**CHAPTER1**

**1.1 자바 개발 간소화**

POJO(Plain Old Java Object) : 간단한 형태에도 불구하고 매우 강력하다.

DI, AOP를 이용하여 조립할 수 있다.

DI(Dependency Injection)는 주입을 통해서 결합도를 낮춰준다.

애플리케이션 컨텍스트(application context)는 애플리케이션을 구성하는 객체의 생성과 와이어링을 전적으로 책임진다.

애스팩트 지향 프로그래밍은 애플리케이션 전체에 걸쳐 사용되는 기능을 재사용할 수 있는 컴포넌트에 담는다.

애스팩트 지향 프로그래밍은 소프트웨어 시스템 내부의 관심사들을 서로 분리하는 기술이라고 설명한다.

템플릿을 이용하여 자바 개발의 복잡성을 줄일 수 있다. 템플릿 내부에서 처리된다.

**1.2 빈을 담는 그릇, 컨테이너**

스프링 기반 애플리케이션에서는 스프링 컨테이너 안에서 객체가 태어나고, 자라고, 소멸한다,

스프링 컨테이너는 크게 두 가지로 분류된다.

1) 빈 팩토리(org.springframework.beans.factory.BeanFactory) 이는 DI에 대한 기본적인 지원을 제공하는 가장 단순한 컨테이너다.

2) 애플리케이션 컨텍스트(org.springframework.context.ApplicationContext) 빈 팩토리를 확장해 프로퍼티(property) 파일에 텍스트 메시지를 읽고 해당 이벤트 리스너에 대한 애플리케이션 이벤트 발생 같은 애플리케이션 프레임워크 서비스를 제공하는 컨테이너다.

빈 팩토리는 지나치게 저수준의 기능을 제공하기 때문에 애플리케이션 컨텍스트를 선호한다.

**1.2.1 또 하나의 컨테이너, 애플리케이션 컨텍스트**

-AnnotationConfigApplicationContext – 하나 이상의 자바 기반 설정 클래스에서 스프링 애플리케이션 컨텍스트를 로드한다.

-AnnotationConfigWebApplicationContext – 하나 이상의 자바 기반 설정 클래스에서 스프링 웹 애플리케이션 컨텍스트를 로드한다.

-ClassPathXmlApplicationContext – 클래스패스(classpath)에 위치한 XML 파일에서 컨텍스트 정의 내용을 로드한다.

-FileSystemXmlApplicationContext – 파일 시스템에서, 즉 파일 경로로 지정된 XML 파일에서 컨텍스트 정의 내용을 로드한다.

-XmlWebApplicationContext – 웹 애플리케이션에 포함된 XML 파일에서 컨텍스트 정의 내용을 로드한다.

**1.2.2 빈의 일생**

보통의 자바 애플리케이션에서 빈의 생명 주기는 매우 단순하다.

자바의 new 키워드를 이용해 빈을 인스턴스화하고 이를 바로 사용한다. 빈이 더 이상 사용되지 않으면 가비지 컬렉션 후보가 되어 언젠가는 사라질 것이다.

반면 스프링 컨테이너 내에서 빈의 생명 주기는 좀더 정교 하다.

**BeanFactory 컨테이너 내에서 빈이 갖는 구동 생명주기를 보여 준다.**

1. 스프링이 빈을 인스턴스화 한다.

2. 스프링이 값과 빈의 레퍼런스를 빈의 프로퍼티에 주입한다.

3. 빈이 BeanNameAware를 구현하면 스프링이 빈의 ID를 setBeanName() 메소드에 넘긴다.

4. 빈이 BeanFactoryAware를 구현하면 setBeanFactory() 메소드를 호출하여 빈 팩토리 자체를 넘긴다.

5. 빈이 ApplicationContextAware를 구현하면 스프링이 setApplicationContext() 메소드를 호출하고 둘러싼 애플리케이션 컨텍스트에 대한 참조를 넘긴다.

6. 빈이 BeanPostProcessor 인터페이스를 구현하면 스프링은 postProcessBeforeInitialization() 메소드를 호출한다.

7. 빈이 InitializingBean 인터페이스를 구현하면 스프링은 afterPropertiesSet() 메소드를 호출한다. 마찬가지로 빈이 init-method와 함께 선언됐으면 지정한 초기화 메소드가 호출된다.

8. 빈이 BeanPostProcessor를 구현하면 스프링은 postProcessAfterInitialization() 메소드를 호출한다.

9. 이 상태가 되면 빈은 애플리케이션에서 사용할 준비가 된 것이며, 애플리케이션 컨텍스트가 소멸될 때까지 애플리케이션 컨텍스트에 남아 있다.

10. 빈이 DiposableBean 인터페이스를 구현하면 스프링은 destroy() 메소드를 호출한다. 마찬가지로 빈이 destroy-method와 함께 선언됐으면 지정된 메소드가 호출된다.

**1.3 스프링 현황**



4.0은 20개의 서로 다른 모듈로 이루어져 있다. 그리고 6개의 기능 카테고리에 속하게 된다.

1) 데이터 액세스와 통합

2) 웹과 리모팅

3) 애스펙트 지향 프로그래밍

4) 인스트루멘테이션

5) 코어 스프링 컨테이너

6) 테스팅

**코어 스프링 컨테이너**

스프링 프레임워크의 핵심은 스프링 애플리케이션의 빈의 생성, 설정 그리고 처리 방법을 관리하는 컨테이너다.

모든 스프링의 모듈은 코어 컨테이너 위에 구축된다.

**스프링의 AOP 모듈**

스프링은 AOP 모듈을 통해 애스펙트 지향 프로그래밍을 풍부하게 지원한다.

**데이터 액세스와 통합**

JDBC를 이용하면 반복 코드가 생기는데, 스프링의 JDBC와 데이터 액세스 객체(DAO) 모듈은 이렇게 반복되는 코드를 추상화하므로 간단하게 만들 수 있고, 데이터베이스 리소스를 닫지 않아서 발생할 수 있는 문제를 예방한다. 또한, 이 모듈에는 여러 종류의 데이터베이스 서버가 제공하는 오류 메시지에 의미 있는 예외 계층이 추가되어 있다.

스프링의 ORM 모듈은 JDBC보다 객체 관계 매핑(ORM) 도구를 선호하는 사람들을 위한 것이다.스프링은 고유한 ORM 솔루션을 구현하지 않고, 하이버네이트, 자바 퍼시스턴스, iBatis 등 널리 사용되는 ORM 프레임워크와의 연결 고리를 제공한다.

**웹과 리모팅**

스프링이 다양한 유명 MVC 프레임워크와 잘 통합되기는 하지만, 웹과 리모팅 모듈에는 애플리케이션의 웹 계층에서 결합도를 낮추는 MVC 프레임워크가 별도로 만들어져 있다.

사용자 접촉 웹 애플리케이션뿐만 아니라 다른 애플리케이션과 상호작용하는 애플리케이션을 개발하기 위한 다양한 리모팅 옵션도 제공한다. 그리고 스프링은 REST API에 대해 최고의 지원을 제공한다.

**인스트루멘테이션**

스프링의 인스트루멘테이션 모듈은 JVM에 에이전트를 추가하는 기능을 제공한다. 정확히는 톰캣용 위빙 에이전트를 제공하는데, 톰캣은 클래스로더에 의해 로드되는 클래스 파일을 변환한다.

**테스팅**

개발자가 작성하는 테스트의 중요성을 인식하여 스프링은 스프링 애플리케이션 테스트에 전념하는 모듈을 제공한다.

통합 테스트의 경우, 이 모듈은 스프링 애플리케이션 컨텍스트에서 빈을 로드하고 이 컨텍스트에 있는 빈과의 작업을 지원한다.

**1.4 스프링의 새로운 기능**

스프링 프레임워크는 세 개의 중요한 릴리스(3.1, 3.2, 4.0)를 선보였고, 각각의 릴리스는 새로운 특징과 개선으로 편리한 애플리케이션 개발을 가져왔다.

**1.4.1 스프링 3.1에서 새로워진 기능**

3.1에서는 여러 가지 유용한 새 기능과 개선이 있었으며, 이 중 대부분이 설정을 간소화하고 개선하는데 초점이 맞춰졌다.

-다양한 환경에서 서로 다른 설정을 선택하는 일반적인 문제를 다루기 위해 환경 프로파일을 도입했다.

-스프링에도 차츰 선언적 캐싱 지원이 쓰이게 되었다.

-새로운 c-네임스페이스 덕분에 생성자 주입이 가능해졌다.

-스프링의 JPA 지원이 개선되었다.

스프링 MVC에 대한 부분 또한 강화되었다.

-애트리뷰트 모델에 대한 경로 변수의 자동 바인딩이 추가되었다.

-요청을 위한 Accept와 Content-Type 헤더에 매칭하기 위한 @RequestMapping의 produces와 consumes 애트리뷰트를 포함한다.

-@RequestPart 애너테이션으로 핸들러 메소드 파라미터에 대한 다중 요청 부분의 바인딩이 가능하다.

**1.4.2 스프링 3.2에서 새로워진 기능**

스프링 3.1이 설정 개선에 초점을 맞추고, 스프링 MVC를 포함한 기타 개선점이 적었던 반면, 스프링 3.2는 주로 스프링 MVC에 초점을 맞춘 릴리즈였다.

-스프링 3.2 컨트롤러는 서블릿 3의 비동기식 요청을 이용하여 요청 처리를 분리시키고, 분리된 스레드 내에서 처리되도록 한다.

-스프링 2.5 이후로 스프링 MVC 컨트롤러 테스트가 POJO처럼 용이해졌다고는 하지만, 스프링 3.2에서는 스프링 MVC 테스트 프레임워크가 포함됨으로써 컨트롤러에 대한 더욱 다양한 테스트 작성이 가능해졌다.

-ResponseEntityExceptionHandler 클래스는 DefaultHandlerExceptionResolver의 대안으로 사용되기 위해 추가되었다.

-매핑된 인터셉터들은 인터셉터 프로세싱으로부터 배제되기 위해 URL 패턴을 지원한다.

**MVC 말고도 다른 개선점들이 추가되었다. 3.2이 가장 흥미로운 새로운 특징을 보자.**

-@Autowired, @Value 그리고 @Bean 애너테이션은 커스텀 주입과 빈 선언 애너테이션을 생성하기 위한 메타-애너테이션으로 사용된다.

-DateTimeFormat 애너테이션은 더 이상 JodaTime에 강력한 의존성을 갖지 않는다.

-스프링의 선언적 캐싱 지원은 JCache 0.5를 위한 초기 지원을 가진다.

-날짜와 시간을 파싱하고 렌더링하는 글로벌 포맷을 정의한다.

-통합 테스트는 WebApplicationContext를 설정하고 로드한다.

-통합 테스트는 요청 범위와 세션 범위의 빈에 대하여 테스트한다.

**1.4.3 스프링 4.0에서 새로워진 기능**

스프링 4.0에는 다음과 같이 흥미롭고 새로운 기능들이 많이 있다.

-스프링은 이제 JSR-356의 웹 소켓을 위한 자바 API 지원을 포함하여 웹 소켓 프로그래밍을 지원한다.

-스프링 4.0은 자바8의 특징인 람다식을 지원하는 첫 번째 프레임워크 중 하나이다. 무엇보다도 읽기에 훨씬 깨끗하고 쉬운 콜백 인터페이스로 작업할 수 있다.

-JSR-310의 데이터 및 시간 API를 포함하는 자바8 지원을 통해 개발자들이 java.util.Date 또는 java.util.Calendar에서 제공되는 것보다 훨씬 풍부한 API로 날짜와 시간을 작업하는 기회를 제공받을 수 있다.

-조건부 빈 생성에 대한 일반적 지원이 추가되면서 개발자가 정의한 조건이 만족되었을 때만 빈이 생성되도록 선언할 수 있다.

-스프링 4.0이 포함하는 또 한 가지는 스프링의 RestTemplate에 대한 새로운 비동기식 구현으로, RestTemplate는 응답을 반환하고, 연산이 끝났을 때 콜백을 호출한다.

**CHAPTER2**

스프링을 사용하는 애플리케이션에서는 각 객체가 자신의 일을 하기 위해 필요한 다른 객체를 직접 찾거나 생성할 필요가 없다. 컨테이너가 협업할 객체에 대한 레퍼런스를 주기 때문이다.

애플리케이션 객체 간의 이러한 연관관계 형성 작업이 바로 종속객체 주입(DI) 개념의 핵심이며, 이를 보통 **와이어링** 이라고 한다.

**2.1. 스프링 설정 옵션 알아보기**

스프링 컨테이너는 애플리케이션 내에서의 빈 생성 및 DI를 통해, 그러한 객체 사이의 관계 조정에 책임을 지고 있다. 하지만 어떤 빈을 생성할지 그들을 어떻게 엮을지 스프링에게 말해주는 것은 개발자 책임이다.

세 가지 기본적인 와이어링 메커니즘을 제공한다.

-XML에서의 명시적 설정 (ref)

-자바에서의 명시적 설정 (@Configuration, @Bean)

-내재되어 있는 빈을 찾아 자동으로 와이어링하기 (Scan)

🡺선택은 개인 취향의 문제이며 마음에 드는 것으로 선택한다.

하지만 할 수만 있다면 자동 설정을 추천한다. 명시적인 설정이 적을수록 좋다.

자동으로 와이어링하기 > 자바에서의 명시적 설정 (JavaConfig) > XML

**2.2 자동으로 빈 와이어링하기**

명시적 와이어링 테크닉의 많은 쓰임새를 보겠지만, 사용의 용이성 측면에서 스프링 자동 설정보다 나은 것은 없다. 스프링이 자동으로 설정된다면 명시적 와이어링 빈을 건드릴 필요가 있을까?

**스프링은 두 가지 방법으로 오토와이어링을 수행한다.**

-컴포넌트 스캐닝 – 스프링은 애플리케이션 컨텍스트에서 생성되는 빈을 자동으로 발견한다.

-오토와이어링 – 스프링은 자동으로 빈 의존성을 충족시킨다.

🡺컴포넌트 스캐닝과 오토와이어링을 모두 사용하면 강력하고, 명시적 설정을 최소한으로 유지하는데 도움이 된다.

**2.2.1 발견 가능한 빈 만들기**

@Component  
public class SgtPeppers implements CompactDisc {  
 private String title = "Sgt. Pepper's Lonely Hearts Club Band";  
 private String artist = "The Beatles";  
  
 @Override  
 public void play() {  
 System.*out*.println("Playing " + title + " by " + artist);  
 }  
}

@Component는 클래스를 빈으로 만들어야 함을 스프링에 단서로 제공한다.

하지만 컴포넌트 스캐닝은 기본적으로 켜 있지는 않다.

다음 코드는 이를 가능케 할 최소한의 설정을 보여준다.

@Configuration  
@ComponentScan  
public class CDPlayerConfig {  
}

@ComponentScan은 설정 클래스로서 동일한 클래스를 기본 스캐닝한다.

따라서 CDPlayerConfig는 soundsystem 패키지에 있으므로 스프링은 그 패키지와 하위 패키지를 스캔하고 @Component로 애너테이트된 클래스를 찾는다.

위의 CompactDisc 클래스를 찾고 자동으로 스프링으로 빈을 만든다.

XML 설정을 통해서 컴포넌트 스캐닝을 활성화하려면 그때는 스프링의 컨텍스트 네임스페이스로부터 <comtext:component-scan> 요소를 사용한다.

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context-4.2.xsd">  
  
 <context:component-scan base-package="com.sjb.chapter2.soundsystem"/>  
</beans>

XML이 컴포넌트 스캐닝을 가능케 하는 옵션이지만, 이 책에서는 더 선호하는 자바 기반의 설정을 사용하는데 집중한다. XML이 더 스타일리시하긴 해도 <context:component-scan> 요소에는

@ComponentScan을 사용할 때 애트리뷰터 값과 사용할 애트리뷰트를 미러링 하는 하위 요소가 있다. (?)

@RunWith(SpringJUnit4ClassRunner.class) // 자동으로 생성되는 스프링 애플리케이션 컨텍스트를 갖는다.  
@ContextConfiguration(classes = CDPlayerConfig.class) // 설정을 로드한다. 설정 클래스는 @ComponentScan을 가지므로 CompactDisc도 포함한다.  
public class CDPlayerTest {  
  
 @Autowired  
 private CompactDisc cd;  
  
 @Test  
 public void cdShouldNotBeNull() {  
 *assertNotNull*(cd);  
 }  
}

CDPlayerTest는 테스트 시작 시 자동으로 생성되는 스프링 애플리케이션 컨텍스트를 가지는 스프링 **SpringJUnit4ClassRunner**을 이용한다.

@ContextConfiguration 애너테이션은 CDPlayerConfig 클래스를 통해서 설정을 로드한다.

@ComponentScan 한줄로 수많은 빈을 자동으로 생성하게 하는 것은 좋은 거래다.

**2.2.2 컴포넌트 스캔된 빈 명명하기**

@ComponentScan을 사용하면 빈에 ID가 명시적으로 주어지지 않아도 동일 패키지 밑에 클래스를 기본 스캔해서 유추되어 할당한다.

특히 구체적으로 빈은 클래스 명의 첫 글자를 소문자로 바꾼 ID를 가진다.

만약 빈에 다른 ID를 주고 싶다면 해야 할 일은 @Component 애너테이션에 원하는 ID를 값으로 넣어 주는 것이 전부다.

@Component("lonelyHeartsClub")  
public class SgtPeppers implements CompactDisc {

...  
}

빈에 이름을 부여하는 다른 방법으로는 Named 어노테이션을 사용한다.

(@Autowired 사용 가능한가?)

@Named("lonelyHeartsClub")  
public class SgtPeppers implements CompactDisc {

...  
}

하지만 @Component를 주로 사용한다.

**2.2.3 컴포넌트 스캐닝을 위한 베이스 패키지 세팅**

애트리뷰트 없이 @ComponentScan을 사용하면 컴포넌트 검색을 위한 베이스 패키지로서 설정 클래스 패키지가 기본이라는 의미다.

다른 패키지를 원한다면?

베이스 패키지를 명시적으로 세팅해야 하는 한 가지 공통 이유는 애플리케이션 코드와 분리하여 설정 코드를 패키지 안에서 보관할 수 있기 때문이다. 그 경우 기본 베이스 패키지를 이용하는 것은 불가능하다.

다음과 같이 @ComponentScan의 값 속성에 기술하면 된다.

@Configuration  
@ComponentScan(com.sjb.chapter2.soundsystem")  
public class CDPlayerConfig {  
}

이건 좀더 명확하게 basePackages를 사용한 것이다.

@Configuration  
@ComponentScan(basePackages = "com.sjb.chapter2.soundsystem")  
public class CDPlayerConfig {  
}

여러 개의 베이스 패키지를 지정하고 싶을 때 사용한다.

@Configuration  
@ComponentScan(basePackages = {"com.sjb.chapter2.soundsystem", "com.sjb.chapter2.soundsystem2"})  
public class CDPlayerConfig {  
}

단점으로 String 값으로 설정하기 때문에 타입 세이프하지 않다. 패키지 이름을 변경한다면 작성했던 베이스 패키지가 잘못될 수 있다.

패키지를 간단한 String 값으로 지정하지 않고, @ComponentScan에서 옵션으로 제공하는 패키지 내의 클래스나 인터페이스를 사용할 수 있다.

@Configuration  
@ComponentScan(basePackageClasses = {SgtPeppers.class})  
public class CDPlayerConfig {  
}

비록 basePackageClasses에 컴포넌트 클래스를 지정했지만, 스캔될 패키지 안의 비어 있는 마커 인터페이스 생성을 고려해야 한다.

SgtPeppers 빈처럼 애플리케이션의 모든 객체가 혼자이고 의존성이 없다면, 컴포넌트 스캔으로 충분하다. 그러나 많은 객체는 일을 처리하기 위해 다른 객체에 의존관계를 가진다. 의존성을 가지고 컴포넌트 스캔된 빈을 묶기 위한 방법이 필요하다.

이를 위해 자동 스프링 설정의 다른 측면인 오토와이어링이 필요하다.

@Component()  
public class SgtPeppers implements CompactDisc {  
 private String title = "Sgt. Pepper's Lonely Hearts Club Band";  
 private String artist = "The Beatles";  
  
 @Override  
 public void play() {  
 System.*out*.println("Playing " + title + " by " + artist);  
 }  
}

**2.2.4 오토와이어링되는 빈의 애너테이션**

오토와이어링은 스프링이 빈의 요구 사항과 매칭되는 애플리케이션 컨텍스트상에서 다른 빈을 찾아 **빈 간의 의존성을 자동으로 만족시키도록 하는 수단이다.**

CDPlayer 클래스의 생성자에 @Autowired가 걸려 있는데, 이 애너테이션은 스프링이 CDPlayer 빈을 생성할 때, 생성자를 통해서 인스턴스화하고 CompcatDisc에 대입 가능한 빈을 전달해 준다.

@Component  
public class CDPlayer implements MediaPlyer {  
 private CompactDisc cd;  
  
 @Autowired  
 public CDPlayer(CompactDisc cd) {  
 this.cd = cd;  
 }  
  
 public void play() {  
 cd.play();  
 }  
}

생성자 뿐만 아니라 프로퍼티 세터 메소드에도 사용된다.

insertDiscfmf 사용해도 동일하게 동작한다.

// 프러퍼티 세터  
@Autowired  
public void setCompactDisc(CompactDisc cd) {  
 this.cd = cd;  
}

@Autowired  
public void insertDisc(CompactDisc cd) {  
 this.cd = cd;  
}

🡺 어떤 메소드이든 스프링은 메소드 파라미터에 의존성을 가진다. 한 개의 빈이 일치하면 그 빈은 와이어링 된다.

