessageListenerContainer

이 메시지 리스너 컨테이너는 3개의 하위클래스에서 가장 간단하다. 이것은 시작시 고정된 수의

JMS세션을 간단히 생성하고 컨테이너의 수명을 통해 그것들을 사용한다. 이 하위클래스는 런타임 요구에 동적인 적응을 허용하지 않고 메시지의 트랜잭션 성격을 지니는 수령에 참가한다. 어쨌든, JMS제공자에서 가장 적은 요구사항을 가진다. 이 하위클래스는 오직 간단한 JMS API행위를 요구한다.

19.2.4.2. DefaultMessageListenerContainer

이 메시지 리스너 컨테이너는 가장 많은 경우에 사용되는 것이다. SimpleMessageListenerContainer를 가지고, 이 하위클래스는 런타임 요구에 동적 적응을 허용하지 않는다. 어쨌든 이 종류는 트랜잭션내 참가한다. 각각 받아진 메시지는 XA 트랜잭션으로 등록되고 XA 트랜잭션 문법의 장점을 가질수 있다. 이 하위클래스는 JMS 제공자의 적은 요구사항과 트랜잭션 참가를 포함하는 좋은 기능사이에서 좋은 균형을 가진다.

19.2.4.3. ServerSessionMessageListenerContainer

이 하위클래스는 3개중 가장 강력한 것이다. JMS세션의 동적 관리를 허용하는 JMS ServerSessionPool SPI에 영향을 끼친다. 이 구현물 또한 트랜잭션에 참가한다. 다양한 메시지 리스너 컨테이너의 사용은 강력한 런타임 튜닝을 가능하게 하지만 사용되는 JMS제공자의 좀더 큰 요구사항(ServerSessionPool SPI)이 둔다. 런타임 성능 튜닝이 필요없다면, DefaultMessageListenerContainer나 SimpleMessageListenerContainer를 찾는다.

19.2.5. 트랜잭션 관리

Spring은 하나의 JMS ConnectionFactory를 위한 트랜잭션을 관리하는 JmsTransactionManager을 제공한다. 이것은 JMS 애플리케이션이 Chapter 9, 트랜잭션 관리에서 언급된것처럼 Spring의 관리 트랜잭션 기능에 영향을 끼치는 것을 허용한다. JmsTransactionManager는 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 Session을 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 Session을 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.

19.3. 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.MessageCreator;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.JmsTemplate102;

public class JmsQueueSender {

  private JmsTemplate jmsTemplate;

  private Queue queue;

  public void setConnectionFactory(ConnectionFactory cf) {
    jt = new JmsTemplate102(cf, false);
  }

  public void setQueue(Queue q) {
    queue = q;
  }

  public void simpleSend() {
    this.jmsTemplate.send(this.queue, new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        return session.createTextMessage("hello queue world");
      }
    });
  }
}
```

이 예제는 제공되는 Session객체로부터 텍스트 메시지를 생성하기 위한 MessageCreator 콜백과 ConnectionFactory의 참조를 전달하여 생성되는 JmsTemplate, 메시지 도메인을 명시하는 boolean값을 사용한다. 0(zero) 인자 생성자와 connectionFactory / queue bean프라퍼티는 인스턴스를 생성(BeanFactory나 명백한 자바코드를 사용하여)하기 위해 제공되고 사용될수 있다. 대안으로, JMS설정을 위한 미리 빌드된 bean프라퍼티를 제공하는 Spring의 JmsGatewaySupport 편리한 base클래스로부터 끌어내보자.

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 creator) 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.

19.3.1. 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}
  }
}
```

19.3.2. 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.

19.4. 메시지 받기

19.4.1. 동기적인 수령

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.

JMS가 대개 비동기적인 처리에 관련되는 동안, 메시지를 동기적으로 받는것이 가능하다. 오버로드된 receive(..) 메소드는 이 기능을 제공한다. 동기적으로 받는동안, 메시지가 사용가능할때까지 쓰레드 블럭을 호출한다. 이것은 호출 쓰레드가 막연히 블럭될수 있기 때문에 위험한 작업이 될수 있다. receiveTimeout 프라퍼티는 메시지 받는것을 포기하기 전에 리시버(receiver)가 얼마나 오래 기다려야 하는지 명시한다.

19.4.2. 비동기적인 수령 - 메시지-기반 POJO

EJB에서 메시지-기반 Bean(MDB)와 유사한 형태로, 메시지-기반 POJO(MDP)는 JMS메시지를 위한 리시버로 작동한다. MDP에서 하나의 제약(MessageListenerAdapter 클래스에 대한 논의를 위해 아래를 보라)은 javax.jms.MessageListener 인터페이스를 구현해야만 한다는 것이다. POJO가 다중 쓰레드에서 메시지를 받는 경우를 알아보라. 이것은 구현물이 쓰레드에 안전하다는것을 확인하는 것이 중요하다.

아래는 MDP의 간단한 구현물이다.

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

  public void onMessage(Message message) {
    if (message instanceof TextMessage) {
      try {
        System.out.println(((TextMessage) message).getText());
      } catch (JMSException ex) {
        throw new RuntimeException(ex);
      }
    } else {
      throw new IllegalArgumentException("Message must be of type TextMessage");
    }
  }
}
```

MessageListener를 구현했을때가 메시지 리스너 컨테이너를 생성할 시간이다.

아래는 Spring에 포함된 메시지 리스너 컨테이너중 하나(이 경우 DefaultMessageListenerContainer)를 정의하고 설정하는 방법에 대한 예제이다.

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener" />

<!-- and this is the attendant message listener container -->
<bean id="listenerContainer"
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="concurrentConsumers" value="5"/>
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
```

```
</bean>
```

각각의 구현물에 의해 지원되는 기능에 대한 상세한 설명을 위해서 다양한 메시지 리스너 컨테이너의 Spring Javadoc를 참조하라.

19.4.3. SessionAwareMessageListener 인터페이스

SessionAwareMessageListener 인터페이스는 JMS MessageListener 인터페이스를 유사하게 축소한 Spring특유이 인터페이스이다. 하지만 받은 Message로부터 JMS Session에 접근하는 메시지 핸들링 메소드를 제공한다.

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSException;
}
```

받은 메시지에 응답하기 위한 MDP를 원한다면 이 인터페이스(표준 JMS MessageListener 인터페이스에 우선하는)를 구현하는 MDP를 가지도록 선택할수 있다(onMessage(Message, Session)에서 제공되는 Session을 사용하는). Spring을 가진 모든 메시지 리스너 컨테이너 구현물은 MessageListener 나 SessionAwareMessageListener 인터페이스를 구현하는 MDP를 지원한다. SessionAwareMessageListener를 구현하는 클래스는 caveat가 되고 인터페이스를 통해 Spring에 묶인다. 이것을 사용할지 말지에 대한 선택은 애플리케이션 개발자나 아키텍트와 같은 당신에게 남게된다.

SessionAwareMessageListener 인터페이스의 'onMessage(..)' 메소드가 JMSException가 던지는것을 알아두라. 표준 JMS MessageListener에 대해 대조적으로, SessionAwareMessageListener 인터페이스를 사용할때, 던져진 예외를 다루기 위한 클라이언트 코드의 책임이다.

19.4.4. MessageListenerAdapter

MessageListenerAdapter 클래스는 Spring의 비동기적인 메시지 지원내 마지막 컴포넌트이다. 간단히 말해, MDP처럼 대부분의 어떤 클래스를 나타내는것을 허용한다.

Note

JMS 1.0.2 API를 사용한다면, JMS 1.0.2 API를 위한것만을 제외하고 MessageListenerAdapter처럼 같은 기능과 값 추가를 제공하는 MessageListenerAdapter102 클래스를 사용하는 것을 원할것이다.

다음의 인터페이스 정의를 잘 보라. 비록 인터페이스가 MessageListener뿐 아니라 SessionAwareMessageListener 인터페이스를 확장하지 않는다면, MessageListenerAdapter 클래스의 사용을 통해 MDP로 사용될수 있다. 다양한 메시지 핸들링 메소드가 받고 다룰수 있는 다양한 Message타입의 내용에 따라 강력하게 타입화된다.

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
```

```
}
```

```
public class DefaultMessageDelegate implements MessageDelegate {
    // implementation elided for clarity...
}
```

특별히, 위 MessageDelegate 인터페이스의 구현물(위 DefaultMessageDelegate클래스)이 JMS 의존성을 전혀 가지지 않는 방법을 노트하라. 우리가 다음의 설정을 통해 MDP로 만들 POJO이다.

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the attendant message listener container... -->
<bean id="listenerContainer"
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="concurrentConsumers" value="5"/>
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
</bean>
```

아래는 JMS TextMessage 메시지를 받는것을 다룰수 있는 다른 MDP의 예제이다. 메시지 핸들링 메소드가 실제로 호출된 'receive'(MessageListenerAdapter내 메시지 핸들링 메소드의 이름이 'handleMessage'가 디폴트)이지만 설정가능한 방법을 알라. 'receive(..)' 메소드가 오직 JMS TextMessage 메시지에만 받고 응답하기 위해 강력하게 타입화된 방법또한 알라.

```
public interface TextMessageDelegate {

    void receive(TextMessage message);
}
```

```
public class DefaultTextMessageDelegate implements TextMessageDelegate {
    // implementation elided for clarity...
}
```

MessageListenerAdapter의 설정은 다음처럼 보일것이다.

```
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultTextMessageDelegate"/>
    </constructor-arg>
    <property name="defaultListenerMethod" value="receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="messageConverter">
        <null/>
    </property>
</bean>
```

위 'messageListener'가 TextMessage와 다른 타입의 JMS Message를 받는다면, IllegalStateException가 던져질것이다.

MessageListenerAdapter의 다른 기능은 핸들러 메소드가 void가 아닌 다른 값을 반환한다면 응답 Message를 자동으로 돌려주는 것이다.

다음의 인터페이스와 구현물을 보라.

```
public interface ResponsiveTextMessageDelegate {

    // notice the return type...
    String receive(TextMessage message);
}
```

```
public class DefaultResponsiveTextMessageDelegate implements ResponsiveTextMessageDelegate {
    // implementation elided for clarity...
}
```

만약 위 DefaultResponsiveTextMessageDelegate가 MessageListenerAdapter와 함께 결합되어 사용된다면, 'receive(..)'메소드의 수행으로부터 반환되는 어느 null이 아닌 값이 TextMessage로 변환될것이다. 결과 TextMessage는 원래 Message의 JMS Reply-To 프라퍼티로 정의되거나 디폴트 Destination이 MessageListenerAdapter에 셋팅된다면 Destination로 보내어질것이다. 만약 Destination이 발견되지 않는다면, InvalidDestinationException가 던져질것이다(이 예외는 삼켜지지 않을것이고 호출 스택을 전할것이다.).

19.4.5. 트랜잭션내 참여

트랜잭션내 참여는 오직 두개의 작은 변경만이 요구된다. 트랜잭션 관리자를 생성하고 트랜잭션내 참여할수 있는 하위클래스중 하나를 가지고 트랜잭션 관리자를 등록할 필요가 있을것이다(DefaultMessageListenerContainer 또는 ServerSessionMessageListenerContainer).

트랜잭션 관리자를 생성하기 위해, JmsTransactionManager의 인스턴스를 생성하고 XA트랜잭션 성능을 가지는 connection factory를 주길 원할것이다.

```
<bean id="transactionManager" class="org.springframework.jms.connection.JmsTransactionManager">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

그리고 나서 이전 컨테이너 설정에 다음을 추가할 필요가 있다. 컨테이너는 나머지를 다룰것이다.

```
<bean id="listenerContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="concurrentConsumers" value="5" />
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
  <property name="transactionManager" ref="transactionManager" />
</bean>
```

Chapter 20. JMX

20.1. 소개

Spring내 JMX지원은 당신에게 쉽게 기능을 제공하고 당신의 Spring애플리케이션을 JMX내부구조와 투명하게 통합한다.

> JMX?
>
> 이 장에서 JMX를 소개하지는 않는다. JMX를 사용하길 원하는 이유에 대한 동기를 언급하지 않을것이다. JMX가 처음이라면 이 장의 마지막에 있는 Section 20.9, "더 많은 자원" 부분을 체크해보라.

특별히 Spring의 JMX지원은 4개의 핵심 기능을 제공한다.

☒ JMX MBean처럼 어느 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 JMX인터페이스 와 클래스를 위한 애플리케이션 컴포넌트를 커플링하지 않는 작업을 위해 다지인되었다. 물론 당신 애플리케이션 클래스의 대부분을 위해 Spring JMX기능의 장점을 가져가기 위한 Spring이나 JMX를 인식할 필요는 없다.

20.2. JMX에 당신의 bean을 내보내기(Exporting)

Spring의 JMX 프레임워크내 핵심(core) 클래스는 MBeanExporter이다. 이 클래스는 당신의 Spring bean을 가져오고 그것들을 JMX MBeanServer에 등록하는 책임이 있다. 예를 들어, 다음의 클래스를 보라.

```
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();
    }
}
```

MBean의 속성과 작업(operation)처럼 이 bean의 프라퍼티와 메소드를 드러내기 위해 당신은 당신의 설정파일내 MBeanExporter 클래스의 인스턴스를 간단하게 설정하고 밑에서 보여지는것처럼 bean으로 전달한다.

```
<beans>

  <!-- this bean must not be lazily initialized if the exporting is to happen -->
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-init="false">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

위 설정조각의 적절한 bean정의는 exporter bean이다. beans 프라퍼티는 JMX MBeanServer에 내보내져야하는 당신의 bean이 어떠한 것인지 MBeanExporter에 알리기 위해 사용된다. 디폴트 설정내에서, beans Map의 각각의 항목 key는 관련 항목값에 의해 참조되는 bean을 위한 ObjectName처럼 사용된다. 이 행위는 Section 20.5, “당신의 bean을 위한 ObjectName 제어하기” 부분에서 언급된것처럼 변경될수 있다.

testBean bean를 설정하는것은 ObjectNamebean:name=testBean1 하위의 MBean처럼 드러난다. 디폴트에 의해, bean의 모든 public성격의 프라퍼티는 속성과 모든 public성격의 메소드(Object 클래스로 부터 상속된)가 작업(operation)처럼 드러나는 만큼 드러난다.

20.2.1. MBeanServer 생성하기

위 설정은 애플리케이션이 이미 실행중인 하나의(그리고 오직 하나의) MBeanServer를 가진 환경에서 실행중이라는것을 가정한다. 이 경우 Spring은 실행중인 MBeanServer를 할당하고 그 서버와 함께 당신의 bean을 등록할것이다. 이 행위는 당신의 애플리케이션이 자신만의 MBeanServer를 가진 톰캣이나 IBM웹스피어와 같은 컨테이너내부에서 실행중일때 유용하다.

어쨌든, 이 접근법은 독립형(standalone) 환경이나 MBeanServer를 제공하지 않는 컨테이너내부에서 실행될때는 필요없다. 이것을 할당하기 위해 당신은 당신의 설정을 위한 org.springframework.jmx.support.MBeanServerFactoryBean의 인스턴스를 추가하여 선언적으로 MBeanServer인스턴스를 생성할수 있다. 당신은 또한 MBeanServer가 MBeanExporter의 server 프라퍼티의 값을

MBeanServerFactoryBean에 의해 반환되는 MBeanServer값으로 셋팅하여 사용하는지 확인할수 있다.

```
<beans>

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"/>

  <!--
    this bean needs to be eagerly pre-instantiated in order for the exporting to occur;
    this means that it must not be marked as lazily initialized
  -->
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="server" ref="mbeanServer"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

여기, MBeanServer의 인스턴스는 MBeanServerFactoryBean에 의해 생성되고 서버 프라퍼티를 통해 MBeanExporter에 제공된다. 당신이 당신 자신의 MBeanServer 인스턴스를 제공할때 MBeanExporter는 수행중인 MBeanServer에 할당하기 위해 시도하지 않을것이고 제공되는 MBeanServer 인스턴스를 사용할것이다. 이 작업을 정확하게 하기 위해 당신은 classpath에 JMX구현물의 위치시켜야만 한다.

20.2.2. 존재하는 MBeanServer를 재사용하기

명시된 서버가 없다면, MBeanExporter는 구동중인 MBeanServer를 자동으로 감지하도록 시도한다. 이것은 오직 하나의 MBeanServer 인스턴스가 사용된 환경에서 작동한다. 어쨌든 다중 인스턴스가 존재할때, exporter는 나쁜 서버를 잡을것이다. 이러한 경우, 사용된 인스턴스를 표시하기 위해 MBeanServer agentId를 사용할것이다.

```
<beans>
   <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
     <!-- indicate to first look for a server -->
     <property name="locateExistingServerIfPossible" value="true"/>
     <!-- search the MbeanServer instance with the given agentId -->
     <property name="agentId" value="<MBeanServer instance agentId>"/>
   </bean>

   <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
     <property name="server" ref="mbeanServer"/>
   ...
   </bean>
</beans>
```

존재하는 MBeanServer이 lookup메소드를 통해 가져오는 동적/알려지지 않은 agentId를 가지는 플랫폼/경우를 위해 factory-method를 사용해야만 한다:

```
<beans>
   <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
     <property name="server">
       <!-- Custom MBeanServerLocator -->
```

```
      <bean class="platform.package.MBeanServerLocator" factory-method="locateMBeanServer"/>
    </property>
  ...
  </bean>
</beans>
```

20.2.3. 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 프록시의 첫번째 호출이 발생할때까지 컨테이너로 부터 bean을 얻는것을 미룰것이다.

20.2.4. MBean의 자동 등록

MBeanExporter와 이미 유효한 MBeans으로 부터 보내어진 bean은 Spring으로 부터 더이상의 조정(intervention) 없이 MBeanServer처럼 등록된다. MBean은 autodetect 프라퍼티를 true로 셋팅하여 MBeanExporter에 의해 자동적으로 감지될수 있다.

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="autodetect" value="true"/>
</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 자동으로 등록될것이다. 디폴트에 의하면 JMX등록을 위해 자동감지된 bean은 ObjectName으로 사용된 bean이름을 가진다. 이 행위는 Section 20.5, “당신의 bean을 위한 ObjectName 제어하기” 부분에서 상세화된것처럼 오버라이드될수 있다.

20.3. 등록 행위 제어하기

Spring MBeanExporter가 ObjectName ‘bean:name=testBean1’를 사용하여 MBeanServer로 MBean을 등록하는 시나리오를 생각해보자. 만약 MBean 인스턴스가 같은 ObjectName 아래에서 이미 등록되었다면, 디폴트 행위는 실패한다(그리고 InstanceAlreadyExistsException를 던진다.).

MBean이 MBeanServer와 등록될때 발생하는 것의 행위를 제어하는 것은 가능하다. Spring의 JMX지원은 등록절차가 MBean이 같은 ObjectName아래에서 이미 등록된것을 찾았을때 등록행위를 제어하기 위한 3가지의 다른 등록행위를 허용한다. 이 등록행위는 아래의 테이블에서 나열된다.

Table 20.1. 등록 행위

등록 행위	설명
REGISTRATION_FAIL_ON_EXISTING	이것은 디폴트 등록행위이다. MBean인스턴스가 같은 ObjectName아래에서 이미 등록되었다면, 등록된 MBean은 등록되지 않을것이고 InstanceAlreadyExistsException가 던져질것이다.

등록 행위	설명
	존재하고 있는 MBean은 영향을 받지 않는다.
REGISTRATION_IGNORE_EXISTING	MBean인스턴스가 같은 ObjectName아래에서 이미 등록되었다면, 등록된 MBean은 등록되지 않을것이다. 존재하는 MBean은 영향을 받지 않는다. 그리고 던져지는 Exception은 없다. 이것은 다중 애플리케이션이 공유 MBeanServer내 공통 MBean을 공유하길 원하는 셋팅에 유용하다.
REGISTRATION_REPLACE_EXISTING	MBean인스턴스가 같은 ObjectName아래 이미 등록되었다면, 이미 등록된 존재하는 MBean은 등록되지 않을것이고 새로운 MBean은 대신 등록될것이다.(새로운 MBean은 이전 인스턴스를 효과적으로 대체한다.)

위 값들(REGISTRATION_FAIL_ON_EXISTING, REGISTRATION_IGNORE_EXISTING, 그리고 REGISTRATION_REPLACE_EXISTING)은 MBeanRegistrationSupport클래스의 상수처럼 정의된다(MBeanExporter클래스는 이것의 수퍼클래스에서 나온다.). 만약 당신이 디폴트 등록 행위를 변경하길 원한다면, 당신은 MBeanExporter정의의 registrationBehaviorName프라퍼티의 값을 이러한 값중 하나로 셋팅할 필요가 있다.

아래의 예제는 디폴트 등록 행위로부터 REGISTRATION_REPLACE_EXISTING 행위로 변경하는 효과를 주는 방법을 보여준다.

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="registrationBehaviorName" value="REGISTRATION_REPLACE_EXISTING"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

20.4. 당신 bean의 관리 인터페이스를 제어하기

이전의 예제에서 당신 bean의 관리 인터페이스를 조금(public 프라퍼티 모두와 JMX속성과 개별 작업처럼 드러나는 각각의 bean) 제어한다. 내보내어진 bean의 프라퍼티와 메소드를 잘 제어하는 것은 JMX속성과 작업처럼 실질적으로 드러난다. Spring JMX는 당신 bean의 관리 인터페이스를 제어하기 위한 포괄적이고 확장가능한 기법을 제공한다.

20.4.1. MBeanInfoAssembler 인터페이스

MBeanExporter는 드러난 각각의 bean의 관리 인터페이스를 정의하는 책임을 지닌 org.springframework.jmx.export.assembler.MBeanInfoAssembler 인터페이스의 구현물을 위임한다. 디폴트 구현물인 org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler 는 (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.

20.4.2. 소스레벨 메타데이타(metadata) 사용하기

MetadataMBeanInfoAssembler를 사용하여 당신은 소스레벨 메타데이타를 사용하는 당신의 bean을 위한 관리 인터페이스를 정의할수 있다. 메타데이타 읽기는 org.springframework.jmx.export.metadata.JmxAttributeSource 인터페이스에 의해 캡슐화된다. Spring JMX는 Commons Attributes를 위한 org.springframework.jmx.export.metadata.AttributesJmxAttributeSource 인터페이스와 JDK 5.0 annotations을 위한 org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource 인터페이스의 두가지 구현물을 위한 지원을 제공한다. MetadataMBeanInfoAssembler는 정확하게 기능을 위한 JmxAttributeSource의 구현물이 설정되어야만 한다. 이 예제를 위해 우리는 Commons Attributes 메타데이타 접근법을 사용할것이다.

JMX로 내보내기 위한 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 작업(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 20.4.4, “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)메소드가 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.

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>
```

```
  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource">
    <property name="attributes">
      <bean class="org.springframework.metadata.commons.CommonsAttributes"/>
    </property>
  </bean>

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

</beans>
```

여기서 당신은 MetadataMBeanInfoAssembler bean이 AttributesJmxAttributeSource 클래스의 인스턴스로 설정되고 assembler프라퍼티를 통해 MBeanExporter로 전달되는것을 볼수 있다. 이것은 당신의 Spring-노출 MBean을 위한 데타데이타-기반 관리 인터페이스의 장점을 가져오기 위해 요구되는 모든것이다.

20.4.3. JDK 5.0 Annotations 사용하기

관리 인터페이스 정의를 위한 JDK 5.0 annotation의 사용을 가능하게 하기 위해 Spring은 그것들을 읽기 위해 MBeanInfoAssembler을 허용하는 Commons Attribute속성 클래스와 JmxAttributeSource전략 인터페이스의 구현물인 AnnotationsJmxAttributeSource를 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")
  @ManagedOperationParameters({
    @ManagedOperationParameter(name = "x", description = "The first number"),
    @ManagedOperationParameter(name = "y", description = "The second number")})
  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.

위 주석처리된(annotated) 클래스를 위한 부수적인 XML설정은 아래에서 볼수 있다.

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"
          class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

    <!-- will create management interface using annotation metadata -->
    <bean id="assembler"
          class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <!-- will pick up ObjectName from annotation -->
    <bean id="namingStrategy"
          class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

20.4.4. 소스레벨 메타데이타 타입들

다음의 소스레벨 메타데이타 타입들은 Spring JMX내에서 사용되기 위해 사용가능하다.

Table 20.2. 소스레벨 메타데이타 타입

목적	Commons Attributes 속성	JDK 5.0 Annotation	Attribute / Annotation 타입
Class의 모든 인스턴스를 JMX관리 자원처럼 표시	ManagedResource	@ManagedResource	Class
메소드를 JMX작업처럼 표시하기	ManagedOperation	@ManagedOperation	Method
JMX속성의 반을 getter나 setter로 표시하기	ManagedAttribute	@ManagedAttribute	Method (오직 getters 와 setters)
작업 파라미터를 위한 상세설명 정의하기	ManagedOperationParameter	@ManagedOperationParameter 와 @ManagedOperationParameters	Method

다음의 설정 파라미터는 소스레벨 메타데이타 타입들의 사용을 위해 사용가능하다.

Table 20.3. 소스레벨 메타데이타 파라미터

파라미터	설명	적용
objectName	관리되는 자원의 ObjectName를 결정하기 위한 MetadataNamingStrategy에 의해 사용	ManagedResource
description	자원, 속성 또는 작업(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	ManagedResource

파라미터	설명	적용
	셋팅하기	
name	작업(operation) 파라미터의 표시명(display name)을 셋팅하기	ManagedOperationParameter
index	작업(operation) 파라미터의 인덱스 셋팅하기	ManagedOperationParameter

20.4.5. AutodetectCapableMBeanInfoAssembler 인터페이스

좀더 간단한 설정을 위해, Spring은 MBean자원의 자동감지를 위한 지원을 추가하는 MBeanInfoAssembler인터페이스를 확장하는 AutodetectCapableMBeanInfoAssembler인터페이스를 소개한다. 만약 당신이 AutodetectCapableMBeanInfoAssembler의 인스턴스를 사용해서 MBeanExporter을 설정한다면 이것은 JMX에 드러내기 위한 bean의 포함(inclusion)에 '결정(vote)'하는것을 허용한다.

특히, 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.

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <!-- notice how no 'beans' are explicitly configured here -->
    <property name="autodetect" value="true"/>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <!-- (for Commons Attributes-based metadata) -->
  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource">
    <property name="attributes">
      <bean class="org.springframework.metadata.commons.CommonsAttributes"/>
    </property>
  </bean>

  <!-- (for Java5+ annotations-based metadata) -->
  <!--
  <bean id="attributeSource"
        class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>
  -->

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource"/>
  </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 20.5, “당신의 bean을 위한 ObjectName 제어하기” 에서 정의되는것처럼 ObjectName 생성을 위한 디폴트 행위를 변경하여 이 문제를 해결할수 있다.

20.4.6. 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 작업(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" value-ref="testBean"/>
      </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"/>
    <property name="age" value="100"/>
  </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 key가 bean이름이고 각각의 항목의 값은 bean사용을 위한 인터페이스 이름의 콤마로 분리된 리스트인 interfaceMappings 을 통한 프라퍼티를 통해 Properties에 InterfaceBasedMBeanInfoAssembler을 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.

20.4.7. MethodNameBasedMBeanInfoAssembler 사용하기

MethodNameBasedMBeanInfoAssembler는 당신에게 속성과 작업(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" value-ref="testBean"/>
      </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 JMX작업(operation)처럼 드러나고 getName(), setName(String) 그리고 getAge() 메소드가 선호되는 JMX속성의 절반처럼 드러날것을 볼수 있다. 위 코드에서 메소드 맵핑은 JMX에 드러나는 bean에 적용한다. bean대 bean이라는 기본이 드러나는 메소드를 제어하는 것은 메소드명의 리스트에 bean이름을 맵핑하기 위한 MethodNameMBeanInfoAssembler의 methodMappings프라퍼티를 사용한다.

20.5. 당신의 bean을 위한 ObjectName 제어하기

MBeanExporter는 이것이 등록하는 각각의 bean을 위한 ObjectName를 얻기 위해 ObjectNamingStrategy의 구현물로 위임한다. 디폴트 구현물인 KeyNamingStrategy는 디폴트에 의해 ObjectName처럼 beans Map의 key를 사용할것이다. 추가적으로 KeyNamingStrategy는 ObjectName을 분석하기 위한 Properties 파일내 항목에 beans Map의 key를 맵핑할수 있다. KeyNamingStrategy에 추가적으로, Spring은 두가지 추가적인 ObjectNamingStrategy구현물(bean의 JVM확인에 기반하는 ObjectName을 빌드하는 IdentityNamingStrategy와 ObjectName를 얻기 위해 소스레벨 메타데이타를 사용하는 MetadataNamingStrategy)을 제공한다.

20.5.1. 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.

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </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 병합된 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 key명은 ObjectName처럼 사용된다.

20.5.2. MetadataNamingStrategy 사용하기

MetadataNamingStrategy 는 ObjectName을 생성하기 위한 각각의 bean의 ManagedResource속성의 objectName프라퍼티를 사용한다. 밑의 코드는 MetadataNamingStrategy를 위한 설정을 보여준다.

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
```

```
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

  <bean id="attributeSource"
      class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource"/>

</beans>
```

20.6. JSR-160 연결자(Connectors)

원격 접속을 위해 Spring JMX 모듈은 서버측과 클라이언트측 연결자를 생성하기 위한 org.springframework.jmx.support패키지내 두가지의 FactoryBean 구현물을 제공한다.

20.6.1. 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 160 스펙에 의해 선택적으로 표시된다는것을 알라. 현재 인기있는 오픈소스 구현물인 MX4J와 JMXMP를 지원하지 않는 자바 5.0에 의해 제공되는 것이다.

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"/>
  <property name="serviceUrl"
            value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector"/>
</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"/>
  <property name="serviceUrl"
              value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>
  <property name="threaded" value="true"/>
  <property name="daemon" value="true"/>
```

```
  <property name="environment">
    <map>
      <entry key="someKey" value="someValue"/>
    </map>
  </property>
</bean>
```

RMI기반 connector를 사용할때, 당신은 명명등록을 위해 시작하는 서비스를 룩업할 필요가 있다(스스로이거나 rmiregistry). 만약 당신이 RMI를 통해 당신을 위한 원격 서비스를 내보내기 위해 Spring을 사용한다면, Spring은 RMI registry를 생성할것이다. 그렇지 않다면, 당신은 다음의 설정조각을 사용하여 registry를 쉽게 시작할수 있다.

```
<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
  <property name="port" value="1099"/>
</bean>
```

20.6.2. 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"/>
</bean>
```

20.6.3. 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 JMX구현물(MX4J와 같은)을 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"/>
  <property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>
</bean>
```

위 예제의 경우, MX4J 3.0.0이 사용되었다.좀더 다양한 정보를 위해서는 MX4J문서를 보라.

20.7. 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"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

여기서 당신은 프록시가 ObjectName(bean:name=testBean)하위에 등록된 MBean을 위해 생성되는것을 볼수 있다. 프록시가 구현할 인터페이스의 세트는 proxyInterfaces 프라퍼티에 의해 제어되고 MBean의 작업(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"/>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
  <property name="objectName" value="bean:name=testBean"/>
  <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
  <property name="server" ref="clientConnector"/>
</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.

20.8. 통지

Spring의 JMX는 JMX통지를 위한 포괄적인 지원을 포함한다.

20.8.1. 통지를 위한 리스너 등록하기

Spring의 JMX지원은 많은 MBeans을 가지는 많은 수의 NotificationListeners를 등록하는 것을 쉽게 만든다(이것은 Spring의 MBeanExporter에 의해 나오는 MBeans와 몇몇 다른 기법을 통해 등록된 MBeans를 포함한다). 예제는 NotificationListeners의 등록에 영향을 주는 간단한 방법을 잘 보여줄것이다. 각각 그리고 항상 대상 MBean변경의 속성을 알리고자(Notification를 통해) 하는 시나리오를 생각해보자.

```
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
                implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notification) {
        return AttributeChangeNotification.class.isAssignableFrom(notification.getClass());
    }
```

```
}
```

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListenerMappings">
        <map>
            <entry key="bean:name=testBean1">
                <bean class="com.example.ConsoleLoggingNotificationListener"/>
            </entry>
        </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

위 설정에서, 항상 JMX Notification은 대상 MBean(bean:name=testBean1)으로부터 퍼트려진다. notificationListenerMappings 프라퍼티를 통해 리스너로 등록된 ConsoleLoggingNotificationListener bean은 통지될것이다. ConsoleLoggingNotificationListener bean은 Notification에 적절하게 응답하는 액션이 무엇이든 가질수 있다.

둘러싸는 MBeanExporter가 보내는 모든 bean을 위한 하나의 NotificationListener 인스턴스를 등록하길 원한다면, notificationListenerMappings 프라퍼티 맵내 항목을 위한 key로 특별한 와일드카드 '*'를 사용할수 있다.

```
    <property name="notificationListenerMappings">
        <map>
            <entry key="*">
                <bean class="com.example.ConsoleLoggingNotificationListener"/>
            </entry>
        </map>
    </property>
```

반대로 할 필요가 있다면(이를테면, MBean에 대해 많은 수의 구별되는 리스너를 등록한다면), 대신 notificationListeners 리스트 프라퍼티를 사용해야만 한다(그리고 notificationListenerMappings 프라퍼티 보다는 오히려). 하나의 MBean을 위한 NotificationListener를 설정하는 대신, NotificationListenerBean 인스턴스를 설정한다. NotificationListenerBean은 MBeanServer에 대해 등록되기 위한 NotificationListener 와 ObjectName(또는 ObjectNames)를 캡슐화한다. NotificationListenerBean는 NotificationFilter와 고급 JMX통지 시나리오에서 사용될수 있는 임의의 handback객체 같은 다른 많은 프라퍼티를 캡슐화한다.

NotificationListenerBean 인스턴스를 사용할때 설정은 이전에 존재하는 것과 무턱대고 다르지 않다.

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
```

```
      </map>
    </property>
    <property name="notificationListeners">
        <list>
            <bean class="org.springframework.jmx.export.NotificationListenerBean">
                <constructor-arg>
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </constructor-arg>
                <property name="mappedObjectNames">
                    <list>
                        <bean class="javax.management.ObjectName">
                            <constructor-arg value="bean:name=testBean1"/>
                        </bean>
                    </list>
                </property>
            </bean>
        </list>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

위 예제는 첫번째 통지 예제와 동등하다. Notification이 발생할때마다 handback객체가 주어지길 원하고 추가적으로 우리는 NotificationFilter를 제공하여 관계없는 Notifications을 걸러내길 원한다고 가정해보자. (handback객체가 무엇인지, 그리고 NotificationFilter가 무엇인지에 대해 완전한 정보를 원한다면, 'JMX 통지 모델'이라는 JMX스펙(1.2) 부분을 보라).

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean1"/>
        <entry key="bean:name=testBean2" value-ref="testBean2"/>
      </map>
    </property>
    <property name="notificationListeners">
        <list>
            <bean class="org.springframework.jmx.export.NotificationListenerBean">
                <constructor-arg ref="customerNotificationListener"/>
                <property name="mappedObjectNames">
                    <list>
                        <!-- let's handle notifications from two distinct MBeans -->
                        <bean class="javax.management.ObjectName">
                            <constructor-arg value="bean:name=testBean1"/>
                        </bean>
                        <bean class="javax.management.ObjectName">
                            <constructor-arg value="bean:name=testBean2"/>
                        </bean>
                    </list>
                </property>
                <property name="handback">
                    <bean class="java.lang.String">
                        <constructor-arg value="This could be anything..."/>
                    </bean>
                </property>
                <property name="notificationFilter" ref="customerNotificationListener"/>
            </bean>
        </list>
```

```
    </property>
  </bean>

  <!-- implements both the 'NotificationListener' and 'NotificationFilter' interfaces -->
  <bean id="customerNotificationListener" class="com.example.ConsoleLoggingNotificationListener"/>

  <bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="ANOTHER TEST"/>
    <property name="age" value="200"/>
  </bean>

</beans>
```

20.8.2. 통지 발행하기(Publishing)

Spring은 Notifications을 받는 것을 등록할뿐 아니라 Notifications을 발행하는 지원을 제공한다.

Note

이 부분은 MBeanExporter를 통해 MBean으로 나타나는 Spring관리 bean에만 관계된다. 존재하고 사용자 정의 MBean은 통지 발행을 위한 표준 JMX API를 사용할것이다.

Spring의 JMX 통지 발행지원내 핵심 인터페이스는 NotificationPublisher(org.springframework.jmx.export.notification 패키지에서 정의되는)인터페이스이다. MBeanExporter 인스턴스를 통해 MBean으로 나타날 bean은 NotificationPublisher 인스턴스에 접근하기 위한 관련 NotificationPublisherAware 인터페이스를 구현할수 있다. NotificationPublisherAware 인터페이스는 Notifications를 발행하기 위해 사용할수 있는 bean을 말하는 간단한 setter메소드를 통해 bean을 구현하는 NotificationPublisher의 인스턴스를 간단히 제공한다.

NotificationPublisher클래스를 위한 Javadoc내 상태처럼, NotificationPublisher 기법을 통해 이벤트를 발행하는 관리 bean은 어느 통지 리스너의 상태 관리를 책임지지 않는다. Spring의 JMX지원은 모든 JMX구조적인 이슈를 다룰것이다. 애플리케이션 개발자처럼 할 필요가 있는 모든것은 NotificationPublisherAware인터페이스를 구현하고 제공되는 NotificationPublisher 인스턴스를 사용하여 이벤트를 발행하는것을 시작한다. NotificationPublisher는 관리 bean이 MBeanServer로 등록된 후에 셋팅될것이다.

NotificationPublisher 인스턴스를 사용하는 것은 매우 일관적이다. JMX Notification 인스턴스(또는 적절한 Notification 하위클래스의 인스턴스)를 간단히 생성한다. 이것은 발행되기 위한 이벤트에 적절한 데이터로 통지하는 것을 활성화한다. 그리고 Notification를 전달하는 NotificationPublisher인스턴스에서 sendNotification(Notification)를 호출한다.

간단한 예제를 보자. 이 시나리오에서, JmxTestBean의 인스턴스는 add(int, int)작업이 호출될때마다 NotificationEvent를 발행할것이다.

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.jmx.export.notification.NotificationPublisher;
import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {
```

```
    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        this.publisher.sendNotification(new Notification("add", this, 0));
        return answer;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

    public void setNotificationPublisher(NotificationPublisher notificationPublisher) {
        this.publisher = notificationPublisher;
    }
}
```

NotificationPublisher 인터페이스와 작업하는 부수적인 장치는 Spring의 JMX지원의 좋은 기능중 하나이다. 이것은 Spring과 JMX모두를 위해 당신의 클래스를 커플링의 price태그가 딸려나온다. 여기 advice는 실용적이다. 만약 당신이 NotificationPublisher에 의해 제공되는 기능이 필요하고 Spring과 JMX모두를 커플링하는 것을 받아들일수 있다면, 그렇게 하라.

20.9. 더 많은 자원

이 부분은 JMX에 대한 더 많은 자원에 대한 링크를 포함한다.

☒ Sun의 JMX 홈페이지

☒ JMX 스펙 (JSR-000003)

☒ JMX Remote API 스펙 (JSR-000160)

☒ MX4J 홈페이지 (다양한 JMX스펙의 오픈소스 구현물)

☒ JMX로 시작하기 - Sun의 소개글

Chapter 21. JCA CCI

21.1. 소개

J2EE는 EIS(JCA(자바 연결자 아키텍쳐-Java Connector Architecture)) 에 대한 표준적인 접근을 위한 스펙을 제공한다. 이 스펙은 여러개의 다른 파트로 나뉜다.

☒ SPI (서비스 제공자 인터페이스-Service provider interfaces) 는 연결자 제공자를 구현해야만 한다. 이러한 인터페이스는 J2EE애플리케이션 서버에 배치될수 있는 자원 어댑터를 구성한다. 이러한 상황에서, 서버는 connection pooling, 트랜잭션 과 보안(관리모드)을 다룬다. 애플리케이션 서버는 클라이언트 애플리케이션 외부에 유지되는 설정을 관리하는 책임을 진다. 연결자는 애플리케이션 서버가 없이도 잘 사용될수 있다. 이러한 경우, 애플리케이션은 직접(비-관리모드) 이것을 설정해야만 한다.

☒ CCI (공통 클라이언트 인터페이스-Common Client Interface) 는 애플리케이션이 연결자와 상호작동하고 EIS와 통신하기 위해 사용할수 있다. local트랜잭션구분을 위한 API도 마찬가지로 제공된다.

Spring CCI 지원의 목적은 전형적인 Spring스타일로 Spring의 일반적인 자원과 트랜잭션 관리 기능에 영향력이 미치는 CCI연결자에 접근하는 클래스를 제공하는 것이다.

연결자의 클라이언트측은 언제나 CCI를 사용하는것은 아니다. 몇몇 연결자는 그들 자신만의 API를 가지고, J2EE컨테이너의 시스템 계약(connection pooling, 전역 트랜잭션및 보안)을 사용하기 위한 JCA자원 어댑터만을 제공한다. Spring은 이러한 연결자에 종속적인 API를 위한 특별한 지원은 하지 않는다.

21.2. CCI 설정하기

21.2.1. 연결자 설정

JCA CCI를 사용하기 위한 기본 자원은 ConnectionFactory 인터페이스이다. 사용되는 연결자는 이 인터페이스의 구현물을 제공해야만 한다.

당신의 연결자를 사용하기 위해, 당신은 애플리케이션 서버로 이것을 배치하고 서버의 JNDI환경(관리모드)으로부터 ConnectionFactory를 가져올수 있다. 연결자는 RAR파일(자원 어댑터 압축파일)처럼 패키징되어야만 하고 배치속성을 언급하기 위한 ra.xml 파일을 포함해야만 한다. 자원의 실질적인 이름은 당신이 배치할때 명시된다. Spring에서 이것에 접근하기 위해서, JNDI명으로 factory를 가져오기 위한 Spring의 JndiObjectFactoryBean를 간단히 사용하라.

연결자를 사용하기 위한 다른 방법은 이것을 배치하고 설정하기 위한 애플리케이션 서버를 사용하지 않고 당신의 애플리케이션에 이것을 내장하는 것이다(비-관리모드). Spring은 제공되는 FactoryBean (LocalConnectionFactoryBean)을 통해 bean처럼 연결자를 설정하는 것이 가능하다. 이 방법으로, 당신은 classpath내 연결자 라이브러리를 둘 필요가 있다(RAR파일과 ra.xml 설명자를 둘 필요가 없다). 필요하다면 라이브러리는 연결자의 RAR파일로부터 추출해야만 한다.

ConnectionFactory 인스턴스로 접근했다면, 당신은 이것을 컴포넌트로 삽입할수 있다. 이러한 컴포넌트는 명백한 CCI API에 대해 코딩되거나 CCI접근을 위한 Spring지원 클래스(이를테면, CciTemplate)에 영향을

끼칠수 있다.

비-관리 모드에서 연결자를 사용할때, 당신은 전역 트랜잭션을 사용할수 없다. 자원이 현재 쓰레드의 현재 전역 트랜잭션에 결코 리스트화되거나 리스트에사 빠지지 않을것이다. 자원은 단순히 수행중인 전역 J2EE트랜잭션을 인지하지 않는다.

21.2.2. Spring내 ConnectionFactory 설정

EIS를 위한 connection을 만들기 위해, 만약 당신이 관리모드에 있다면 당신은 애플리케이션 서버로부터 ConnectionFactory를 얻거나 비-관리모드에 있다면 Spring으로부터 직접 ConnectionFactory를 얻을 필요가 있다.

관리 모드에서, 당신은 JNDI로부터 ConnectionFactory에 접근한다. 이것의 프라퍼티는 애플리케이션 서버에 설정될것이다.

```
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>
```

비-관리모드에서, 당신은 Spring설정내에서 사용하길 원하는 ConnectionFactory를 JavaBean처럼 설정해야만 한다. 당신 연결자의 ManagedConnectionFactory구현물로 전달하는 이러한 셋업 스타일을 제공하는 LocalConnectionFactoryBean 클래스는 애플리케이션-레벨의 CCI ConnectionFactory를 나타낸다.

```
<bean id="eciManagedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="tcp://localhost/"/>
  <property name="portNumber" value="2006"/>
</bean>

<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>
```

Note

당신은 특정 ConnectionFactory을 직접 인스턴스화할수 없다. 당신은 당신 연결자를 위한 ManagedConnectionFactory 인터페이스의 관련 구현물을 통해 수행할 필요가 있다. 이 인터페이스는 JCA SPI스펙의 일부이다.

21.2.3. CCI connection 설정하기

JCA CCI는 개발자엑 연결자의 ConnectionSpec 구현물을 사용하여 EIS에 대한 연결을 설정하도록 해준다. 이것의 프라퍼티를 설정하기 위해, 당신은 전용 어댑터를 가진 목표 connection factory인 ConnectionSpecConnectionFactoryAdapter를 포장할 필요가 있다. 그래서 전용 ConnectionSpec은 connectionSpec 프라퍼티로 설정(내부 bean처럼)될수 있다.

이 프라퍼티는 CCI ConnectionFactory인터페이스가 CCI connection을 얻기 위한 서로 다른 두개의 메소드를 정의하기 때문에 필수가 아니다. 몇몇 ConnectionSpec 프라퍼티는 애플리케이션 서버나 관련된 local ManagedConnectionFactory구현물에서 설정될수 있다.

```
public interface ConnectionFactory implements Serializable, Referenceable {
  ...
  Connection getConnection() throws ResourceException;
```

```
  Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;
  ...
}
```

Spring은 주어진 factory에서 모든 작업을 위해 사용하는 ConnectionSpec 인스턴스를 명시하는 ConnectionSpecConnectionFactoryAdapter를 제공한다. 어댑터의 connectionSpec프라퍼티가 명시된다면, 어댑터는 인자가 없거나 ConnectionSpec인자를 가지는 getConnection 의 다른 형태를 사용한다.

```
<bean id="managedConnectionFactory"
    class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>
```

21.2.4. 하나의 CCI connection 사용하기

하나의 CCI connection을 사용하길 원하다면, Spring은 이것을 관리하기 위한 더 나은 ConnectionFactory 어댑터를 제공한다. SingleConnectionFactory 어댑터는 하나의 connection을 늦게(lazy) 열고 이 bean이 애플리케이션 종료시점에 사라질때 닫힌다. 이 클래스는 기본적인 같은 물리적 connection을 공유하는 특별한 Connection 프록시를 나타낸다.

```
<bean id="eciManagedConnectionFactory"
    class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TEST"/>
  <property name="connectionURL" value="tcp://localhost/"/>
  <property name="portNumber" value="2006"/>
</bean>

<bean id="targetEciConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>

<bean id="eciConnectionFactory"
    class="org.springframework.jca.cci.connection.SingleConnectionFactory">
  <property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>
</bean>
```

Note

ConnectionFactory 어댑터는 ConnectionSpec로 직접 설정될수 없다. SingleConnectionFactory가

당신이 하나의 connection을 요구하는지를 알리는 중계수단의 ConnectionSpecConnectionFactoryAdapter를 사용하라.

21.3. Spring의 CCI 접근 지원 사용하기

21.3.1. 레코드 전환(Record conversion)

JCA CCI지원의 목적중 하나는 CCI레코드를 변경하기 위한 편리한 기능을 제공하는 것이다. 개발자는 Spring의 CciTemplate을 사용하기 위해 레코드를 생성하고 레코드로부터 데이터를 추출하기 위한 전략을 명시할수 있다. 다음의 인터페이스는 당신이 애플리케이션에 직접 레코드를 가지고 작업하길 원하지 않는다면 입력및 출력 레코드를 사용하기 위한 전략을 설정할것이다.

입력 Record를 생성하기 위해, 개발자는 RecordCreator인터페이스의 전용 구현물을 사용할수 있다.

```
public interface RecordCreator {

  Record createRecord(RecordFactory recordFactory) throws ResourceException, DataAccessException;
}
```

당신이 볼수 있는것처럼, createRecord 메소드는 사용되는 ConnectionFactory의 RecordFactory에 관련되는 파라미터로 RecordFactory인스턴스를 가진다. 이 참조는 IndexedRecord 인스턴스나 MappedRecord인스턴스를 생성하기 위해 사용될수 있다. 다음의 샘플은 RecordCreator 인터페이스를 사용하고 레코드를 인덱싱하고 맵핑하는 방법을 보여준다.