매칭되는 빈이 없으면 애플리케이션 컨텍스트가 생성될 때 예외를 발생시킨다.

이를 피하기 위해 required 애트리뷰트를 false로 설정한다.

@Autowired(required = false)  
public CDPlayer(CompactDisc cd) {  
 this.cd = cd;  
}

🡺주의 깊게 생각해야 되는 것이 코드에서 널 체크하지 않으면 NullPointerException이 발생할 수 있다.

스프링 기반의 애너테이션을 사용하기 힘든 경우에는 대신 @inject 애너테이션 사용을 고려한다.

@Named  
public class CDPlayer {  
 private CompactDisc cd;  
  
 @Inject  
 public CDPlayer(CompactDisc cd) {  
 this.cd = cd;  
 }  
  
}

**2.3. 자바로 빈 와이어링하기**

타사 라이브러리의 컴포넌트를 애플리케이션으로 와이어하고자 한다. 그 라이브러리의 소스 코드를 가지고 있지 않으므로 클래스를 @Component와 @Autowired를 사용하여 애너테이트할 수 없다.

그 경우 명시적인 설정을 해야 한다. 명시적 설정에는 두 가지 선택 방법이 있다. 자바와 XML.

JavaConfig는 설정용 코드다. 어떠한 비즈니스 로직도 포함하지 않고, 비즈니스 로직이 있는 코드에 영향을 주지도 않는다.

JavaConfig는 애플리케이션 로직 중 비즈니스 로직 외의 다른 부분과는 분리된 패키지이며, 원래의 목적과 혼동되어 사용되지 않는다.

**2.3.3 JavaConfig 주입하기**

@Bean  
public CompactDisc sgtPeppers() {  
 return new SgtPeppers();  
}

CompactDisc는 sgtPeppers를 호출해서 생성되는 것처럼 보이지만, 항상 그렇진 않다.

sgtPeppers() 메소드는 @Bean으로 애너테이트되므로 스프링은 콜을 중간에 인터셉트하고, 메소드에 의해 만들어진 빈은 다시 만들어지지 않고 이미 만들어진 것을 리턴해 주는 것을 보장한다.

@Bean  
public CDPlayer cdPlayer() {  
 return new CDPlayer(sgtPeppers());  
}  
  
@Bean  
public CDPlayer anotherCDPlayer() {  
 return new CDPlayer(sgtPeppers());  
}

기본적으로 스프링의 모든 빈은 싱글톤이고, 두 번째 CDPlayer 빈에 대한 중복 인스턴스를 생성할 필요가 없다. 따라서 스프링은 sgtPeppers() 호출을 중간에 인터셉트하고, 반환 값이 스프링 자기 자신이 CompactDisc 빈을 만들기 위해 sgtPeppers()를 호출하였을 때 만들어진 스프링 빈임을 확인한다. 따라서 CDPlayer 빈은 SgtPeppers의 동일 인스턴스가 된다.

**2.4 빈을 XML로 와이어링하기**

지금까지 스프링이 어떻게 빈을 자동적으로 찾고 와이어링하는지 살펴 보았다.

<context:component-scan base-package="com.sjb.chapter2.soundsystem"/>

또는

@Configuration // 설정 클래스로 식별한다.  
@ComponentScan  
public class CDPlayerConfig {  
}

그리고 어떻게 사용하고, JavaConfig를 사용하여 빈을 명시적으로 와이어링하는지 알아보았다.

XML 설정이 어떻게 동작하는지를 이해하는데 목적을 두며, 새로 작성하는 스프링은 자동 설정과 JavaConfig를 사용하길 바란다.

**2.4.1 XML 설정 스펙 만들기**

<?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.xsd  
 http://www.springframework.org/schema/context">  
</beans>

스프링을 설정하기 위한 XML 요소는 XML 설정 파일의 프리앰블(preamble) 내에서 선언되는 여러 개의 XML 스키마(XSD) 파일에서 정의된다.

<bean id="compactDisc" class="com.sjb.chapter2.soundsystem.SgtPeppers"/>

JavaConfig를 사용할 때처럼, SgtPeppers의 인스턴스를 생성할 책임을 직접적으로 가질 필요는 없다는 것이다.

스프링이 <bean> 요소를 찾았을 때, 기본 생성자를 호출하여 SgtPeppers 빈을 만든다.

단점으로 스프링의 XML 설정은 참조할 자바 타입의 컴파일 타임 검증의 도움을 얻지 못한다.

(IDE를 사용하여 유효성 체크를 할 수 있다.)

**2.3.4 생성자 주입을 사용하여 빈 초기화 하기**

빈 레퍼런스를 사용한 생성자 주입 (보통 많이 쓰는 방법)

<bean id="compactDisc" class="com.sjb.chapter2.soundsystem.SgtPeppers"/>  
<bean id="cdPlayer" class="com.sjb.chapter2.soundsystem.CDPlayer">  
 <constructor-arg ref="compactDisc"/>  
</bean>

c-네임스페이스와 스키마를 선언하는 방법

xmlns:c="http://www.springframework.org/schema/c"

<bean id="cdPlayer2" class="com.sjb.chapter2.soundsystem.CDPlayer"  
 c:cd-ref="compactDisc"/>

빈 레퍼런스와 리터럴 값의 와이어링에 대해서 <constructor-arg>와 c-네임스페이스 애트리뷰트는 모두 다 가능하다.

하지만 <constructor-arg>는 c-네임스페이스가 할 수 없는 한 가지를 할 수 있다.

**와이어링 컬렉션**

public class BlankDisc implements CompactDisc {  
 private String title;  
 private String artist;  
 private List<String> tracks;  
  
 public BlankDisc(String title, String artist, List<String> tracks) {  
 this.title = title;  
 this.artist = artist;  
 this.tracks = tracks;  
 }

...  
}

<list>요소를 사용하여 리스트를 지정할 수 있다.

<bean id="compactDisc" class="com.sjb.chapter2.soundsystem.BlankDisc">  
 <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band"/>  
 <constructor-arg value="The Beatles"/>  
 <constructor-arg>  
 <list>  
 <value>Sgt. Pepper's Lonely Hearts Club Band</value>  
 <value>With a Little Help from My Friends</value>  
 <value>Lucy in the Sky with Diamonds</value>  
 </list>  
 </constructor-arg>  
</bean>

Bean 형태

<bean id="compactDisc" class="com.sjb.chapter2.soundsystem.BlankDisc">  
 <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band"/>  
 <constructor-arg value="The Beatles"/>  
 <constructor-arg>  
 <list>  
 <ref bean="sgtPeppers" />  
 </list>  
 </constructor-arg>  
</bean>

Set 형태

<bean id="compactDisc" class="com.sjb.chapter2.soundsystem.BlankDisc">  
 <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band"/>  
 <constructor-arg value="The Beatles"/>  
 <constructor-arg>  
 <set>  
 <value>Sgt. Pepper's Lonely Hearts Club Band</value>  
 <value>With a Little Help from My Friends</value>  
 <value>Lucy in the Sky with Diamonds</value>  
 <value>Getting Better</value>  
 <value>Fixing a Hole</value>  
 </set>  
 </constructor-arg>  
</bean>

<list>는 java.util.List, <set>은 java.util.Set

와이어링 컬렉션에서 <constructor-arg>가 c-네임스페이스보다 유리하다.

**2.4.4 프로퍼티 세팅**

생성자가 아닌 프로퍼티 주입 방식

public class BlankDisc2 implements CompactDisc {  
 private String title;  
 private String artist;  
 private List<String> tracks;  
  
 public void setTitle(String title) {  
 this.title = title;  
 }  
  
 public void setArtist(String artist) {  
 this.artist = artist;  
 }  
  
 public void setTracks(List<String> tracks) {  
 this.tracks = tracks;  
 }  
}

Bean 형태

<bean id="compactDisc" class="com.sjb.chapter2.soundsystem.BlankDisc2">  
 <property name="title" value="Sgt. Pepper's Lonely Hearts Club Band"/>  
 <property name="artist" value="The Beatles"/>  
 <property name="tracks">  
 <list>  
 <value>Sgt. Pepper's Lonely Hearts Club Band</value>  
 <value>With a Little Help from My Friends</value>  
 <value>Lucy in the Sky with Diamonds</value>  
 <value>Getting Better</value>  
 <value>Fixing a Hole</value>  
 </list>  
 </property>  
</bean>

리터럴 형태

<bean id="compactDisc" class="com.sjb.chapter2.soundsystem.BlankDisc2"  
 p:title="Sgt. Pepper's Lonely Hearts Club Band"  
 p:artist="The Beatles"  
 p:tracks-ref="trackList"/>  
  
<util:list id="trackList">  
 <value>Sgt. Pepper's Lonely Hearts Club Band</value>  
 <value>With a Little Help from My Friends</value>  
 <value>Lucy in the Sky with Diamonds</value>  
 <value>Getting Better</value>  
 <value>Fixing a Hole</value>  
</util:list>

**2.5 설정 가져오기와 믹싱하기**

JavaConfig 또는 XML 설정에서 컴포넌트 스캐닝 및 오토와이어링을 혼합하는 것은 자유다.

**2.6 요약**

와이어링 하는 세가지 기본 방법

-자동 설정(scan) : 명시적인 설정과 관련된 유지보수 비용을 피하기 위해

-명시적인 Java 기반 설정(JavaConfig) : XML 보다 더 강력하고 타입세이프하며 리팩토링 가능

-명시적인 XML 기반 설정(XML)

자동 > 명시적인 Java > 명시적인 XML 순으로 사용하자.

**3. 고급 와이어링**

**3.1 환경과 프로파일**

@Bean(destroyMethod = "shutdown")  
public DataSource dataSource() {  
 return new EmbeddedDatabaseBuilder()  
 .addScript("classpath:schema.sql")  
 .addScript("classpath:test-data.sql")  
 .build();  
}  
  
@Bean  
public DataSource dataSource2() {  
 JndiObjectFactoryBean jndiObjectFactoryBean =  
 new JndiObjectFactoryBean();  
 jndiObjectFactoryBean.setJndiName("jdbc/myDS");  
 jndiObjectFactoryBean.setResourceRef(true);  
 jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);  
 return (DataSource) jndiObjectFactoryBean.getObject();  
}  
  
@Bean(destroyMethod = "close")  
public DataSource dataSource3() {  
 BasicDataSource dataSource = new BasicDataSource();  
 dataSource.setUrl("jdbc:h2:tcp://dbserver/~/test");  
 dataSource.setDriverClassName("org.h2.Driver");  
 dataSource.setUsername("sa");  
 dataSource.setPassword("password");  
 dataSource.setInitialSize(20);  
 dataSource.setMaxActive(30);  
 return dataSource;  
}

3개다 javax.sql.DataSource 타입인데 빈을 생성하기 위한 전략은 완전히 다르다.

서로 다른 환경에서는 차이가 있다.

**3.1.1 빈 프로파일 설정하기 (스프링 3.1에서 빈 프로파일을 도입)**

@Configuration  
@Profile("dev")  
public class DevelopmentProfileConfig {  
  
 @Bean(destroyMethod = "shutdown")  
 public DataSource dataSource() {  
 return new EmbeddedDatabaseBuilder()  
 .setType(EmbeddedDatabaseType.*H2*)  
 .addScript("classpath:schema.sql")  
 .addScript("classpath:test-data.sql")  
 .build();  
 }  
}

@Profile 애너테이션은 설정 클래스의 빈이 dev 프로파일이 활성화된 경우에만 작성되어야 함을 스프링에 알려 준다.

dev 프로파일이 활성화되지 않은 경우, @Bean 메소드는 무시된다.

스프링 3.1은 클래스 수준에서 @Profile 애너테이션을 사용했지만

3.2 이상에서는 메소드 수준에서 사용한다.

**XML로 프로파일 설정하기**

<beans> 요소는 다음과 같이 하나의 XML 파일에 모든 프로파일 빈 정의를 수집하는데 도움이 된다.

<beans profile="dev">  
 <jdbc:embedded-database id="dataSource">  
 <!-- <jdbc:script location="classpath:schema.sql"/>  
 <jdbc:script location="classpath:test-data.sql"/>-->  
 </jdbc:embedded-database>  
</beans>  
  
<beans profile="qa">  
 <bean id="dataSource2"  
 class="org.apache.commons.dbcp.BasicDataSource"  
 destroy-method="close"  
 p:url="jdbc:h2:tcp://dbserver/~/test"  
 p:driverClassName="org.h2.Driver"  
 p:username="sa"  
 p:password="password"  
 p:initialSize="20"  
 p:maxActive="30"/>  
</beans>  
  
<beans profile="prod">  
 <jee:jndi-lookup id="dataSource3" jndi-name="jdbc/myDatabase"  
 resource-ref="true"  
 proxy-interface="javax.sql.DataSource"/>  
  
</beans>

**3.1.2 프로파일 활성화하기**

스프링은 프로파일이 활성 상태인지를 결정하는 두 가지 다른 프로퍼티를 가진다.

spring.profile.active와 spring.profile.default 이다.

spring.profile.active가 설정되어 있는 경우, 그 값은 프로파일이 활성 상태인지를 결정한다.

Spring.profile.active가 설정되어 있지 않으면 스프링은 spring.profile.default가 된다.

🡺둘다 설정되저 있지 않으면 프로파일에 정의되지 않은 빈만 만들어진다.

**web.xml**

<context-param> // 컨텍스트를 위한 기본 프로파일 설정하기  
 <param-name>spring.profiles.default</param-name>  
 <param-value>dev</param-value>  
</context-param>  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>  
<servlet>  
 <servlet-name>dispatcherServlet</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param> // 서블릿을 위한 기본 프로파일 설정하기  
 <param-name>spring.profiles.default</param-name>  
 <param-value>dev</param-value>  
 </init-param>  
</servlet>

**3.2 조건부 빈**

스프링 4.0에서는 @Bean을 적용할 수 있는 새로운 @Conditional 애너테이션이 소개되었다.

소정의 조건이 참으로 평가된 경우 빈이 생성된다. 그렇지 않으면 빈은 무시된다.

@Conditional은 Condition 인터페이스와 같이 사용된다.

public interface Condition {  
 boolean matches(ConditionContext var1, AnnotatedTypeMetadata var2);  
}

Condition의 matches() 메소드가 true를 돌려주었을 경우, @Conditional로 애너테이션된 빈이 생성된다. Matches()가 false를 반환하면 그 빈은 작성되지 않는다.

@Profile 애너테이션은 다음과 같다.

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Conditional({ProfileCondition.class})  
public @interface Profile {  
 String[] value();  
}

@Profile 자체가 @Conditional을 사용하여 애너테이션되고, Condition 구현으로 ProfileCondition을 참조한다.

**3.3 오토와이어링의 모호성**

정확히 하나의 빈이 원하는 결과와 일치할 때 오토와이어링은 동작한다.

다음과 같이 setDessert() 메소드가 애너테이션 되고

@Autowired  
public void setDessert(Dessert dessrt) {  
 this.dessert = dessert;  
}

Cake, Cookie, IceCream로 구현되었다.

@Component  
class Cake implements Dessert {  
}  
@Component  
class Cookies implements Dessert {  
}  
@Component  
class IceCream implements Dessert {  
}

스프링이 setDessert()에서 Dessert 파라미터를 오토와이어링할 때 하나의 명확한 대안이 있지는 않다. 스프링은 선택할 수 없어서 예외를 발생 시킨다. (NoUniqueBeanDefinitionException)

**3.3.1 기본 빈 지정**

@Primary 애너테이션을 사용하여 스프링에서 선택할 수 있다.

@Primary는 컴포넌트 스캐닝된 빈을 위한 @Component와 자바 설정에서 선언된 빈의 @Bean을 함께 사용한다.

@Component  
@Primary  
class IceCream implements Dessert {  
}

또는

@Bean  
@Primary  
public Dessert iceCream() {  
 return new IceCream();  
}

XML로도 설정할 수 있다.

<bean id="iceCream"  
 class="com.sjb.chapter3.autowired.IceCream" primary="true"/>

만약 primary bean이 여러 개라면?

**3.3.2 오토와이어링 빈의 자격**

@**Qualifier** 애너테이션은 수식자를 사용하는 주된 방법이다.

그것은 주입 대상 빈을 지정할 주입 지점에서 @Autowired나 @Inject와 함께 적용된다.

@Autowired  
@Qualifier("iceCream")  
public void setDessert(Dessert dessrt) {  
 this.dessert = dessert;  
}

Qualifier의 파라미터는 주입할 빈의 ID다.

하지만 IceCream Class 이름을 Gelato로 변경하면 어떻게 될까? 그 경우 빈의 ID 및 기본 수식자는 gelato이며, setDessert()에서 수식자와 일치하지 않는다. 따라서 오토와이어링은 실패한다.

**맞춤형 수식자 만들기**

빈 ID에 의존하는 대신, 빈에 자신의 수식자를 지정한다.

빈 선언에서 @Qualifier 애너테이션을 배치하는 것이다.

@Component  
@Qualifier("cold")  
class IceCream implements Dessert {  
}

@Autowired  
@Qualifier("cold")  
public void setDessert(Dessert dessert) {  
 this.dessert = dessert;  
}

그러면 동일한 Qualifier(“cold”)가 생기면?

**맞춤형 수식자 애너테이션 정의하기**

// cold

@Target({ElementType.*CONSTRUCTOR*, ElementType.*FIELD*, ElementType.*METHOD*, ElementType.*TYPE*})  
@Retention(RetentionPolicy.*RUNTIME*)  
@Qualifier  
public @interface Cold {  
}

@Target({ElementType.*CONSTRUCTOR*, ElementType.*FIELD*, ElementType.*METHOD*, ElementType.*TYPE*})  
@Retention(RetentionPolicy.*RUNTIME*)  
@Qualifier  
public @interface Cremy {  
}

사용자 지정 수식자 애너테이션을 정의하여, 자바 컴파일러에서 제한 없이 또는 불만 없이 여러 개의 수식자를 사용한다.

// IceCream

@Component  
@Cold  
@Cremy  
class IceCream implements Dessert {  
}

// AutowiredExample

@Component  
public class AutowiredExample {  
 private Dessert dessert;  
  
 @Autowired  
 @Cold  
 @Cremy  
 public void setDessert(Dessert dessert) {  
 this.dessert = dessert;  
 }  
}

하지만 스프링의 애너테이션이 많아 진다.

**3.4 빈 범위**

스프링은 빈이 생성될 수 있는 여러 개의 범위를 정의하며 다음을 포함한다.

-싱글톤(Singleton) – 전체 애플리케이션을 위해 생성되는 빈의 인스턴스 하나

-프로토타입(Prototype) – 빈이 주입될 때마다 생성되거나 스프링 애플리케이션 컨텍스트에서 얻는 빈의 인스턴스 하나

-세션(Session) – 웹 애플리케이션에서 각 세션용으로 생성되는 빈의 인스턴스 하나

-요청(Request) – 웹 애플리케이션에서 각 요청용으로 생성되는 빈의 인스턴스 하나

@Scope를 사용하여 bean 클래스를 애너테이션한다.

@Component  
@Scope(ConfigurableBeanFactory.*SCOPE\_PROTOTYPE*)  
public class Notepad {  
}

자바 설정에서(JavaConfig) 프로토타입으로서 Notepad 빈을 설정한다면, 원하는 범위를 지정하기 위해서 @Bean과 함께 @Scope를 사용한다.

@Bean  
@Scope(ConfigurableBeanFactory.*SCOPE\_PROTOTYPE*)  
public Notepad notepad() {  
 return new Notepad();  
}

XML 설정

<bean id="notepad" class="com.sjb.chapter3.scope.Notepad" scope="prototype" />

**3.4.1 요청과 세션 범위 작업하기**

다음과 같이 싱글톤 StoreService 빈이 setter 메소드로 ShoppingCart 빈을 주입한다고 가정할 때

@Component  
public class StoreService {  
  
 private ShoppingCart shoppingCart;  
  
 @Autowired  
 public void setShoppingCart(ShoppingCart shoppingCart) {  
 this.shoppingCart = shoppingCart;  
 }  
}

스프링 애플리케이션 컨텍스트가 로드될 수 있도록 SotreService 싱글톤 빈이 생성된다.

그러나 세션 범위를 가지는 ShoppingCart는 존재하지 않는다. 사용자가 함께 와서 세션이 만들어질 때까지 ShoppingCart 인스턴스는 존재하지 않는다.

이번에는 범위 프록시에 대한 이해를 가지고 proxyMode 애트리뷰트를 설명한다.

설정대로 proxyMode는 ShoppingCart의 인터페이스를 구현하고 구현 빈에 위임할 필요가 있다는 의미의 ScopedProxyMode.INTERFACES로 설정된다.

@Scope(  
 value = WebApplicationContext.*SCOPE\_SESSION*,  
 proxyMode = ScopedProxyMode.*INTERFACES*)

ShoppingCart가 구상 클래스인 경우, 스프링은 인터페이스 기반의 프록시를 만들 수 있다. 대신 클래스 기반의 프록시 생성에 CGLIB를 사용한다. 빈의 타입이 구체적인 클래스인 경우, 프록시가 타깃 클래스 확장으로 생성되어야 함을 나타내기 위해서 ScopedProxyMode.TARGET\_CLASS를 설정한다.

**3.4.2 XML로 범위 프록시 선언하기**

XML로 세션 범위 또는 요청 범위의 빈을 선언하는 경우 @Scope 애너테이션 또는 proxyMode 애트리뷰트를 사용할 수 없다.

프록시모드를 사용하려면 스프링의 AOP 네임스페이스에서 새로운 요소를 사용해야만 한다.