```
public class MyRecordCreator implements RecordCreator {

  public Record createRecord(RecordFactory recordFactory) throws ResourceException {
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }
}
```

출력 Record는 EIS로부터 데이터를 가져오기 위해 사용될수 있다. 나아가, RecordExtractor인터페이스의 특정 구현물은 출력 Record로부터 데이터를 추출하기 위한 Spring의 CciTemplate로 전달될수 있다.

```
public interface RecordExtractor {

  Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;
}
```

다음의 샘플은 RecordExtractor를 사용하는 방법을 보여준다.

```
public interface RecordExtractor {

  Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;
}
```

The following sample shows how to use the RecordExtractor interface.

```
public class MyRecordExtractor implements RecordExtractor {

  public Object extractData(Record record) throws ResourceException {
```

```
    CommAreaRecord commAreaRecord = (CommAreaRecord) record;
    String str = new String(commAreaRecord.toByteArray());
    String field1 = string.substring(0,6);
    String field2 = string.substring(6,1);
    return new OutputObject(Long.parseLong(field1), field2);
  }
}
```

21.3.2. CciTemplate

CciTemplate은 핵심 CCI지원 패키지(org.springframework.jca.cci.core)의 중심 클래스이다. 이것은 자원의 생성과 반환을 다루기 때문에 CCI의 사용을 단순화한다. 이것은 connection을 닫는것을 잊었을때와 같이 공통적인 에러를 피하도록 도와준다. connection의 생명주기와 객체간의 상호작동을 다루고 애플리케이션 데이터로부터 입력 레코드를 생성하고 출력 레코드로부터 애플리케이션 데이터를 추출하는 것에 집중하는 애플리케이션 코드를 둔다.

JCA CCI스펙은 EIS의 작업을 호출하기 위한 두가지 구별되는 메소드를 정의한다. CCI Interaction 인터페이스는 두개의 execute메소드를 제공한다.

```
public interface javax.resource.cci.Interaction {
  ...
  boolean execute(InteractionSpec spec, Record input, Record output) throws ResourceException;

  Record execute(InteractionSpec spec, Record input) throws ResourceException;
  ...
}
```

호출되는 템플릿 메소드에 의존하여, CciTemplate은 상호작동에서 호출하기 위한 execute메소드를 알게될것이다. 어떤 경우, InteractionSpec 인스턴스를 정확하게 초기화하는것은 필수이다.

CciTemplate.execute는 두가지 방법으로 사용될수 있다.

☒ 직접 Record 인자 사용하기. 이 경우, 당신은 CCI 입력 레코드를 전달할 필요가 있다. 그리고 반환되는 객체는 CCI 출력 레코드와 관련된다.

☒ 레코드 맵핑을 사용하여 애플리케이션 객체 사용하기. 이 경우, 당신은 관련 RecordCreator 인스턴스와 RecordExtractor 인스턴스를 제공할 필요가 있다.

첫번째 접근법에서, 다음의 템플릿 메소드가 사용될것이다. 이러한 메소드는 Interaction 인터페이스의 메소드와 직접 대응된다.

```
public class CciTemplate implements CciOperations {
  ...
  public Record execute(InteractionSpec spec, Record inputRecord)
      throws DataAccessException { ... }

  public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)
      throws DataAccessException { ... }
  ...
}
```

두번째 접근법에서, 우리는 인자로 레코드 생성과 레코드 추출 전략을 명시할 필요가 있다. 사용되는 인터페이스는 레코드 전환의 이번 부분에서 언급되었다. 대응되는 CciTemplate 메소드는 다음과 같다.

```
public class CciTemplate implements CciOperations {
  ...
  public Record execute(InteractionSpec spec, RecordCreator inputCreator)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, Record inputRecord, RecordExtractor outputExtractor)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, RecordCreator creator, RecordExtractor extractor)
      throws DataAccessException { ... }
  ...
}
```

만약 outputRecordCreator 프라퍼티가 템플릿에 셋팅되지 않는다면, 모든 메소드는 두개의 파라미터(InteractionSpec 과 입력 Record)를 가지고 CCI Interaction의 반환값으로 출력 Record를 가지는 execute 메소드를 호출할것이다.

CciTemplate 은 createIndexRecord 와 createMappedRecord메소드를 통해 RecordCreator 구현물 외부에서 IndexRecord 와 MappedRecord을 생성하는 메소드를 제공한다. 이것은 적절한 CciTemplate.execute 메소드에 전달하기 위한 Record인스턴스를 생성하는 DAO구현물내에서 사용될수 있다.

```
public class CciTemplate implements CciOperations {
  ...
  public IndexedRecord createIndexedRecord(String name) throws DataAccessException { ... }

  public MappedRecord createMappedRecord(String name) throws DataAccessException { ... }
  ...
}
```

21.3.3. DAO 지원

Spring이 CCI지원은 ConnectionFactory 인스턴스나 CciTemplate인스턴스의 삽입을 지원하는 DAO를 위한 추상 클래스를 제공한다. 클래스명은 CciDaoSupport이다. 이것은 단순히 setConnectionFactory메소드와 setCciTemplate 메소드를 제공한다. 내부적으로, 이 클래스는 ConnectionFactory를 전달하기 위한 CciTemplate인스턴스를 생성하고 하위클래스에서 견고한 데이터 접근 구현을 나타낸다.

```
public abstract class CciDaoSupport {
  ...
  public void setConnectionFactory(ConnectionFactory connectionFactory) { ... }
  public ConnectionFactory getConnectionFactory() { ... }

  public void setCciTemplate(CciTemplate cciTemplate) { ... }
  public CciTemplate getCciTemplate() { ... }
  ...
}
```

21.3.4. 자동화된 출력 레코드 생성

사용되는 연결자는 단지 파라미터로 입력및 출력 레코드를 가지는 Interaction.execute 메소드만을 지원한다(이것은 적절한 출력 레코드를 반환하는 대신에 전달되도록 기대하는 출력 레코드를 요구한다.). 당신은 응답을 가져왔을때 JCA연결자에 의해 채워지는 출력 레코드를 자동으로 생성하는 CciTemplate의 outputRecordCreator 프라퍼티를 볼수 있다. 이 레코드는 템플릿의 호출자로 반환될것이다.

이 프라퍼티는 RecordCreator의 구현물을 가진다. RecordCreator인터페이스는 이미 이전 부분에서

언급되었다. outputRecordCreator프라퍼티는 CciTemplate에서 직접 명시되어야만 한다. 이것은 애플리케이션 코드에서 수행될수 있다.

```
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());
```

또는 Spring설정에서, CciTemplate이 전용 bean처럼 설정된다면

```
<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>

<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
  <property name="outputRecordCreator" ref="eciOutputRecordCreator"/>
</bean>
```

Note

CciTemplate 클래스가 쓰레드에 안전한것처럼, 이것은 공유 인스턴스처럼 설정될것이다.

21.3.5. 개요

다음의 테이블은 CciTemplate와 CCI Interaction 인터페이스에서 호출되는 대응 메소드의 기법을 개략적으로 설명한다.

Table 21.1. Interaction execute 메소드의 사용법

CciTemplate 메소드 시그너처	CciTemplate outputRecordCreator 프라퍼티	CCI Interaction에서 호출되는 execute메소드
Record execute(InteractionSpec, Record)	셋팅안됨	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, Record)	셋팅됨	boolean execute(InteractionSpec, Record, Record)
void execute(InteractionSpec, Record, Record)	셋팅안됨	void execute(InteractionSpec, Record, Record)
void execute(InteractionSpec, Record, Record)	셋팅됨	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, RecordCreator)	셋팅안됨	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, RecordCreator)	셋팅됨	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, Record, RecordExtractor)	셋팅안됨	Record execute(InteractionSpec, Record)

CciTemplate 메소드 시그너처	CciTemplate outputRecordCreator 프라퍼티	CCI Interaction에서 호출되는 execute메소드
Record execute(InteractionSpec, Record, RecordExtractor)	셋팅됨	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, RecordCreator, RecordExtractor)	셋팅안됨	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, RecordCreator, RecordExtractor)	셋팅됨	void execute(InteractionSpec, Record, Record)

21.3.6. CCI Connection 과 Interaction을 직접 사용하기

CciTemplate 은 JdbcTemplate 과 JmsTemplate처럼 같은 방법으로 CCI connections 과 interactions을 가지고 직접 작동하는것이 가능하다. 예를 들어, 이것은 당신이 CCI connection 이나 interaction에서 다중 작업을 수행하길 원할때 유용하다.

인터페이스 ConnectionCallback는 인자로 CCI Connection를 제공한다. 사용자정의 작업을 수행하기 위해, Connection를 생성하는 CCI ConnectionFactory를 더한다. 후자는 관련 RecordFactory인스턴스를 얻고 인덱싱되고 맵핑된 레코드를 생성하기 위한 예제를 위해 유용하다.

```
public interface ConnectionCallback {

  Object doInConnection(Connection connection, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

인터페이스 InteractionCallback은 CCI Interaction을 제공한다. 사용자정의 작업을 수행하기 위해, 대응하는 CCI ConnectionFactory를 더한다.

```
public interface InteractionCallback {

  Object doInInteraction(Interaction interaction, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

Note

InteractionSpec 객체는 다중 템플릿 호출을 통해 공유될수 있고 모든 콜백 메소드내부에서 새롭게 생성될수 있다. 이것은 DAO구현을 완벽하게 한다.

21.3.7. CciTemplate 사용을 위한 예제

이 부분에서, CciTemplate의 사용은 IBM CICS ECI연결자를 사용하여 ECI모드로 CICS에 접근하는 것을 보여준다.

첫째로, CCI InteractionSpec에서의 몇몇 초기화는 접근하는 CICS 프로그램과 이것과 상호작동하는 방법을 명시해야만 한다.

```
ECIInteractionSpec interactionSpec = new ECIInteractionSpec();
interactionSpec.setFunctionName("MYPROG");
interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
```

그리고나서 프로그램은 Spring의 템플릿을 통해 CCI를 사용하고 사용자정의 객체와 CCI Records사이 맵핑을 명시할수 있다.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ECIInteractionSpec interactionSpec = ...;

    OutputObject output = (ObjectOutput) getCciTemplate().execute(interactionSpec,
        new RecordCreator() {
          public Record createRecord(RecordFactory recordFactory) throws ResourceException {
            return new CommAreaRecord(input.toString().getBytes());
          }
        },
        new RecordExtractor() {
          public Object extractData(Record record) throws ResourceException {
            CommAreaRecord commAreaRecord = (CommAreaRecord)record;
            String str = new String(commAreaRecord.toByteArray());
            String field1 = string.substring(0,6);
            String field2 = string.substring(6,1);
            return new OutputObject(Long.parseLong(field1), field2);
          }
        });

    return output;
  }
}
```

이미 언급된것처럼, 콜백은 CCI connections 이나 interactions에서 직접 작동하기 위해 사용될수 있다.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ObjectOutput output = (ObjectOutput) getCciTemplate().execute(
        new ConnectionCallback() {
          public Object doInConnection(Connection connection, ConnectionFactory factory)
              throws ResourceException {
            ...
          }
        });
    }
    return output;
  }
}
```

Note

사용되는 ConnectionCallback을 가지고, Connection은 CciTemplate에 의해 관리되고 닫힐것이다. 하지만 connection에서 생성된 interaction은 콜백 구현물에 의해 관리되어야만 한다.

특정 콜백에 대한 좀더 많은 정보를 위해, 당신은 InteractionCallback을 구현할수 있다. 전달된 Interaction은 이 경우 CciTemplate에 의해 관리되고 닫힐것이다.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public String getData(String input) {
    ECIInteractionSpec interactionSpec = ...;

    String output = (String) getCciTemplate().execute(interactionSpec,
        new InteractionCallback() {
          public Object doInInteraction(Interaction interaction, ConnectionFactory factory)
              throws ResourceException {
            Record input = new CommAreaRecord(inputString.getBytes());
            Record output = new CommAreaRecord();
            interaction.execute(holder.getInteractionSpec(), input, output);
            return new String(output.toByteArray());
          }
        });

    return output;
  }
}
```

위 예제를 위해, 포함된 Spring bean의 대응되는 설정은 이것이 비-관리 모드에서처럼 보일수 있다.

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="mypackage.MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

관리모드(J2EE환경에서)에서, 설정은 다음과 같을것이다.

```
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

21.4. 작업(operation) 객체로 CCI접근 모델링하기

org.springframework.jca.cci.object 패키지는 Spring의 JDBC작업 객체와 비슷한 재사용가능한 작업객체를 통해 다른 스타일로 EIS에 접근하도록 허용하는 지원 클래스를 포함한다. 이것은 언제나 CCI API를 캡슐화할것이다. 애플리케이션 레벨의 입력 객체는 작업객체로 전달될것이다. 그래서 이것은 입력 레코드를 생성하고 가져온 레코드 데이터를 애플리케이션 레벨 출력 객체로 변환하고 이것을 반환한다.

Note

이 접근법은 내부적으로 CciTemplate클래스와 RecordCreator / RecordExtractor 인터페이스에

기초한다. Spring의 핵심 CCI지원의 장치를 재사용한다.

21.4.1. MappingRecordOperation

MappingRecordOperation는 기본적으로 CciTemplate과 같은 작업을 수행하지만, 객체로 특별하고 미리 설정된 작업을 표시한다. 이것은 입력 객체를 입력 레코드를 변환하는 방법과 출력 레코드를 출력 객체로 변환하는 방법(레코드 맵핑)을 명시하는 두개의 템플릿 메소드를 제공한다.

☒ 입력 객체를 입력 Record로 변환하는 방법을 명시하는 createInputRecord

☒ 출력 Record로 부터 출력 객체를 추출하는 방법을 명시하는 extractOutputData

이것은 이러한 메소드의 시그너처이다.

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  protected abstract Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException, DataAccessException { ... }

  protected abstract Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException, DataAccessException { ... }
  ...
}
```

그 후, EIS작업을 수행하기 위해, 당신은 결과처럼 애플리케이션 레벨의 입력 객체를 전달하고 애플리케이션 레벨의 출력 객체를 가져오는 하나의 execute메소드를 사용할 필요가 있다.

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  public Object execute(Object inputObject) throws DataAccessException {
  ...
}
```

당신이 보는것처럼, CciTemplate클래스에 반대로, 이 execute메소드는 인자로 InteractionSpec를 가지지 않는다. 대신 InteractionSpec은 작업에 범용적이다. 다음의 생성자는 특별한 InteractionSpec을 가진 작업 객체를 인스턴스화하기 위해 사용되어야만 한다.

```
InteractionSpec spec = ...;
MyMappingRecordOperation eisOperation = new MyMappingRecordOperation(getConnectionFactory(), spec);
...
```

21.4.2. MappingCommAreaOperation

몇몇 연결자는 EIS에 보내는 파라미터와 이것에 의해 반환되는 데이터를 포함하는 바이트의 배열을 표시하는 COMMAREA에 기초한 레코드를 사용한다. Spring은 레코드보다 COMMAREA에 직접 작동하기 위한 특별한 작업 클래스를 제공한다. MappingCommAreaOperation클래스는 특별한 COMMAREA지원을 제공하기 위한 MappingRecordOperation을 확장한다. 이것은 입력및 출력 레코드 타입으로 CommAreaRecord클래스를 사용하고 입력 객체를 입력 COMMAREA로 변환하고 출력 COMMAREA를 출력 객체로 전환하기 위한 두개의 새로운 메소드를 제공한다.

```
public abstract class MappingCommAreaOperation extends MappingRecordOperation {
  ...
  protected abstract byte[] objectToBytes(Object inObject)
```

```
        throws IOException, DataAccessException;

    protected abstract Object bytesToObject(byte[] bytes)
        throws IOException, DataAccessException;
    ...
}
```

21.4.3. 자동 출력 레코드 작업

모든 MappingRecordOperation 하위클래스가 내부적으로 CciTemplate에 기초하는것처럼, CciTemplate에서처럼 자동으로 출력 레코드를 생성하기 위한 같은 방법은 사용가능하다. 모든 작업 객체는 대응하는 setOutputRecordCreator메소드를 제공한다. 좀더 많은 정보를 위해, 이전의 "자동 출력 레코드 생성" 부분을 보라.

21.4.4. 개요

작업 객체 접근법은 CciTemplate클래스처럼 같은 방법으로 레코드를 사용한다.

Table 21.2. Interaction execute메소드의 사용

MappingRecordOperation 메소드 시그너처	MappingRecordOperarion outputRecordCreator 프라퍼티	CCI Interaction에서 호출되는 execute메소드
Object execute(Object)	셋팅안됨	Record execute(InteractionSpec, Record)
Object execute(Object)	셋팅됨	boolean execute(InteractionSpec, Record, Record)

21.4.5. MappingRecordOperation 사용을 위한 예제

이 부분에서, MappingRecordOperation의 사용이 블랙박스 CCI연결자를 가지고 데이터베이스에 접근하는 것을 보여줄것이다.

Note

이 연결자의 원래의 버전은 Sun에서 사용가능한 J2EE SDK(버전 1.3)에 의해 제공된다.

첫번째, CCI InteractionSpec에서의 몇몇 초기화는 수행하기 위한 SQL요청을 명시하기 위해 수행되어야만 한다. 이 샘플에서, 우리는 요청의 파라미터를 CCI레코드로 변환하는 방법과 CCI결과 레코드를 Person클래스의 인스턴스로 변환하는 방법을 직접 정의한다.

```
public class PersonMappingOperation extends MappingRecordOperation {

  public PersonMappingOperation(ConnectionFactory connectionFactory) {
    setConnectionFactory(connectionFactory);
    CciInteractionSpec interactionSpec = new CciConnectionSpec();
    interactionSpec.setSql("select * from person where person_id=?");
    setInteractionSpec(interactionSpec);
```

```
  }

  protected Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException {
    Integer id = (Integer) inputObject;
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }

  protected Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException {
    ResultSet rs = (ResultSet) outputRecord;
    Person person = null;
    if (rs.next()) {
      Person person = new Person();
      person.setId(rs.getInt("person_id"));
      person.setLastName(rs.getString("person_last_name"));
      person.setFirstName(rs.getString("person_first_name"));
    }
    return person;
  }
}
```

그리고 나서 애플리케이션은 인자로 person id를 가지는 작업객체를 수행할수 있다. 작업객체는 공유 인스턴스이고 쓰레드에 안전한것처럼 셋업될수 있다.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public Person getPerson(int id) {
    PersonMappingOperation query = new PersonMappingOperation(getConnectionFactory());
    Person person = (Person) query.execute(new Integer(id));
    return person;
  }
}
```

Spring bean에 대응되는 설정은 다음의 비-관리모드처럼 볼수 있다.

```
<bean id="managedConnectionFactory"
    class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

관리 모드(J2EE환경에서)에서, 설정은 다음과 같을것이다.

```
<bean id="targetConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/blackbox"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

21.4.6. MappingCommAreaOperation 사용을 위한 예제

이 부분에서, IBM CICS ECI연결자로 ECI모드에서 CICS접근하는 MappingCommAreaOperation의 사용을 보여줄것이다.

첫번째, CCI InteractionSpec는 접근하기 위한 CICS프로그램과 상호작동하는 방법을 명시하기 위해 초기화할필요가 있다.

```
public abstract class EciMappingOperation extends MappingCommAreaOperation {

  public EciMappingOperation(ConnectionFactory connectionFactory, String programName) {
    setConnectionFactory(connectionFactory);
    ECIInteractionSpec interactionSpec = new ECIInteractionSpec();
    interactionSpec.setFunctionName(programName);
    interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
    interactionSpec.setCommareaLength(30);
    setInteractionSpec(interactionSpec);
    setOutputRecordCreator(new EciOutputRecordCreator());
  }

  private static class EciOutputRecordCreator implements RecordCreator {
    public Record createRecord(RecordFactory recordFactory) throws ResourceException {
      return new CommAreaRecord();
    }
  }
}
```

추상 EciMappingOperation 클래스는 사용자정의 객체와 Records사이의 맵핑을 명시하기 위해 하위클래스화될수 있다.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(Integer id) {
    EciMappingOperation query = new EciMappingOperation(getConnectionFactory(), "MYPROG") {
      protected abstract byte[] objectToBytes(Object inObject) throws IOException {
        Integer id = (Integer) inObject;
        return String.valueOf(id);
      }
      protected abstract Object bytesToObject(byte[] bytes) throws IOException;
```

```
        String str = new String(bytes);
        String field1 = str.substring(0,6);
        String field2 = str.substring(6,1);
        String field3 = str.substring(7,1);
        return new OutputObject(field1, field2, field3);
      }
    });

    return (OutputObject) query.execute(new Integer(id));
  }
}
```

Spring bean에 대응되는 설정은 비-관리모드에서 다음과 같을것이다.

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

관리모드(J2EE환경에서)에서, 설정은 다음과 같을것이다.

```
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

21.5. 트랜잭션

JCA는 자원 어댑터를 위한 다양한 레벨의 트랜잭션 지원을 명시한다. 당신의 자원 어댑터가 지원하는 종류의 트랜잭션은 ra.xml파일에 명시된다. 여기엔 3개의 선택사항이 있다. none (CICS EPI 연결자의 예제를 위해), local 트랜잭션 (CICS ECI 연결자의 예제를 위해), global 트랜잭션 (IMS 연결자의 예제를 위해).

```
<connector>
  ...
  <resourceadapter>
    ...
    <!-- transaction-support>NoTransaction</transaction-support -->
    <!-- transaction-support>LocalTransaction</transaction-support -->
    <transaction-support>XATransaction</transaction-support>
    ...
  <resourceadapter>
  ...
<connector>
```

global트랜잭션을 위해, 당신은 JtaTransactionManager로 트랜잭션을 구분하는 Spring의 일반적인 트랜잭션 구조를 사용할수 있다(J2EE서버의 분산 트랜잭션 조정자에 위임한다.).

하나의 CCI ConnectionFactory에서 local트랜잭션을 위해, Spring은 JDBC를 위한 DataSourceTransactionManager와 유사한 CCI를 위한 특정 트랜잭션 관리 전략을 제공한다. CCI API는 local트랜잭션 객체와 대응되는 local트랜잭션 구분 메소드를 정의한다. Spring의 CciLocalTransactionManager는 Spring의 일반적인 PlatformTransactionManager추상화와 완벽하게 호환되는 local CCI트랜잭션을 수행한다.

```
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="eciTransactionManager"
    class="org.springframework.jca.cci.connection.CciLocalTransactionManager">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
</bean>
```

선언적이거나 프로그램으로 정의되는 두가지의 트랜잭션 전략은 Spring의 트랜잭션 구분 기능과 함께 사용될수 있다. 이것은 실질적인 수행 전략으로부터 트랜잭션 구분을 디커플링하는 Spring의 일반적인 PlatformTransactionManager추상화의 결과이다. 필요하다면 당신의 트랜잭션 구분을 유지하고 JtaTransactionManager 와 CciLocalTransactionManager를 간단히 교체하라.

Spring의 트랜잭션 기능에 대한 좀더 많은 정보를 위해서, 트랜잭션 관리를 보라.

Chapter 22. Spring 메일 추상 계층

22.1. 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.

22.2. Spring 메일 추상화 구조

Spring 메일 abstraction 계층의 메인 패키지는 org.springframework.mail패키지이다. 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.

```
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;

}
```

```
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;

}
```

```
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;

}
```

22.3. 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"/>
</bean>

<bean id="mailMessage" class="org.springframework.mail.SimpleMailMessage">
  <property name="from" value="customerservice@mycompany.com"/>
  <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.OrderManagerImpl">
  <property name="mailSender" ref="mailSender"/>
  <property name="message" ref="mailMessage"/>
</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 Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍를 보도록 하라.

22.3.1. 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.

22.4. 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.

22.4.1. 간단한 MimeMessage 를 생성하고 보내기

MimeMessageHelper를 사용하면, MimeMessage를 셋업하고 보내는 것이 매우 간단해진다.

```
// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("test@host.com");
helper.setText("Thank you for ordering!");

sender.send(message);
```

22.4.2. 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.createMimeMessage();

// 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
// the various addAttachment() methods on the MimeMessageHelper

sender.send(message);
```

Warning

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 23. Spring을 사용한 스케줄링과 쓰레드 풀링

23.1. 소개

Spring은 스케쥴링을 지원하는 통합 클래스들을 제공한다. 현재적으로, Spring은 1.3 이후버전 JDK의 일부분인 Timer와 Quartz 스케쥴러 (http://www.opensymphony.com/quartz/)를 지원하고 있다. 이 두개의 스케쥴러들은 각각 Timer 혹은 Trigger에 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.) Spring은 Java 1.3, 1.4, 5와 J2EE환경간의 차이점을 추상화하는 쓰레드 풀링을 위한 클래스를 제공한다.

23.2. OpenSymphony Quartz 스케쥴러 사용하기

Quartz는 Trigger, Job 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.

23.2.1. 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" />
  <property name="jobDataAsMap">
    <map>
      <entry key="timeout" value="5" />
    </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. v

```
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.)