<bean id="cart" class="com.sjb.chapter3.scope.ShoppingCart" scope="session">  
 <aop:scoped-proxy proxy-target-class="false"/>  
</bean>

기본적으로 타깃 클래스 프록시를 생성하기 위해 CGLIB를 사용한다.

하지만 proxy-target-class 애트리뷰트를 false로 설정하여 인터페이스 기반의 프록시를 생성할 수 있다.

**3.5. 런타임 값 주입**

런타임에서 결정되길 원할 때 스프링은 런타임에 값을 평가하는 두 가지 방법을 제공한다.

-프로퍼티 플레이스홀더(Property placeholders) // 주로 내가 사용했던 방법

-스프링 표현 언어(SpEL, Spring Expression Language)

목표와 동작은 다르지만, 두 기술의 응용은 비슷하다.

**3.5.1 외부 값 주입**

@PropertySource는 클래스패스 applicationProperty.properties 파일을 참조한다.

@Configuration  
@PropertySource("classpath:applicationProperty.properties")  
public class ExpressiveConfig {  
  
 @Autowired  
 Environment env;  
  
 @Bean  
 public BlankDisc disc() {  
 return new BlankDisc(  
 env.getProperty("disc.title", "Rattle and Hum"),  
 env.getProperty("disc.artist", "U2")  
 );  
 }  
}

// applicatinoProperty.properties

disc.title=DevTitle  
disc.artist=DevArtist

**스프링 환경에 대해 더 살펴 보기**

getProperty()는 네 가지 변형으로 오버로드된다.

-String getProperty(String key)

-String getProperty(String key, String defaultValue)

-T getProperty(String key, Class<T> type)

-T getProperty(String key, Class<T> type, T defaultValue)

**T getProperty(String key, Class<T> type, T defaultValue)**

만인 프로퍼티 파일에서 String 값을 찾으면, 사용하기 전에 Integer로 변환할 필요가 있다.

int connectionCount = env.getProperty("db.connection.count", Integer.class, 30);

**getRequiredProperty()**

프로퍼티 정의가 필요하다면 getRequiredProperty를 사용한다.

정의되저 있지 않으면 IllegalStateException이 발생한다.

@Bean  
public BlankDisc disc2() {  
 return new BlankDisc(  
 env.getRequiredProperty("disc.title"),  
 env.getRequiredProperty("disc.artist")  
 );  
}

프로퍼티 존재를 확인해야 할 경우 Environment에서 containsProperty를 호출한다.

boolean titleExist = env.containsProperty("disc.title");

**프로퍼티 플레이스홀더 처리하기**

XML 설정

<bean id="sgtPeppers"  
 class="com.sjb.chapter3.scope.BlankDisc"  
 c:\_title="${disc.title}"  
 c:\_artist="${disc.artist"/>

@Value 애너테이션

BlankDisc(  
 @Value("${disc.title}") String title,  
 @Value("${disc.artist}") String artist) {  
 this.title = title;  
 this.artist = artist;  
}

플레이스홀더 값을 사용하기 위해 PropertyPlaceholderConfigurer 빈 또는

PropertySourcesPlaceholderConfigurer를 사용한다.

자바 설정

@Bean  
public PropertySourcesPlaceholderConfigurer placeholderConfigurer() {  
 return new PropertySourcesPlaceholderConfigurer();  
}

XML 설정(PropertySourcesPlaceholderConfigurer)

<context:property-placeholder/>

**3.5.2 스프링 표현식 와이어링**

SpEL(Spring Expression Language)는 다음 기능을 포함한다.

-ID로 빈을 참조하는 기능

-메소드 호출과 객체의 프로퍼티 액세스

-값에서의 수학적인 동작, 관계와 관련된 동작, 논리연산 동작

-정규 표현식 매칭

-컬렉션 처리

SpEL은 종속객체 주입보다는 다른 용도로 사용된다. 예를 들면 스프링 시큐리티는 SpEL표현식을 포함하는 보안을 정의하는데 도움된다.

SpEL 방식

BlankDisc(  
 @Value("#{systemProperties['disc.title']}") String title,  
 @Value("#{systemProperties['disc.artist']}") String artist) {  
 this.title = title;  
 this.artist = artist;  
}

SpEL에서 지원되는 몇 가지 프리미티브 표현식

**리터럴값 표시하기**

부동소수

#{3.14159}

과학적 표기법 (98,700)

#{9.87E4}

리터럴 String

#{'Hello'}

불리언 리터럴

#{false}

**빈, 프로퍼티, 메소드 참조**

ID를 사용하여 빈 와이어링

#{sgtPeppers}

표현식에서 sgtPeppers 빈의 artist 프로퍼티 참조

#{sgtPeppers.artist}

selectArtist 메소드 호출

#{sgtPeppers.selectArtist}

NullPointer 방지를 위한 타입세이프

#{sgtPeppers.selectArtist?.toUpperCase}

**CHAPTER4 애스펙트 지향 스프링**

소프트웨어 개발자들은 한 애플리케이션의 여러 부분에 걸쳐 있는 기능을 가리켜 횡단 관심사라고 한다. (cross-cutting concerns)

보통 횡단 관심사는 애플리케이션의 비즈니스 로직과는 개념적으로 분리된다. 애스팩트 지향 프로그래밍(AOP)는 바로 이러한 횡단 관심사의 분리를 위한 것이다.

-DI : 애플리케이션 객체 간 결합도를 낮추는데 목적이 있다.

-AOP : 횡단 관심사와 이에 영향 받는 객체 간 결합도를 낮추는데 있다.

**4.1 AOP란 무엇인가?**

애스펙트의 주목적은 횡단 관심사의 모듈화에 있다.

공통 기능을 재사용하기 위해 일반적으로 사용하는 객체 지향 기법은 상속이나 위임이다.

하짖만 상속은 객체의 정적 구조에 의존하므로 복잡하고 깨지기 쉬운 구조가 되기 십상이며,

위임은 대상 객체에 대한 복잡한 호출로 인해 번거롭다.

🡺애스펙트는 대부분 상속이나 위임보다 더 깔끔한 해결책을 제공한다.

AOP는 이 기능을 어디에 어떻게 적용할지를 **선언적으로** 정의할 수 있고, 대상 클래스를 전혀 수정할 필요가 없다는 점이 큰 차이점이다.

AOP 장점

1) 전체 코드 기반에 흩어져 있는 관심 사항이 하나의 장소로 응집된다는 점

2) 여타 서비스 모듈이 자신의 주요 관심 사항에 대한 코드만 포함하고 그 외 관심 사항은 모두 애스펙트로 옮겨지므로 코드가 깔끔해진다는 점

**4.1.1 AOP 용어 정의**

AOP 가장 중요한 용어는 어드바이스(advice), 포인트커트(pointcut), 조인 포인트(join point)

**어드바이스(무엇을 언제)**

애스펙트가 해야 할 작업을 AOP 용어로는 어드바이스(advice)라고 한다.

어드바이스는 애스펙트가 ‘무엇’을 ‘언제’ 할지를 정의한다.

애스펙트는 다섯 가지 종류의 어드바이스를 가진다.

-이전(before) : 어드바이스 대상 메소드가 호출되기 전에 어드바이스 기능을 수행

-이후(after) : 결과에 상관없이 어드바이스 대상 메소드가 완료된 후에 어드바이스 기능을 수행

-반환 이후(after-return) : 어드바이스 대상 메소드가 성공적으로 완료된 후에 어드바이스 기능을 수행

-예외 발생 이후(after-throwing) : 어드바이스 대상 메소드가 예외를 던진 후에 어드바이스 기능 수행

-주위(around) : 어드바이스가 어드바이스 대상 메소드를 감싸서 어드바이스 대상 메소드 호출 전과 후에 몇가지 기능을 제공한다.

**조인 포인트 (지점)**

어드바이스를 적용할 수 있는 곳을 조인 포인트(join point)라고 한다. 즉, 조인 포인트는 애플리케이션 실행에 애스펙트를 끼워 넣을 수 있는 지점(point)을 말한다.

지점으로는 메소드 호출 지점이나 예외 발생, 필드 값 수정 등이 있다.

**포인트커트(어디서)**

포인트커트(point cut)는 바로 애스펙트가 어드바이스할 조인 포인트의 영역을 좁히는 일을 한다.

포인트커트를 지정하는 방법은 클래스나 메소드 명을 직접 사용하는 것이지만, 매칭 패턴을 나타내는 정규 표현식을 정의하는 방법도 있다.

**애스펙트(무엇을 언제 어디서)**

어드바이스와 포인트커트를 합친(merge) 것이다.

두 가지 정보가 합쳐지면 애스펙트가 무엇을 언제 어디서 할지, 즉 애스펙트에 필요한 모든 정보가 정의된다.

**인트로덕션**

기존 클래스에 코드 변경 없이도 새 메소드나 멤버 변수를 추가하는 기능이다.

**위빙**

위빙(weaving)은 타깃 객체에 애스펙트를 적용해서 새로운 프록시 객체를 생성하는 절차다.

애스펙트는 타깃 객체의 조인 포인트로 위빙된다. 위빙은 대상 객체의 생애 중 다음과 같은 몇가지 시점에서 수행된다.

-컴파일 시간(compile time) : 타깃 클래스가 컴파일될 때 애스펙트가 위빙되며, 별도의 컴파일러가 필요하다.

Ex) AspectJ의 위빙 컴파일러

-클래스로드 시간(classload time) : 클래스가 JVM에 로드될 때 애스펙트가 위빙된다. 이렇게 하려면 애플리케이션에서 사용되기 전에 타깃 클래스의 바이트 코드를 인핸스(소스가 아닌 컴파일된 바이트 코드에 직접 메소드나 멤버 변수등을 추가) 하는 특별한 ClassLoader가 필요하다.

Ex) AspectJ 5의 로드 시간 위빙

-실행 시간(runtime) : 애플리케이션 실행 중에 애스펙트가 위빙된다. 보통 타깃 객체에 호출을 위임하는 구조의 프록시 객체를 위빙 중에 AOP 컨테이너가 동적으로 만들어 낸다

Ex) 스프링 AOP 애스펙트가 위빙되는 방식

**포인트커트는 어드바이스 대상이 될 조인 포인트를 정의하는것이다.**

**4.1.2 스프링의 AOP 지원**

스프링은 다음과 같은 네 가지 형태로 AOP를 지원한다.

-고전적인(classic) 스프링 프록시 기반 AOP

-Pure-POJO 애스펙트

-@AspectJ 애너테이션 기반 애스펙트

-AspectJ 애스펙트에 빈 주입(스프링의 모든 버전에서 지원)

**스프링 어드바이스는 자바로 작성**

스프링에서 생성하는 모든 어드바이스는 표준 자바 클래스로 작성한다.



스프링 애스펙트는 타깃 객체를 감싸는 프록시 형태로 구현된다. 이 프록시는 먼저 호출을 가로챈 후 추가적인 애스펙트 로직을 수행하고 나서야 타깃 메소드를 호출한다.

**실행 시간에 만드는 스프링 어드바이스**

스프링에서 빈을 감싸는 프록시 객체를 실행 시간에 생성함으로써 애스펙트가 스프링 관리빈에 위빙된다. 프록시 객체는 타깃 객체로 위장해서 어드바이스 대상 메소드의 호출을 가로채고, 타깃 객체로 호출을 전달한다.

스프링은 런타임 시에 프록시를 생성하므로, 스프링 AOP의 애스펙트를 위빙하기 위한 별도의 컴파일러는 필요하지 않다.

**스프링은 메소드 조인 포인트만 지원**

스프링의 경우에는 동적 프록시를 기반으로 AOP를 구현하므로 메소드 조인 포인트만 지원한다.

메소드 조인 포인트만으로 100%는 아니더라도 필요한 대부분이 충족된다. 게다가 더 세밀한 제어가 필요하다면 그때는 AspectJ를 사용한 스프링 AOP를 이용하여 보충하면 된다.

**4.2 포인트커트를 이용한 조인 포인트 선택**

스프링 AOP와 관련된 AspectJ 포인트커트에 대해 알아야 하는 가장 중요한 사항은, 스프링은 AspectJ에서 사용할 수 있는 포인트커트 지정자에 속하는 것만 지원한다는 사실이다.

**스프링은 스프링 애스펙트를 정의하기 위해 AspectJ의 포인트커트 표현식 언어를 사용한다.**

|  |  |
| --- | --- |
| 지정자 | 설명 |
| args() | 인자가 주어진 타입의 인스터스인 조인 포인트 매칭을 정의한다. |
| @args() | 전달된 인자의 런타임 타입이 주어진 타입의 애너테이션을 갖는 조인 포인트 매칭을 정의한다. |
| execution() | 메소드 실행 조인 포인트와 일치시키는데 사용된다. |
| this() | AOP 프록시의 빈 레퍼런스가 주어진 타입의 인스턴스를 갖는 조인 포인트를 정의한다. |
| target() | 대상 객체가 주어진 타입을 갖는 조인 포인트를 정의한다. |
| @target() | 수행 중인 객체의 클래스가 주어진 타입의 애너테이션을 갖는 조인 포인트를 정의한다. |
| within() | 특정 타입에 속하는 조인 포인트를 정의한다. |
| @within() | 주어진 애너테이션을 갖는 타입 내 조인 포인트를 정의한다. |
| @annotation | 조인 포인트의 대상 객체가 주어진 애너테이션을 갖는 조인 포인트를 정의한다. |

AspectJ의 다른 지정자를 사용하면 IllegalArgumentException이 발생한다.

execution 지정자만 실제로 일치시키는 작업을 수행한다. 다른 지정자는 일치를 제한하는데 사용된다.

**perform() 메소드가 실행될 때마다 어드바이스를 하기 위해 사용될 수 있는 포인트커트 표현식**

package com.sjb.chapter4.aop;public interface Performance {  
 public void perform();  
}

execution(\* com.sjb.chapter4.aop.Performance.perform(..))

execution : 메소드 실행 시작

\* : 모든 타입 리턴

com.sjb.chapter4.aop.Performance : 메소드가 속하는 타입

perform : 메소드

(..) : 모든 인자 받기

**포인트커트의 범위를 패키지로만 제한**

execution(\* com.sjb.chapter4.aop.Performance.perform(..) && within(com.sjb.chapter4.aop.\*))

&& : 조합 및 연산자

within(com.sjb.chapter4.aop.\*) : 메소드가 패키지의 operator 클래스에서 호출되는 경우

&&(and), ||(or) 를 이용하여 결합시킬 수 있다.

**4.3 애스펙트 애너테이션 만들기**

**4.3.1 애스펙트 정의하기**

**Target Class**

package com.sjb.chapter4.aop;  
public interface Performance {  
 void perform();  
}

**애스펙트**

package com.sjb.chapter4.aop;  
  
import org.aspectj.lang.annotation.AfterReturning;  
import org.aspectj.lang.annotation.AfterThrowing;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
  
@Aspect  
public class Audience {  
  
 // 공연 이전  
 @Before("execution(\*\* com.sjb.chapter4.aop.Performance.perform(..))")  
 public void silenceCellPhones() {  
 System.*out*.println("Silencing cell phones");  
 }  
  
 // 공연 이전  
 @Before("execution(\*\* com.sjb.chapter4.aop.Performance.perform(..))")  
 public void takeSeats() {  
 System.*out*.println("Taking seats");  
 }  
  
 // 공연 후  
 @AfterReturning("execution(\*\* com.sjb.chapter4.aop.Performance.perform(..))")  
 public void applause() {  
 System.*out*.println("CLAP CALP CALP!!!");  
 }  
  
 // 공연 실패 후  
 @AfterThrowing("execution(\*\* com.sjb.chapter4.aop.Performance.perform(..))")  
 public void demandRefund() {  
 System.*out*.println("Demanding a refund");  
 }  
}

**스프링은 어드바이스 메소드를 선언하기 위해 AspectJ 애너테이션을 사용한다.**

|  |  |
| --- | --- |
| 애너테이션 | 어드바이스 |
| @After | 어드바이스 메소드는 어드바이스된 메소드가 반환되거나 예외 상황이 발생한 이후에 호출된다. |
| @AfterReturning | 어드바이스 메소드는 어드바이스된 메소드가 반환된 이후에 호출된다. |
| @AfterThrowing | 어드바이스 메소드는 어드바이스된 메소드가 예외 상황을 발생시킨 이후에 호출된다. |
| @Around | 어드바이스 메소드는 어드바이스된 메소드를 감싼다. |
| @Before | 어드바이스 메소드는 어드바이스된 메소드가 호출되기 이전에 호출된다. |

Pointcut을 이용해서 선언할 수 있다.

@Aspect  
public class Audience {  
  
 @Pointcut("execution(\*\* com.sjb.chapter4.aop.Performance.perform") // 명명된 포인트커트 정의  
 public void performance() {  
 }  
  
 // 공연 이전  
 @Before("performance()")  
 public void silenceCellPhones() {  
 System.*out*.println("Silencing cell phones");  
 }  
  
 // 공연 이전  
 @Before("performance()")  
 public void takeSeats() {  
 System.*out*.println("Taking seats");  
 }  
  
 // 공연 후  
 @AfterReturning("performance()")  
 public void applause() {  
 System.*out*.println("CLAP CALP CALP!!!");  
 }  
  
 // 공연 실패 후  
 @AfterThrowing("performance()")  
 public void demandRefund() {  
 System.*out*.println("Demanding a refund");  
 }  
}

Performance 메소드는 실제로 비어 있으며, 메소드 자체는 마커이고 @Pointcut애너테이션을 제공

@Pointcut("execution(\*\* com.sjb.chapter4.aop.Performance.perform(..))") // 명명된 포인트커트 정의  
public void performance() { // 실제로는 비어 있다.  
}

위의 Audience 클래스는 POJO다.

하지만 실제 위의 @Aspect 코드 선언 만으로는 스프링 컨테이너 내의 빈일 뿐이다.

애스펙트로 변경하는 프록시를 생성하고, 애너테이션을 해석하는 무엇인가 없이는 애스펙트로 취급되지 않는다.

**JavaConfig 방식**

@Configuration  
@EnableAspectJAutoProxy  
@ComponentScan  
public class ConcertConfig {  
 @Bean  
 public Audience audience() {  
 return new Audience();  
 }  
}

@EnableAspectJAutoProxy 애너테이션을 적용하여 오토-프록싱을 사용한다.

**Xml 방식**

<beans>

<!-- chapter4 -->  
 <context:component-scan base-package="com.sjb.chapter4.aop.xml"/>  
  
 <!-- AspectJ 오토-프록싱 활성화 -->  
 <aop:aspectj-autoproxy/>  
  
 <bean class="com.sjb.chapter4.aop.xml.Audience"/>  
</beans>

<aop:aspectj-autoproxy/> 적용하여 오토-프록싱을 사용한다.

JavaConfig 또는 XML을 사용하면 AspectJ 오토-프록싱은 애스펙트의 포인트커트가 일치하는 다른 빈의 프록시를 만들기 위해 @Aspect 애너테이션된 빈을 사용한다.

**4.3.2 around 어드바이스 만들기**

@Around("performance()")  
public void watchPerformance(ProceedingJoinPoint jp) {  
 try {  
 System.*out*.println("Sliencing cell phones");  
 System.*out*.println("Taking seats");  
 jp.proceed();  
 System.*out*.println("CLAP CLAP CLAP!!!");  
 } catch (Throwable e) {  
 System.*out*.println("Demanding a refund");  
 }  
}

@Around 애너테이션은 around 어드바이스로서 가장 주목할 점은 파라미터로 ProceedingJoinPoint를 받는다는 사실이다. 이 객체는 어드바이스 내에서 어드바이스 대상 메소드를 호출할 수 있는 방법을 제공한다.

Around 어드바이스를 구현할 때는 반드시 proceed()를 호출해야 한다는 점을 기억하자.

**4.3.3 어드바이스에서 파라미터 처리하기**

private Map<Integer, Integer> trackCounts =  
 new HashMap<Integer, Integer>();  
  
@Pointcut("execution(\* com.sjb.chapter4.aop.parameter.CompactDisc.playTrack(int)) " + "&& args(trackNumber)")  
public void trackPlayed(int trackNumber) {  
  
}  
  
@Before("trackPlayed(trackNumber)")  
public void countTrack(int trackNumber) {  
 int currentCount = getPlayCount(trackNumber);  
 trackCounts.put(trackNumber, currentCount + 1);  
}  
  
public int getPlayCount(int trackNumber) {  
 return trackCounts.containsKey(trackNumber) ? trackCounts.get(trackNumber) : 0;  
}

중점적인 것은 포인트커트 표현식에서 args(trackNumber) 식별자(qualifier)다.

이는 playTrack()에 전달되는 int 인자는 어드바이스로도 전달됨을 나타낸다. 파라미터 명 trackNumber는 포인트커트 메소드 시그너처(signature)의 파라미터와도 매칭된다.

**4.3.4 인트로덕션 애너테이션**

스프링에서 애스펙트는 래핑하는 빈과 동일한 인터페이스를 구현하는 프록시일 뿐이라는 사실을 기억하자.

**4.4 XML에서 애스펙트 선언하기**

**4.4.1 before 어드바이스와 after 어드바이스 선언하기**

**Target Class**

@Component  
public class Musical implements Performance {  
 @Override  
 public void perform() {  
 System.*out*.println("공연중");  
 }  
}

**Advice Class**

public class Audience {  
 // 공연 이전  
 public void silenceCellPhones() {  
 System.*out*.println("Silencing cell phones");  
 }  
 // 공연 이전  
 public void takeSeats() {  
 System.*out*.println("Taking seats");  
 }  
 // 공연 후  
 public void applause() {  
 System.*out*.println("CLAP CALP CALP!!!");  
 }  
 // 공연 실패 후  
 public void demandRefund() {  
 System.*out*.println("Demanding a refund");  
 }  
}

**애스펙트 XML로 선언 (before, after-returning, throwing)**

<bean id="audience" class="com.sjb.chapter4.aop.xml2.Audience"/>  
<context:component-scan base-package="com.sjb.chapter4.aop.xml2"/>  
  
<!-- chapter4 -->  
<aop:config> <!-- 최상위 개념 -->  
 <aop:aspect ref="audience"> <!-- 애스펙트 정의 -->  
 <aop:pointcut id="performance"  
 expression="execution(\* com.sjb.chapter4.aop.xml2.Performance.perform(..))"/>  
 <aop:before  
 pointcut-ref="performance"  
 method="silenceCellPhones"/>  
 <aop:before  
 pointcut-ref="performance"  
 method="takeSeats"/>  
 <aop:after-returning  
 pointcut-ref="performance"  
 method="applause"/>  
 <aop:after-throwing  
 pointcut-ref="performance"  
 method="demandRefund"/>  
 </aop:aspect>  
</aop:config>

|  |  |
| --- | --- |
| AOP 설정 요소 | 용도 |
| <aop:advisotr> | AOP 어드바이저 정의 |
| <aop:after> | 어드바이스 이후 AOP 정의(성공적인지 아닌지 상관 없음) |
| <aop:after-returning> | AOP 반환 후(after-returning) 어드바이스 정의 |
| <aop:after-throwing> | AOP 발생 후(after-throwing) 어드바이스 정의 |
| <aop:around> | AOP around 어드바이스 정의 |
| <aop:aspectj-autoproxy> | @AspectJ를 사용한 애너테이션 기반 애스펙트 사용 |
| <aop:before> | 어드바이스 전 AOP 정의 |
| <aop:config> | 톱 레벨 AOP 요소. 대부분 config 내에 포함 |
| <aop:declare-parents> | 추가 인터페이스를 명확히 구현된 어드바이스된 객체에 도입 |
| <aop:pointcut> | 포인트커트 정의 |

**4.4.2 around 어드바이스 선언**

**Advice Class**

public class Audience {  
 public void watchPerformance(ProceedingJoinPoint jp) {  
 try {  
 System.*out*.println("Silencing cell phones"); // 공연 전  
 System.*out*.println("Taking seats");  
 jp.proceed();  
 System.*out*.println("CLAP CLAP CLAP!!!");  
 } catch (Throwable e) {  
 System.*out*.println("Demanding a refund");  
 }  
 }  
}

**애스펙트 XML로 선언 (around)**

<bean id="audience" class="com.sjb.chapter4.aop.xml2.Audience"/>  
<context:component-scan base-package="com.sjb.chapter4.aop.xml2"/>  
  
<!-- chapter4 -->  
<aop:config> <!-- 최상위 개념 -->  
 <aop:aspect ref="audience"> <!-- 애스펙트 정의 -->  
 <aop:pointcut id="performance"  
expression="execution(\* com.sjb.chapter4.aop.xml2.Performance.perform(..))"/>  
 <aop:around pointcut-ref="performance" method="watchPerformance"/>  
 </aop:aspect>  
</aop:config>

around는 위의 before, after, throwing 기능을 다 포함하고 있다.

**4.4.3 어드바이스에 파라미터 전달**

**Aspect를 제거한 TrackCounter**

public class TrackCounter {  
  
 private Map<Integer, Integer> trackCounts =  
 new HashMap<Integer, Integer>();  
  
 public void countTrack(int trackNumber) {  
 int currentCount = getPlayCount(trackNumber);  
 trackCounts.put(trackNumber, currentCount + 1);  
 }  
  
 public int getPlayCount(int trackNumber) {  
 return trackCounts.containsKey(trackNumber) ? trackCounts.get(trackNumber) : 0;  
 }  
}

**XML로 파라미터 설정**

<bean id="trackCounter" class="com.sjb.chapter4.aop.parameter.xml.TrackCounter"/>  
<bean id="cd" class="com.sjb.chapter4.aop.parameter.xml.BlankDisc">  
 <property name="title" value="Sgt. Pepper's Lonely Hearts Club Band"/>  
 <property name="artist" value="The Beatles"/>  
 <property name="tracks">  
 <list>  
 <value>Sgt. Pepper's Lonely Hearts Club Band</value>  
 <value>With a Little Help from My Friends</value>  
 <value>Lucy in the Sky with Diamonds</value>  
 <value>Getting Better</value>  
 <value>Fixing a Hole</value>  
 <value>Fixing a Hole2</value>  
 <value>Fixing a Hole3</value>  
 <value>Fixing a Hole4</value>  
 </list>  
 </property>  
</bean>  
  
<aop:config>  
 <aop:aspect ref="trackCounter">  
 <aop:pointcut id="trackPlayed" expression=  
 "execution(\* com.sjb.chapter4.aop.parameter.xml.CompactDisc.playTrack(int)) and args(trackNumber)"/>  
 <aop:before pointcut-ref="trackPlayed" method="countTrack"/>  
 </aop:aspect>  
</aop:config>

앞에서 본 javaconfig와 차이점은 && 대신 and 키워드를 사용한다.

**4.5 AspectJ 애스펙트 주입**

AspectJ는 스프링 AOP에서는 불가능했던 많은 포인트커트를 제공한다.

**CHAPTER5**

**5.1 스프링 MVC 시작하기**

**5.1.1 스프링 MVC를 이용한 요청 추적**

**1) DispatcherServlet**

스프링 MVC 역시 많은 요청들을 깔대기처럼 하나의 프런트 컨트롤러 서블릿에서 처리한다.

단일 프런트 컨트롤러는 단일 서블릿이 실제 처리를 수행하기 위해 다른 컴포넌트에 대한 요청 책임을 위임하는 웹 애플리케이션의 일반적 패턴을 제공한다.

DispatcherServlet이 이런 역할을 한다.

DispatcherServlet의 임무는 요청을 스프링 MVC 컨트롤러에 전달해 주는 것이다.

**2) 핸들러 매핑**

일반적으로 애플리케이션은 여러 개의 컨트롤러를 가지고 있기 때문에 DispatcherServlet은 요청을 전달할 컴포넌트를 선택하기 위한 도움이 필요하다. 그래서 요청이 가야 할 다음 정거장을 찾기 위해 여러 개의 핸들러 매핑에게 도움을 요청한다.

요청을 받은 매핑들은 결정을 내릴 때 요청이 가져온 URL에 특별히 주목한다.

**3) 컨트롤러**

적절한 컨트롤러가 선택되면, DispatcherServlet은 선택된 컨트롤러에 요청을 보낸다.

컨트롤러에서 처리되는 로직의 결과는 사용자의 브라우저에 표시되기 위한 형태의 정보로 반환된다. 이 정보를 일반적으로 모델이라고 한다.

**4) 모델과 논리 뷰 이름**

컨트롤러가 하는 마지막 일은 모델을 패키징하는 일과 결과물을 렌더링하기 위한 뷰의 이름을 확인하는 것이다. 그리고 나서 모델과 뷰 이름을 포함하여 다시 DispatcherServlet 에게 요청을 돌려보낸다.

**5) 뷰 리졸버**

DispatcherServlet은 뷰 리졸버에게 논리적으로 주어진 뷰의 이름과 실제로 구현된 뷰를 매핑해 줄 것을 요청한다.

**6) 뷰**

뷰는 모델 데이터를 사용하여 결과를 렌더링하고 이것은 그다지 어렵지 않게 응답 객체에 의해 클라이언트로 전달된다.

**5.1.2 스프링 MVC 설정하기**

**DispatcherServlet 설정하기**

**기존 방식 (web.xml)**

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0">  
 <display-name>skeleton\_api</display-name>  
 <context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>  
 classpath:spring/applicationContext-bean.xml  
 classpath:spring/applicationContext-config.xml  
 classpath:spring/applicationContext-database.xml  
 </param-value>  
 </context-param>  
 <listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
 </listener>  
 <servlet>  
 <servlet-name>dispatcherServlet</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:mvc-config.xml</param-value>  
 </init-param>  
 </servlet>  
 <servlet-mapping>  
 <servlet-name>dispatcherServlet</servlet-name>  
 <url-pattern>/</url-pattern>  
 </servlet-mapping>  
</web-app>

root-context와 servlet-context

**스프링 MVC 설정하기**

public class SpitterWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {  
  
 // DispatcherServlet이 매핑되기 위한 하나 혹은 여러개의 패스를 지정한다.  
 @Override  
 protected String[] getServletMappings() {  
 return new String[]{"/"}; // 이것은 애플리케이션으로 들어오는 모든 요청을 처리한다.  
 }  
  
 @Override  
 protected Class<?>[] getServletConfigClasses() { // 이것은 servlet-context  
 return new Class<?>[]{WebConfig.class};  
 }  
  
 @Override  
 protected Class<?>[] getRootConfigClasses() { // 이것은 root-context  
 return new Class<?>[]{RootConfig.class};  
 }  
}

**dispatcherServlet context (servlet-context) XML 형식**

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">  
  
 <!-- # dispatcherServlet Context -->  
 <context:component-scan base-package="com.sjb" use-default-filters="false">  
 <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller" />  
 </context:component-scan>  
 <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/view/" />  
 <property name="suffix" value=".jsp" />  
 </bean>  
 <bean id="restTemplate" class="org.springframework.web.client.RestTemplate" />  
</beans>

**스프링 MVC 설정 (servlet-context)**

@Configuration  
@EnableWebMvc  
@ComponentScan("com.sjb.chapter5.web")  
public class WebConfig extends WebMvcConfigurerAdapter {  
 @Bean  
 public ViewResolver viewResolver() {  
 InternalResourceViewResolver resolver =  
 new InternalResourceViewResolver();  
 resolver.setPrefix("/WEB-INF/views/");  
 resolver.setSuffix(".jsp");  
 resolver.setExposeContextBeansAsAttributes(true);  
 return resolver;  
 }  
  
 @Override  
 public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {  
 configurer.enable();  
 }  
}

주의점은 web.xml이 활성화 되어 있으면 충돌난다.

**MockMvc 이용**

public class HomeControllerTest {  
 @Test  
 public void testHomePage() throws Exception {  
 HomeController controller = new HomeController();  
 MockMvc mockMvc =  
 *standaloneSetup*(controller).build();  
  
 mockMvc.perform(*get*("/"))  
 .andExpect(*view*().name("home"));  
 }  
}

Controller 테스트 시, standaloneSetup 을 이용하여 요청을 할 수 있다.

**5.3 요청 입력받기**

스프링 MVC는 클라이언트가 컨트롤러의 핸들러 메소드에 데이터를 전달해 줄 몇 가지 방법을 제공한다.

-쿼리 파라미터

-폼 파라미터

-패스 변수

**5.3.1 쿼리 파라미터 입력받기**

**SpittleController class**

private static final String *MAX\_LONG\_AS\_STRING* = "9223372036854775807";

@RequestMapping(method = RequestMethod.*GET*)  
public List<Spittle> spittles(  
 @RequestParam(value = "max", defaultValue = *MAX\_LONG\_AS\_STRING*) long max,  
 @RequestParam(value = "count", defaultValue = "20") int count) {  
 return spittleRepository.findSpittles(max, count);  
}

**Test**

mockMvc.perform(*get*("/spittles?max=238900&count=50"))  
 .andExpect(*view*().name("spittles"))  
 .andExpect(*model*().attributeExists("spittleList"))  
 .andExpect(*model*().attribute("spittleList", *hasItems*(expectedSpittles.toArray())));

쿼리 파라미터는 항상 String 타입이므로 defaultValue 애트리뷰트 역시 String 값을 필요한다.

**5.3.2 패스 파라미터를 통한 입력받기**

*/\*\*  
 \* 쿼리 파라미터 방식 (RequestParam 이용)  
 \*/*

@RequestMapping(value = "/show", method = RequestMethod.*GET*)  
public String showSpittle(  
 @RequestParam("spittle\_id") long spittleId, Model model) {  
 model.addAttribute(spittleRepository.findOne(spittleId));  
 return "spittle";  
}

이 핸들러 메소드는 /spittles/show?spittle\_id=12345 와 같은 요청을 처리한다.

리소스 지향 관점에서 보면 이상적인 방식은 아니다. 리소스가 쿼리 파라미터가 아닌 URL 패스에 의해 식별되어야 이상적인 방식이 된다. 일반적인 규칙에 따르면 쿼리 파라미터는 리소스를 식별하는데 사용하지 않는다.

/spittles/12345가 위 방식보다 더 나은 GET 요청 방식이다.

/spittles/show?spittle\_id=12345 : 검색될 리소스를 식별한다.

/spittles/12345 : 파라미터에 대한 동작을 서술하는데 기본적으로 HTTP상의 RPC다.

*/\*\*  
 \* 패스 방식 (PathVariable 이용)  
 \*/*@RequestMapping(value = "/{spittleId}", method = RequestMethod.*GET*)  
public String spittle(  
 @PathVariable("spittleId") long spittleId, Model model) {  
 model.addAttribute(spittleRepository.findOne(spittleId));  
 return "spittle";  
}

패스 방식을 사용하려면 스프링 MVC는 @RequestMapping 패스에 플레이스홀더를 사용한다.

플레이스홀더는 중괄호({와}) 로 묶여 있는 이름들이다.

ex) "/{spittleId}"

@RequestMapping(value = "/{spittleId}", method = RequestMethod.*GET*)  
public String spittle(  
 @PathVariable long spittleId, Model model) {  
 model.addAttribute(spittleRepository.findOne(spittleId));  
 return "spittle";  
}

다음과 같이 플레이스 홀더와 파라미터 이름을 같이 사용하면 @PathVariable의 value 파라미터를 생략하는 것도 가능하다.

하지만 파라미터의 이름을 바꾸려고 결정한다면 반드시 플레이스홀더 이름과 일치하도록 바꿔 주어야 한다는 것에 주의하자.

**5.4 폼 처리하기**

폼은 두 가지 과정으로 나눌 수 있다. 폼을 보여 주기와 사용자가 폼을 통해 제출한 데이터

처리하기다.

@RequestMapping(value ="/register", method= RequestMethod.*POST*)  
public String processRegistration(Spitter spitter) {  
 spitterRepository.save(spitter);  
  
 return "redirect:/spitter/" + spitter.getUsername();  
}

InternalResourceViewResolver가 redirect 접두사(prefix)를 뷰 명세에서 확인하면 뷰 이름이 아닌 리다이렉션 명세로 인식한다.

**CAHPTER6**

**6.1. 뷰 리졸루션 이해하기**

컨트롤러가 단지 논리적인 이름으로만 뷰를 알 수 있다면, 스프링이 모델을 실제로 렌더링하기 위해 구현되어 있는 뷰를 어떻게 결정할까? 바로 이것이 스프링의 뷰 리졸버(resolver)의 역할이다.

5장에서 InternalResourceViewResolver를 사용했었다. 이 리졸버는 모델을 렌더링하게 될 뷰의 물리적인 위치로 전달되는 논리적 뷰 이름에 접두사(prefix)로 /WEB-INF/views/를 붙이고 접미사(suffix)로 .jsp를 붙이도록 되어 있다.

스프링 MVC는 아래와 같은 내용의 ViewResolver 인터페이스를 정의한다.

public interface ViewResolver {  
 View resolveViewName(String var1, Locale var2) throws Exception;  
}

ResolveViewName(), 메소드는 뷰의 이름과 Locale을 넘겨 받고 View 인스턴스를 리턴해 준다.

View의 다른 인터페이스는 아래와 같다.

public interface View {  
 String getContentType();  
  
 void render(Map<String, ?> var1, HttpServletRequest var2, HttpServletResponse var3) throws Exception;  
}

**6.2 JSP 뷰 만들기**

스프링은 JSP를 아래의 두 가지 방식으로 지원한다. (JSTL이나 스프링 JSP 태그 라이브러리)

-InternalResourceViewResolver는 JSP 파일에 뷰 이름을 결정하기 위해서 사용한다.

-스프링은 form-to-model 바인딩을 위한 것과 일반적인 유틸리티 기능을 제공하는 두 개의 JSP 태그 라이브러리를 제공한다.

**6.2.1 JSP-ready 뷰 리졸버 설정하기**

**XML 방식**

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/view/" />  
 <property name="suffix" value=".jsp" />  
</bean>

**JavaConfig 방식**

@Bean  
public ViewResolver viewResolver() {  
 InternalResourceViewResolver resolver =  
 new InternalResourceViewResolver();  
 resolver.setPrefix("/WEB-INF/view/");  
 resolver.setSuffix(".jsp");  
 resolver.setExposeContextBeansAsAttributes(true);  
 return resolver;  
}

- books/detail은 /WEB-INF/view/books/detail.jsp 로 결정된다.

**JSTL 뷰 결정하기**

InternalResourceViewResolver는 궁극적으로 논리적 뷰 이름으로 JSP 파일을 참조하는 InternalResourceView 인스턴스를 결정한다. 하지만 JSP 파일에서 포매팅이나 메시지에 JSTL을 사용한다면 InternalResourveViewResolver가 JstlView의 인스턴스를 결정하도록 설정하는 것이 더 바람직하다.

**XML 방식 (JSTL)**

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/view/" />  
 <property name="suffix" value=".jsp" />  
 <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />  
</bean>

**JavaConfig 방식 (JSTL)**

@Bean  
public ViewResolver viewResolver() {  
 InternalResourceViewResolver resolver =  
 new InternalResourceViewResolver();  
 resolver.setPrefix("/WEB-INF/view/");  
 resolver.setSuffix(".jsp");  
 resolver.setViewClass(org.springframework.web.servlet.view.JstlView.class);  
 resolver.setExposeContextBeansAsAttributes(true);  
 return resolver;  
}

**6.2.2 스프링의 JSP 라이브러리 사용하기**

태그 라이브러리는 스크립트 블록에 직접 자바 코드를 작성할 필요 없이 JSP 템플릿에 기능을 추가하는 강력한 방법이다.

스프링은 스프링 MVC 웹 뷰의 정의를 용이하게 하는 두 가지 JSP 태그 라이브러리를 제공한다.

-첫 번째 태그 라이브러리는 model 애트리뷰트와 바인딩되어 있는 HTML의 폼 태그를 렌더링해 준다.

-다른 하나는 종종 편리하게 사용할 수 있는 유틸리티 태그의 혼합 형태이다.

**폼에 모델 바인딩하기**

폼 바인딩 태그를 사용하기 위해서는 JSP 페이지 안에 아래와 같은 내용을 선언한다.

<%@ **taglib** uri="http://www.springframework.org/tags/form" prefix="sf" %>

<**sf:form** method="POST" commandName="spitter">  
 First Name: <**sf:input** path="firstName"/><br/>  
 Last Name: <**sf:input** path="lastName"/><br/>  
 Email: <**sf:input** path="email"/><br/>  
 Username: <**sf:input** path="username"/><br/>  
 Password: <**sf:password** path="password"/><br/>  
 <input type="submit" value="Register"/>  
</**sf:form**>

<sf:form> 태그는 HTML <form> 태그를 렌더링한다. 그리고 이 태그는 commandName 애트리뷰트에 지정되는 모델 객체의 컨텍스트를 설정한다.

@RequestMapping(value = "/register", method = RequestMethod.*GET*)  
public String showRegistrationForm(Model model) {  
 model.addAttribute(new Spitter());  
 return "springForm";  
}

예를 들어 모델의 Spitter 객체가 firstName 프로퍼티에 Jack 이라는 값을 갖고 있다면,

<sf:input path=”firstName”/>은 value=”Jack”인 <input> 태그를 렌더링한다.

**오류 표시하기**

검증 오류가 발생하면 오류의 상세한 내용들은 모델 데이터를 포함하고 있는 요청에 의해 전달된다. 그리고 모델을 파헤쳐서 사용자에게 표시해 줄 에러들을 추출한다.

<sf:errors> 태그는 이러한 작업을 간단하게 만든다.

<**sf:form** method="POST" commandName="spitter">  
 First Name: <**sf:input** path="firstName"/>  
 <**sf:errors** path="firstName" /><br/>

…

</sf:form>

유효성 결과



**6.3 아파치 타일즈 뷰로 레이아웃 정의하기**

아파치 타일즈와 같은 레이아웃 엔진을 사용하여 모든 페이지에 적용될 공통 페이지 레이아웃을 정의하는 것이 더 좋은 방법이다.

**6.3.1 타일 뷰 리졸버 설정하기**

**JavaConfig 방식**

*\*/*@Configuration  
public class TilesConfig {  
 @Bean  
 public TilesConfigurer tilesConfigurer() {  
 TilesConfigurer tilesConfigurer = new TilesConfigurer();  
 tilesConfigurer.setDefinitions("/WEB-INF/layouts/definitions/tiles.xml");  
 tilesConfigurer.setCheckRefresh(true);  
  
 return tilesConfigurer;  
 }  
}

**XML 설정 방식**

<!-- TilesConfigurer 방식 -->  
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">  
 <property name="definitions">  
 <value>/WEB-INF/layout/tiles.xml</value>  
 <value>/WEB-INF/views/\*\*/tiles.xml</value>  
 </property>  
</bean>  
<bean id="viewResolver" class="org.springframework.web.servlet.view.tiles3.TilesViewResolver" />

TilesConfigurer가 아파치 타일즈로 타일 정의를 불러오고 배치시키는 반면, TilesViewResolver는 논리적 뷰 이름으로 타일 정의를 참조하는 뷰를 결정하도록 되어 있다.