23.2.2. MethodInvokingJobDetailFactoryBean 사용하기

종종 당신은 특정한 객체의 메써드를 호출할 필요가 있을 것이다. 당신은 MethodInvokingJobDetailFactoryBean을 사용하여 다음과 같이 할 수 있다.

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
</bean>
```

위의 예는 (아래에 있는) exampleBusinessObject메소드에서 호출되는 doIt메소드의 결과일것이다.

```
public class ExampleBusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

MethodInvokingJobDetailFactoryBean을 사용할 때, 메소드를 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 Stateful 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="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
  <property name="concurrent" value="false" />
</bean>
```

Note

기본적으로 jobs는 concurrent 옵션에 따라 실행될 것이다.

23.2.3. 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">
    <!-- see the example of method invoking job above -->
    <property name="jobDetail" ref="jobDetail" />
    <!-- 10 seconds -->
    <property name="startDelay" value="10000" />
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000" />
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="exampleJob" />
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?" />
</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 bean="cronTrigger" />
            <ref bean="simpleTrigger" />
        </list>
    </property>
</bean>
```

당신이 세팅할 수 있는 더욱 많은 프라퍼티들이 SchedulerFactoryBean에 있다. 이를테면, job details에 의해 사용되는 calendars라던가, Quartz를 커스터마이징할 수 있게 하는 프라퍼티같은 것들이 말이다. 더 많은 정보를 위해서는 JavaDoc(SchedulerFactoryBean Javadoc)을 참조하도록 해라.

23.3. JDK Timer 지원 사용하기

Spring에서 스케쥴링 업무를 처리하는 또다른 방법은 JDK Timer 객체들을 사용하는 것이다. Timer 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.

23.3.1. 사용자정의 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" />
    <!-- run every 50 seconds -->
    <property name="period" value="50000" />
    <property name="timerTask" ref="checkEmail" />
</bean>
```

task를 단지 한번만 실행하고자 한다면, period 속성을 0(혹은 음수값)으로 바꿔주면 된다.

23.3.2. MethodInvokingTimerTaskFactoryBean 사용하기

Quartz support와 비슷하게, Timer 역시 당신이 주기적으로 메써드를 호출할 수 있도록 하는 요소들을 기술한다.

```
<bean id="doIt" class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject" />
    <property name="targetMethod" value="doIt" />
</bean>
```

위 결과는 exampleBusinessObject 에서 호출되는 doIt 메소드의 결과가 될것이다.

```
public class BusinessObject {
```

```
  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

ScheduledTimerTask 예제의 timerTask 참조를 bean doIt으로 변경하는 것은 고정된 스케줄에서 수행되는 doIt메소드의 결과가 될것이다.

23.3.3. 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 bean="scheduledTask" />
        </list>
    </property>
</bean>
```

23.4. Spring TaskExecutor 추상화

Spring 2.0은 실행자(Executor)를 다루기 위해 새로운 추상화를 소개한다. 실행자(Executor)는 쓰레드 풀링의 개념을 위한 Java 5의 이름이다. 뜻밖의(odd) 명명은 참조하는 구현물이 실제로 풀(pool)인것을 보증하지 않는 사실에 기반한다. 사실, 많은 경우, 실행자(executor)는 쓰레드 하나로 이루어진다. Spring의 추상화는 1.3, 1.4, 5와 Java EE환경간의 상세한 구현을 감추는 만큼 Java 1.3, 1.4환경에 쓰레드 풀링을 적용하도록 도와준다.

23.4.1. TaskExecutor 인터페이스

Spring의 TaskExecutor 인터페이스는 java.util.concurrent.Executor에 일치한다. 사실, 이것이 존재하는 가장 큰 이유는 쓰레드 풀을 사용할때 Java 5에 필요한 추상화이다. 인터페이스는 의미론적인 수행과 쓰레드 풀의 설정을 위한 작업을 받는 하나의 메소드인 execute(Runnable task)를 가진다.

23.4.2. TaskExecutor를 사용하는 곳

TaskExecutor는 필요한 쓰레드 풀링을 위한 추상화에 다른 Spring컴포넌트를 주기 위해 생성되었다. ApplicationEventMulticaster와 같은 컴포넌트인 JMS의 AbstractMessageListenerContainer와 Quartz통합은 모두 풀 쓰레드를 위해 TaskExecutor 추상화를 사용한다. 어쨌뜬, bean이 쓰레드 풀링이 필요하다면, 자체적인 필요를 위해 이 추상화를 사용하는 것이 가능하다.

23.4.3. TaskExecutor 타입들

Spring배포판에 포함된 TaskExecutor의 미리 빌드된 많은 구현물이 있다. 모든 가능성에서, 자체적으로 구현할 필요는 없을것이다.

☒ SimpleAsyncTaskExecutor

이 구현물은 어떤 쓰레드도 재사용하지 않는다. 각각의 호출에 새로운 쓰레드를 시작한다. 어쨌든, 슬롯이 자유로워질때까지 제한을 넘어선 호출은 블럭할 동시 제한을 지원한다. 진정한 풀링을 찾는다면, 페이지 아래로 스크롤하라.

☒ SyncTaskExecutor

이 구현물은 호출을 비동기적으로 수행하지 않는다. 대신, 각각의 호출은 호출 쓰레드로 대체된다. 간단한 테스트케이스와 같이 필요하지 않은 멀티쓰레드 상황에서 사용된다.

☒ ConcurrentTaskExecutor

이 구현물은 Java 5 java.util.concurrent.Executor를 위한 래퍼(wrapper)이다. 대안으로 bean프라퍼티로 Executor 설정 파라미터를 나타내는 ThreadPoolTaskExecutor가 있다. ConcurrentTaskExecutor를 사용할 필요는 드물지만 ThreadPoolTaskExecutor가 필요한 것만큼 충분히 견고하지 않다면, ConcurrentTaskExecutor이 대안이 된다.

☒ SimpleThreadPoolTaskExecutor

이 구현물은 실제로 Spring의 생명주기 콜백을 듣는 Quartz의 SimpleThreadPool의 하위클래스이다. 이것은 Quartz와 Quartz가 아닌 컴포넌트간에 공유될필요가 있는 쓰레드 풀을 가질때 사용된다.

☒ ThreadPoolTaskExecutor

> 어느 이전버전을 위한 것이나 이 구현물을 가지고 java.util.concurrent 패키지의 대안이 되는 버전을 사용하는 것은 가능하지 않다. Lea와 Dawid Kurzyniec의 구현물은 제대로 작동하는 것으로 부터 막을 다른 패키지 구조를 사용한다.

이 구현물은 Java 5 환경에서 사용될수 있지만 그 환경에서 가장 공통적으로 사용되는 것이다. java.util.concurrent.ThreadPoolExecutor를 설정하고 TaskExecutor에 그것을 포장하기 위한 bean프라퍼티를 나타낸다. ScheduledThreadPoolExecutor처럼 향상된 어떤것이 필요하다면, 대신 ConcurrentTaskExecutor를 사용하도록 추천한다.

☒ TimerTaskExecutor

이 구현물은 지원 구현물로 하나의 TimerTask를 사용한다. 이것은 비록 쓰레드에서 동기적이더라도 메소드 호출이 개별 쓰레드에서 수행되어 SyncTaskExecutor와 다르다.

☒ WorkManagerTaskExecutor

> CommonJ는 BEA와 IBM사이에서 함께 개발된 스펙들이다. 이러한 스펙은 Java EE표준은 아니지만 BEA와 IBM의 애플리케이션 서버 구현물에서는 표준이다.

이 구현물은 지원 구현물로 CommonJ WorkManager을 사용하고 Spring 컨텍스트에서 CommonJ WorkManager참조를 셋팅하기 위한 중심적이고 편리한 클래스이다. SimpleThreadPoolTaskExecutor와 유사하게, 이 클래스는 WorkManager인터페이스를 구현하고 WorkManager만큼 직접 사용될수 있다.

23.4.4. TaskExecutor 사용하기

Spring의 TaskExecutor 구현물은 간단한 JavaBean처럼 사용된다. 아래의 예제에서, 우리는 메시지의 세트를 비동기적으로 프린트하기 위한 ThreadPoolTaskExecutor을 사용하는 bean을 정의한다.

```
import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

  private class MessagePrinterTask implements Runnable {

    private String message;

    public MessagePrinterTask(String message) {
      this.message = message;
    }

    public void run() {
      System.out.println(message);
    }

  }

  private TaskExecutor taskExecutor;

  public TaskExecutorExample(TaskExecutor taskExecutor) {
    this.taskExecutor = taskExecutor;
  }

  public void printMessages() {
    for(int i = 0; i < 25; i++) {
      taskExecutor.execute(new MessagePrinterTask("Message" + i));
    }
  }

}
```

당신이 볼수 있는것처럼, 풀에서 쓰레드를 가져오고 스스로 수행하는 것보다, 큐에 Runnable를 추가하고 TaskExecutor는 작업이 수행될때 결정할 내부 규칙을 사용한다.

TaskExecutor가 사용할 규칙을 설정하기 위해, 간단한 bean프라퍼티로 나타낸다.

```
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5" />
  <property name="maxPoolSize" value="10" />
  <property name="queueCapacity" value="25" />
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
  <constructor-arg ref="taskExecutor" />
</bean>
```

Chapter 24. 동적 언어 지원

24.1. 소개

Spring 2.0은 Spring으로 동적 언어를 사용하여 정의된 bean을 가지는 클래스와 객체를 사용하기 위한 편리한 지원을 소개한다.

> 왜 오직 이러한 언어만인가.?
>
> 지원되는 언어는 a) 자바 기업용 커뮤니티에서 많은 매력을 가지는 언어, b) Spring 2.0개발자내 다른 언어에 대한 요청이 없고, c) Spring개발자가 그것들에 대부분 친숙하기 때문에 선택되었다.
>
> 더 많은 언어를 포함하는 것을 멈추지는 않는다. 만약 당신이 <여기에 당신이 선호하는 동적언어를 추가하기> 위한 지원을 보길 원한다면, 당신은 Spring JIRA 페이지에 이슈를 언제나 제기할수 있다.

이 지원은 지원되는 동적언어로 많은 수의 클래스를 작성하는 것을 허용하고 Spring컨테이너를 투명하게 인스턴스화하고 설정하며 결과객체를 의존성 삽입한다.

현재 지원하는 동적언어는 다음과 같다.

☒ Groovy

☒ BeanShell

☒ JRuby

이 동적언어 지원이 즉시 유용할수 있는 완벽하게 작동하는 예제는 Section 24.7, “시나리오” 부분에서 언급된다.

이 장에서 상세화되는 동적언어 지원은 오직 Spring 2.0과 그 이상의 버전에서만 사용가능하다. 현재 이전버전에 대한 동적언어 지원은 계획에 없다.

24.2. 첫번째 예제

이 장의 대부분은 동적언어 지원을 상세하게 다루고 있다. 지원에 대해 나누기 전에, 동적언어에 정의된 bean의 빠른 예제를 보자.

이 첫번째 bean을 위한 동적언어는 Groovy다(이 예제를 위한 기본은 Spring테스트 모음로 부터 얻어진다. 그래서 지원되는 다른 언어와 동등한 예제를 보길 원한다면, 스스로 직접 소스코드를 보라.)

Groovy bean이 구현되는 Messenger 인터페이스를 보자. 이 인터페이스는 명백한 Java로 정의된다. 기초적인 구현물을 알지않는 Messenger에 대한 참조를 가지고 삽입되는 의존적인 객체는 Groovy 스크립트이다.

```
package org.springframework.scripting;

public interface Messenger {
```

```
    String getMessage();
}
```

그리고 이것은 Messenger 인터페이스에 의존성을 가지는 클래스의 정의이다.

```
package org.springframework.scripting;

public class DefaultBookingService implements BookingService {

    private Messenger messenger;

    public void setMessenger(Messenger messenger) {
        this.messenger = messenger;
    }

    public void processBooking() {
        // use the injected Messenger object...
    }
}
```

이것은 Groovy내 Messenger 인터페이스의 구현물이다.

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

// import the Messenger interface (written in Java) that is to be implemented
import org.springframework.scripting.Messenger

// define the implementation in Groovy
class GroovyMessenger implements Messenger {

    @Property String message;
}
```

마지막으로 이것은 Groovy스크립트 Messenger 구현물을 DefaultBookingService 클래스의 인스턴스로 삽입하는 것에 영향을 끼칠 bean정의이다.

Note

동적언어를 지원하는 bean을 정의하기 위한 사용자정의 동적언어 태그를 사용하기 위해, Spring XML설정파일의 가장 위에 XML스키마 서문을 가질 필요가 있다. 또한 IoC컨테이너로 Spring ApplicationContext 구현물이 사용될 필요가 있다. 명백한 BeanFactory 구현물과 함께 동적언어를 지원하는 bean을 사용하는 것은 지원된다. 하지만 그렇게 하기 위해 Spring내부의 구현을 관리해야만 한다.

스키마-기반의 설정에 대한 좀더 많은 정보를 위해서 Appendix A, XML 스키마-기반 설정를 보라.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

    <!-- this is the bean definition for the Groovy-backed Messenger implementation -->
    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
```

```
    </lang:groovy>

    <!-- an otherwise normal bean that will be injected by the Groovy-backed Messenger -->
    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

삽입되는 Messenger 인스턴스가 Messenger 인스턴스이기 때문에 bookingService bean(DefaultBookingService)은 대개 private messenger 멤버 변수를 사용할수 있다. 여기에 특별한 것은 없다. 이것은 명백한 Java이고 명백한 Groovy이다.

위 XML조각은 이름 자체가 설명을 하도록 바라지만, 그렇지 않더라도 걱정하지 말라. 위 설정의 이유와 원인에 대한 깊은 상세사항을 위해 계속 읽어라.

24.3. 동적언어에 의해 지원되는 bean정의하기

이 부분은 지원되는 동적언어로 Spring관리 bean을 정의하는 방법을 언급한다.

이 부분은 지원되는 동적언어의 문법과 표현형식을 설명하는 것을 시도하지 않는다. 예를 들어, 당신이 애플리케이션에서 어떤 클래스를 작성하기 위해 Groovy를 사용하길 원한다면, 당신은 이미 Groovy를 알고 있다고 가정한다. 만약 당신이 동적언어에 대해 좀더 상세하게 다룰필요가 있다면, 이 장의 마지막 부부인 Section 24.8, “더 많은 자원” 를 참조하라.

24.3.1. 공통적인 개념

동적언어-지원 bean을 사용하는 것을 포함하는 단계는 다음과 같다.

1. 동적언어를 위한 테스트 작성
2. 그리고 나서 동적언어 자체를 작성 :)
3. XML설정내 적절한 <lang:language/> 요소를 사용하여 동적언어를 지원하는 bean을 정의하라(비록 당신이 이 장에서는 다루어지지 않는 고급 설정에서 이러한 타입으로 수행하는 방법에 대한 지시를 위한 소스코드를 언급하더라도 당신은 Spring API를 사용하여 이러한 bean을 프로그램처리로 정의할수 있다.). 이것은 반복적인 단계이다. 당신은 동적언어 소스파일마다 적어도 하나의 bean정의가 필요할 것이다(비록 같은 동적언어 소스파일이 다중 bean정의에 의해 참조될수 있더라도).

첫번째 두단계(동적언어에 대한 테스팅과 작성)는 이 장의 범위를 넘어서는 것이다. 언어 스펙과/또는 당신이 선택한 동적언어를 위한 참조문서를 보고 동적언어를 개발하라. 당신은 먼저 이 장의 나머지를 읽기를 원할것이다.

24.3.1.1. <lang:language/> 요소

> XML 스키마
>
> 이 장의 모든 설정예제는 Spring 2.0에 추가된 새로운 XML스키마 지원을 사용한다.

XML스키마 사용을 선행하고 Spring XML파일의 예전 스타일의 DTD기반의 유효성체크에 충실하는것은 가능하다. 하지만 당신은 <lang:language/> 요소가 제공하는 편리함을 잃게된다. XML스키마-기반의 유효성체크를 요구하지 않는 예전 스타일의 설정 예제를 위해서는 Spring테스트 모음을 보라(이것은 다소 장황하고 참조하는 Spring구현물을 숨기지 않는다.).

마지막 단계는 설정(이것은 대개의 Java bean설정과는 차이점이 없다.)하고자 하는 각각의 bean을 위한 동적언어를 지원하는 bean정의를 정의하는 것을 포함한다. 어쨌든, 컨테이너에 의해 인스턴스화되고 설정되는 클래스의 전체경로를 포함한 클래스명을 명시하는 것 대신에, 동적언어를 지원하는 bean을 정의하기 위해 <lang:language/>요소를 사용한다.

지원되는 각각의 언어는 관련된 <lang:language/> 요소를 가진다.

☒ <lang:jruby/> (JRuby)

☒ <lang:groovy/> (Groovy)

☒ <lang:bsh/> (BeanShell)

설정에서 사용가능한 속성과 자식 요소는 bean이 정의되는 언어에 의존한다.(아래의 언어에 종속한 부분은 이것에 대해 완전하게 다룬다.)

24.3.1.2. 갱신가능한(Refreshable) beans

Spring에서 동적언어지원에 추가되는 가장 마음에 드는(compelling) 값중에 하나는 'refreshable bean'기능이다.

갱신가능한 bean은 적은 양의 설정을 가지는 동적언어를 지원하는 bean이다. 동적언어를 지원하는 bean은 참조하는 소스파일 자원의 변경을 모니터링하고 동적언어 소스파일이 변경되면 자체적으로 리로드(예를 들어 개발자가 파일시스템의 파일을 편집하고 저장할때)한다.

이것은 애플리케이션의 일부로 많은 수의 동적언어 소스 파일을 배치하고 동적언어 소스파일에 의해 지원되는 bean을 생성하기 위해 Spring컨테이너를 설정(이 장에서 언급되는 기법을 사용하여) 하는것을 가능하게 한다. 그리고 나서, 요구사항 변경이나 몇몇 다른 외부 요소는 작동하기 시작한다. 동적언어 소스파일을 간단히 편집하고 동적언어 소스파일을 변경하여 지원되는 bean을 반영한다. 여기서는 구동중인 애플리케이션을 중지할 필요가 없다(또는 웹 애플리케이션의 경우 다시 배치할 필요가 없다.). 동적언어를 지원하는 bean은 변경된 동적언어 소스파일로부터 새로운 상태와 로직을 간단히 가져올것이다. 이것은 빠른 프로토타이핑과 같은 시나리오를 위해 명백하다.

Note

이 기능은 디폴트로 사용되지 않는다.

갱신가능한 bean을 사용하여 시작하는 방법을 보기 위한 예제를 먼저 보자. 갱신가능한 bean기능을 사용하기 위해, bean정의의 <lang:language/> 요소에 하나의 추가적인 속성을 명시해야만 한다. 그리고 이 장의 앞부분으로 부터 예제에 충실하다면, 이것은 갱신가능한 bean에 영향을 끼치는 Spring XML설정내 변경일것이다.

```
<beans>

    <!-- this bean is now 'refreshable' due to the presence of the 'refresh-check-delay' attribute -->
```

```
    <lang:groovy id="messenger"
          refresh-check-delay="5000" <!-- switches refreshing on with 5 seconds between checks -->
          script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

'messenger' bean정의에 정의된 'refresh-check-delay' 속성은 bean이 참조하는 동적언어 소스파일에 대한 변경으로 갱신되는 초단위 값이다. 당신은 'refresh-check-delay' 속성에 음수를 할당하여 갱신행위를 끌수 있다. 디폴트는 갱신행위가 작동하지 않는다. 당신이 갱신행위를 원하지 않는다면, 속성을 정의하지 말라.

우리가 다음의 애플리케이션을 실행한다면, 우리는 refreshable기능을 사용할수 있다. 이 코드의 다음 부분에서 'jumping-through-hoops-to-pause-the-execution'를 실행해보라. System.in.read() 호출은 내가 기초적인 스크립트 파일을 진행하고 편집하는 동안 프로그램의 실행이 잠시 멈춘다. 그래서 스크립티-지원 bean은 프로그램이 다시 실행될때 이것의 상태를 갱신할것이다.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger.getMessage());
        // pause execution while I go off and make changes to the source file...
        System.in.read();
        System.out.println(messenger.getMessage());
    }
}
```

변경되는 Messenger 구현물의 getMessage() 메소드를 호출한다고 가정해보자. 프로그램의 실행이 멈추었을때, Messenger.groovy 스크립트에 대한 변경을 아래에서 보라.

```
package org.springframework.scripting

class GroovyMessenger implements Messenger {

    private String message = "Bingo"

    public String getMessage() {
        // change the implementation to surround the message in quotes
        return "'" + this.message + "'"
    }

    public void setMessage(String message) {
        this.message = message
    }
}
```

프로그램이 실행되었을때, 입력이 중단되기 전 출력은 I Can Do The Frug 일것이다. 소스파일을 변경하고 저장한후에, 프로그램 수행을 다시 한다면, 동적언어를 지원하는 Messenger 구현물의 getMessage() 메소드

호출의 결과는 'I Can Do The Frug' 가 될것이다(따옴표를 포함한다는 것에 주의하라.).

언급된 변경이 'refresh-check-delay' 값의 창에서 발생한다면 스크립트에 대한 변경이 갱신을 처리하지 않을것이라는 것을 이해하는 것이 중요하다. 메소드가 동적언어를 지원하는 bean에서 호출될때까지 스크립트에 대한 변경이 실제로 '가지지(picked up)' 않는 것을 이해하는 것도 동등하게 중요하다. 메소드가 동적언어를 지원하는 bean에서 호출될때 참조하는 스크립트 소스가 변경될때만 보는것을 체크한다. 스크립트를 갱신하는것에 관련된 예외(컴파일 에러나 스크립트 파일이 삭제된 것을 찾는 것과 같은)는 호출 코드에 위임되는 치명적인(fatal) 예외의 결과가 될것이다.

위에서 언급된 갱신가능한 bean행위는 <lang:inline-script/> 요소을 사용하여 정의된 동적언어 소스파일에 적용되지 않는다(Section 24.3.1.3, "즉시 처리되는(Inline-인라인) 동적언어 소스파일" 를 보라). 추가적으로, 실질적으로 감지될수 있는 참조하는 소스파일에 대한 변경이 있는 bean에만 적용된다. 예를 들어, 코드에 의해 파일시스템에 존재하는 동적언어 소스파일의 마지막 변경날짜를 체크한다.

24.3.1.3. 즉시 처리되는(Inline-인라인) 동적언어 소스파일

동적언어 지원은 Spring bean정의내 직접 내장되는 동적언어를 제공할수 있다. 좀더 특별히, <lang:inline-script/> 요소는 Spring 설정파일내부에서 즉시 동적언어 소스를 정의하는 것을 허용한다. 예제는 인라인 스크립트 기능을 보여줄것이다.

```
<lang:groovy id="messenger">
    <lang:property name="message" value="I Can Do The Frug" />
    <lang:inline-script>
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    @Property String message;
}
    </lang:inline-script>
</lang:groovy>
```

만약 우리가 Spring설정파일 내부에 동적언어 소스를 정의하는 좋은 상황에 둘러쌓인 이슈에 놓여있다면, <lang:inline-script/> 요소는 몇가지 시나리오에서 유용할수 있다. 예를 들어, 우리가 Spring Validator 구현물을 Spring MVC Controller 에 추가하길 원할때, 이것은 인라인 스크립트를 사용하는 상황이 된다(이러한 예제를 위해서는 Section 24.7.2, "스크립트된 Validators" 를 보라.)

물론 위에서 보여진것보다 좀더 복잡한 클래스의 경우도 잊지 말라. 당신은 <![CDATA[]]> 영역내 인라인 스크립트를 둘러싸야 할지도 모른다.