**6.4 Thymeleaf로 작업하기**

JSP에 대한 가장 큰 문제는 내부가 HTML 또는 XML의 형태로 되어 있지만, 실상은 어느 쪽에도 해당되지 않는다는 점이다.

ex)

<input type="text" value="<**c:out** value="**${**this.name**}**"/>" />

또한 JSP의 스펙은 서블릿 스펙과 강하게 결합되어 있다. 이것은 JSP가 서블릿 기반 웹 애플리케이션의 웹 뷰에 사용된다는 것을 의미한다.

Thymeleaf 템플릿은 형태가 자연스럽고 태그 라이브러리에 의존하지 않는다. 또한 HTML이 사용 가능한 곳에서는 언제든 편집하거나 렌더링할 수 있다. 그리고 서블릿 스펙과 관계가 없으므로 Thymeleaf 템플릿은 JSP가 감히 사용될 수 없는 곳에서도 사용된다.

**6.4.1 Thymeleaf 뷰 리졸버 설정하기**

세 개의 빈을 설정해 주어야 한다.

-논리적 뷰 이름으로 Thymeleaf 템플릿 뷰를 결정하는 ThymeleafViewResolver

-템플릿을 처리하고 결과를 렌더링하는 SpringTemplateEngine

-Thymeleaf 템플릿을 불러오는 TemplateResolver

**JavaConfig 방식**

public class Thymeleaf {  
  
 // Tyhemleaf 뷰 리졸버  
 @Bean  
 public ViewResolver viewResolver(SpringTemplateEngine templateEngine) {  
 ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();  
 viewResolver.setTemplateEngine(templateEngine);  
 return viewResolver;  
 }  
  
 // 템플릿 엔진  
 @Bean  
 public TemplateEngine templateEngine(TemplateResolver templateResolver) {  
 SpringTemplateEngine templateEngine = new SpringTemplateEngine();  
 templateEngine.setTemplateResolver(templateResolver);  
 return templateEngine;  
 }  
  
 // 템플릿 리졸버  
 @Bean  
 public TemplateResolver templateResolver() {  
 TemplateResolver templateResolver = new ServletContextTemplateResolver();  
 templateResolver.setPrefix("/WEB-INF/templates/");  
 templateResolver.setSuffix(".html");  
 templateResolver.setTemplateMode(TemplateMode.HTML5);  
 return templateResolver;  
 }  
}

**XML 설정 방식**

<!-- Thymeleaf 방식 -->  
<bean id="viewResolver" class="org.thymeleaf.spring3.view.ThymeleafViewResolver" p:templateEngine-ref="templateEngine" />  
<bean id="templateEngine" class="org.thymeleaf.spring3.SpringTemplateEngine" p:templateResolver-ref="templateResolver" />  
<bean id="templateResolver" class="org.thymeleaf.templateresolver.ServletContextTemplateResolver"  
 p:prefix="/WEB-INF/templates/"  
 p:suffix=".html"  
 p:templateMode="HTML5" />

**6.4.2 Thymeleaf 템플릿 정의하기**

Thymeleaf 템플릿은 기본적으로 일반 HTML 파일이다. JSP와 같은 태그 라이브러리나 특별한 태그를 사용하지도 않는다. 하짖만 Thymeleaf는 사용자가 지정한 네임스페이스를 통해 표준 HTML 태그에 Thymeleaf 애트리뷰트를 추가해 주는 것으로 동작한다.

<html xmlns="http://www.w3.org/1999/xhtml"  
 xmlns:th="http://www.thymeleaf.org"> <!—네임스페이스 선언 --> <title>Spitter</title>  
</head>  
<body>  
<h1>Welcome to Spitter</h1>  
<a th:href="@{/spittles}">Spittles</a>  
<a th:href="@{/spitter/register}">Register</a>  
</body>  
</html>

**CHAPTER7**

**7.1 다양한 스프링 MVC 설정**

JavaConfig 형태를 사용하면 항상 요구 사항을 만족하는 것은 아니다. 아마도 DispatcherServlet 뿐만 아니라 서블릿과 필터가 필요할지도 모르고 DispacherServlet 에도 추가적인 설정이 필요할 수 있다. 아니면 서블릿 3.0 이전 버전의 컨테이너로 애플리케이션을 배포하려고 한다면 고전적인 web.xml 파일로 DispatcherServlet 을 설정한다.

대항스럽게도 이를 위한 몇 가지 제어 방법을 스프링에서 제공한다.

**7.1.1 DispatcherServlet 설정 사용자 정의하기**

이러한 메소드들 중 하나가 customizeRegistration()이다.

ex) 멀티파트 설정

@Override  
protected void customizeRegistration(ServletRegistration.Dynamic registration) {  
 registration.setMultipartConfig(  
 new MultipartConfigElement("/tmp/spitter/uploads")  
 );  
}

**7.1.2 서블릿과 필터 추가하기**

AbstractAnnotationConfigDispatcherServletInitializer의 정의에 따르면, 기본적으로

DispatcherServlet과 ContextLoaderListener가 생성된다. 하지만 서블릿, 필터, 리스너를 추가적으로 등록하고 싶다면?

**서블릿을 등록하기 위한 WebApplicationInitializer 구현**

public class MyServletInitializer implements WebApplicationInitializer {

// 서블릿 등록  
@Override  
public void onStartup(ServletContext servletContext) throws ServletException {  
 ServletRegistration.Dynamic myServlet = servletContext.addServlet("myServlet", MyServlet.class);  
 myServlet.addMapping("/custom/\*\*");  
}

}

**필터도 등록할 수 있는 WebApplicationInitializer**

// 필터 등록

public class MyServletInitializer implements WebApplicationInitializer {

@Override  
public void onStartup(ServletContext servletContext) throws ServletException {  
 FilterRegistration.Dynamic filter = servletContext.addFilter("myFilter", MyFilter.class);  
 filter.addMappingForUrlPatterns(null, false, "/custom\*//\*");  
}

}

**AbstractAnnotationConfigDispatcherServletInitializer에서 간단하게 필터 사용하기**

public class SpitterWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override  
protected Filter[] getServletFilters() {  
 return new Filter[]{  
 new MyFilter()  
 };  
}

}

**7.1.3 web.xml에서 DispatcherServlet 선언하기**

**web.xml (root context 설정)**

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>  
 classpath:spring/applicationContext-bean.xml  
 </param-value>  
</context-param>  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>

**web.xml (servlet context 설정-기본형)**

<servlet>  
 <servlet-name>appServlet</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
 <servlet-name>appServlet</servlet-name>  
 <url-pattern>/</url-pattern>  
</servlet-mapping>

서블릿 이름이 appServlet이면 /WEB-INF/appServlet-context.xml 파일에서 애플리케이션 컨텍스트를 불러오도록 되어 있다.

**web.xml(servlet context 설정-지정형)**

<servlet>  
 <servlet-name>appServlet</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
 <servlet-name>appServlet</servlet-name>  
 <url-pattern>/</url-pattern>  
</servlet-mapping>

**자바 설정을 사용하기 위한 web.xml 설정하기 (혼합형)**

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0">  
 <display-name>skeleton\_api</display-name>  
 <context-param>  
 <param-name>contextClass</param-name>  
 <param-value>  
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext  
 </param-value>  
 </context-param>  
 <context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>com.sjb.chapter7.config.RootConfig</param-value>  
 </context-param>  
 <listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
 </listener>  
  
 <servlet>  
 <servlet-name>appServlet</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextClass</param-name>  
 <param-value>  
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext  
 </param-value>  
 </init-param>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>com.sjb.chapter7.config.WebConfig</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
 </servlet>  
 <servlet-mapping>  
 <servlet-name>appServlet</servlet-name>  
 <url-pattern>/</url-pattern>  
 </servlet-mapping>  
</web-app>

**7.2. 멀티파트 폼 데이터 처리하기**

멀티 파트는 주로 이미지 업로드 할 때 사용되며, 텍스트가 아닌 바이너리로 되어 있다.

**7.2.1 멀티파트 리졸버 설정하기**

DispatchServlet에는 멀티파트 요청의 데이터를 파싱하기 위한 로직이 구현되어 있지 않다.

대신 스프링의 MultipartResolver 전략 인터페이스의 구현부에 멀티파트 요청의 내용을 결정하도록 위임한다. 스프링 3.1 부터 두가지를 제공한다.

-CommonsMultipartResolver : 멀티파트 요청을 Jakarta Commons FileUpload를 사용하여 결정

-StandardServletMultipartResolver : 서블릿 3.0을 사용한 멀티파트 요청에 대한 지원 (spring3.1부터)

**서블릿 3.0으로 멀티파트 요청 결정하기**

서블릿 3.0 호환의 StandardServletMultipartResolver는 생성자 인자나 설정해 주어야 할 프로퍼티가 없다.

@Configuration  
public class MultipartConfig {  
 @Bean  
 public MultipartResolver multipartResolver() throws IOException {  
 return new StandardServletMultipartResolver();  
 }  
}

**AbstractAnnotationConfigDispatcherServletInitializer 방식 (multipart)**

@Override  
 protected void customizeRegistration(ServletRegistration.Dynamic registration) {  
 registration.setMultipartConfig(  
 new MultipartConfigElement("/tmp/spitter/uploads", 2097152, 4194304, 0)  
 );  
 }

-업로드되는 파일의 최대 크기(bytes)

-파트의 개수나 개별 파트들의 크기와 관계없는 전체 멀티파트 요청의 최대 크기

-임시 저장 위치에 쓰지 않고 업로드할 수 있는 파일의 최대 크기(bytes)

**web.xml 방식 (multipart)**

<multipart-config>  
 <location>/tmp/spitter/uploads</location>  
 <max-file-size>52428800</max-file-size>  
 <max-request-size>52428800</max-request-size>  
 <file-size-threshold>0</file-size-threshold>  
</multipart-config>

**Jakarta Commons FileUpload 멀티파트 리졸버 설정하기**

서블릿 3.0 컨테이너로 배포되지 않는다면 대안이 필요하다.

스프링에서는 StandardServletMultipartResolver의 대안으로 CommonsMultipartResolver를 제공한다.

@Configuration  
public class MultipartConfig {  
 @Bean  
 public MultipartResolver multipartResolver() throws IOException {  
 CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver();  
 multipartResolver.setUploadTempDir(  
 new FileSystemResource("/tmp/spitter/uploads")  
 );  
 multipartResolver.setMaxUploadSize(2097152);  
 multipartResolver.setMaxInMemorySize(0);  
 return multipartResolver;  
 }  
}

StandardServletMultipartResolver 달리 임시 파일 위치를 설정해 줄 필요가 없다. 기본적으로 서블릿 컨테이너의 임시 디렉터리로 설정되기 때문이다.

MultipartConfigElement와 달리 멀티파트 요청의 최대 크기를 명시하는 방법이 없다.

**7.2.2 멀티파트 요청 처리하기**

@RequestMapping(value = "/register", method = RequestMethod.*POST*)  
public String processRegistration(  
 @RequestPart("profilePicture") byte[] profilePicutre,  
 @Valid Spitter spitter,  
 BindingResult errors) {

...  
}

@RequestPart를 사용해서 byte 형태로 데이터를 받으면 이미지의 크기는 얻을 수 있지만, 파일의 타입이나 이름 같은 것을 알 수 없다.

**MultipartFile 받기**

**MultipartFile 인터페이스**

public interface MultipartFile extends InputStreamSource {  
 String getName();  
  
 String getOriginalFilename();  
  
 String getContentType();  
  
 boolean isEmpty();  
  
 long getSize();  
  
 byte[] getBytes() throws IOException;  
  
 InputStream getInputStream() throws IOException;  
  
 void transferTo(File var1) throws IOException, IllegalStateException;  
}

**부분으로 업로드된 파일받기**

MultipartFile 대신 Part를 사용하는 것으로 processRegistration() 메소드는 다음과 같이 사용된다.

@RequestMapping(value = "/register", method = RequestMethod.*POST*)  
public String processRegistration(  
 @RequestPart("profilePicture") Part profilePicture,  
 @Valid Spitter spitter,  
 BindingResult errors) {  
…  
}

**Part 인터페이스**

public interface Part {  
 InputStream getInputStream() throws IOException;  
  
 String getContentType();  
  
 String getName();  
  
 String getSubmittedFileName();  
  
 long getSize();  
  
 void write(String var1) throws IOException;  
  
 void delete() throws IOException;  
  
 String getHeader(String var1);  
  
 Collection<String> getHeaders(String var1);  
  
 Collection<String> getHeaderNames();  
}

Part 인자를 사용한다면 StandardServletMultipartResolver 빈을 설정해 줄 필요가 없다.

StandardServletMultipartResolver는 MultipartFile로 동작할 때만 필요하기 때문이다.

**7.3. 예외 처리하기**

스프링은 예외를 응답으로 변환시켜 주는 유용한 방법들을 제공한다.

-몇 가지 스프링의 예외들은 자동으로 명시된 HTTP 상태 코드로 매핑된다.

-HTTP 상태 코드로 매핑하기 위해 예외에는 @ResponseStatus 애너테이션을 붙여 줄 수 있다.

-예외 처리를 위한 메소드에는 @ExceptionHandler 애너테이션을 붙여 줄 수 있다.

@ExceptionHandler 메소드에서 특별히 흥미로운 부분은 같은 컨트롤러 내의 모든 핸들러 메소드에서 발생한 예외를 처리한다는 점이다.

**7.4 어드바이징 컨트롤러**

애플리케이션의 모든 컨트롤러에 광범위하게 적용되어 있다면 컨트롤러 클래스의 특정 부분은 더 편리해질 수 있다. 그 예로 위에서 얘기한 @ExceptionHandler 메소드는 다수의 컨트롤러에 걸쳐서 예외를 처리하기에 매우 유용함을 증명할 수 있을 것이다.

스프링 3.2는 컨트롤러 어드바이스(controller advice)라는 다른 선택을 제공한다.

@ControllerAdvice 애너테이션이 붙어 있는 클래스로 아래의 메소드들을 포함한다.

-@ExceptionHandler 애너테이션이 붙어 있는 메소드

-@InitBinder 애너테이션이 붙어 있는 메소드

-@ModelAttribute 애너테이션이 붙어 있는 메소드

@ControllerAdvice 애너테이션이 붙어 있는 클래스에서 위의 메소드들은 애플리케이션 내의 모든 컨트롤러의 **@RequestMapping 애너테이션이 붙어 있는 모든 메소드에 전역적으로 적용이 된다.**

@ControllerAdvice 자체에 @Component 애너테이션이 붙어 있어서 컴포넌트-스캐닝에 의해 선택된다.

**모든 컨트롤러의 예외 처리를 위한 @controllerAdvice의 사용**

@ControllerAdvice  
public class AppWideExceptionHandler {  
  
 @ExceptionHandler(DuplicateSpittleException.class)  
 public String duplicateSpittleHandler() {  
 return "error/duplicate";  
 }  
}

**7.5 리다이렉션되는 요청 간의 데이터 전달하기**

일반적으로 핸들러 메소드가 종료될 때, 메소드 내에서 명시된 모든 모델 데이터는 요청 애트리뷰트로서 요청에 복사되고, 이 요청은 렌더링을 위해 뷰에 포워딩(forwarding)된다.

컨트롤러 메소드와 뷰에서 처리되는 요청이 같으므로 요청 애트리뷰트는 포워딩에도 살아남게 된다.

하지만 메소드의 결과가 리다이렉션이라면 원래의 요청은 종료되고 새로운 HTTP GET 요청이 시작된다. 원본 요청에서 전달되던 모든 모델 데이터는 요청과 함께 소멸된다. 새로운 요청은 애트리뷰트에 어떠한 모델 데이터도 없는 상태가 되어 자체적으로 이를 찾아내야 한다.

하지만 몇가지 방법이 있다.

-데이터를 URL 템플릿을 사용하여 패스 변수나 쿼리 인자로 전달

-데이터를 플래시 애트리뷰트에 넣어서 전달

**7.5.1 URL 템플릿과 함께 리다이렉션하기**

@RequestMapping(value = "/register", method = RequestMethod.*POST*)  
public String processRegistrationByRedirect(  
 @Valid Spitter spitter,  
 BindingResult errors,  
 Model model) {  
  
 model.addAttribute("username", spitter.getUsername());  
 return "redirect:/spitter/{username}";  
}

리다이렉션 String에 접합되는 것이 아니라 URL 템플릿의 플레이스홀더에 채워지므로 username 프로퍼티에서 안전하지 않은 문자들은 빠진다. 이런 방식은 사용자에게 사용자명을 직접 입력받아 패스에 추가하는 방식보다 안전하다.

@RequestMapping(value = "/register", method = RequestMethod.*POST*)  
public String processRegistrationByRedirect(  
 @Valid Spitter spitter,  
 BindingResult errors,  
 Model model) {  
  
 model.addAttribute("username", spitter.getUsername());  
 model.addAttribute("spitterId", spitter.getId());  
 return "redirect:/spitter/{username}";  
}

이런 경우는 spitterId 애트리뷰트가 URL 플레이스홀더에 매핑되지 않으므로 리다이렉션에 자동적으로 쿼리 인자로 붙는다.

/spitter/username?spitterId=42 요런식으로..

패스 변수나 쿼리 인자를 이용하여 리다이렉션 데이터를 전달하는 방식은 쉽고 직관적이지만 한계가 있다. String이나 숫자 값과 같은 간단한 데이터의 전달에만 용이하다는 것이다.

이럴 때 플래시 애트리뷰트는 도움을 준다

**7.5.2 플래시 애트리뷰트로 작업하기**

세션 안에 보관하는 것도 한 가지 방법이다. 세션은 다수의 요청이 연속되는 동안에도 오래 지속된다.

그래서 리다이렉션하기 전에 세션에 Spitter를 넣어 두고 리다이렉션 후에 세션으로부터 얻어온다.

물론 리다이렉션 후에 세션에서 깨끗하게 제거해 주는 것도 필요하다.

🡺세션은 손실 없는 방법이지만 사용자가 데이터를 잘 관리할 수 있을 것이라고 생각 안 한다.

🡺대신 플래시 애트리뷰트로 데이터를 전달

RedirectAttributes는 모델과 동일한 기능을 제공하고 추가적으로 플래시 애트리뷰트를 위한 메소드를 제공한다.

@RequestMapping(value = "/register", method = RequestMethod.*POST*)  
public String processRegistrationByRedirect(  
 Spitter spitter,  
 RedirectAttributes model) {  
  
 model.addAttribute("username", spitter.getUsername());  
 model.addFlashAttribute("spitter", spitter);  
 return "redirect:/spitter/{username}";  
}

리다이렉션 되기 전에 모든 플래시 애트리뷰트는 세션으로 복사되어 들어간다.

리다이렉션 후에 세션에 저장되었던 플래시 애트리뷰트는 모델로 저장되고 세션에서 제거된다.

받는쪽

@RequestMapping(value = "/{username}", method = RequestMethod.*GET*)  
public String showSpitterProfileByRedirect(  
 @PathVariable String username, Model model) {  
 if (!model.containsAttribute("spitter")) {  
 model.addAttribute(  
 spitterRepository.findByUsername(username));  
 }  
 return "profile";  
}

**CHAPTER8**

스프링 웹 플로는 정해진 플로에 따른 요소의 개발을 가능하게 해주는 웹 프레임워크다.

**8.1 스프링에 웹 플로 설정하기**

아직까지는 자바로 스프링 웹 플로를 설정하는 방법은 지원되지 않으므로 XML로 설정하는 것 말고는 방법이 없다.

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:flow="http://www.springframework.org/schema/webflow-config"  
 xsi:schemaLocation="   
 http://www.springframework.org/schema/webflow-config  
 http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd” >

네임스페이스 선언이 되엇다면 플로 실행기(executor)로부터 시작되는 웹 플로를 연결한 준비가 된 것이다.

**8.1.1 플로 실행기 연결하기**

<flow:flow-executor> 요소는 스프링에 플로 실행기를 만든다.

<flow:flow-executor id="flowExecutor"/>

플로 실행기는 플로를 생성하고 실행하는 역할을 수행하지만, 플로 정의를 불러오지는 않는다.

플로 정의는 다음으로 만들어 볼 플로 저장소에서 불러온다.

**8.1.2 플로 설정하기**

플로 저장소는 플로 정의를 불러와서 플로 실행기에서 사용이 가능하도록 만든다.

<flow:flow-registry id="flowRegistry" base-path="/WEB-INF/flows">  
 <flow:flow-location-pattern value="order-flow.xml" />  
</flow:flow-registry>

**8.1.3 플로 요청 처리하기**

DispatcherServlet이 플로 요청을 스프링 웹 플로어로 보내도록 하기 위해서 FlowHandlerMapping이 필요 하다.

<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">  
 <property name="flowRegistry" ref="flowRegistry" />  
</bean>

FlowHandlerMapping 역할이 플로 요청을 스프링 웹 플로에 보내 주는 것이라고 한다면

FlowHandlerAdapter의 역할은 호출에 대한 응답니다.

FlowHandlerAdapter는 스프링 MVC 컨트롤러의 흐름과 프로세스로 오는 요청들을 처리한다는 점에서 비슷한다.

<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">  
 <property name="flowExecutor" ref="flowExecutor" />  
</bean>

**8.2 플로의 컴포넌트**

웹 플로에서 플로는 세 가지 기본 요소, 상태(state), 전이(transition), 플로 데이터(flow data)로

정의된다.

-상태는 어떤 일이 발생되는 플로 지점이다.

-전이는 각 위치들을 연결해 주는 길과 같다.

-플로데이터는 상태에 따라 진행되면서 데이터를 소지하게 된다.