아래는 inline: 주석을 사용하여 Spring XML설정파일에 직접 JRuby기반 bean을 위한 소스를 정의하는 예제이다.('<' 문자를 나타내는 < 문자의 사용에 유의하라.)

```
<lang:jruby id="messenger" script-interfaces="org.springframework.scripting.Messenger">
    <lang:inline-script>
require 'java'

include_class 'org.springframework.scripting.Messenger'

class RubyMessenger &lt; Messenger

 def setMessage(message)
  @@message = message
 end
```

```
 def getMessage
  @@message
 end

end
    </lang:inline-script>
    <lang:property name="message" value="Hello World!" />
</lang:jruby>
```

24.3.2. 동적언어를 지원하는 bean의 컨텍스트내 생성자 삽입을 이해하기

Spring의 동적언어 지언에 관련하여 인지되는 매우 중요한 것이 있다. 명명하여, 이것은 동적언어를 지원하는 bean에 생성자의 인자를 제공하는 것이 가능하지 않다(나아가 생성자 삽입은 동적언어를 지원하는 bean에 사용가능하지 않다).

생성자를 특별히 다루는 것과 프라퍼티를 100%로 명백하게 만들기 위한 관심으로, 다음의 코드와 설정의 혼합은 작동하지 않을것이다.

이것은 Groovy내 Messenger 인터페이스의 구현물이다.

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

GroovyMessenger() {}

// this constructor is not available for Constructor Injection...
GroovyMessenger(String message) {
        this.message = message;
}

@Property String message;

@Property String anotherMessage
}
```

```
<lang:groovy id="badMessenger"
script-source="classpath:Messenger.groovy">

<!-- this next constructor argument will *not* be injected into the GroovyMessenger -->
<!--      in fact, this isn't even allowed according to the schema -->
<constructor-arg value="This will *not* work" />

<!-- only property values are injected into the dynamic-language-backed object -->
<lang:property name="anotherMessage" value="Passed straight through to the dynamic-language-backed object" />

</lang>
```

사실 이 제한은 setter삽입이 대다수의 개발자에 의해 선호되는 삽입 스타일이 된 이후에 처음으로 나타난것처럼 명백하지는 않다(이것은 나중에 좋은 것이 무엇인지에 대한 논의로 남겨두자.)

24.4. JRuby beans

JRuby 라이브러리 의존성

Spring내 JRuby 스크립트 지원은 애플리케이션의 클래스패스에 다음의 추가적인 jar를 요구한다. (목록화된 특정 라이브러리 버전은 스크립트 지원 개발에서 사용되는 버전이다. 당신은 이전버전이나 이후 버전을 사용할수 있을것이다.)

☒ jruby.jar

☒ cglib-nodep-2.1_3.jar

이러한 라이브러리는 Spring-with-dependecies 배포판에서 사용가능하다(추가적인 것은 웹에서 자유롭게 구할수 있다.)

JRuby 홈페이지로부터...
" [JRuby is an] Java로 작성된 Ruby인터프리터를 재생성하기 위한 영향을 끼치고 Java 바이트코드 컴파일러를 위한 Ruby이다. "

Spring의 동적언어 지원은 JRuby언어에 정의된 bean도 지원한다. JRuby언어는 Ruby언어에 기초를 둔다. 그리고 인라인 정규표현식, 블럭(클로져), 개발을 좀더 쉽도록 몇가지 도메인 문제를 위한 해결책을 만들어주는 다른 기능들을 지원한다.

Spring내 JRuby동적언어 지원의 구현물은 무엇이 발생하는지에 흥미를 가진다. Spring은 <lang:ruby> 요소의 'script-interfaces' 속성에 명시된 모든 인터페이스의 JDK동적 프록시 구현물을 생성한다. (이것은 당신이 언급된 속성값의 적어도 하나의 인터페이스를 제공하고 JRuby를 지원하는 bean을 사용할때 인터페이스에 대한 프로그래밍을 해야만 하는 이유이다.)

JRuby기반의 bean을 사용한 예제를 보자. 이것은 이전에 먼저 정의한 Messenger 인터페이스의 JRuby구현물이다.

```
package org.springframework.scripting;

public interface Messenger {

String getMessage();

}
```

```
require 'java'

include_class 'org.springframework.scripting.Messenger'

class RubyMessenger < Messenger

def setMessage(message)
@@message = message
end

def getMessage
@@message
end

end

RubyMessenger.new # this last line is not essential (but see below)
```

그리고 이것은 RubyMessenger JRuby bean의 인스턴스를 정의한 Spring XML이다.

```
<lang:jruby id="messageService"
        script-interfaces="org.springframework.scripting.Messenger"
        script-source="classpath:RubyMessenger.rb">

<lang:property name="message" value="Hello World!" />

</lang:jruby>
```

JRuby소스의 마지막 라인('RubyMessenger.new')을 노트하자. Spring의 동적언어 지원의 컨텍스트에서 JRuby를 사용할때, 당신은 JRuby소스의 수행의 결과로 동적언어를 지원하는 bean으로 사용하길 원하는 JRuby클래스의 새로운 인스턴스를 인스턴스화하거나 반환하기 위해 격려된다. 소스파일의 마지막 라인에서 JRuby클래스의 새로운 인스턴스를 간단히 인스턴스화하여 이것을 달성할수 있다.

```
require 'java'

include_class 'org.springframework.scripting.Messenger'

# class definition same as above...

# instantiate and return a new instance of the RubyMessenger class
RubyMessenger.new
```

당신이 이것을 하는것을 잊었다면, 이것이 끝이 아니다. 이것은 인스턴스화하기 위한 클래스를 찾는 JRuby클래스의 타입 표시를 통해 Spring이 (반사적으로) 견지질하는(trawl) 결과를 만든다. 이 중요한 스키마에서, 이것은 결코 알리지 않아서 빠를것이다. 하지만 이것은 때때로 JRuby스크립트의 마지막 줄과같은 것을 가져서 피할수 있다. 이러한 줄을 제공하지 않거나 Spring이 스크립트에서 인스턴스화하기 위한 JRuby클래스를 찾을수 없다면 ScriptCompilationException은 소스가 JRuby 인터프리터에 의해 수행된 직후 던져질것이다. 예외의 가장 위에 있는 이유로 이것을 식별하는 핵심 텍스트는 밑에서 즉시 찾을수 있다(그래서 동적언어를 지원하는 bean을 생성할때 Spring컨테이너가 다음의 예외를 던지고 관련 stacktrace에서 다음 텍스트가 있다면, 이것은 확인하는 것을 허용하고 이슈를 쉽게 개정한다.)

```
org.springframework.scripting.ScriptCompilationException: Compilation of JRuby script returned ''
```

이것을 개정하기 위해, JRuby-동적언어를 지원하는 bean처럼 나타나길 원하는 클래스의 새로운 인스턴스를 간단히 인스턴스화하라. 당신이 JRuby스크립트에서 원하는 만큼의 클래스와 객체를 정의할수 있다는 것을 노트하라. 중요한 것은 소스파일이 객체를 반환해야만 한다는 것이다.

JRuby-기반의 bean를 사용하기 위한 몇가지 시나리오를 위해 Section 24.7, “시나리오” 를 보라.

24.5. Groovy beans

Groovy 라이브러리 의존성

Spring내 Groovy스크립트 지원은 애플리케이션의 클래스패스에 다음의 추가적인 jar파일을 요구한다.

☒ groovy-1.0-jsr-04.jar

☒ asm-2.2.2.jar

☒ antlr-2.7.6.jar

Groovy홈페이지로부터....
"Groovy는 Python, Ruby와 Smalltalk처럼 사람들이 좋아하는 많은 기능을 가진 Java 2 플랫폼을 위한 활기찬(agile) 동적 언어이다. Java와 같은 문법을 사용하여 Java개발자를 위해 사용가능하다. "

이 문서를 처음부터 계속 읽었다면, Groovy 동적언어를 지원하는 bean의 예제를 보았을것이다. 다른 예제를 보자.

Note

Groovy 자체는 JDK 1.4이상을 요구한다는 것에 주의하라.

```
package org.springframework.scripting;

public interface Calculator {

    int add(int x, int y);
}
```

이것은 Groovy내 Calculator 인터페이스의 구현물이다.

```
// from the file 'calculator.groovy'
package org.springframework.scripting.groovy

class GroovyCalculator implements Calculator {

    int add(int x, int y) {
        x + y
    }
}
```

그리고 이것은 부수적인 Spring설정이다(XML).

```
<-- from the file 'beans.xml' -->
<beans>
    <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>
</beans>
```

마지막으로 이것은 위 설정을 수행하기 위한 작은 애플리케이션이다.

```
package org.springframework.scripting;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void Main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Calculator calc = (Calculator) ctx.getBean("calculator");
        System.out.println(calc.add(2, 8));
    }
}
```

위 프로그램의 실행으로부터의 결과는 10이 될것이다(놀라운 예제인가.? 의도는 개념을 전달하는것이라는것을 기억하라. 좀더 실질적인 예제를 위해서는 동적언어 전시 프로젝트를 보거나 이 장의 마지막의 Section 24.7, “시나리오” 를 보라).

Groovy 소스파일마다 한개 이상의 클래스를 정의하지 않는다는 것이 중요하다. Groovy내에서는 완전한 반면에 좋지 않은 상황이다. 일관적인 접근법에 관심을 가지고 당신은 소스파일마다 한개의 (public)클래스라는 표준 Java규칙을 반영해야만 할것이다(저자의 의견으로).

이것은 Groovy기반 bean이 작동하는데 충분하다. 그럼 Groovy를 사용해보라.

24.6. BeanShell beans

BeanShell 라이브러리 의존성

Spring내 BeanShell스크립트 지원은 애플리케이션 클래스패스에 다음의 추가적인 jar파일을 요구한다. (목록화된 특정 라이브러리 버전은 스크립트 지원 개발에서 사용되는 버전이다. 당신은 이전버전이나 이후 버전을 사용할수 있을것이다.)

☒ bsh-2.0b4.jar

☒ cglib-nodep-2.1_3.jar

이러한 모든 라이브러리는 Spring-with-dependecies배포판에서 사용가능하다(추가적인 것은 웹에서 자유롭게 구할수 있다.)

BeanShell 홈페이지로부터...
“ Java로 작성된 BeanShell은 객체 동적언어 기능을 가진 작고, free이며, 내장된 Java소스 인터프리퍼이다. BeanShell은 표준 Java문법을 동적으로 수행하고 느슨한(loose) 타입, 커멘드, 그리고 Perl과 자바스크립트의 그것과 같은 메소드 closure와 같은 공통 동적언어 편리성으로 이것을 확장한다. ”

Groovy와는 반대로, BeanShell을 지원하는 bean정의는 몇가지 추가적인 설정을 요구한다. Spring내 BeanShell 동적언어 지원의 구현물은 무엇이 발생하는지에 대해 흥미를 가진다. Spring은 <lang:bsh> 요소의 ‘script-interfaces’ 속성값에 명시된 모든 인터페이스의 JDK 동적 프록시 구현물을 생성한다(이것은 당신이 언급된 속성값의 적어도 하나의 인터페이스를 제공하고 BeanShell을 지원하는 bean을 사용할때 인터페이스에 대한 프로그래밍을 해야만 하는 이유이다.). 이것은 BeanShell을 지원하는 객체의 모든 메소드 호출이 JDK 동적 프록시 호출 기법을 통한다는 것을 의미한다.

이 장에서 이미 정의된 Messenger 인터페이스를 구현하는 BeanShell기반 bean을 사용하는 예제를 보자(편리성을 위해 밑에서 반복될것이다.).

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

이것은 Messenger 인터페이스의 BeanShell ‘implementation’이다(여기서는 매우 느슨하게 사용되는것을

뜻한다).

```
String message;

String getMessage() {
    return message;
}

void setMessage(String aMessage) {
    message = aMessage;
}
```

그리고 이것은 위 'clsss'의 'instance'를 정의하는 Spring XML이다(다시 말해, 여기서는 매우 느슨하게 사용되는것을 뜻한다.).

```
<lang:bsh id="messageService" script-source="classpath:BshMessenger.bsh"
    script-interfaces="org.springframework.scripting.Messenger">

    <lang:property name="message" value="Hello World!" />
</lang:bsh>
```

BeanShell 기반 bean을 사용하길 원하는 몇몇 시나리오를 위해 Section 24.7, "시나리오" 를 보라.

24.7. 시나리오

스크립트 언어내 Spring관리 bean을 정의하는 가능한 시나리오는 이익이 될것이다. 물론 많고 다양한 이익이 될것이다. 이 부분은 Spring에서 동적언어 지원을 위한 두가지의 가능한 사용 케이스를 언급한다.

Spring 배포판은 Spring은 배포판의 관련 부분내 Spring의 동적언어 지원을 위한 보여주기 위한 프로젝트를 가진다는 것을 알아달라(보여주기 위한 프로젝트는 Spring프레임워크의 특정 관점을 다루는 범위내에서 제한되는 작은 프로젝트이다.)

24.7.1. 스크립트된 Spring MVC 컨트롤러

동적언어를 지원하는 bean을 사용하는 이득을 얻는 하나의 클래스 그룹은 Spring MVC컨트롤러이다. 단순히 Spring MVC애플리케이션에서, 웹 애플리케이션을 통한 탐색이 가능한 흐름(navigational flow)은 하나의 Spring MVC 컨트롤러에서 캡슐화되는 코드에 의해 결정되는 큰 범위이다. 웹 애플리케이션의 탐색가능한 흐름과 다른 표현 레이어 로직은 지원 이슈에 응하거나 비지니스 요구사항을 변경하기 위해 업데이트 될 필요가 있다. 이것은 하나 이상의 동적언어 소스파일을 편집하고 수행중인 애플리케이션의 상태가 즉시 반영되는 변경을 보는 이러한 요구되는 변경에 좀더 쉽게 영향을 끼친다.

Spring과 같은 프로젝트가 채택한 경량 구조 모델(lightweight architectural model)에서, 도메인과 서비스 레이어 클래스에 포함된 모든 애플리케이션의 비지니스 로직과 함께 가벼운 표현 레이어를 대개 지향한다. 동적언어를 지원하는 bean처럼 Spring MVC컨트롤러를 개발하는 것은 텍스트 파일을 간단히 편집하고 저장하여 표현 레이어를 변경하는 것을 허용한다. 이러한 동적언어 소스파일의 변경(설정에 따라)은 언급된 스크립트에 의해 지원되는 bean에 자동적으로 영향을 끼칠것이다.

Note

동적언어를 지원하는 bean에 대한 변경을 자동적으로 'pickup' 하는데 영향을 끼치지 위해, '갱신가능한(refreshable) beans' 기능을 사용가능하게 해야 한다는 것을 알아두라. 이 기능에 대한 완전하고 상세한 처리를 Section 24.3.1.2, "갱신가능한(Refreshable) beans"

를 보라.

Groovy 동적언어를 사용하여 구현된 org.springframework.web.servlet.mvc.Controller의 예제는 아래에서 찾아보라(이 예제는 Spring 배포판과 함께 제공되는 동적언어 지원 전시 프로젝트의 일부에 있다. Spring 배포판의 'samples/showcases/dynamvc/' 디렉토리내 프로젝트를 보라).

```
<!-- from the file '/WEB-INF/groovy/FortuneController.groovy' -->
package org.springframework.showcase.fortune.web;

import org.springframework.showcase.fortune.service.FortuneService;
import org.springframework.showcase.fortune.domain.Fortune;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FortuneController implements Controller {

    @Property FortuneService fortuneService

    public ModelAndView handleRequest(
            HttpServletRequest request, HttpServletResponse httpServletResponse) {

        return new ModelAndView("tell", "fortune", this.fortuneService.tellFortune())
    }

}
```

위 Controller의 부수적인 bean정의는 다음과 같을것이다.

```
<lang:groovy id="fortune"
             refresh-check-delay="3000"
             script-source="/WEB-INF/groovy/FortuneController.groovy">
    <lang:property name="fortuneService" ref="fortuneService"/>
</lang:groovy>
```

24.7.2. 스크립트된 Validators

동적언어를 지원하는 bean으로 인해 가져지는 유연성을 가지는 Spring을 사용한 애플리케이션 개발의 다른 영역은 유효성체크이다. 이것은 아마도 일반적인 Java에 비해 느슨하게 타이핑된 동적언어를 사용하여 복잡한 유효성체크 로직을 표현하는 것을 좀더 쉽게 할것이다.

다시 말해, 동적언어를 지원하는 bean처럼 하나의 validator를 개발하는 것은 간단한 텍스트 파일을 편집하고 저장하여 유효성 체크 로직을 변경하는 것을 허용한다. 이러한 변경(설정에 따라)은 수행중인 애플리케이션의 실행에 자동적으로 영향을 끼치고 애플리케이션의 재시작을 요구하지는 않는다.

Note

스크립트-지원 bean에 대한 변경을 자동적으로 'pickup' 하는데 영향을 끼치지 위해, 'refreshable beans' 기능을 사용가능하게 해야 한다는 것을 알아두라. 이 기능에 대한 완전하고 상세한 처리를 Section 24.3.1.2, "갱신가능한(Refreshable) beans" 를 보라.

Groovy 동적언어를 사용하여 구현되는 Spring org.springframework.validation.Validator 예제는 아래에서 찾아보라. (Validator 인터페이스에 대한 언급은 Section 5.2, "Spring의 Validator인터페이스를 사용하여 유효성 체크하기" 에서 보라.)

```
import org.springframework.validation.Validator
import org.springframework.validation.Errors
import org.springframework.beans.TestBean

public class TestBeanValidator implements Validator {

    boolean supports(Class clazz) {
        return TestBean.isAssignableFrom(clazz)
    }

    void validate(Object bean, Errors errors) {
        String name = bean.getName()
        if(name == null || name.trim().length == 0) {
            errors.reject("whitespace", "Cannot be composed wholly of whitespace")
        }
    }
}
```

24.8. 더 많은 자원

이 장에서 언급된 다양한 동적언어에 대한 더 많은 자원을 위한 링크를 밑에서 알아보라.

☒ JRuby 홈페이지

☒ Groovy 홈페이지

☒ BeanShell 홈페이지

Spring커뮤니티의 몇 활동적인 멤버는 위 추가적인 동적언어의 지원과 이 장에서 다루는것 이상을 추가했다. 이러한 여타의 기여는 주요 Spring배포판에 의해 지원되는 언어의 목록에 추가되는것이 가능한 동안, 당신이 선호하는 스크립트 언어를 보기 위한 베팅은 Spring Modules project이다.

Chapter 25. 어노테이션(annotation)과 소스 레벨 메타데이타 지원

25.1. 소개

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

예를 들어, 우리는 다음처럼 클래스에 메타데이타를 추가할수 있다.

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

우리는 다음처럼 메소드에 데타데이타를 추가할수 있다.

```
/**
 * Normal comments here
 * @@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.

☒ 표준 Java Annotations: 표준 Java 메타데이터 구현물(JSR-175로 개발되고 Java 5에서 사용가능한). Spring은 트랜잭션 구분과 JMX를 위한 특정 Java 5 Annotation을 이미 지원한다. 하지만 우리는 Java 1.4와 1.3을 위한 솔루션도 필요하다. Spring 메타데이타 지원은 이러한 솔루션을 제공한다.

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

☒ 자바 1.3 그리고 1.4를 위한 다양한 오픈소스 속성 구현물들, Commons Attribute의 것은 대부분 완벽한 구현물이다. 모든것들은 특별한 pre-(선) 또는 post-(후) 컴파일 단계를 요구한다.

25.2. 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.

☒ Java 5는 언어 레벨에서 메타데이타 지원을 제공한다. 여기엔 추상화처럼 제공되는데 가치가 있을것이다.

  ☒ Java 5 메타데이타는 정적이다. 이것은 컴파일 시점에 클래스와 관련된다. 그리고 배치된 환경에서 변경될수 없다. 여기엔 배치내 어떤 속성값을 오버라이드하기 위한 능력을 제공하는 구조적인 메타데이타를 위해 필요하다. 예를 들면, XML파일.

  ☒ Java 5 메타데이타는 자바의 리플렉션(reflection) API를 통해 반환된다. 이것은 테스트 시간동안 흉내내는것은 불가능하다. Spring은 이것을 허용하기 위한 간단한 인터페이스를 제공한다.

  ☒ 여기엔 1.3과 1.4애플리케이션에서 메타데이타 지원을 위해 필요할것이다. Spring은 지금 작동하는 솔루션을 제공하는것이 목적이다. Java 5의 사용을 강요하는 것은 중요한 영역에서 선택사항은 아니다.

☒ Commons Attributes(Spring 1.0-1.2에서 사용되는)과 같이 근래의 메타데이타 API는 테스트하기가 어렵다. Spring은 흉내내기 좀더 쉬운 간단한 메타데이타 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)를 사용하여, 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.

25.3. 어노테이션

Spring은 많은 수의 사용자정의 (Java5이상) 어노테이션을 가진다.

25.3.1. @Required

org.springframework.beans.factory.annotation패키지내 @Required 어노테이션은 'required-to-be-set'되도록 프라퍼티를 표시하기 위해 사용될수 있다(이를테면, 클래스에서 어노테이션 처리된 (setter)메소드는 값을 가지고 의존성이 삽입되도록 설정되어야만 한다.). 그렇지 않다면 런타임시 컨테이너에 의해 Exception가 던져져야 한다.

이 어노테이션의 사용법을 보여주는 가장 좋은 방법은 간단하게 예제를 보여주는 것이다.

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can 'inject' a MovieFinder
    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

위 클래스정의를 눈으로 쉽게 읽기를 바란다. 기본적으로 SimpleMovieLister클래스를 위한 어떤 그리고 모든 BeanDefinitions은 값으로 가진채 제공되어야 한다(Spring API의 개념에서는 이것은 PropertyValue가 그 프라퍼티를 위해 셋팅되어야 한다는 것을 의미한다.).

유효성 체크를 지나치지 않을 몇가지 XML설정의 예제를 보자.

```
<bean id="movieLister" class="x.y.SimpleMovieLister">
    <!-- whoops, no MovieFinder is set (and this property is @Required) -->
</bean>
```

런타임시 다음의 메시지는 Spring 컨테이너에 의해 생성될것이다(스택 추적의 나머지는 잘릴것이다.).

```
Exception in thread "main" java.lang.IllegalArgumentException:
    Property 'movieFinder' is required for bean 'movieLister'.
```

여기에는 이 행위를 '(교체)switch on'하기 위해 필요한 Spring 설정의 작은 조각이 있다. 당신의 클래스에 'setter' 프라퍼티를 간단히 어노테이션 처리하는 것은 이 행위를 얻기 위해 충분하지 않다.

당신은 @Required 어노테이션을 알아차리고 적절하게 처리할수 있는 어떤것이 필요하다.

RequiredAnnotationBeanPostProcessor 클래스로 들어가라. 이것은 @Required를 인식하고 실제로 '필수 프라퍼티가 셋팅되지 않는다면 엉망이 되는'로직을 처리하는 Spring에 의해 특별히 제공되는 BeanPostProcessor 구현물이다. 이것은 설정하기 매우 쉽다. Spring XML설정에 다음 bean정의를 간단히 추가하라.

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

마지막으로, 다른 Annotation 타입을 찾기 위해 RequiredAnnotationBeanPostProcessor 클래스의 인스턴스를 설정할수 있다. 당신 스스로가 작성한 @Required 스타일의 어노테이션이 있다면 멋진일이다. 간단히 그것을 RequiredAnnotationBeanPostProcessor 정의에 추가하면 된다.

예제에서의 방법에 의해, @Mandatory라 불리는 속성을 정의하자. 당신은 @Mandatory를 인식하는 RequiredAnnotationBeanPostProcessor 인스턴스를 만들수 있다.

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor">
    <property name="requiredAnnotationType" value="your.company.package.Mandatory"/>
</bean>
```

25.3.2. Spring내 다른 @Annotations

어노테이션은 Spring도처에 여기서 언급하는 것보다 많이 사용된다. 이러한 어노테이션은 관련된 참조문서의 다음 부분에서 언급된다.

☒ Section 9.5.6, "@Transactional 사용하기"

☒ Section 6.8.1, "Using AspectJ to dependency inject domain objects with Spring"

☒ Section 6.2, "@AspectJ 지원"

25.4. 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.

☒ 필요한 파라미터로 나타내기. 이것은 생성자의 인자나 자바빈 프라퍼티를 통해 수행된다.

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 생성된 파일을 저장하기 위해 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.

25.5. 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.

25.5.1. 기초

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

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

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="txInterceptor" />
</bean>

<bean id="txInterceptor" class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager" />
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
      <property name="attributes" ref="attributes" />
    </bean>
  </property>
</bean>

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

여기의 기본적인 개념은 AOP장의 자동프록시에서 언급된것과 유사하다.

가장 중요한 bean정의는 자동 프록시(auto-proxy) 생성자와 advisor이다. 실질적인 bean이름은 중요하지 않다. 클래스가 중요하다.

org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator 클래스의 bean정의는 Advisor구현물에 대응되는 현재 factory내 모든 bean인스턴스에 자동적으로

advise("autoproxy")할것이다. 이 클래스는 속성에 대해 아무것도 모르지만 Advisor의 pointcut대응에 의존한다. pointcut은 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.

25.5.2. 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.

25.5.3. 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)
 */
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">
  <property name="attributes" ref="attributes" />
</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 class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
  <property name="customTargetSourceCreators">
    <list>
      <ref bean="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.

25.5.4. 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.)

25.6. 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
 *
 * @@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 request, HttpServletResponse response) 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
 *
 * @@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 request, HttpServletResponse response) 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

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.

25.7. 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.

25.8. 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.

Appendix A. XML 스키마-기반 설정

A.1. 설정

이 추가된 부분은 Spring 2.0에서 소개된 XML스키마 기반의 설정을 상세하게 언급한다.

> DTD 는 지원하는가.?
>
> 오래된 DTD스타일을 사용하여 Spring 설정파일을 작성하는 것은 여전히 완벽히 지원된다.
>
> 만약 당신이 Spring XML설정파일을 작성하기 위해 새로운 XML스키마-기반 접근법을 사용하는데 앞장선다면 문제가 되는것은 아무것도 없을것이다. 당신이 잃게되는 것은 좀더 간결하고 명백한 설정을 가질수 있는 기회이다. XML설정이 DTD- 나 스키마-기반인것에는 상관없이, 마지막에는 컨테이너내 같은 객체 모델이 될것이다.(즉 하나 이상의 BeanDefinition 인스턴스)

XML 스키마 기반의 설정파일로 이동하기 위한 중심이 되는 동기는 Spring XML설정을 좀더 쉽게 만들고자 함이다. 'classic' <bean/>-기반의 접근법도 좋지만, 설정이라는 개념에서 오버헤드라는 비용을 치루어야 한다.

Spring IoC 컨테이너 관점에서, 모든것은 bean이다. 모든것이 bean이라면 모든것은 정확히 같은 형태로 처리될수 있기 때문에 그것은 Spring IoC컨테이너를 위해 굉장히 좋은 소식이다. 어쨌든 이것은 개발자의 관점에서는 참이 되지 않는다. Spring XML설정파일내 정의된 객체는 모두 일반적이지는 않다. 언제나 각각의 bean은 몇가지 특정 설정의 단계를 요구한다.

Spring 2.0의 새로운 XML스키마-기반의 설정은 이러한 이슈를 할당한다. <bean/> 요소는 여전히 존재한다. 당신이 원한다면, 오직 <bean/> 요소를 사용하는 Spring XML설정을 계속 사용할수 있다. 어쨌든 새로운 XML스키마-기반 설정은 Spring XML설정파일을 특히 읽기 명백하도록 해준다. 추가적으로 이것은 bean정의의 목적을 표시하도록 해준다.

기억해야하는 핵심사항은 예를 들어, AOP, collection, transaction, Mule와 같은 이기종 프레임워크와의 통합과같이 새로운 사용자정의 태그가 하부조직(infrastructure)이나 통합(integration) bean을 위한 가장 잘 작동한다는 것이다. 현재 존재하는 bean태그는 DAO, 서비스 레이어 객체, validator등등 애플리케이션 특유의 bean에 가장 적합하다.

아래의 예제는 Spring 2.0의 XML스키마지원이 좋은 생각이라고 당신을 납득시킬것이다. 또한 이 새로운 설정기법이 전체적으로 사용자정의가능하고 확장가능하다는 것을 노트하라. 이것은 당신의 애플리케이션 도메인을 좀더 잘 표현할 자체적인 도메인 속성의 설정 태그를 작성할수 있다는 것을 의미한다. 이 절차는 Appendix B, 확장가능한 XML제작에서 다루어진다.

A.2. XML 스키마-기반 설정

A.2.1. 스키마 참조하기

DTD-스타일에서 새로운 XML스키마-스타일로의 전환을 위해, 당신은 다음을 변경할 필요가 있다.

```
<!-- DTD-style -->
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<!-- <bean/> definitions here -->

</beans>
```

XML스키마-스타일에서 같은 파일은.....

```
<!-- XML Schema-style -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

Note

'xsi:schemaLocation' 조각은 실제로 필수는 아니지만, 스키마의 local복사 참조에 포함될수 있다.(개발하는 동안 유용하다.).

위 Spring XML설정 조각은 다소 반복적 어구를 사용한다. 기존에 한것처럼 이것을 간단히 사용하고 <bean/> 정의를 작성하는 것을 지속할수 있다. 어쨌든, 전체적인 전환은 새로운 Spring 2.0 XML태그의 장점을 가져서 좀더 설정이 쉽다. Section A.2.2, "util 스키마" 는 당신이 몇가지 공통적인 유틸리티 태그를 사용하여 즉시 시작할수 있는 방법을 보여준다.

이 장의 나머지는 새로운 모든 태그의 사용을 적어도 한번은 하는 새로운 Spring XML스키마 기반의 설정의 예제를 보여준다. 형태는 before 과 after 스타일을 따른다. 여기서 before는 이전것을 보여주고 after는 새로운 XML스키마 기반의 스타일을 보여준다.

A.2.2. util 스키마

첫번째는 util 태그이다. 이름이 뜻하는 것처럼, util태그는 설정집합, 참조상수와 같은 utility 설정이슈를 공통으로 다룬다.

util 스키마내 태그를 사용하기 위해, Spring XML설정파일의 가장 위에 다음의 머릿말을 둘 필요가 있다. 아래의 텍스트는 정확한 스키마를 참조하기 때문에 util 명명공간내 태그는 사용가능하다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:util="http://www.springframework.org/schema/util"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

A.2.2.1. <util:constant/>

Before...

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

위 설정은 bean의 'isolation' 프라퍼티 값을 'java.sql.Connection.TRANSACTION_SERIALIZABLE'상수의 값으로 셋팅하는 Spring FactoryBean 구현물인 FieldRetrievingFactoryBean을 사용한다. 이것은 모두 잘되고 좋지만, 다소 장황하고 마지막 사용자에게 Spring의 내부 작동을 보여준다.

다음의 XML 스키마-기반의 버전은 좀더 간결하고 명백하게 개발자의 의도('inject this constant value')를 표시한다. 그리고 이것은 읽기가 더 좋다.

```
<bean id="..." class="...">
  <property name="isolation">
    <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
  </property>
</bean>
```

A.2.2.1.1. 필드값으로 부터 bean프라퍼티나 생성자의 인자를 셋팅하기

FieldRetrievingFactoryBean는 정적이거나(static) 비-정적인 필드값을 가져오는 FactoryBean이다. 이것은 대개 다른 bean을 위한 프라퍼티 값이나 생성자의 인자를 셋팅하기 위해 사용된 public static final 상수를 가져오기 위해 사용된다.

staticField 를 사용하여 static 필드를 나타내는 방법을 보여주는 예제를 아래에서 보라. property:

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

여기엔 또한 static 필드가 bean이름처럼 명시되는 편리한 사용형태가 있다.

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

이것은 bean id가 무엇인지 더이상 선택하지 않는다는 것을 의미한다(그래서 이것을 참조하는 다른 bean은 더 긴 이름을 사용할것이다.). 하지만 이 형태는 정의하기 위해 매우 간결하고 id는 bean참조를 명시할 필요가 없기 때문에 내부 bean처럼 사용하기 위해 매우 편리하다.

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

FieldRetrievingFactoryBean를 위한 API문서에서 언급하는 것처럼 다른 bean의 비-정적인 필드에 접근하는

것이 가능하다.

프라퍼티나 생성자의 인자처럼 bean으로 enum값을 삽입하는 것은 Spring에서 쉬운일이다. Spring내부(또는 FieldRetrievingFactoryBean와 같은 클래스에 대해)에 대해 어떤것을 하거나 알지 않는다. enum값을 쉽게 삽입하는 방법을 보여주는 예제를 보자. 이것은 JDK 5의 enum을 살펴보는 것이다.

```
package javax.persistence;

        public enum PersistenceContextType {

        TRANSACTION,
        EXTENDED

        }
```

지금 PersistenceContextType타입의 setter를 생각해보자.

```
package example;

        public class Client {

        private PersistenceContextType persistenceContextType;

        public void setPersistenceContextType(PersistenceContextType type) {
                this.persistenceContextType = type;
        }
        }
```

.. 그리고 일치하는 bean정의는

```
<bean class="example.Client">
        <property name="persistenceContextType" value="TRANSACTION" />
        </bean>
```

이것은 전통적인 방식의 type-safe모방의 enum(JDK 1.4와 JDK1.3)에 잘 작동한다. Spring은 string프라퍼티 값을 enum클래스의 상수에 일치시키도록 자동으로 시도할것이다.

A.2.2.2. <util:property-path/>

Before...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

위 설정은 'testBean'bean의 'age' 프라퍼티와 같은 값을 가지는 'testBean.age'라고 불리는 bean(int 타입의)을 생성하기 위한 Spring FactoryBean 구현물인 PropertyPathFactoryBean를 사용한다.

After...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

<property-path/>태그의 'path' 속성값은 'beanName.beanProperty'폼을 따른다.

A.2.2.2.1. bean 프라퍼티나 생성자의 인자를 셋팅하기 위해 <util:property-path/> 사용하기

PropertyPathFactoryBean은 대상 객체의 프라퍼티 경로를 평가하는 FactoryBean이다. 대상 객체는 직접적으로나 bean이름을 통해 명시될수 있다. 이 값은 프라퍼티 값이나 생성자의 인자로 다른 bean정의내 사용될수 있다.

이것은 경로가 이름에 의해, 다른 bean에 대해 사용되는 예제이다.

```
// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </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"/>
  <property name="propertyPath" value="spouse.age"/>
</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"/>
    </bean>
  </property>
  <property name="propertyPath" value="age"/>
</bean>
```

여기엔 또한 bean이름이 프라퍼티 경로인 단축형태가 존재한다.

```
<!-- 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 이름으로 더이상 선택하지 않는다는 것을 의미한다. 이것에 대한 참조는 경로인 같은 id를 사용할것이다. 물론, 내부 bean처럼 사용된다면, 이것에 대해 전혀 참조할 필요가 없다.

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

결과타입은 실제 정의내 명시적으로 셋팅될것이다. 이것은 대개의 경우를 위해 필요하지 않다. 하지만 몇가지 경우에 사용될수 있다. 이 기능의 좀더 많은 정보를 위해 Javadoc를 보라.

A.2.2.3. <util:properties/>

Before...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

위 설정은 제공된 Resource위치로부터 로그된 값을 가지는 java.util.Properties인스턴스를 구체적으로 나타내기 위한 Spring FactoryBean구현물인 PropertiesFactoryBean을 사용한다.

After...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>
```

A.2.2.4. <util:list/>

Before...

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
      <list>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </list>
  </property>
</bean>
```

위 설정은 제공된 'sourceList'로부터 얻어진 값을 가지고 초기화된 java.util.List인스턴스를 생성하기 위해 Spring FactoryBean 구현물인 ListFactoryBean을 사용한다.

After...

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
```

```
<util:list id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:list>
```

<util:list/> 요소의 'list-class' 속성의 사용을 통해 인스턴스화되고 활성화될 List의 정확한 타입을 명시적으로 제어할수 있다. 예를 들어, 인스턴스화되는 java.util.LinkedList가 필요하다면, 우리는 다음의 설정을 사용할수 있다.

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>jackshaftoe@vagabond.org</value>
    <value>eliza@thinkingmanscrumpet.org</value>
    <value>vanhoek@pirate.org</value>
    <value>d'Arcachon@nemesis.org</value>
</util:list>
```

'list-class' 속성이 제공되지 않는다면, List 구현물은 컨테이너에 의해 선택될것이다.

마지막으로, 당신은 <util:list/> 요소의 'merge' 속성을 사용하여 병합(merge)행위를 제어할수 있다. collection병합은 Section 3.3.3.4.1, "Collection 병합" 에서 좀더 상세하게 언급된다.

A.2.2.5. <util:map/>

Before...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
  <property name="sourceMap">
      <map>
        <entry key="pechorin" value="pechorin@hero.org"/>
        <entry key="raskolnikov" value="raskolnikov@slums.org"/>
        <entry key="stavrogin" value="stavrogin@gov.org"/>
        <entry key="porfiry" value="porfiry@gov.org"/>
      </map>
  </property>
</bean>
```

위 설정은 제공된 'sourceMap'로부터 얻어진 key-value쌍으로 초기화된 java.util.Map 인스턴스를 생성하기 위해 Spring FactoryBean 구현물인 MapFactoryBean을 사용한다.

After...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

<util:map/> 요소의 'map-class' 속성의 사용을 통해 인스턴스화되고 활성화될 Map의 정확한 타입을 명시적으로 제어할수 있다. 예를 들어, 인스턴스화되는 java.util.TreeMap가 필요하다면, 다음의 설정을 사용할수 있다.

```
<util:map id="emails" map-class="java.util.TreeMap">
```

```
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

'map-class' 속성이 제공되지 않는다면, Map 구현물은 컨테이너에 의해 선택될것이다.

마지막으로, <util:map/> 요소의 'merge' 속성을 사용하여 병합행위를 제어할수 있다. collection병합은 Section 3.3.3.4.1, "Collection 병합" 에서 상세히 언급된다.

A.2.2.6. <util:set/>

Before...

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
  <property name="sourceSet">
      <set>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </set>
  </property>
</bean>
```

위 설정은 제공된 'sourceSet'으로부터 얻어진 값으로 초기화된 java.util.Set 인스턴스를 생성하기 위해 Spring FactoryBean 구현물인 SetFactoryBean을 사용한다.

After...

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

<util:set/> 요소의 'set-class' 속성의 사용을 통해 인스턴스화되고 활성화될 Set의 정확한 타입을 명시적으로 제어할수 있다. 예를 들어, 인스턴스화되는 java.util.TreeSet가 필요하다면, 다음의 설정을 사용할수 있다.

```
<util:set id="emails" set-class="java.util.TreeSet">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

'set-class' 속성이 제공되지 않는다면, Set 구현물은 컨테이너에 의해 선택될것이다.

마지막으로, <util:set/> 요소의 'merge' 속성을 사용하여 병합행위를 제어할수 있다. collection병합은 Section 3.3.3.4.1, "Collection 병합" 에서 상세히 언급된다.

A.2.3. jee 스키마

jee 태그는 JNDI객체를 룩업하고 EJB참조를 정의하는 것과 같은 EE (Java Enterprise Edition)에 관련된 설정이슈를 다룬다.

jee 스키마에 태그를 사용하기 위해, Spring설정파일의 가장 상위에 다음의 서문을 둘 필요가 있다. 다음의 텍스트는 정확한 스키마를 참조해서 jee 명명공간내 태그는 사용가능하다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jee="http://www.springframework.org/schema/jee"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

A.2.3.1. <jee:jndi-lookup/> (simple)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource"/>
```

A.2.3.2. <jee:jndi-lookup/> (하나의 JNDI환경셋팅으로)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>foo=bar</jee:environment>
</jee:jndi-lookup>
```

A.2.3.3. <jee:jndi-lookup/> (다중 JNDI환경셋팅으로)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties format) -->
    <jee:environment>
        foo=bar
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

A.2.3.4. <jee:jndi-lookup/> (complex)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="cache" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="expectedType" value="com.myapp.DefaultFoo"/>
    <property name="proxyInterface" value="com.myapp.Foo"/>
</bean>
```

After...

```
<jee:jndi-lookup id="simple"
                 jndi-name="jdbc/MyDataSource"
                 cache="true"
                 resource-ref="true"
                 lookup-on-startup="false"
                 expected-type="com.myapp.DefaultFoo"
                 proxy-interface="com.myapp.Foo"/>
```

A.2.3.5. <jee:local-slsb/> (simple)

<jee:local-slsb/> 태그는 EJB 비상태유지 세션빈에 대한 참조를 설정한다.

Before...

```
<bean id="simple"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/RentalServiceBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

After...

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
    business-interface="com.foo.service.RentalService"/>
```

A.2.3.6. <jee:local-slsb/> (complex)

```
<bean id="complexLocalEjb"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/RentalServiceBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
</bean>
```

After...

```
<jee:local-slsb id="complexLocalEjb"
    jndi-name="ejb/RentalServiceBean"
    business-interface="com.foo.service.RentalService"
    cache-home="true"
    lookup-home-on-startup="true"
    resource-ref="true">
```

A.2.3.7. <jee:remote-slsb/>

<jee:remote-slsb/> 태그는 remote EJB 비상태유지 세션빈에 대한 참조를 설정한다.

Before...

```
<bean id="complexRemoteEjb"
      class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/MyRemoteBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
  <property name="homeInterface" value="com.foo.service.RentalService"/>
  <property name="refreshHomeOnConnectFailure" value="true"/>
</bean>
```

After...

```
<jee:remote-slsb id="complexRemoteEjb"
    jndi-name="ejb/MyRemoteBean"
    business-interface="com.foo.service.RentalService"
    cache-home="true"
    lookup-home-on-startup="true"
    resource-ref="true"
    home-interface="com.foo.service.RentalService"
    refresh-home-on-connect-failure="true">
```

A.2.4. lang 스키마

lang태그는 Spring컨테이너내에서 bean처럼 JRuby나 Groovy와 같은 동적언어내 작성되는 노출된 객체를 다룬다.

이러한 태그(와 동적언어지원)는 Chapter 24, 동적 언어 지원에서 완벽하게 다루어진다. 이 지원과 lang 태그 자체에 대한 상세정보를 위해서는 그 장을 참조하라.

lang 스키마내 태그를 사용하기 위해, Spring XML설정파일의 가장 상위에 다음의 서문을 둘 필요가 있다. 다음의 텍스트는 정확한 스키마를 참조해서 lang 명명공간내 태그는 당신에게 사용가능하다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

A.2.5. tx (트랜잭션) 스키마

tx태그는 트랜잭션을 위한 Spring의 완벽한 지원을 위한 모든 bean의 설정을 다룬다. 이러한 태그는 Chapter 9, 트랜잭션 관리에서 상세히 다루어진다.

Tip

Spring배포판에 포함된 'spring-tx-2.0.xsd' 파일을 보는것을 강력히 권한다. 이 파일은 Spring의 트랜잭션 설정을 위한 XML스키마이고 tx 명명공간내 다양한 모든 태그를 다룬다. 이 파일은 그때마다 문서화되었고 정보는 여기서 반복되지 않는다.

tx 스키마에서 태그를 사용하기 위해, Spring XML설정파일의 가장 위에 다음의 서문을 둘 필요가 있다. 다음의 텍스트가 정확한 스키마를 참조해서 tx 명명공간내 태그는 사용가능하다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

Note

tx 명명공간내 태그를 사용할때 (Spring내 선언적인 트랜잭션 지원이 AOP를 사용하여 구현되었기 때문에)당신은 aop 명명공간으로부터 태그를 사용할것이다. 위 XML은 aop 스키마를 참조할 필요가 있는 관련된 줄을 포함해서 aop 명명공간내 태그는 사용가능할것이다.

A.2.6. aop 스키마

aop 태그는 Spring내 모든 AOP 설정을 다룬다. 이것은 Spring 자체의 프록시-기반 AOP프레임워크와 AspectJ AOP 프레임워크와의 통합을 포함한다. 이 태그는 Chapter 6, Spring을 이용한 Aspect 지향 프로그래밍에서 상세히 다루어진다.

aop 스키마내 태그를 사용하기 위해, 당신은 Spring XML 설정파일의 가장 위에 다음의 서문을 둘 필요가 있다. 다음의 텍스트는 정확한 스키마를 참조해서 aop 명명공간내 태그는 사용가능하다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:aop="http://www.springframework.org/schema/aop"
            xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

A.2.7. tool 스키마

tool 태그는 사용자정의 설정 요소에 툴성격의 메타데이타를 추가하길 원할때 사용하기 위한 것이다. 이 메타데이타는 이 메타데이타를 인식하는 툴에 의해 소비될수 있고 툴은 하고자 하는 것(유효성 체크및 기타)이 무엇이든지 잘 수행할수 있다.

tool 태그는 Spring의 이 버전에 문서화되지 않았기 때문에 현재는 리뷰를 제공한다. 당신이 이기종 툴 업체거나 이 리뷰 프로세스에 기여하고자 한다면, Spring메일링 리스트에 메일을 보내달라. 현재 지원되는 tool 태그는 Spring소스 배포판의 'src/org/springframework/beans/factory/xml' 디렉토리의 'spring-tool-2.0.xsd' 파일에서 찾을수 있다.

A.2.8. beans 스키마

마지막이지만 적어도 우리는 beans 스키마내 태그를 가진다. 프레임워크가 발달한 이후 Spring에서 bean을 가지는 같은 태그가 있다. 예를 들어 Section 3.3.3, "상세화된 bean프라퍼티와 생성자의 인자" (그리고 전체 장에서는 말할것도 없이)에서 포괄적으로 다루어지기 때문에 beans 스키마내 다양한 태그는 여기서 보여지지 않는다.

Spring 2.0내 bean태그 자체에 새로운 것은 임의의 bean메타데이타에 대한 생각이다. Spring 2.0에서, <bean/> XML정의에 키/값의 쌍을 추가하지 않거나 여러개를 추가하는 것이 가능하다. 어떤것이라도 발생한다면, 이 추가적인 메타데이타는 자체적인 사용자정의 로직에 수행한다(그래서 Appendix B, 확장가능한 XML제작에 언급된 사용자정의 태그를 작성하면 사용된다).

둘러싸는 <bean/>의 문구내 <meta/> 태그의 예제를 아래에서 보라.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="foo" class="x.y.Foo">
        <meta key="cacheName" value="foo"/>
        <property name="name" value="Rick"/>
    </bean>
```

```
</beans>
```

위 예제의 경우, bean정의를 소비할 몇가지 로직과 제공된 메타데이타를 사용하여 몇가지 캐싱 내부구조를 셋팅한다고 가정할것이다.

A.3. 당신의 IDE 셋업하기

이 마지막 부분은 Spring의 XML 스키마-기반의 설정파일의 편집을 좀더 쉽게 하도록 해주는 많은 수의 Java IDE를 셋업하는 단계를 보여준다. 만일 당신이 선호하는 Java IDE 나 편집기가 다음의 문서화된 부분에 포함되지 않았다면, Spring 프레임워크 JIRA 이슈 관리시스템에 이슈화하고 다음 릴리즈에 기능화될 예제를 달라.

A.3.1. eclipse 셋업하기

다음 단계는 XSD 를 인식하도록 Eclipse 를 셋업하는 것을 보여준다. 다음 단계는 당신이 이미 eclipse 프로젝트를 열었다고 가정한다.

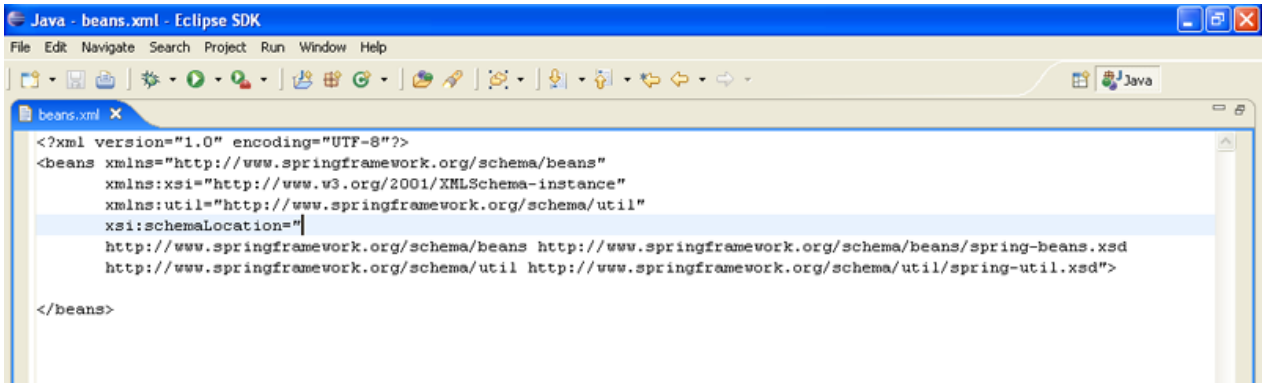
Note

다음 단계는 eclipse 3.2를 사용하여 만들었다. 셋업방법은 아마 이전버전과 차후버전에도 같거나 유사할것이다.

1. 1 단계

   새 XML파일을 생성한다. 당신이 원하는 파일명을 지정한다. 아래의 예제에서는, 파일명이 'context.xml'이다. 다음의 텍스트를 복사하고 파일에 넣는다. 그러면 다음의 화면과 같을것이다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-2.0.xsd">

</beans>
```



2. 2 단계

위 화면에서 본것처럼(어떤 플러그인으로 설치한 사용자정의된 eclipse를 사용하지 않는한) XML파일은 일반 텍스트로 처리될것이다. eclipse에는 특별한 XML편집 지원이 없다. 그래서 요소나 속성의 문법적인 하이라이트가 없다. 이것을 할당하기 위해, 당신은 eclipse를 위한 XML편집기를 설치해야할것이다.