**CHAPTER9**

**9.1 스프링 시큐리티 시작하기**

스프링 시큐리티는 스프링 기반의 애플리케이션에 선언적 보안을 제공하는 보안 프레임워크다.

스프링 시큐리티는 웹 요청 수준과 메소드 실행 수준의 인증과 권한을 처리하는 종합적인 보안 솔루션을 제공한다.

스프링 시큐리티는 Acegi 보안으로부터 시작된다.

-스프링 2.0은 스프링에서 보안을 설정하기 위한 새로운 보안 특성 XML 네임스페이스를 제공한다

🡺일반적인 보안 설정을 수백 줄에서 수 줄 정도로 간소화 시켰다.

-스프링 3.0은 SpEL을 추가한 보안 설정을 한층 더 간소화시켰다.

-스프링 3.2 에서는 시큐리티를 두 가지 각도로 해결한다.

1)웹 요청에 대한 보안과 URL 수준의 접근을 제한하기 위해서 스프링 시큐리티는 서블릿 필터를 사용한다. (웹 요청)

2)스프링 AOP, 객체 프록싱, 완전한 권한을 가진 사용자에 의해 메소드를 실행하도록 하는 어드바이스 적용을 통해 메소드 실행에 대한 보안을 유지한다. (메소드 실행 수준)

**9.1.1 스프링 시큐리티 모듈의 이해**

|  |  |
| --- | --- |
| ACL | 접근 제어 목록(ACL, Access Control List)를 통한 도메인 객체 보안 지원 |
| Aspects | 스프링 시큐리티 애너테이션을 사용 시, 표준 스프링 AOP 대신 AspectJ 기반의 관점을 지원 |
| CAS Client | 통합 인증 서비스(CAS)를 사용하여 통합 인증 로그인 (SSO, Single Sign-On)을 지원한다. |
| Configuration | XML과 자바를 통한 스프링 시큐리티 설정을 지원한다. |
| Core | 스프링 시큐리티 라이브러리의 핵심 기능을 제공한다. |
| Cryptography | 암호화 및 암호 부호화를 지원한다. |
| LDAP | LDAP 기반 인증을 지원한다. |
| OpenID | OpenID로 중앙 집중 인증을 지원한다. |
| Remoting | 스프링 리모팅 통합을 지원한다. |
| Tag Library | 스프링 시큐리티의 JSP 태그 라이브러리한다. |
| Web | 스프링 시큐리티의 필터 기반 웹 보안을 지원한다. |

**9.1.2 웹 요청 필터하기**

**web.xml**

<filter>  
 <filter-name>springSecurityFilterChain</filter-name>  
 <filter-class>  
 org.springframework.web.filter.DelegatingFilterProxy  
 </filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>springSecurityFilterChain</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

**JavaConfig**

public class SecurityWebInitializer extends AbstractSecurityWebApplicationInitializer {  
   
}

DelegatingFilterProxy 설정 두 가지 중(appendFilters(), insertFilters()) 어떠한 방식을 사용했든지, 애플리케이션으로 들어오는 요청을 가로채 ID가 springSecurityFilterChain인 빈에게 위임시킨다.

springSecurityFilterChain 빈 자체에 대해 말하면 FilterChainProxy라고 알려진, 또 다른 특별한 필터이다.

**9.1.3 간단한 보안 설정 작성하기**

-스프링 시큐리티의 초기 (Acegi) 단계에서는 웹 애플리케이션에 간단한 보안을 활성화하기 위해서 수백 줄 이상의 XML 설정을 작성해야 한다.

-스프링 시큐리티 2.0은 보안 특성 XML 설정 네임스페이스를 제공하여 이를 개선했다.

-스프링 시큐리티 3.2는 새로운 자바 설정 방식을 소개했고 XML 보안 설정의 필요성을 제거했다.

**스프링 MVC 웹 보안을 활성화하는 가장 간단한 클래스 설정**

package com.sjb.chapter9;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
@Configuration  
@EnableWebSecurity // 웹 보안 활성  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
}

WebSecurityConfigurerAdapter를 확장하여 클래스를 설정하는 것이 가장 편하고 자주 쓰이는 방법이다.

@EnableWebSevurity는 일반적으로 모든 웹 애플리케이션에서 보안을 활성화시키는데 유용하다.

하지만 스플핑 MVC 애플리케이션을 개발할 때는 아래의 코드처럼

@EnableWebMvcSecuriy(deprecated 됨 enablewebsecurity 사용)를 대신 사용하는 것을 고려해야 한다.

@Configuration  
// @EnableWebSecurity // 웹 보안 활성  
@EnableWebMvcSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
}

**WebSecurityConfigurerAdapter에 configure() 메소드 오버라이딩하기**

|  |  |
| --- | --- |
| Configure(WebSecurity) | 스프링 시큐리티의 필터 연결을 설정하기 위한 오버라이딩 |
| Configure(HttpSecurity) | 인터셉터로 요청을 안전하게 보호하는 방법을 설정하기 위한 오버라이딩 |
| Configure(AuthenticationManagerBuilder) | 사용자 세부 서비스를 설정하기 위한 오버라이딩 |

protected void configure(HttpSecurity http) throws Exception {  
 this.logger.debug("Using default configure(HttpSecurity). If subclassed this will potentially override subclass configure(HttpSecurity).");  
 ((HttpSecurity)((HttpSecurity)((AuthorizedUrl)http.authorizeRequests().anyRequest()).authenticated().and()).formLogin().and()).httpBasic();  
}

Configure(AuthenticationManagerBuilder) 메소드는 오버라이딩하지 않았으므로 인증 절차를 지원하기 위한 사용자 저장소가 없다.

애플리케이션에서 요구하는 것을 맞춰 주기 위해 설정에 약간 추가되는 것이 있어야 한다.

-사용자 저장소 설정

-인증이 필요한 요청과 아닌 요청을 명시하고, 필요한 권한이 무엇인지 명시

-평범한 기본 로그인 화면을 대체하기 위한 수정된 로그인 화면 제공

**9.2 사용자 상세 서비스 선택**

스프링 시큐리티는 지극히 유연하여 어떠한 데이터 저장소에 대해서도 가상으로 사용자 인증이 가능하다. 몇 가지 일반적인 사용자 저장 방식인 인메모리, 관계형 데이터베이스, LDAP에 대해서는 이미 훌륭한 기능을 제공하고 있고, 사용자 저장에 대한 구현을 새롭게 만들어 사용할 수도 있다.

**9.2.1 인메모리 사용자 저장소로 작업하기**

@EnableWebMvcSecurity // 스프링 MVC 보안 활성  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.inMemoryAuthentication() // 인메모리 사용자 저장소 활성  
 .withUser("user").password("password").roles("USER").and()  
 .withUser("admin").password("password").roles("USER", "ADMIN");  
 }  
}

AuthenticationMangerBuilder는 인증 설정을 만들기 위한 빌더 스타일 인터페이스를 사용하는 configure()에 인자로 사용된다.

간단히 inMemoryAuthentication()를 호출하는 것으로 인메모리 사용자 저장소가 활성화 된다.

**UserDetailsBuilder**

|  |  |
| --- | --- |
| accountExpired(boolean) | 계정이 만료되었는지 아닌지를 정의 |
| accountLocked(boolean) | 계정이 잠겨 있는지 아닌지를 정의 |
| and() | 설정을 연결하기 위해 사용 |
| authorities(GrantedAuthority..) | 사용자에게 부여된 권한들을 명시 |
| authorities(List<? extends GrantedAuthority> | 사용자에게 부여된 권한들을 명시 |
| authorities(String…) | 사용자에게 부여된 권한들을 명시 |
| credentialsExpired(boolean) | 자격이 만료되었는지 아닌지를 정의 |
| disabled(boolean) | 계정이 비활성화되었는지 아닌지를 정의 |
| password(String) | 사용자의 암호를 명시 |
| roles(String…) | 사용자에게 부여된 역할을 명시 |

인메모리 사용자 저장소가 디버깅이나 개발 테스트 목적으로는 유용하지만, 애플리케이션 상품화를 위해서는 좋은 선택이 될 수는 없다. 상품화를 목적으로 한다면 일반적으로 사용자 저장소를 데이터베이스와 같은 곳을 선택한다.

**9.2.2 데이터베이스 테이블로 인증하기**

@Autowired  
DataSource dataSource;  
  
*/\*\*  
 \* 데이터 베이스 방식  
 \*/*@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.jdbcAuthentication().dataSource(dataSource);  
}

쿼리에 대해 추가적인 제어가 필요할 경우

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.jdbcAuthentication()  
 .dataSource(dataSource)  
 .usersByUsernameQuery(  
 "select username, password, true " +  
 "from Spitter where username=?")  
 .authoritiesByUsernameQuery(  
 "select username, 'ROLE\_USER' from Spitter where username=?");  
}

**부호화된 암호로 작업하기**

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth  
 .jdbcAuthentication()  
 .dataSource(dataSource)  
 .usersByUsernameQuery(  
 "select username, password, true " +  
 "from Spitter where username=?")  
 .authoritiesByUsernameQuery(  
 "select username, 'ROLE\_USER' from Spitter where username=?")  
 .passwordEncoder(new StandardPasswordEncoder("53cr3t"));  
}

passwordEncoder() 메소드는 스프링 시큐리티의 PassEncoder 인터페이스에 대한 어떠한 구현도 받을 수 있다.

스프링 시큐리티의 암호화 모듈은 다음 세가지의 구현들을 포함한다.

- BCryptPasswordEncoder, NoOpPasswordEncoder, StandardPasswordEncoder

세가지 구현이 상속받는 PasswordEncoder 인터페이스는 꽤 단순하다.

public interface PasswordEncoder {  
 String encode(CharSequence var1);  
  
 boolean matches(CharSequence var1, String var2);  
}

**9.2.3 LDAP 기반 인증 적용하기**

아래는 LDAP 인증을 위한 간단할 설정이다.

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
 .userSearchFilter("(uid={0}")  
 .groupSearchFilter("member={0}");  
}

Ex)LDAP

|  |
| --- |
| ## FIRST Level hierarchy - people  ## uses mixed upper and lower case for objectclass  # this is an ENTRY sequence and is preceded by a BLANK line  dn: ou=people, dc=example,dc=com  ou: people  description: All people in organisation  objectclass: organizationalunit |

**9.2.4 사용자 정의 사용자 서비스 설정하기**

**SpitterUserService**

public class SpitterUserService implements UserDetailsService {  
 private final SpitterRepository spitterRepository;  
  
 // SpitterRepository 주입  
 public SpitterUserService(SpitterRepository spitterRepository) {  
 this.spitterRepository = spitterRepository;  
 }  
  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 Spitter spitter = spitterRepository.findByUsername(username);  
  
 if (spitter != null) {  
 List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();  
 authorities.add(new SimpleGrantedAuthority("ROLE\_SPITTER"));  
  
 return new User(  
 spitter.getUsername(),  
 spitter.getPassword(),  
 authorities  
 );  
 }  
  
 throw new UsernameNotFoundException(  
 "User '" + username + "' not found."  
 );  
 }  
}

@Autowired  
SpitterRepository spitterRepository;  
  
*/\*\*  
 \* 사용자 정의 서비스 설정  
 \*/*@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth  
 .userDetailsService(new SpitterUserService(spitterRepository));  
}

**9.3 요청 가로채기**

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 .antMatchers("/spitters/me").authenticated()  
 .antMatchers(HttpMethod.*POST*, "/spittles").authenticated()  
 .anyRequest().permitAll();  
}

-/spitters/me인 요청은 인증되어야 함

-/spittles에 대한 HTTP POST 요청이 인증되어야 함

-나머지는 권한 없이 허용

접근해 보면 HTTP Status 403 – Forbidden 에러남

**9.3.1 스프링 표현식 보안**

|  |  |
| --- | --- |
| 보안 표현 | 평가 내용 |
| authentication | 사용자의 인증 객체 |
| denyAll | 항상 거짓으로 평가함 |
| hasAnyRole(역할 목록) | 사용자가 역할 목록 중 하나라도 역할이 있는 경우 참 |
| hasRole(역할) | 사용자가 주어진 역할이 있는 경우 참 |
| hasIpAddress(IP 주소) | 주어진 IP 주소로부터 요청이 오는 경우 참 |
| isAnymous() | 사용자가 익명인 경우 참 |
| isAuthenticated() | 사용자가 인증된 경우 참 |
| isFullyAuthenticated() | 사용자가 완전히 인증된 경우 참(기억하기(rememver-me)로는 인증되지 않음 |
| isRememberMe() | 사용자가 기억하기(remember-me)로 인증된 경우 참 |
| permitAll | 항상 참으로 평가함 |
| pricipal | 사용자 주체 객체 |

**9.3.2 채널 보안 적용하기**

중요한 정보들은 반드시 암호화되어서 HTTP를 통해 전달 되어야 한다.

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 .antMatchers("/spitter/me").hasAuthority("ROLE\_SPITTER")  
 .antMatchers(HttpMethod.*POST*, "/spittles").hasAuthority("ROLE\_SPITTER")  
 .anyRequest().permitAll()  
  
 .and()  
 .requiresChannel()  
 .antMatchers("/spitter/form").requiresSecure() // HTTPS 통해 전송  
 .antMatchers("/").requiresInsecure(); // HTTP 통해 전송  
}

-requiresSecure() : http로 요청되면 https로 리다이렉션

-requiresInsecure() : https로 요청되면 http로 리다이렉션

채널에 대한 보안 시행에 패스를 선택하는 방법이 authorizeRequests()와 같아다는 것에 주목하자.

채널은 requreidsChannel() 을 사용한다.

**9.3.3 사이트 간 요청 위조 방지하기**

|  |
| --- |
| <form method=”POST” action=<http://www.spitter.com/spittles>></form> |

클릭 시, <http://www.spitter.com/spittles>에 폼이 제출된다. 위의 내용은 간단한 사이트 간 요청 위조 (CSRF, Cross-Site Request Forgery)의 예다.

스프링 시큐리티 3.2부터 CSRF 보안은 기본 설정으로 활성화되어 있다. 사실 CSRF 보호를 위한 조취를 취하지 않거나 이 기능을 비활성화한다면 애플리케이션에 제출되는 폼을 성공적으로 얻어 오는데 문제가 발생한다.

상태 변경 요청(GET, HEAD, OPTION, TRACE가 아닌 모든 요청)들은 가로채여서 CSRF 토큰을 확인받게 된다.

CSRF를 비활성화하는 것은 일반적으로 좋은 생각이 아니다.

**9.4 사용자 인증하기**

**기본 로그인 페이지를 활성화 시키는 formLogin() 메소드**

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http  
 .formLogin() // 기본 로그인 페이지 활성화  
 .and()  
 .authorizeRequests()  
 .antMatchers("/spitter/me").hasAuthority("ROLE\_SPITTER")  
 .antMatchers(HttpMethod.*POST*, "/spittles").hasAuthority("ROLE\_SPITTER")  
 .anyRequest().permitAll()  
 .and()  
 .requiresChannel()  
 .antMatchers("/spitter/form").requiresSecure() // HTTPS 통해 전송  
 .antMatchers("/").requiresInsecure(); // HTTP 통해 전송  
}

**9.4.1 사용자 정의 로그인 페이지 추가하기**

<html><head><title>Login Page</title></head>

<body onload="document.f.username.focus();">  
<h3>Login with Username and Password</h3><form name="f" action="/login" method="POST">  
 <table>  
 <tbody><tr><td>User:</td><td><input type="text" name="username" value=""></td></tr>  
 <tr><td>Password:</td><td><input type="password" name="password"></td></tr>  
 <tr><td colspan="2"><input name="submit" type="submit" value="Login"></td></tr>  
 <input name="\_csrf" type="hidden" value="3e8b0489-c217-4d03-837a-e6e97b1ad575">  
 </tbody></table>  
</form></body></html>

-기본 login html을 확인해 보면 csrf 토큰으로 된 \_csrf 필드가 포함되어 있다는 것을 확인해야 한다.

**9.4.2 HTTP 기본 인증 활성화하기**

HTTP 기본 인증 활성화 방법은 configure() 메소드로 전달되는 HttpSecurity 객체의 httpBasic()

메소드의 호출이다. 원하는 대로 범위를 명시하기 위해서 realmName()을 호출하는 것도 가능하다.

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http  
 .formLogin() // 기본 로그인 페이지 활성화  
 .loginPage("/login")  
 .and()  
 .httpBasic()  
 .realmName("Spitter")  
 …

**9.4.3 기억하기 기능 활성화**

한번 로그인 하면 재방문 시에는 애플리케이션이 사용자를 기억할 수 있도록 한다.

HttpSecurity 객체에서 rememberMe()를 호출해 주면 된다.

http  
 .formLogin() // 기본 로그인 페이지 활성화  
 .loginPage("/login")  
 .and()  
 .rememberMe()  
 .tokenValiditySeconds(2419200)  
 .key("spitterKey")  
 .and()

form에 remember\_me 와 매칭된다.

|  |
| --- |
| <input id=”remember\_me” name=”remember-me” type=”checkbox” /> |

**9.4.4 로그아웃하기**

로그 아웃은(기본 설정으로) /logout으로 가능 요청을 가로채는 서블릿 필터로 구현되어 있다.

따라서 애플리케이션에 로그아웃을 추가하는 것은 아래의 링크를 추가하면 되는 정도로 간단하다.

|  |
| --- |
| <a th:href=”@{/logout}”>Logout</a> |

/logout에 대한 요청이 스프링 시큐리티의 LogoutFilter에 의해 처리된다.

로그아웃 시, 모든 기억하기 토큰도 정리된다.

**CHAPTER10**

**10.1 스프링의 데이터 액세스 철학**

애플리케이션 내 모든 컴포넌트에 걸쳐서 퍼시스턴스 로직이 산란(scattering)되는 것을 피하기 위해서 데이터 액세스는 해당 태스크에 집중된 하나 이상의 컴포넌트로 이루어진다.

해당 컴포넌트들은 일반적으로 DAO(Data-Access Obejct) 또는 저장소라 부른다.

서비스 객체는 인터페이스를 통해서 저장소에 액세스한다. 먼저, 특정 데이터 액세스 구현에 밀착되어 있지 않으므로 서비스 객체는 쉽게 테스트할 수 있다.

**인터페이스와 스프링**

인터페이스는 느슨한 코드 커플링을 위해 작성되어야 하며, 데이터 액세스 계층에서만 아니라 애플리케이션의 모든 계층을 사용해야 한다.

**10.1.1 스프링의 데이터 액세스 예외 계층 구조**

다음은 SQLException을 발생시키는 흔한 문제이다

-애플리케이션이 데이터베이스에 연결될 수 없다.

-수행할 쿼리 문장에 문법적인 오류가 있다.

-쿼리 문장에서 참조하는 테이블이나 컬럼(column)이 존재하지 않는다.

-레코드 삽입이나 업데이트 시도가 데이터베이스 제약 사항을 위배했다.

**스프링이 제공하는 퍼시스턴스 플랫폼 – 독립적인 예외**

스프링 JDBC는 JDBC와는 대조적으로 스프링은 예외가 발생한 문제 상황을 잘 설명하는 데이터 액세스 예외들을 제공한다.

**catch블록이 없다**

DataAccessException이 특별한 점은 이 예외가 비검사형 예외(unchecked exception)라는 점이다.

바꿔 말하면, 스프링이 던지는 모든 데이터 액세스 예외는 반드시 잡아서 처리할 필요가 없다.

스프링은 많은 예외가 catch 블록 내에서 해결할 수 없는 문제 때문에 발생한다고 보고 있다.

따라서 스프링은 개발자에게 catch 블록의 작성을 강제하는 대신 비검사형 예외를 사용하도록 촉진한다.

**10.1.2 데이터 액세스 템플릿화 (서비스 템플릿)**

소프트웨어 용어로 말하면 템플릿 메소드는 어떤 절차의 구현 종속적인 부분을 특정 인터페이스에 위임한다. 이 인터페이스를 서로 다르게 구현함으로써 위임된 부분에 특화된 부분을 정의하게 된다.



이것이 바로 스프링 데이터 액세스에 적용한 것과 동일한 패턴이다.

스프링은 데이터 액세스 절차상에서 고정된 단계와 가변적인 단계를 템플릿(template)과 콜백(callback)이라는 두 가지의 별도 클래스로 분리했다.

스프링의 템플릿 클래스는 트랜잭션 제어, 자원 관리 및 예외 처리와 같은 데이터 액세스의 고정된 부분을 담당한다. 반면에 애플리케이션에서 담당해야 할, 예를 들어 질의객체(statement) 생성이나 파라미터 바인딩, 질의 결과 추출과 변환 등의 내용은 콜백을 구현해서 처리한다.

**10.2 데이터 소스 설정**

데이터 액세스가 어떤 형태라도 데이터 소스에 대한 래퍼런스를 설정해야 한다.

스프링 애플리케이션에 데이터 소스 빈을 설정하는 방법은 다음과 같다.

-JDBC 드라이버를 통해 선언된 데이터 소스

-JNDI에 등록된 데이터 소스

-커넥션 풀링(pooling)을 하는 데이터 소스

**10.2.1 JNDI 소스 이용**

JNDI는 DB pool을 미리 Naming 시켜두는 방법

JDBC는 DB을 연결할 때 쓰는 API패키지로 DB연결할 때 쓰인다.

스프링에서는 JNDI에 등록된 데이터 소스의 레퍼런스를 설정하고, 이를 데이터 소스가 필요한 클래스에 일반 빈과 동일한 방식으로 와이어링한다.

**XML**

<jee:jndi-lookup id="dataSource"

jndi-name="jdbc/SpitterDS" resource-ref="true" />

jndi-name 프로퍼티만 지정된 경우에는 데이터 소스는 지정된 jndi-name을 그대로 이용해서 검색한다.