Table A.1. eclipse XML 편집기

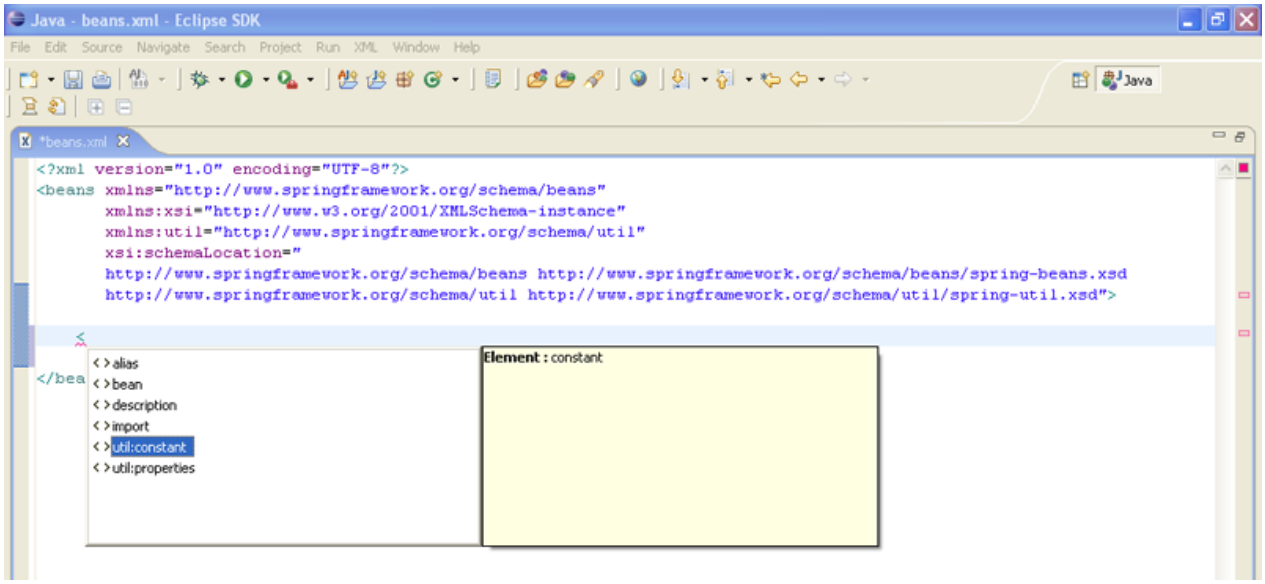
XML Editor	Link
Eclipse Web Tools Platform (WTP)	http://www.eclipse.org/webtools/
eclipse XML플러그인 목록	http://eclipse-plugins.2y.net/eclipse/plugins.jsp?cate

> 기여된 문서...
>
> eclipse XML편집기를 설정하는 방법을 보여주는 것을 환영한다. 이러한 기여는 Spring 프레임워크 JIRA 이슈 관리시스템를 통해 관리되는 것이 가장 좋고 다음 릴리즈에서 기능화될것이다.

불행히도, eclipse를 위한 표준적인 XML편집기가 없기 때문에, eclipse내 XML 스키마 지원을 설정하는 방법을 보여줄 그 이상의 단계를 없다. 각각의 XML 편집기는 전용으로 할당된 부분에서 요구될것이고 이것은 eclipse XML편집기 문서가 아닌 Spring 참조문서이다. 당신은 XML편집기 플러그인을 사용한 문서를 읽을것이고 자체적으로 설정할것이다.

3. 3 단계

eclipse 를 위한 WTP(Web Tools Platform)를 사용한다면, WTP 플랫폼의 XML편집기를 사용하여 Spring XML설정파일을 열수 있다. 아래의 화면에서 보듯이, 당신은 태그를 자동완성하기 위한 IDE레벨의 지원을 얻게 된다. “[WTP] rocks!”



A.3.2. IntelliJ IDEA 셋업하기

다음 단계는 XSD를 인식하도록 IntelliJ IDEA를 셋업하는 방법을 보여준다. 그 방법은 매우 간단하다.

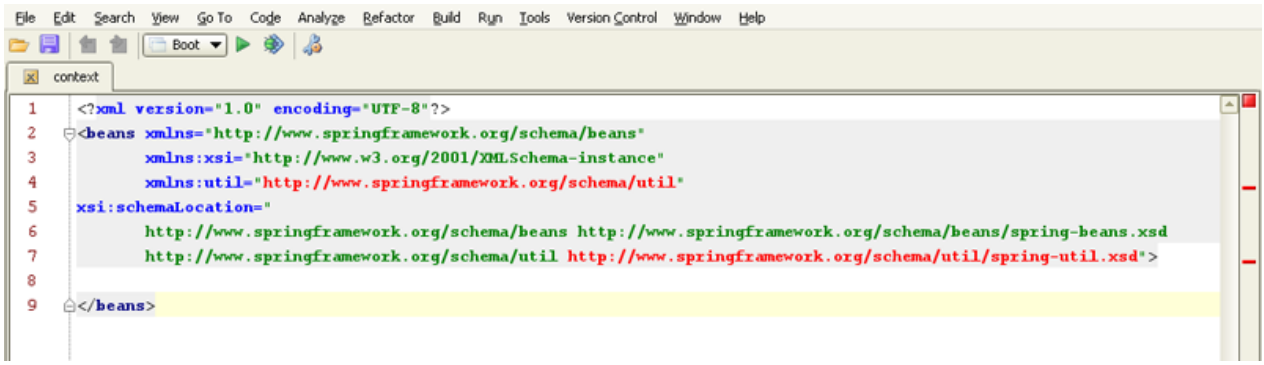
다음 단계는 당신이 이미 IDEA 프로젝트를 열었다고 가정한다.

다른 Spring XSD파일을 참조하는 IDEA를 셋업하기 위해 반복하라.

1. 1 단계

   새로운 XML파일을 생성한다. 당신이 원하는 파일명을 사용할수 있다. 아래의 예제에서, 파일명은 'context.xml'이다. 다음 텍스트를 복사하고 파일에 넣자. 그러면 다음 화면과 같을것이다.

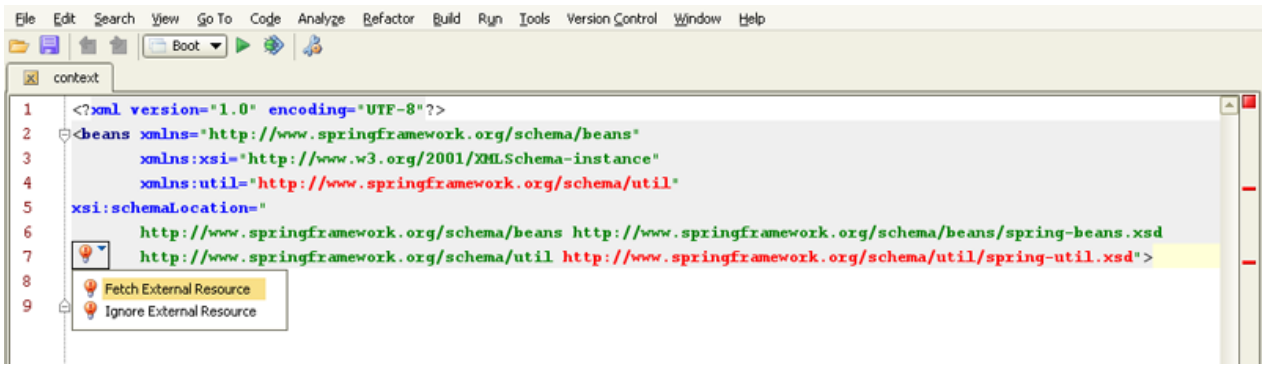
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-2.0.xsd">

</beans>
```



2. 2 단계

   위 화면에서 본것처럼, XML파일은 많은 수의 빨간색의 에러 표시자를 가진다. 이것을 고치기 위해, IDEA는 참조된 XSD명명공간의 위치를 인식해야만 한다.

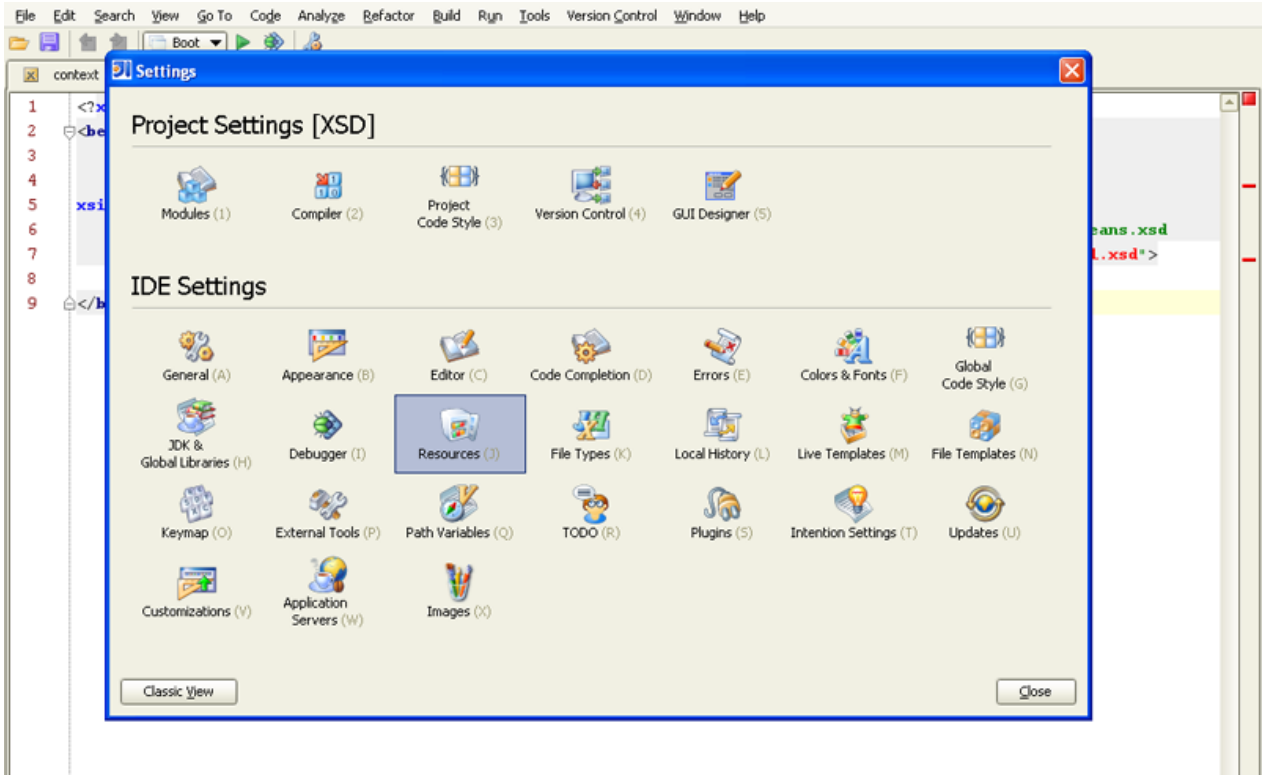
   이것을 하기 위해, (아래의 화면에서 보듯이)빨간색 영역으로 커서를 옮기고 Alt-Enter 기 조합을 누른다. 그리고 나서 팝업이 외부자권을 가져오기 위해 활성화될때 Enter 키를 다시 누른다.



3. 3 단계

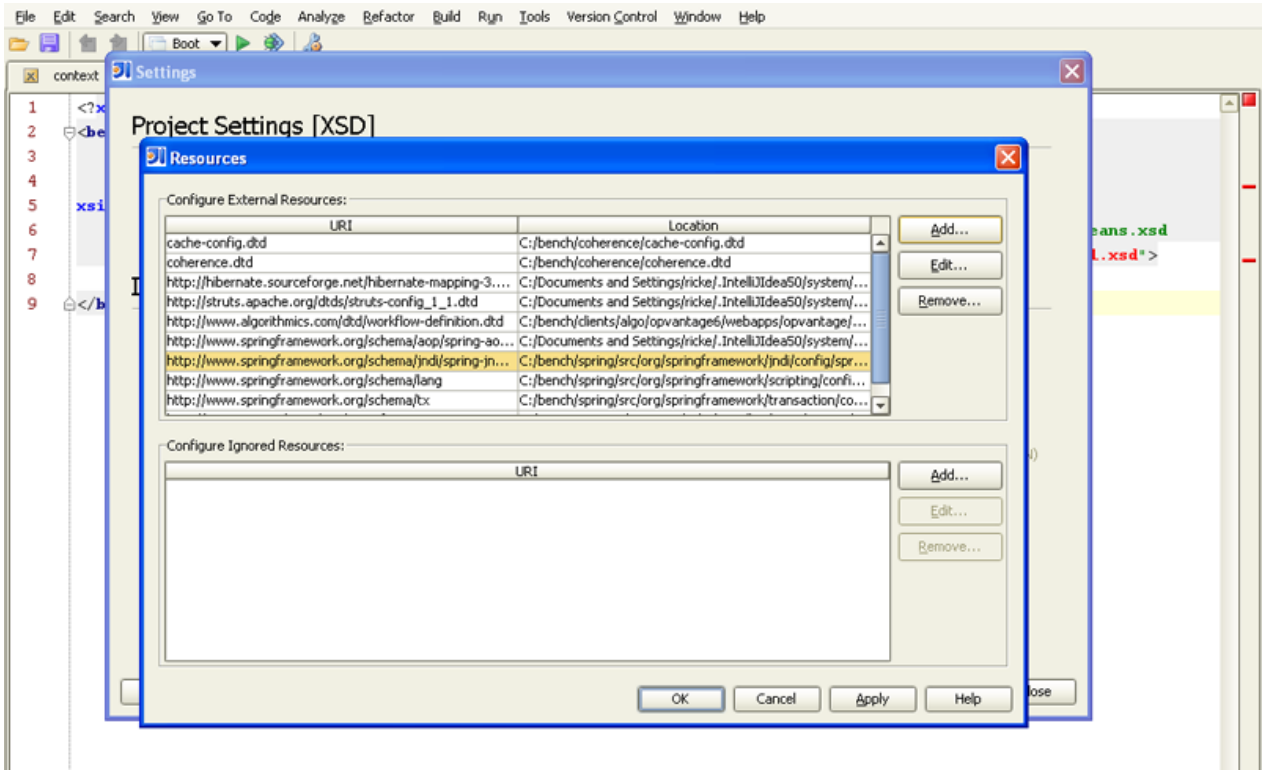
   만약 외부 자원을 가져올수 없다면(아마도 실질적인 인터넷 연결이 사용가능하지 않을때), 관련

XSD파일의 로컬 복사본을 참조하도록 자원을 수동으로 설정할수 있다. 간단히 'Settings' 다이얼로그를 열고(Ctrl-A-S 키조합을 사용하거나 'File|Settings' 메뉴를 통해) 'Resources' 버튼을 클릭한다.



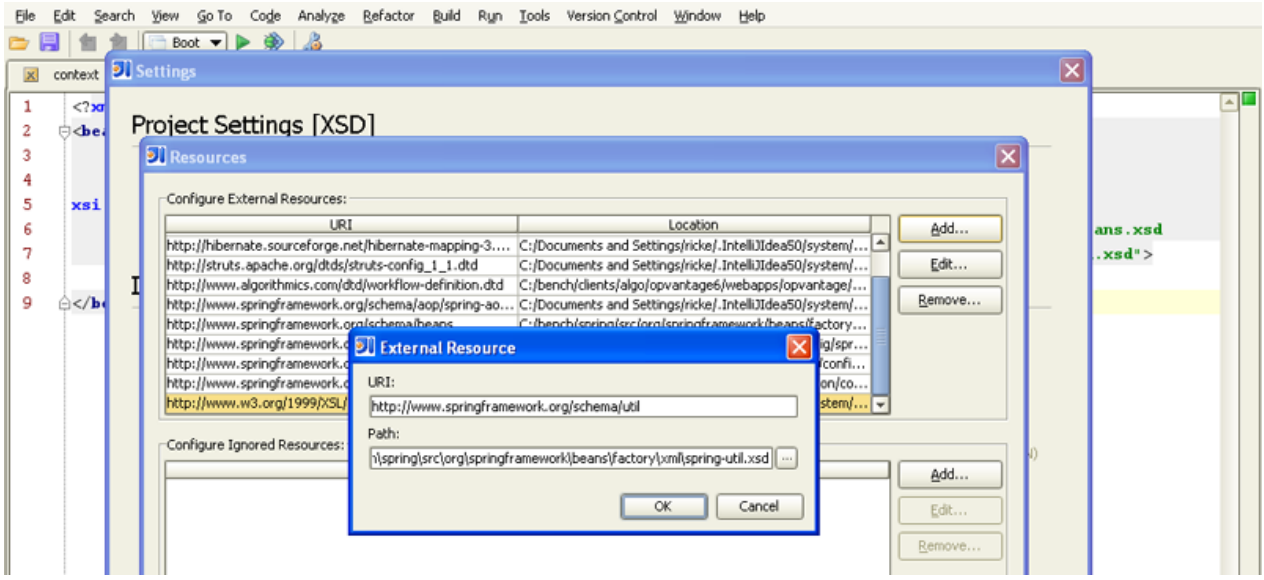
4. 4 단계

다음 화면에서 볼수 있는것처럼, 이것은 util 스키마 파일의 로컬 복사본을 명시적으로 참조하도록 추가하는 것을 허용하는 다이얼로그를 가져올것이다(당신은 Spring배포판의 'src' 디렉토리의 다양한 Spring XSD파일을 찾을수 있을것이다.).



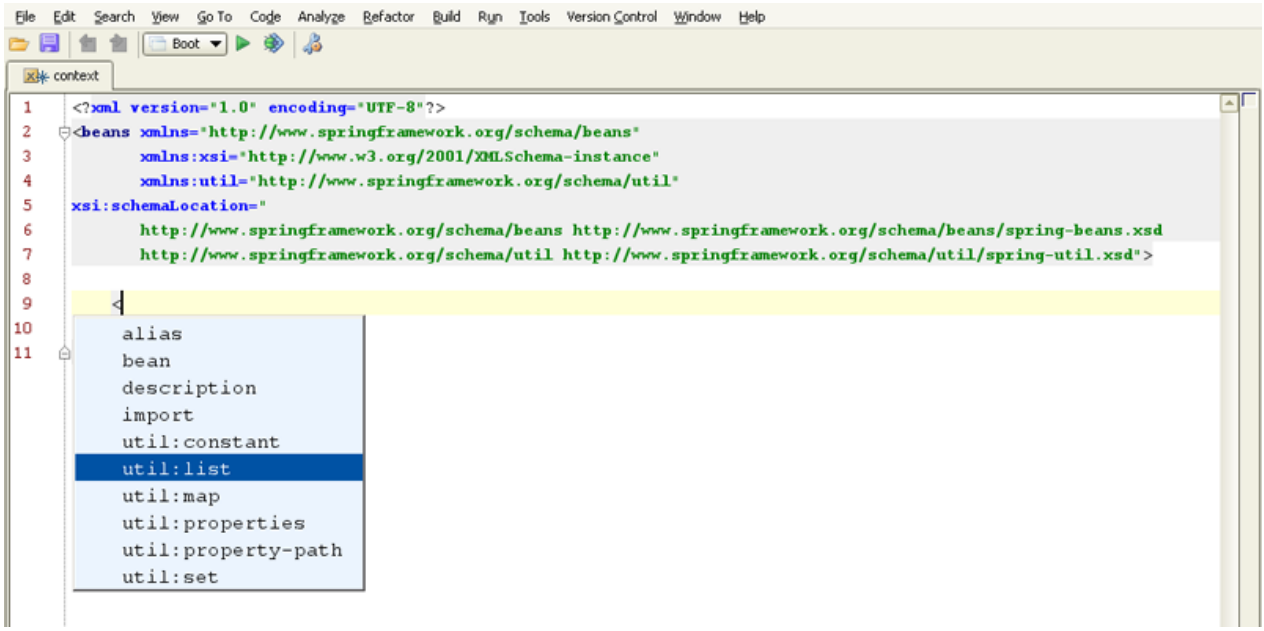
5. 5 단계

'Add' 버튼을 클릭하는 것은 관련 XSD파일을 위한 경로를 가진 명명공간 URI 관련시키도록 하는 다른 다이얼로그를 가져올것이다. 다음 화면에서 볼수 있듯이, 'http://www.springframework.org/schema/util' 명명공간은 'C:₩bench₩spring₩src₩org₩springframework₩beans₩factory₩xml₩spring-util-2.0.xsd' 파일 자원과 관련된다.



6. 6 단계

'OK' 버튼을 클릭하여 내포된 다이얼로그를 빠져나가는 것은 메인 편집창으로 돌아갈것이다. 다음 화면에서 볼수 있듯이, 문법적 에러 표시자는 사라진다. 지금 편집창에서 '<' 문자를 타이핑하면 util 명명공간으로 부터 가져온 태그를 모두 포함하는 드랍다운 형태의 박스를 보게될것이다.



A.3.3. 통합 이슈

이 마지막 부분은 Spring 2.0설정을 위해 XSD스타일을 사용하는 것으로 교체할때 발생할수 있는 통합 이슈를 언급한다.

이 부분은 지금 시점에서는 매우 작다. 이것은 Spring사용자에게 편리하도록 Spring문서에 포함된다. 그래서 몇가지 환경에서 XSD스타일로 교체할때 이슈가 발생한다면, 그에 대한 방법을 보기 위해 이 부분을 참조할수 있다.

A.3.3.1. Resin v.3 애플리케이션 서버의 XML파일 에러

Spring 2.0 XML설정을 위해 XSD스타일을 사용하고 Caucho의 Resin애플리케이션서버의 v.3에 배치한다면, 애플리케이션 서버 시작시 XSD을 인식하는 파서가 Spring에서 사용가능하도록 셋팅할 필요가 있을것이다.

좀더 많은 정보를 위해 Caucho Resin웹사이트의 다음 자원을 참조하라.

http://www.caucho.com/resin-3.0/xml/jaxp.xtp#xerces

Appendix B. 확장가능한 XML제작

B.1. 소개

2.0버전 이후, Spring은 bean을 정의하고 설정하기 위한 기본적인 Spring XML포맷에 대해 스키마-기반의 확장을 위한 기법을 제공한다. 이 부분은 자체적인 사용자정의 bean정의 파서를 작성하고 Spring IoC컨테이너로 파서를 통합하는 방법을 다룬다.

스키마를 인식하는 XML편집기를 사용하여 설정파일 작업을 돕기 위해, Spring의 확장가능한 XML설정 기법은 XML스키마에 기초를 둔다. Spring의 최근 XML설정 확장에 친숙하지 않다면 Appendix A, XML 스키마-기반 설정를 먼저보라.

다음의 간단한 XML스키마 작성 절차로 새로운 XML설정 확장을 생성할수 있다. 새로운 XML설정 확장을 생성하는 것은 다음 XML스키마의 간단한 처리를 통해 가능하다. NamespaceHandler 구현물을 코딩하고, 전용 프라퍼티 파일에 하나 이상의 BeanDefinitionParser 인스턴스를 코딩하고 NamespaceHandler와 스키마를 등록한다. 다음은 이러한 단계의 각각에 대한 상세설명이다. 예제에서, 우리는 Spring IoC컨테이너내 SimpleDateFormat타입의 객체를 직접 설정하는 것을 허용하는 XML확장(사용자정의 XML요소)을 생성할것이다.

B.2. 스키마 작성하기

Spring의 IoC컨테이너를 사용하기 위한 XML설정 확장을 생성하는 것은 확장을 설명하는 XML스키마를 작성하는 것을 시작한다. 다음은 우리가 SimpleDateFormat 객체를 설정하기 위해 사용할 스키마이다. 강조된 줄은 확인될(컨테이너에서 bean 식별자로 사용될 id 속성을 가진다는 것을 의미한다.) 모든 태그를 위한 확장의 기본사항을 포함한다.