그러나 애플리케이션이 자바 애플리케이션 서버에서 기동되는 경우에는 resource-ref 프로퍼티를 true로 설정하여 jndi-name 값의 앞에 java:comp/env/가 붙은 이름을 사용한다.

**JavaConfig**

@Bean  
public JndiObjectFactoryBean dataSource() {  
 JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();  
 jndiObjectFB.setJndiName("jdbc/SpitterDS");  
 jndiObjectFB.setResourceRef(true);  
 jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);  
 return jndiObjectFB;  
}

자바 설정을 사용한다면 JNDI에서 DataSource를 검색하기 위해 JndiObjectFactoryBean을 사용한다.

**Was에 DataSource를 설정하는 이유**

이클립스에서 개발을 할때는 순수JDBC또는 Spring에서 제공하는 기본적인 JDBC설정을 사용해서 개발을 합니다. 하지만 웹 애플리케이션을 운영 서버로 deploy할 경우 JDBC 설정이 달라집니다.

그 이유는?

1) 실제 서비스 운영은 개발자들이 만든 어플리케이션을 잘 모르는 사람이 운영을 하게 된다.

그래서 소스 레벨에서 설정되어 있는 것보다 WAS에 설정되어 있는 것을 선호

2) WAS(1) : 웹 어플리케이션(N) 인 경우 WAS가 관리하지 않고 n개의 애플리케이션들이 동일 유를 본다면 DB Pool도 N개 나오게 된다. 그러면 관리도 어렵고, 자원 또한 낭비가 된다.

그래서 WAS에 설정되어 있는 것이 자원 낭비를 줄일 수 있다.

3) 장애가 나거나 성능이 정상적이지 못할 때 다른 한 서버가 대신 일을 해주는 failover 기능이다.

**즉 운영, 관리, 최적화, 장애대처 등 다양한 이점이 있기 때문에 AWS에 DB Source를 설정하는 것이다.**

**10.2.2 풀링 기능이 있는 데이터 소스 사용하기**

JNDI에서 데이터 소스를 가져올 수 없는 경우, 차선책은 풀링 기능이 있는 데이터 소스를 직접 스프링에 설정하는 것이다. 다음 오픈 소스를 사용할 수 있다.

-Apache 공통 DBCP

-c3P0

-BoneCP

**DBCP 설정방법**

**XML**

<!-- DBCP -->  
<bean id="dataSoruce" class="org.apache.commons.dbcp.BasicDataSource">  
 <property name="driverClassName" value="org.h2.Driver"/>  
 <property name="url" value="jdbc:h2:tcp://localhost/~/spitter"/>  
 <property name="username" value="sa"/>  
 <property name="password" value=""/>  
 <property name="initialSize" value="5"/>  
 <property name="maxActive" value="10"/>  
</bean>

**JavaConfig**

@Bean  
public BasicDataSource dataSource2() {  
 BasicDataSource ds = new BasicDataSource();  
 ds.setDriverClassName("org.h2.Driver");  
 ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");  
 ds.setUsername("sa");  
 ds.setPassword("");  
 ds.setInitialSize(5);  
 ds.setMaxActive(10);  
 return ds;  
}

**BasicDataSource의 풀 설정 프로퍼티**

|  |  |
| --- | --- |
| 풀 설정 프로퍼티 | 설 명 |
| initialSize | 해당 풀이 시작될 때 생성할 커넥션 수 |
| maxActive | 해당 풀에서 동시에 제공할 수 있는 최대 커넥션 수. 0은 무제한 |
| maxIdle | 해당 풀에서 동시에 휴먼(idle) 상태로 유지될 수 있는 최대 커넥션 수. 0은 무제한 |
| maxOpenPreparedStatements | 질의객체(statement) 풀에서 동시에 제공할 수 있는 최대  PreparedStatement 수. 0은 무제한 |
| maxWait | 해당 풀에 커넥션을 요청했을 때 대기 가능한 최대 시간.  이 시간이 지나면 예외가 발생한다. |
| minEvictableIdleTimeMillis | 해당 풀에서 커넥션을 제거하기 전에 휴먼 상태로 남아 있을 수 있는 시간 |
| minIdle | 해당 풀에서 커넥션이 휴먼 상태로 유지될 수 있는 커넥션 수 |
| poolPreparedStatements | PreparedStatement의 풀링 여부를 나타내는 boolean 값 |

**10.2.3 JDBC 드라이버 기반 데이터 소스**

스프링에 설정할 수 있는 가장 단순한 데이터 소스는 JDBC 드라이버를 통해 정의된 것이다.

스프링은 이러한 유형으로 두 가지의 데이터 소스 클래스를 제공한다. (org.springframework, jdbc.datasource 패키지 안에 있다.)

-DriverManagerDataSource : 애플리케이션이 커넥션을 요청할 때마다 새로운 커넥션을 반환한다.

DBCP의 BasicDataSource와는 달리 커넥션은 풀링되지 않는다.

-SimpleDriverDataSource : OSGi 컨테이너와 같은 특정 환경에서 발생할 수 있는 클래스 로딩 문제를 극복하기 위해 직접 JDBC 드라이버를 사용하는 경우를 제외하면, DriverManager와 거의 동일

-SingleConnectionDataSource : 커넥션을 요청하면 항상 동일한 커넥션을 반환한다.

**XML**

<bean id="dataSource2" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
 <property name="driverClassName" value="org.h2.Driver"/>  
 <property name="url" value="jdbc:h2:tcp://localhost/~/spitter"/>  
 <property name="username" value="sa"/>  
 <property name="password" value=""/>  
</bean>

**JavaConfig**

@Bean  
public javax.sql.DataSource dataSource3() {  
 DriverManagerDataSource ds = new DriverManagerDataSource();  
 ds.setDriverClassName("org.h2.Driver");  
 ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");  
 ds.setUsername("sa");  
 ds.setPassword("");  
 return ds;  
}

SingleConnectionDataSource는 사용할 수 있는 데이터베이스 커넥션이 오직 하나뿐

🡺멀티스레드 애플리케이션에서 정상적으로 동작하지 않는다.

DriverManagerDataSource의 경우, 멀티스레드에는 동작하지만

🡺커넥션이 필요할 때마다 새로운 커넥션을 생성하므로 심각한 성능 저하 유발

결론 : 스프링 커넥션풀을 사용하지 말고 오픈소스 사용

**10.2.4 임베디드 데이터 소스 사용하기**

제품화 세팅에서는 유용하지 않을 수도 있지만, 개발과 테스트 목적으로는 괜찮은 선택이다.

**XML**

<jdbc:embedded-database id="dataSource3" type="H2">  
 <jdbc:script location="com/sjb/chapter10/schema.sql" />  
</jdbc:embedded-database>

**JavaConfig**

@Bean  
public javax.sql.DataSource dataSource4() {  
 return new EmbeddedDatabaseBuilder()  
 .setType(EmbeddedDatabaseType.*H2*)  
 .addScript("classpath:schema.sql")  
 .addScript("classpath:test-data.sql")  
 .build();  
}

**10.2.5 데이터 소스 선택을 위한 프로파일링하기**

// 임베디드  
@Profile("development")  
@Bean  
public javax.sql.DataSource embeddedDataSource() {  
 return new EmbeddedDatabaseBuilder()  
 .setType(EmbeddedDatabaseType.*H2*)  
 .addScript("classpath:schema.sql")  
 .addScript("classpath:test-data.sql")  
 .build();  
}  
  
// DBCP  
@Profile("qa")  
@Bean  
public BasicDataSource data() {  
 BasicDataSource ds = new BasicDataSource();  
 ds.setDriverClassName("org.h2.Driver");  
 ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");  
 ds.setUsername("sa");  
 ds.setPassword("");  
 ds.setInitialSize(5);  
 ds.setMaxActive(10);  
 return ds;  
}  
  
// JNDI  
@Profile("production")  
@Bean  
public JndiObjectFactoryBean dataSource() {  
 JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();  
 jndiObjectFB.setJndiName("jdbc/SpitterDS");  
 jndiObjectFB.setResourceRef(true);  
 jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);  
 return jndiObjectFB;  
}

개발 시 임베디드, QA 시 DBCP, 제품 적용시 JNDI를 사용할 수 있다.

프로파일링이 유효하다는 가정하에 데이터 소스는 프로파일링을 통해 런타임 시에 선택할 수 있다.

**10.3 스프링과 JDBC**

JDBC는 다른 프레임워크의 질의 언어를 배워야 할 필요가 없다. 또한 다른 기술 대산 JDBC를 사용하면 데이터 액세스 성능을 더 세밀하게 조율한다.

더욱이 JDBC는 퍼시스턴스 프레임워크를 사용했을 때보다 훨씬 낮은 수준에서 데이터를 조작할 수 있다.

**10.3.1 지저분한 JDBC 코드 해결**

JDBC가 데이터베이스와 밀접하게 동작하는 API를 제공하는 한편, 데이터베이스 액세스와 관련된 모든 처리는 전적으로 개발자의 손에 달렸다.

**전통적인 JDBC**

public class OldJdbc {  
 private static final String *SQL\_INSERT\_SPITTER* = "insert into spitter (username, password, fullname) values (?, ?, ?)";  
 private DataSource dataSource;  
  
 public void addSpitter(Spitter spitter) {  
 Connection conn = null;  
 PreparedStatement stmt = null;  
  
 try {  
 conn = dataSource.getConnection(); // 커넥션 얻기  
 stmt = conn.prepareStatement(*SQL\_INSERT\_SPITTER*);  
 stmt.setString(1, spitter.getUsername());  
 stmt.setString(2, spitter.getPassword());  
 stmt.setString(3, spitter.getFullName());  
 stmt.execute();  
 } catch (SQLException e) {  
  
 } finally {  
 try {  
 if (stmt != null) {  
 stmt.close();  
 }  
  
 if (conn != null) {  
 conn.close();  
 }  
 } catch (SQLException e) {  
  
 }  
 }  
 }  
}

정말 이렇게 많은 양의 코드가 필요한 것일까? 사실 그렇지 않다. 오직 몇 줄의 코드만이 실질적인 입력 작업을 수행한다. 하지만 JDBC는 개발자로 하여금 커넥션과 질의객체(statement), 그리고 적절히 SQLException을 처리하는 코드를 작성하게 만든다.

ex) insert, update, select 보면 20%의 코드만이 실제 레코드에 대한 쿼리고 나머지는 단순 반복

**10.3.2 JDBC 템플릿과 놀아 보자**

스프링의 JDBC 프레임워크는 자원 관리와 예외 처리 같은 무거운 짐을 대신 어깨에 짊어지므로 JDBC 코드가 훨씬 간결해진다. 따라서 개발자는 오직 데이터베이스로부터 데이터를 가져오거나 내보내는 데 필요한 코드만을 작성한다.

스프링은 단순 반복적인 데이터 액세스 코드를 템플릿 클래스 뒤로 추상화해 숨긴다.

스프링은 다음과 같이 세 개의 선택 가능한 JDBC 템플릿 클래스를 제공한다.

-JdbcTemplate : 스프링의 가장 기본적인 JDBC 템플릿으로서, 색인된 파라미터(indexed parameter) 기반의 쿼리를 통해 데이터베시르르 쉽게 액세스하는 기능을 제공한다.

-NamedParameterJdbcTemplate : SQL과 값들을 색인된 파라미터 대신 명명된 파라미터로 바인딩하여 쿼리를 수행할 수 있게 해주는 JDBC 템플릿 클래스다.

-SimpleJdbcTemplate : 이 JDBC 템플릿은 자바5가 제공하는 오토박싱(autoboxing), 제너릭스(generics), 가변 파라미터 목록(variable parameter list) 등을 활용해서 더 쉽게 템플릿을 사용한다.

**JdbcTemplate을 사용하여 데이터 추가하기**

모든 JdbcTemplate은 DataSource가 있어야 작동한다.

**JdbcTemplate 기반 Update**

public Spitter save(Spitter spitter) {  
 Long id = spitter.getId();  
 if (id == null) {  
 long spitterId = insertSpitterAndReturnId(spitter);  
 return new Spitter(spitterId, spitter.getUsername(), spitter.getPassword(), spitter.getFullName(), spitter.getEmail(), spitter.isUpdateByEmail());  
 } else {  
 jdbcTemplate.update("update Spitter set username=?, password=?, fullname=?, email=?, updateByEmail=? where id=?",  
 spitter.getUsername(),  
 spitter.getPassword(),  
 spitter.getFullName(),  
 spitter.getEmail(),  
 spitter.isUpdateByEmail(),  
 id);  
 }  
 return spitter;  
}

매우 단순한다. 커넥션, 질의객체 생성 코드가 더 이상 없으며, 예외 처리 코드도 없다.

Update () 메소드가 호출될 때, JdbcTemplate은 커넥션을 얻고, 질의객체를 생성하고 SQL 추가를 수행한다.

SQLException이 어떻게 처리되는지 볼 수 없다. 내부적으로 처리한다.

**JdbcTemplate 으로 데이터 읽기**

public Spitter findOne(long id) {  
 return jdbcTemplate.queryForObject(  
 *SELECT\_SPITTER* + " where id=?", new SpitterRowMapper(), id);  
}

...  
private static final class SpitterRowMapper implements RowMapper<Spitter> {  
 public Spitter mapRow(ResultSet rs, int rowNum) throws SQLException {  
 long id = rs.getLong("id");  
 String username = rs.getString("username");  
 String password = rs.getString("password");  
 String fullName = rs.getString("fullname");  
 String email = rs.getString("email");  
 boolean updateByEmail = rs.getBoolean("updateByEmail");  
 return new Spitter(id, username, password, fullName, email, updateByEmail);  
 }  
}

findOne() 메소드는 쿼리를 수행하기 위해 queryForObject를 사용한다.

queryForObject는 세 가지 파라미터를 가진다.

-데이터베이스에서 데이터를 검색하기 위해서 사용되는 SQL을 포함하는 String

-ResultSet에서 값을 추출하고 도메인 객체를 생성하는 RowMapper 객체

-인덱스된 쿼리 파라미터로 한정된 값의 가변 인자 리스트 (바인딩변수)

**JdbcTemplate을 가지고 자바8 람다 사용하기**

**메소드 레퍼런스 형식**

public Spitter findOne(long id) {  
 return jdbcTemplate.queryForObject(  
 *SELECT\_SPITTER* + " where id=?", this::mapSpitter, id);  
}

private Spitter mapSpitter(ResultSet rs, int row) throws SQLException {  
 return new Spitter(  
 rs.getLong("id"),  
 rs.getString("username"),  
 rs.getString("password"),  
 rs.getString("fullName"),  
 rs.getString("email"),  
 rs.getBoolean("updateByEmail")  
 );  
}

**CHAPTER11**

JDBC로는 한계가 있었고 다음과 같은 좀 더 복잡하고 정교한 기능도 필요해졌다.

-지연 로딩(lazy loading) : 지연 로딩은 우리가 필요로 하는 실제 데이터만 불러올 수 있다.

-조기 인출(eager fetching) : 지연 로딩의 반대 개념이다. 조기 인출은 한 번의 쿼리로 전체 객체 구조를 가져올 수 있다. 이를 통해 데이터베이스와 애플리케이션 사이에서 발생하는 값비싼 왕복 비용을 줄일 수 있다.

-캐스케이딩(cascading) : 데이터베이스 테이블에 변경을 가했을 때 다른 테이블도 동일하게 변경되어야 하는 경우가 있다. 부모 개념의 데이터가 삭제 되면 연관 데이터도 삭제되야 한다.

이런 서비스를 일반적으로 객체 관계 매핑(ORM, Object-Relational Mapping)이라고 하는데,

애플리케이션의 퍼시스턴스 계층에 ORM 도구를 사용하면 많은 양의 코드를 줄이고 개발 시간을 단축한다.

스프링은 ROM 프레임워크를 지원하기 위해 프레임워크에 대한 통합 지점과 함께 아래와 같은 부가적인 서비스를 제공한다.

-스프링의 선언적 트랜잭션에 대한 통합된 지원

-투명한 예외 처리

-스레드 안정성(thread-safe)을 갖춘 경량의 템플릿 클래스

-DAO 지원 클래스

-자원 관리

**11.1 스프링과 하이버네이트 통합**

하이버네이트는 오픈 소스 퍼시스턴스(persistence) 프레임워크로서 개발자 커뮤니티에서 괄목할만한 인기를 얻고 있다. 하이버네이트는 캐시(caching), 지연 로딩(lazy loading), 조기 인출(eager fetching)과 분산 캐시(distributed caching) 등이 있다.

**11.1.1 하이버네이트 세션 팩토리 선언**

태초부터 org.hibernate.Session은 하이버네이트 작업에 있어 가장 중심이 되는 인터페이스다.

Session 인터페이스는 데이터베이스로부터 객체에 대한 저장(save), 업데이트(update), 삭제(delete) 그리고 로드(load)와 같은 기본적인 데이터 액세스 기능을 제공한다.

하이버네이트 Session 객체의 레퍼런스를 얻으려면 하이버네이트의 SessionFactory 인터페이스를 이용해야 한다. SessionFactory는 하이버네이트 Session 객체들을 열고 닫으며, 관리에 대한 책임을 진다.

버전 3.1에서 스프링은 세 개의 세션 팩토리 빈을 가진다.

-org.springframework.orm.hibernate3.LocalSessionFactoryBean

-org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean

-org.springframework.orm.hibernate4.LocalSessionFactoryBean

하이버네이트 세션 팩토리 빈을 설정할 때 선택의 기로에 서게 된다. 결정은 애너테이션을 이용할지 아니면 하이버네이트의 XML 매핑 파일을 이용하여 퍼시스턴스 도메인 객체를 설정할지…

만일 하이버네이트 3.2 또는 더 높은버전을 사용하고 **XML로 매핑한다면** LocalSessionFactoryBean을 설정한다.

@Bean  
public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {  
 LocalSessionFactoryBean sfb = new LocalSessionFactoryBean();  
 sfb.setDataSource(dataSource);  
 sfb.setMappingResources(new String[]{"Spitter.hbm.xml"});  
 Properties props = new Properties();  
 props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");  
 sfb.setHibernateProperties(props);  
 return sfb;  
}

3개의 프러퍼티

-dataSource : DataSource 빈의 레퍼런스로 설정

-mappingResources : 애플리케이션의 퍼시스턴스 전략을 정의하는 하나 이상의 하이버네이트 매핑 파일 나열

-hibernateProperties : 하이버네이트가 작동하는 세세한 방법 설정

애너테이션 지향 퍼시스턴스가 더 적합한 경우는 **AnnotationSessionFactoryBean** 을 사용. 단 하이버네이트4를 아직 사용하지 않은 경우에 한해서…

(4에서는 deprecated 되고 LocalSessionFactoryBean를 사용)

@Bean  
public AnnotationSessionFactoryBean sessionFactory2(DataSource dataSource) {  
 AnnotationSessionFactoryBean sfb = new AnnotationSessionFactoryBean();  
 sfb.setDataSource(dataSource);  
 sfb.setPackagesToScan(new String[] {"com.habuma.spitter.domain"});  
 Properties props = new Properties();  
 props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");  
 sfb.setHibernateProperties(props);  
 return sfb;  
}

어떻게 애너테이션 기반 매핑이 이루어지는지를 살펴본다.

@Bean  
public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {  
 LocalSessionFactoryBean sfb = new LocalSessionFactoryBean();  
 sfb.setDataSource(dataSource);  
 sfb.setPackagesToScan(new String[]{"com.habuma.spitter.domain"});  
 Properties props = new Properties();  
 props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");  
 sfb.setHibernateProperties(props);  
 return sfb;  
}

packagesToScan 프로퍼티를 사용하면 스프링은 패키지들을 검사하여 퍼시스턴스 애너테이션이 적용된 도메인 클래스를 찾는다. 여기에는 JPA의 @Entity나 @MappedSuperclass 그리고 하이버네이트의 자체적인 @Entity 애너테이션이 적용된 클래스가 포함된다.

프로퍼티 annotatedClasses 내 검증된 클래스 명 리스트를 지정하여 애플리케이션의 모든 퍼시스턴스 클래스를 나타낼 수 있다.

sfb.setAnnotatedClasses(  
 new Class<?>[]{Spitter.class, Spittle.class}  
);

**11.1.2 스프링으로부터 해방된 하이버네이트 구성**

스프링과 하이버네이트 초반부에서, 저장소 클래스 작성에서 스프링의 HibernateTemplate 사용을 포함한다. 컨텍스트 의존 세션 없이도 스프링의 하이버네이트 템플릿은 트랜잭션당 하나의 세션을 보장하는 작업을 처리한다. 이 접근법의 단점은 저장소 구현에서 스프링에 직접 커플링이 되어야 한다는 점이다.

하지만 지금은 하이버네이트 상황 세션을 이용하므로 HibernateTemplate이 필요 없다.