```
#### myns.xsd (inside package org/springframework/samples/xml)

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.springframework.org/schema/myns"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:beans="http://www.springframework.org/schema/beans"
    targetNamespace="http://www.mycompany.com/schema/myns"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

  <xsd:import namespace="http://www.springframework.org/schema/beans"/>

  <xsd:element name="dateformat">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="beans:identifiedType">
          <xsd:attribute name="lenient" type="xsd:boolean"/>
          <xsd:attribute name="pattern" type="xsd:string" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

위 스키마는 myns:dateformat 설정 지시를 사용하여 XML 애플리케이션 컨텍스트 파일내 직접

SimpleDateFormat 객체를 설정하기 위해 사용될것이다. 위에서 노트된것처럼, id 속성은(이 경우) SimpleDateFormat bean을 위한 bean식별자로 사용된다.

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

우리가 내불구조를 나타내는 클래스를 생성한 뒤에, XML의 위 일부분은 다음의 XML일부와 정확하게 같을것이다. 반면에, 우리는 컨테이너에 프라퍼티 세트의 쌍을 가지고 SimpleDateFormat 타입의 dateFormat으로 식별되는 bean을 생성한다.

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

Note

설정 포맷을 생성하기 위한 스키마-기반의 접근법은 스키마를 인지하는 XML편집기를 가진 IDE와 통합할수 있다. 작성된 스키마를 사용하여, 예들 들어 당신은 반복내 정의된 여러개의 설정 옵션들간의 사용자선택을 하기 위해 자동완성을 사용할수 있다.

B.3. NamespaceHandler코딩하기

스키마에 추가적으로, 우리는 Spring이 설정파일을 파싱하는 동안 만나게 되는 이 특정 명명공간의 모든 요소를 파싱하는 NamespaceHandler가 필요하다. 우리의 경우 NamespaceHandler는 myns:dateformat 요소를 파싱하는 것을 관리한다.

NamespaceHandler 인터페이스는 오직 3가지 메소드 기능을 가진다는 면에서 매우 간단하다.

- ☒ init() - NamespaceHandler의 초기화와 핸들러가 사용되기 전 Spring에 의해 호출될것이다.
- ☒ BeanDefinition parse(Element element, ParserContext parserContext) - Spring이 가장 상위레벨(bean정의내 내포되거나 다른 명명공간이 아닌)의 요소를 만날때 호출된다. 이 메소드는 bean정의를 등록하고/하거나 bean정의를 반환할수 있다.
- ☒ BeanDefinitionHolder decorate(Node element, BeanDefinitionHolder definition, ParserContext parserContext) - Spring이 다른 명명공간(예를 들어 Spring명명공간내부)의 속성이나 내포된 요소를 만날때 호출된다. 하나 이상의 bean정의의 장식(decoration)은 Spring 2.0에 포함된 특별한 범위(scope - 범위에 대한 좀더 많은 정보를 위해서는 Section 3.5, “Bean scopes” 를 보라.)에서 사용된다. 우리는 간단한 예제를 강조하여 시작할것이다.

전체 명명공간을 위해 자체적인 NamespaceHandler를 코딩하는 것이 완전히 가능하다 하더라도, 이것은 종종 Spring XML설정파일내 각각의 가장 상위레벨의 XML요소는 하나의 bean정의의 결과가 된다. (우리의 경우, myns:dateformat 요소가 있는 곳은 SimpleDateFormat bean 정의의 결과가 된다). Spring은 이러한 시나리오를 지원하는 2개의 편리한 클래스를 제공한다. 이 예제에서, 우리는 NamespaceHandlerSupport 클래스인 가장 자주 사용되는 편리한 클래슬 사용할 것이다.

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
```

```
        registerBeanDefinitionParser("dateformat",
                new SimpleDateFormatBeanDefinitionParser());
    }
}
```

B.4. BeanDefinitionParser 코딩하기

볼수 있는것처럼, 위에서 보여지는 명명공간 핸들러는 BeanDefinitionParsers라고 불리는 것을 등록한다. 이 경우 BeanDefinitionParsers는 Spring 명명공간 핸들러가 이 특정 bean정의 파서에 맵핑되는 타입(이 경우 dateformat)의 XML요소를 만난다면 고려될것이다. 반면에, BeanDefinitionParser는 스키마내 정의된 가장 상위레벨 구별되는 XML요소를 파싱하기 위한 책임이 있다. 파서에서, 우리는 XML요소(그리고 하위요소)와 ParserContext에 접근할것이다. 후자는 아래 예제에서 수행한것처럼 예들 들어 BeanDefinitionRegistry에 대한 참조를 얻기 위해 사용될수 있다.

```
package org.springframework.samples.xml;

import java.text.SimpleDateFormat;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.BeanDefinitionReaderUtils;
import org.springframework.beans.factory.support.RootBeanDefinition;
import org.springframework.beans.factory.xml.BeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

public class SimpleDateFormatBeanDefinitionParser implements BeanDefinitionParser {

   public BeanDefinition parse(Element element, ParserContext parserContext) {

      // create a RootBeanDefinition that will serve as configuration
      // holder for the 'pattern' attribute and the 'lenient' attribute
      RootBeanDefinition beanDef = new RootBeanDefinition();
      beanDef.setBeanClass(SimpleDateFormat.class);

      // never null since the schema requires it
      String pattern = element.getAttribute("pattern");
      beanDef.getConstructorArgumentValues().addGenericArgumentValue(pattern);

      String lenientString = element.getAttribute("lenient");
      if (StringUtils.hasText(lenientString)) {
         // won't throw exception if validation is turned on (boolean type set in schema)
         beanDef.getPropertyValues().addPropertyValue("lenient", new Boolean(lenientString));
      }

      // retrieve the ID attribute that will serve as the bean identifier in the context
      String id = element.getAttribute("id");

      // create a bean definition holder to be able to register the
      // bean definition with the bean definition registry
      // (obtained through the ParserContext)
      BeanDefinitionHolder holder = new BeanDefinitionHolder(beanDef, id);

      // register the BeanDefinitionHolder (which contains the bean definition)
      // with the BeanDefinitionRegistry
      BeanDefinitionReaderUtils.registerBeanDefinition(holder, parserContext.getRegistry());

      return beanDef;
   }
}
```

Note

이 예제에서, 우리는 BeanDefinition를 정의하고 이것을 BeanDefinitionRegistry에 등록한다. 당신은 등록기로 bean정의를 등록하거나 parse() 메소드로부터 bean정의를 반환할필요가 없다. 당신에게 주어진 정보와 ParserContext에서 당신이 원하는 것이 무엇이든 수행하는것에는 자유롭다.

ParserContext 는 다음의 프라퍼티에 접근한다.

- ☒ readerContext - bean factory와 내포된 명명공간을 분석하기 위해 선택적으로 사용될수 있는 NamespaceHandlerResolver에 접근한다.
- ☒ parserDelegate - 설정파일을 파싱하는 컴포넌트를 제어. 대개 당신은 이것에 접근할 필요가 없다.
- ☒ registry - 새롭게 생성된 BeanDefinition인스턴스를 등록하는 것을 허용하는 BeanDefinitionRegistry
- ☒ nested - 처리된 XML요소가 외부(outer) bean정의인지 아닌지 표시(반면에, 이것은 전통적인 내부(inner) bean과 유사하게 정의된다.)

B.5. 핸들러와 스키마 등록하기

우리는 사용자정의 XML스키마를 파싱하는 것을 다룰 NamespaceHandler 와 BeanDefinitionParser를 구현한다. 우리는 다음의 결과물을 가진다.

- ☒ org.springframework.samples.xml.MyNamespaceHandler - 하나 이상의 BeanDefinitionParser 인스턴스를 등록할 명명공간 핸들러
- ☒ org.springframework.samples.xml.SimpleDateFormatBeanDefinitionParser - dateformat 타입의 요소를 파싱하기 위한 사용된 명명공간 핸들러
- ☒ org/springframework/samples/xml/myns.xsd - Spring설정파일내 사용될 실제 스키마(이 파일은 우리가 나중에 볼 명명공간 핸들러와 파서 클래스와 나란히, 클래스패스에 둘 필요가 있다.)

우리가 마지막으로 해야할 필요가 있는 것은 두개의 특별한 목적을 가진 프라퍼티 파일에 이것을 등록하여 사용할 준비가 된 명명공간을 얻는것이다. 이러한 프라퍼티 파일은 META-INF에 두거나 JAR파일내 바이너리 클래스와 나란히 배포될수 있다. 일단 클래스패스에서 프라퍼티 파일을 찾으면 Spring은 새로운 명명공간과 핸들러를 자동으로 가질것이다.

B.5.1. META-INF/spring.handlers

spring.handlers라고 불리는 프라퍼티 파일은 명명공간 핸들러 클래스에 대한 XML스키마 URI의 맵핑을 포함한다. 예제에서, 우리는 다음을 언급할 필요가 있다.

```
http￦://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler
```

B.5.2. META-INF/spring.schemas

spring.schemas라고 불리는 프라퍼티 파일은 클래스패스 자원에 대한 XML스키마 위치의 맵핑을 포함한다(xsi:schemaLocation 속성의 부분처럼 스키마를 사용하는 XML파일내 스키마 선언과 나란히 언급된). 이 파일은 Spring을 스키마 파일을 가져오기 위해 인터넷 접속을 요구하는 디폴트 EntityResolver를 사용하는 것으로부터 막을필요가 있다. 당신이 프라퍼티 파일내 맵핑을 언급한다면, Spring은 스키마(이 경우 'org.springframework.samples.xml'패키지의 'myns.xsd')를 위해 클래스패스를 검색할것이다.

```
http₩://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd
```

Appendix C. spring-beans_2_0.dtd

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

<!--
	Spring XML Beans DTD
	Authors: Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop

	이것은 Spring BeanFactory가 관리하고 XmlBeanDefinitionReader(DefaultXmlBeanDefinitionParser를 가진)가 읽는 자바빈 객체의 명명공간(names

	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 상위

	bean사이의 참조는 제안된다. 이를테면 같은 factory(또는 조상(ancestor) factory)내 다른 bean을 위한 참조를 위해 자바빈 프라퍼티나 생성자의 인

	bean참조의 대안처럼 "내부 bean정의"가 사용될수 있다. 내부 bean정의의 싱글톤 플래그는 효과적으로 무시된다. 내부 bean은 대개 익명 프로토타입

	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 2.0//EN"
		"http://www.springframework.org/dtd/spring-beans_2_0.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-autowire (no | byName | byType | constructor | autodetect) "no">
<!ATTLIST beans default-dependency-check (none | objects | simple | all) "none">
<!ATTLIST beans default-init-method CDATA #IMPLIED>
<!ATTLIST beans default-destroy-method CDATA #IMPLIED>
<!ATTLIST beans default-merge (true | false) "false">

<!--
	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 "mylmport.xml", "includes/mylmport.xml" 또는 "../mylmport.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 콤마 또는 세미콜론에 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". 특

        노트 : 이 속성은 자식 bean정의에 의해 상속되지 않을것이다. 나아가 이것은 견고한 bean정의마다 명시될 필요가 있다.
-->
<!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.

        노트 : 이 속성은 자식 bean정의에 의해 상속되지 않을것이다. 나아가, 이것은 견고한 bean정의마다 명시될 필요가 있다.
-->
<!ATTLIST bean singleton (true | false) "true">

<!--
        bean이 늦게(lazily) 초기화되는지에 대한 값.
        false라면 이것은 singletons의 초기화를 수행하는 bean factory에 의해 시작될때 인스턴스화될것이다.

        노트 : 이 속성은 자식 bean정의에 의해 상속되지 않을것이다. 나아가 이것은 견고한 bean정의마다 명시될 필요가 있다.
-->
<!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.

        노트 : 이 속성은 자식 bean정의에 의해 상속되지 않을것이다. 나아가 이것은 견고한 bean정의마다 명시될 필요가 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.

        노트 : 이 속성은 자식 bean정의에 의해 상속되지 않을것이다. 나아가 이것은 견고한 bean정의마다 명시될 필요가 있다.
-->
<!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.

        노트 : 이 속성은 자식 bean정의에 의해 상속되지 않을것이다. 나아가 이것은 견고한 bean정의마다 명시될 필요가 있다.
-->
<!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)*
```

```
)>

<!-- 부모/자식 bean을 사용할때 collection을 병합하는 것을 가능하게 하기/가능하지 않게 하기 -->
<!ATTLIST list merge (true | false | default) "default">


<!--
        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)*
)>

<!-- 부모/자식 bean을 사용할때 collection을 병합하는 것을 가능하게 하기/가능하지 않게 하기 -->
<!ATTLIST set merge (true | false | default) "default">


<!--
        Spring map은 문자열 key로 객체를 맵핑한다. map은 아마도 비어있을것이다.
-->
<!ELEMENT map (
        (entry)*
)>

<!-- 부모/자식 bean을 사용할때 collection을 병합하는 것을 가능하게 하기/가능하지 않게 하기 -->
<!ATTLIST map merge (true | false | default) "default">

<!--
        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)*
)>

<!-- 부모/자식 bean을 사용할때 collection을 병합하는 것을 가능하게 하기/가능하지 않게 하기 -->
<!ATTLIST props merge (true | false | default) "default">

<!--
	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>
```

Appendix D. spring.tld

D.1. Introduction

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). To help you implement views using Java Server Pages the Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages.

Please note that the various tags generated by this form tag library are compliant with the XHTML-1.0-Strict specification and attendant DTD.

This appendix describes the spring.tld tag library.

D.2. The bind tag

Provides BindStatus object for the given bind path. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.1. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
ignoreNestedPath	false	true

Attribute	Required?	Runtime Expression?
path	true	true

D.3. The escapeBody tag

⊠⊠·⊠½×⊠⊠ body³ ⊠¿⊠⊠⊠ Escape⊠⊠´⊠. Escapes its enclosed body content, applying HTML escaping and/or JavaScript escaping. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.2. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
javaScriptEscape	false	true

D.4. The hasBindErrors tag

Provides Errors instance in case of bind errors. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.3. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
name	true	true

D.5. The htmlEscape tag

☒☒☒☒ Æ☒☒☒☒☒ ¸☒ ☒§☒☒ HTML escape °ª☒☒ ¼☒Æ☒. web.xmlÆ☒☒☒³ ☒ "defaultHtmlEscape" context-param☒☒ ¿☒¹ ☒¶☒☒☒☒☒☒☒ ´☒.

Table D.4. Attributes

Attribute	Required?	Runtime Expression?
defaultHtmlEscape	true	true

D.6. The message tag

Retrieves the message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.5. Attributes

Attribute	Required?	Runtime Expression?
arguments	false	true
argumentSeparator	false	true
code	false	true
htmlEscape	false	true
javaScriptEscape	false	true
message	false	true
scope	false	true
text	false	true

Attribute	Required?	Runtime Expression?
var	false	true

D.7. The nestedPath tag

Sets a nested path to be used by the bind tag's path.

Table D.6. Attributes

Attribute	Required?	Runtime Expression?
path	true	true

D.8. The theme tag

Retrieves the theme message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.7. Attributes

Attribute	Required?	Runtime Expression?
arguments	false	true
code	false	true
htmlEscape	false	true
javaScriptEscape	false	true
scope	false	true

Attribute	Required?	Runtime Expression?
text	false	true
var	false	true

D.9. The transform tag

Provides transformation of variables to Strings, using an appropriate custom PropertyEditor from BindTag (can only be used inside BindTag). The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.8. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
scope	false	true
value	true	true
var	false	true

Appendix E. spring-form.tld

E.1. Introduction

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). To help you implement views using Java Server Pages the Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages.

Please note that the various tags generated by this form tag library are compliant with the XHTML-1.0-Strict specification and attendant DTD.

This appendix describes the spring-form.tld tag library.

E.2. The checkbox tag

Renders an HTML 'input' tag with type 'checkbox'.

Table E.1. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true

Attribute	Required?	Runtime Expression?
HTML Standard Attribute		
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute		
cssErrorClass	false	true
Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.		
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
dir	false	true
HTML Standard Attribute		
disabled	false	true
HTML Optional Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
onblur	false	true
HTML Event Attribute		
onchange	false	true
HTML Event Attribute		

Attribute	Required?	Runtime Expression?
onclick	false	true
HTML Event Attribute		
ondblclick	false	true
HTML Event Attribute		
onfocus	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true
HTML Event Attribute		
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
path	true	true
Path to property for data binding		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		
value	false	true
HTML Optional Attribute		

E.3. The errors tag

Renders field errors in an HTML 'span' tag.

Table E.2. Attributes

Attribute	Required?	Runtime Expression?
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute		
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
delimiter	false	true
Delimiter for displaying multiple error messages. Defaults to the br tag.		
dir	false	true
HTML Standard Attribute		

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
onclick	false	true
HTML Event Attribute		
ondblclick	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
path	false	true
Path to errors object for data binding		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		

E.4. The form tag

Renders an HTML 'form' tag and exposes a binding path to inner tags for binding.

Table E.3. Attributes

Attribute	Required?	Runtime Expression?
action	false	true
HTML Required Attribute		
commandName	false	true
Name of the attribute under which the command name is exposed. Defaults to 'command'.		
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute		

Attribute	Required?	Runtime Expression?
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
dir	false	true
HTML Standard Attribute		
enctype	false	true
HTML Optional Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
method	false	true
HTML Optional Attribute		
name	false	true
HTML Optional Attribute		
onclick	false	true
HTML Event Attribute		
ondblclick	false	true
HTML Event Attribute		
onkeydown	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true
HTML Event Attribute		
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
onreset	false	true
HTML Event Attribute		
onsubmit	false	true
HTML Event Attribute		
title	false	true
HTML Standard Attribute		

E.5. The hidden tag

Renders an HTML 'input' tag with type 'hidden' using the bound value.

Table E.4. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
path	true	true
Path to property for data binding		

E.6. The input tag

Renders an HTML 'input' tag with type 'text' using the bound value.

Table E.5. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
HTML Standard Attribute		
alt	false	true
HTML Optional Attribute		
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute		
cssErrorClass	false	true

Attribute	Required?	Runtime Expression?
Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.		
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
dir	false	true
HTML Standard Attribute		
disabled	false	true
HTML Optional Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
maxlength	false	true
HTML Optional Attribute		
onblur	false	true
HTML Event Attribute		
onchange	false	true
HTML Event Attribute		
onclick	false	true
HTML Event Attribute		

Attribute	Required?	Runtime Expression?
ondblclick	false	true
HTML Event Attribute		
onfocus	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true
HTML Event Attribute		
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
onselect	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
path	true	true
Path to property for data binding		
readonly	false	true
HTML Optional Attribute		
size	false	true
HTML Optional Attribute		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		

E.7. The label tag

Renders a form field label in an HTML 'label' tag.

Table E.6. Attributes

Attribute	Required?	Runtime Expression?
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute.		
cssErrorClass	false	true
Equivalent to "class" - HTML Optional Attribute. Used only when errors are present.		
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		

Attribute	Required?	Runtime Expression?
dir	false	true
HTML Standard Attribute		
for	false	true
HTML Standard Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
onclick	false	true
HTML Event Attribute		
ondblclick	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true
HTML Event Attribute		
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
path	true	true
Path to errors object for data binding		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		

E.8. The option tag

Renders an HTML 'option'. Sets 'selected' as appropriate based on bound value.

Table E.7. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		

Attribute	Required?	Runtime Expression?
label	false	true
HTML Optional Attribute		
value	true	true
HTML Optional Attribute		

E.9. The options tag

Renders a list of HTML 'option' tags. Sets 'selected' as appropriate based on bound value.

Table E.8. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
itemLabel	false	true
Name of the property mapped to the inner text of the 'option' tag		
items	true	true
The Collection, Map or array of objects used to generate the inner 'option' tags		
itemValue	false	true
Name of the property mapped to 'value' attribute of the 'option' tag		

E.10. The password tag

Renders an HTML 'input' tag with type 'password' using the bound value.

Table E.9. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
HTML Standard Attribute		
alt	false	true
HTML Optional Attribute		
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute		
cssErrorClass	false	true
Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.		
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
dir	false	true
HTML Standard Attribute		
disabled	false	true
HTML Optional Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
maxlength	false	true

Attribute	Required?	Runtime Expression?
HTML Optional Attribute		
onblur	false	true
HTML Event Attribute		
onchange	false	true
HTML Event Attribute		
onclick	false	true
HTML Event Attribute		
ondblclick	false	true
HTML Event Attribute		
onfocus	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		

Attribute	Required?	Runtime Expression?
onmouseout	false	true
HTML Event Attribute		
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
onselect	false	true
HTML Event Attribute		
path	true	true
Path to property for data binding		
readonly	false	true
HTML Optional Attribute		
size	false	true
HTML Optional Attribute		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		

E.11. The radiobutton tag

Renders an HTML 'input' tag with type 'radio'.

Table E.10. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
HTML Standard Attribute		
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute		
cssErrorClass	false	true
Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.		
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
dir	false	true
HTML Standard Attribute		
disabled	false	true
HTML Optional Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
onblur	false	true
HTML Event Attribute		

Attribute	Required?	Runtime Expression?
onchange	false	true
HTML Event Attribute		
onclick	false	true
HTML Event Attribute		
ondblclick	false	true
HTML Event Attribute		
onfocus	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true
HTML Event Attribute		
onmouseover	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
path	true	true
Path to property for data binding		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		
value	false	true
HTML Optional Attribute		

E.12. The select tag

Renders an HTML 'select' element. Supports databinding to the selected option.

Table E.11. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
HTML Standard Attribute		
cssClass	false	true
Equivalent to "class" - HTML Optional Attribute		
cssErrorClass	false	true
Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.		

Attribute	Required?	Runtime Expression?
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
dir	false	true
HTML Standard Attribute		
disabled	false	true
HTML Optional Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
itemLabel	false	true
Name of the property mapped to the inner text of the 'option' tag		
items	false	true
The Collection, Map or array of objects used to generate the inner 'option' tags		
itemValue	false	true
Name of the property mapped to 'value' attribute of the 'option' tag		
lang	false	true
HTML Standard Attribute		
multiple	false	true
HTML Optional Attribute		
onblur	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
onchange	false	true
HTML Event Attribute		
onclick	false	true
HTML Event Attribute		
ondblclick	false	true
HTML Event Attribute		
onfocus	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true
HTML Event Attribute		

Attribute	Required?	Runtime Expression?
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
path	true	true
Path to property for data binding		
size	false	true
HTML Optional Attribute		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		

E.13. The textarea tag

Renders an HTML 'textarea'.

Table E.12. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
HTML Standard Attribute		
cols	false	true
HTML Required Attribute		
cssClass	false	true

Attribute	Required?	Runtime Expression?
Equivalent to "class" - HTML Optional Attribute		
cssErrorClass	false	true
Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.		
cssStyle	false	true
Equivalent to "style" - HTML Optional Attribute		
dir	false	true
HTML Standard Attribute		
disabled	false	true
HTML Optional Attribute		
htmlEscape	false	true
Enable/disable HTML escaping of rendered values.		
id	false	true
HTML Standard Attribute		
lang	false	true
HTML Standard Attribute		
onblur	false	true
HTML Event Attribute		
onchange	false	true
HTML Event Attribute		
onclick	false	true
HTML Event Attribute		

Attribute	Required?	Runtime Expression?
ondblclick	false	true
HTML Event Attribute		
onfocus	false	true
HTML Event Attribute		
onkeydown	false	true
HTML Event Attribute		
onkeypress	false	true
HTML Event Attribute		
onkeyup	false	true
HTML Event Attribute		
onmousedown	false	true
HTML Event Attribute		
onmousemove	false	true
HTML Event Attribute		
onmouseout	false	true
HTML Event Attribute		
onmouseover	false	true
HTML Event Attribute		
onmouseup	false	true
HTML Event Attribute		
onselect	false	true

Attribute	Required?	Runtime Expression?
HTML Event Attribute		
path	true	true
Path to property for data binding		
rows	false	true
HTML Required Attribute		
tabindex	false	true
HTML Standard Attribute		
title	false	true
HTML Standard Attribute		