@Repository  
public class HibernateSpitterRepository implements SpitterRepository {  
 private SessionFactory sessionFactory;  
  
 @Inject  
 public HibernateSpitterRepository(SessionFactory sessionFactory) {  
 this.sessionFactory = sessionFactory; // 세션 팩토리 주입  
 }  
  
 private Session currentSession() {  
 return sessionFactory.getCurrentSession();  
 }  
  
 @Override  
 public long count() {  
 return findAll().size();  
 }  
  
 @Override  
 public Spitter save(Spitter spitter) {  
 Serializable id = currentSession().save(spitter);  
 return new Spitter((Long) id,  
 spitter.getUsername(),  
 spitter.getPassword(),  
 spitter.getFullName(),  
 spitter.getEmail(),  
 spitter.isUpdateByEmail());  
 }  
  
 @Override  
 public Spitter findOne(long id) {  
 return (Spitter) currentSession().get(Spitter.class, id);  
 }  
  
 @Override  
 public Spitter findByUsername(String username) {  
 return (Spitter) currentSession()  
 .createCriteria(Spitter.class)  
 .add(Restrictions.*eq*("username", username))  
 .list().get(0);  
 }  
  
 @Override  
 public List<Spitter> findAll() {  
 return (List<Spitter>) currentSession()  
 .createCriteria(Spitter.class).list();  
 }  
}

가장 먼저 주목할 점은 스프링의 @Injection 애너테이션을 이용하여 HibernateSpitterRepository의 sessionFactory를 자동으로 주입한 다음, currentSession() 메소드에서 이 SessionFactory를 이용해 현재 트랜잭션의 세션을 얻었다.

또 하나 주목할 점은 클래스에 @Repository 애너테이션을 적용했다는 점이다. 두가지 기능을 수행한다.

1) @Repository는 또 다른 스프링의 스테레오타입 애너테이션 중 하나로, 스프링의 컴포넌트 스캐닝에 의해 스캔된다.

public class RepositoryExceptions {  
 @Bean  
 public BeanPostProcessor persistenceTranslation() {  
 return new PersistenceExceptionTranslationPostProcessor();  
 }  
}

2) 명시적 설정을 줄이는 데 도움을 주기도 하지만, 또 다른 목적을 위해 사용할 수도 있다.

PersistenceExceptionTranslationPostProcessor는 빈의 후처리기인데, 이는 어드바이저를

@Repository 애너테이션이 적용된 모든 빈에 추가하여 플랫폼에 특화된 모든 예외를 잡은 후 스프링의 비검사형 데이터 액세스 예외로 다시 던진다.

**11.2 스프링과 자바 퍼시스턴스 API**

자바 퍼시스턴스 API(JPA)는 차세대 자바 퍼시스턴트 표준으로 EJB2의 엔티티 빈의 일부분에서 나왔다. JPA는 하이버네이트와 JDO(Java Data Objects)로부터의 아이디어인 POJO 기반의 퍼시스턴스 메커니즘이며, 자바5 애너테이션들을 혼합한다.

**11.2.1 엔티티 관리자 팩토리 설정**

JPA로 만든 애플리케이션은 EntityManagerFactory의 구현 객체를 이용해서 EntityManager의 인스턴스를 획득해야 한다.

JPA 명세에는 다음과 같은 두 가지 유형의 엔티티 관리자가 정의되어 있다.

-애플리케이션 관리형 : 애플리케이션이 엔티티 관리자 팩토리에서 직접 엔티티 관리자를 요청함으로써 엔티티 관리자가 생성되는 유형. 자바 EE 컨테이너 없이 실행되는 독립형(standalone) 애플리케이션에 가장 적합하다.

-컨테이너 관리형 : 자바 EE 컨테이너에 의해 생성되고 관리되는 엔티티 관리자다. 애플리케이션은 엔티티 관리자 팩토리와 전혀 직접 상호작용하지 않고, 대신 엔티티 관리자를 종속객체 주입이나 JNDI를 통해 직접적으로 획득하는 방식이다.

🡺 유형은 다르더라도 모든 엔티티 관리자는 동일한 EntityManager 인터페이스를 구현한다.

-애플리케이션 관리형 EntityManager는 PersistenceProvider의 createEntityManagerFactory() 메소드로 생성한 EntityManagerFactory에 의해 생성되는 반면

-컨테이너 관리형의 경우 EntityManagerFactory를 PersistenceProvider의

createContainerEntityManagerFactory() 메소드를 통해서 획득한다.

스프링은 각 유형에 대응하는 팩토리 빈을 갖추고 있어서 적절하게 엔티티 관리자 팩토리를 생성한다.

-LocalEntityManagerFactoryBean : 애플리케이션 관리형 EntityManagerFactory를 생성

-LocalContainerEntityManagerFactoryBean : 컨테이너 관리형 EntityManagerFacotry를 생성

🡺스프링의 JpaTemplate이 두 가지 유형의 EntityManagerFactory 모두의 복잡한 세부 사항을 감춰 주므로 개발자가 작성하는 데이터 액세스 코드는 본연의 목적인 데이터 액세스에만 충실한다.

**애플리케이션 관리형 JPA 구성하기**

애플리케이션 관리형 엔티티 관리자 팩토리는 설정 대부분을 persistence.xml 설정 파일로부터 가져온다. 이 파일은 반드시 클래스패스(classpath)에 있는 META-INF 디렉터리에 존재 해야 된다.

persistence.xml 파일의 목적은 하나 이상의 퍼시스턴스 유닛을 선언하는데 있다.

<persistence  
 xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">  
 <persistence-unit name="spitterPU">  
 <class>com.sjb.chapter11.domain.Spitter</class>  
 <class>com.sjb.chapter11.domain.Spittle</class>  
 <properties>  
 <property name="toplink.jdbc.driver" value="org.hsqldb.jdbcDriver"/>  
 <property name="toplink.jdbc.url" value="jdbc:hsqldb:hsql://localhost/spitter/spitter"/>  
 <property name="toplink.jdbc.user" value="sa"/>  
 <property name="toplink.jdbc.password" value=""/>  
 </properties>  
 </persistence-unit>  
</persistence>

**LocalEntityManagerFactoryBean**

Bean  
public LocalEntityManagerFactoryBean entityManagerFactoryBean() {  
 LocalEntityManagerFactoryBean emfb  
 = new LocalEntityManagerFactoryBean();  
 emfb.setPersistenceUnitName("spitterPU");  
 return emfb;  
}

persistenceUnitName 프로퍼티에는 persistence.xml 파일에 정의한 퍼시스턴스 유닛 이름을 설정한다.

그런데 JPA에 스프링을 더하게 되면 이젠 애플리케이션 코드 대신 JpaTemplate이 PersistenceProvider와 상호작용을 하는 창구가 된다. 따라서 설정 정보를 persistence.xml 파일에 뽑아 놓는 것은 현명해 보이지 않는다.

🡺스프링으로 설정한 데이터 소스 같은 것도 JAP에서 이용할 수 없게 된다.

**컨테이너 관리형 JPA 구성하기**

persistence.xml에 설정하는 대신에 이 정보를 스프링 애플리케이션 컨텍스트에 설정한다.

**LocalContainerEntityManagerFactoryBean**

@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean(DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {  
 LocalContainerEntityManagerFactoryBean emfb =  
 new LocalContainerEntityManagerFactoryBean();  
 emfb.setDataSource(dataSource);  
 emfb.setJpaVendorAdapter(jpaVendorAdapter);  
 return emfb;  
}

위 소스를 보면 dataSource 프로퍼티에는 스프링에 설정한 데이터 소스를 지정했다.

Javax.sql.DataSource의 어떤 구현객체라도 지정할 수 있다.

jpaVenderAdapter

-EclipseLinkJpaVendorAdapter

-HibernateJpaVendorAdapter

-OpenJpaVendorAdapter

-TopLinkJpaVendorAdapter(스프링 3.1에서 삭제)

@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean(DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {  
 LocalContainerEntityManagerFactoryBean emfb =  
 new LocalContainerEntityManagerFactoryBean();  
 emfb.setDataSource(dataSource);  
 emfb.setJpaVendorAdapter(jpaVendorAdapter);  
 emfb.setPackagesToScan("com.sjb.chapter11.jpa");  
 return emfb;  
}

setPackagesToScan으로 @Entity로 애너테이션된 클래스를 위해 해당 패키지를 스캔한다. 따라서 persistence.xml에서 명시적으로 선언할 필요가 없다.

**11.3 스프링 데이터를 사용한 자동 JPA 저장소**

public interface SpitterRepository extends JpaRepository<Spitter, Long> {

}

여기서 SpitterRepository는 스프링 데이터 JPA의 JpaRepository를 확장한다.

JpaRepository는 파라미터를 가지고, 지속적인 Spitter 객체를 위한 저장소로 사용되며,

Spitters는 Long 타입의 ID를 가진다.

또한 ID로 Spitter를 찾고, Spitter를 삭제하고, Spitter를 저장하는 것과 같은 일반적인 퍼시스턴스 작업을 수행하기 위한 18가지 방법을 상속한다.

다음 목록은 스프링 데이터 JPA를 사용하기 위하여 필요한 XML 구성을 보여 준다.

**스프링 데이터 JPA 설정하기 (XML)**

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:jpa="http://www.springframework.org/schema/data/jpa"  
 xsi:schemaLocation="  
 http://www.springframework.org/schema/data/jpa  
 http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">  
  
 <jpa:repositories base-package="com.sjb.chapter11" />

<context:component-scan> 요소처럼 <jpa:repositories>는 스캔을 위한 기본 패키지다.

하지만 <context:component-scan>이 @Component로 애너테이션된 클래스용 패키지를 스캔하는 반면에 <jpa:repositories>는 스프링 데이터 JPA 저장소의 인터페이스를 확장하는 인터페이스용 기본 패키지를 스캔한다.

**JavaConfig**

@Configuration  
@EnableJpaRepositories(basePackages = "com.sjb.chapter11")  
public class JpaConfiguration {  
   
}

자바 설정 클래스에서 @EnableJpaRepositories를 사용한다.

스프링 데이터에서 저장소 인터페이스를 발견할 때 JpaRepository, PagingAndSortingRepository, CrudRepository로부터 상속된 모든 18가지 방법의 구현을 포함하여 SpitterRepository 구현을 진행한다.

**11.3.1 쿼리 메소드 정의하기**

public interface SpitterRepository extends JpaRepository<Spitter, Long> {  
 Spitter findByUsername(String username);  
}

findByUsername()을 구현하기 위해서 해야 할 것은 없다. 메소드 시그니처는 메소드 구현체를 만들기 위해 알아야 할 모든 것을 스프링 데이터 JPA에 알려 준다.

저장소 구현체를 만들 때, 스프링 데이터는 저장소 인터페이스의 메소드들을 검증하고, 메소드 명을 파싱하고, 퍼시스턴스 객체의 컨텍스트에서의 메소드 용도를 이해한다.

그럼에도 불구하고 스프링 데이터의 미니 DSL(Domain-Specific Language)은 메소드 명에서 원하는 쿼리를 나타낼 때 한계점을 가지며 간편하거나 가능하지도 않다. 이때 스프링 데이터는 @Query 애너테이션을 통해 대응 가능하다.

**11.3.2 맞춤형 쿼리 선언하기**

원하는 데이터가 메소드 명으로 제대로 표현되지 않는 상황에서, 수행 쿼리를 스프링 데이터에 제공하기 위해 @Query 애너테이션을 사용할 수 있다.

@Query("select s from Spitter s where s.email like '%gmail.com'")  
 List<Spitter> findAllGmailSpitters();  
}

아직은 findAllGmailSpitters() 메소드 구현을 만들 수 없다. 단지 스프링 데이터 JPA에 메소드 구현에 대한 힌트를 주기 위해 쿼리를 수행했을 뿐이다.

@Query는 메소드 명명 규칙을 사용하여 원하는 쿼리를 수행하기 어려울 때 유용하다.

명명 규칙을 따른다면 메소드 명이 매우 길 때 유용하다. 예를 들면 다음과 같은

Finder 메소드에서다.

List<Order> findByCustomerAddressZipCodeOrCustomerNameAndCustomerAddressState();

**그러나 단일 JPA 쿼리에 한정된다!!**

**11.3.3 맞춤형 기능 혼합**

스프링 데이터의 메소드 명명 규칙 또는 @Query 애너테이션에서 주어진 쿼리로 나타낼 수 없는 저장소 기능을 원할 경우가 종종 있다.

스프링 데이터 JPA로 할 수 없는 일을 해야 할 때, 스프링 데이터 JPA가 제공할 때보다 낮은 수준에서 JPA와 함께 작동하도록 해야 한다.

스프링 데이터 JPA는 저장소 인터페이스 구현체를 생성하고, Impl 접미어를 가지는 인터페이스의 이름과 같은 클래스를 찾는다. 클래스가 존재하는 경우, 스프링 데이터 JPA는 스프링 데이터 JPA에 의해 생성된 방법과 병합된다.

**내 나름의 결론**

MethodQuery로 처리할 수 없는 경우, QueryDsl

QueryDsl로 처리할 수 없는 경우 @Query (Native Query)

@Query로도 부족한 경우 맞춤형 혼합

**CHAPTER 12**

**12.1 MongoDB의 유지성 도큐먼트**

스프링 애플리케이션이 MongoDB를 사용하는 다음과 같은 세 가지 방법을 나타낸다.

-객체 도큐먼트 매핑을 위한 애너테이션

-MongoTemplate을 사용한 템플릿 기반의 데이터베이스 액세스

-자동 런타임 저장소 생성

**도큐먼트 데이터베이스의 단점**

도큐먼트 데이터베이스는 중요 관계성을 가지는 데이터를 저장하기 위한 튜닝이 잘 되어 있지는 않다.

**12.1.1 MongoDB 활성화**

-MongoClient 빈을 설정해서 MongoDB 데이터베이스를 액세스 할 수 있다.

-MongoTemplate 빈을 사용해서 템플릿 기반 데이터 액세스를 수행할 수 있다.

@Configuration  
@EnableMongoRepositories(basePackages = "com.sjb.chapter12") // MongoDB 저장소 활성화  
public class MongoConfig {  
  
  
 @Bean  
 public MongoFactoryBean mongo() { // MongoClient 빈  
 MongoFactoryBean mongo = new MongoFactoryBean();  
 mongo.setHost("localhost");  
 return mongo;  
 }  
  
 @Bean  
 public MongoOperations mongoTemplate(Mongo mongo) { // MongoTeamplte 빈  
 return new MongoTemplate(mongo, "OrdersDB");  
 }  
  
}

EnableMongoRepositories를 통해 MongoDB 활성화

MongoFactoryBean은 스프링 데이터 MongoDB와 데이터베이스 간의 다리 역할을 한다.

MongoClient를 사용하면 UnknownHostException을 처리 해야 한다.

MongoTemplate 빈은 다른 빈 메소드와 데이터베이스명을 사용하여 생성된 Mongo 인스턴스를 참조하여 생성된다.

@Configuration  
@EnableMongoRepositories(basePackages = "com.sjb.chapter12") // MongoDB 저장소 활성화  
public class MongoConfig extends AbstractMongoConfiguration {  
 @Override  
 public Mongo mongo() throws Exception {  
 return new MongoClient();  
 }  
  
 @Override  
 protected String getDatabaseName() {  
 return "OrdersDB";  
 }  
}

위의 것과 거의 동일. MongoTemplate 빈을 선언하는 대신 getDatabaseName으로 오버라이드.

예외 상황이 발생할 수 있으므로 MongoFactoryBean을 사용하지 않고 직접 MongoClient를 가지고 작업한다.

@Override  
public Mongo mongo() throws Exception {  
 return new MongoClient("mongodbserver");  
}

MongoDB가 다른서버에서 동작 시 지정

@Override  
public Mongo mongo() throws Exception {  
 return new MongoClient("mongodbserver", 37017);  
}

포트도 지정 가능

@Override  
public Mongo mongo() throws Exception {  
 MongoCredential credential =  
 MongoCredential.*createMongoCRCredential*(  
 env.getProperty("mongo.username"),  
 "OrdersDB"  
 env.getProperty("mongo.password").toCharArray());  
  
 return new MongoClient(  
 new ServerAddress("localhost", 37017),  
 Arrays.*asList*(credential));  
}

인증된 MongoDB 서버로 액세스하기 위해 MongoClient 만들기

**스프링 데이터 MongoDB는 XML 설정 옵션을 제공**

<mongo:repositories base-package="com.sjb.chapter12" />  
<mongo:mongo />  
<bean id="mongoTemplate"  
 class="org.springframework.data.mongodb.core.MongoTemplate">  
 <constructor-arg ref="mongo" />  
 <constructor-arg value="OrdersDB" />  
</bean>

지금까지 스프링 데이터 MongoDB를 설정하였으며, 도큐먼트를 저장하고 검색하기 위한 준비를 마쳤다.

**12.1.2 MongoDB 퍼시스턴트를 위한 애너테이션 모델 타입**

JPA를 사용할 때, 자바 엔티티 타입을 관계형 테이블과 컬럼에 매핑해야 한다.

JPA 스펙은 객체-관계형 매핑, JPA를 지원하기 위한 여러 애너테이션을 제공했다.

그러나MongoDB는 객체-도큐먼트 매핑 애너테이션을 가지지 않는다.

스프링 데이터 MongoDB는 애너테이션을 사용하여 갭을 채울 수 있는 기회를 가지며, 자바 타입을 MongoDB 도큐먼트에 매핑한다.

**객체-도큐먼트 매핑을 위한 스프링 데이터 MongoDB 애너테이션**

|  |  |
| --- | --- |
| 애너테이션 | 내용 |
| @Document | MongoDB 도큐먼트로 매칭될 수 있는 도메인 객체 구별 |
| @Id | ID 필드임을 표시 |
| @DbRef | 다른 데이터베이스의 다른 도큐먼트를 참조하기 위한 필드임을 표시 |
| @Field | 도큐먼트 필드를 위한 맞춤형 메타데이터 정의 |
| @Version | 버전 필드로 사용되는 프로퍼티 구별 |

**12.1.3 MongoTempalte을 사용하여 MongoDB 액세스하기**

@Autowired  
MongoOperations mongo;

MongoOperations는 MongoTemplate 구현 인터페이스이며, 주입 시 구체적 구현물로 직접 사용되지 않는다.

MongoOperations는 여러 개의 유용한 메소드를 제공한다.

Ex) mongo.getCollection(), mongo.save, mongo.findById, mongo.find, mongo.remove

전형적으로 개발자들은 자기 자신의 설계의 저장소 클래스로 MongoOperations를 주입하고,

저장소 메소드를 구현한다. 그러나 저장소를 작성하는데 애를 쓰고 싶지 않다면, 스프링 데이터 MongoDB는 자동으로 런타임 시의 저장소 구현을 한다.

**12.1.4 MongoDB 저장소 작성하기**

@EnableMongoRepositories를 가지고 스프링 데이터 MongoDB 저장소를 활성화하는 작업이 이미 수행되었으며, 이제 남은 것은 저장소 구현체 폼을 만들 수 있는 인터페이스를 만드는 작업이다.

**스프링 데이터 MongoDB의 자동 저장소 인터페이스 구현하기**

public interface OrderRepository extends MongoRepository<Order, String> {  
   
}

MongoRepository 인터페이스는 두 개의 파라미터를 가진다.

첫 번째는 저장소가 취급하는 @Document 애너테이션된 객체의 타입이다.

두 번째는 @Id 애너테이션된 프로퍼티 타입이다.

MongoRepository 확장에 의해 저장소 인터페이스는 스프링 데이터 MongoDB에 의해 자동으로 구현된 여러 개의 CRUD 동작을 상속한다.

|  |  |
| --- | --- |
| 메소드 | 내용 |
| long count() | 저장소 타입용 도큐먼트 카운트 반환 |
| void delete(Iterable<? Extends T); | 주어진 객체와 관련된 모든 도큐먼트 삭제 |
| void delete(T) | 주어진 객체와 관련된 도큐먼트 삭제 |
| void delete(ID) | ID로 도큐먼트 삭제 |
| void deleteAll() | 주어진 저장소 타입의 모든 도큐먼트 삭제 |
| boolean exists(Object) | 주어진 객체와 관련된 도큐먼트가 존재하면 참 반환 |
| boolean exists(ID) | 주어진 ID에 대한 도큐먼트가 존재하면 참 반환 |
| List<T> findAll() | 저장소 타입별 모든 도큐먼트 반환 |
| List<T> findAll(Iterable<ID>) | 주어진 도큐먼트 ID별 모든 도큐먼트 반환 |
| List<T> findAll(Pageable) | 저장소 타입에 대한 페이징과 정렬된 도큐먼트 리스트 반환 |
| List<T> findAll(Sort) | 주어진 도큐먼트 ID에 대한 모든 도큐먼트의 정렬된 리스트 반환 |
| T findOne(ID) | 주어진 ID에 대한 단일 도큐먼트 반환 |
| <S extends T> Iterable<S>  Save(Iterable<S>) | 주어진 Iterable의 모든 도큐먼트 저장 |
| <S extends T>S save(S) | 주어진 객체에 대한 단일 도큐먼트 저장 |

**맞춤형 쿼리 메소드 추가**

CRUD는 일반적으로 유용하지만, 메소드를 제공하기 위한 저장소가 필요하다.

MongoDB도 JPA가 지원하는 메소드 명명 규칙과 비슷하다.

public interface OrderRepository extends MongoRepository<Order, String> {  
 List<Order> findByCustomer(String c);  
 List<Order> findByCustomerLike(String c);  
 List<Order> findByCustomerAndType(String c, String t);  
 List<Order> findByCustomerLikeAndType(String c, String t);  
}

이제 네 개의 새 메소드를 가진다.

**쿼리 지정하기**

@Query는 JPA를 사용했을 때와 마찬가지로 MongoDB를 사용한다.

차이점은 @Query는 JPA 쿼리 대신 JSON 쿼리 문자열을 사용한다.

다음은 고객명이 ‘Chuck Wagon’인 타입의 모든 주문을 찾는 메소드이다.

@Query("{'customer' : 'Chuck Wagon', 'type' : ?0}")  
List<Order> findChucksOrders(String t);

@Query JSON은 모든 Order 도큐먼트를 대상으로 매칭되는지 파악하며, 매칭되는 도큐먼트가 반환된다. type 프로퍼티가 쿼리 메소드의 0번째 파라미터임을 나타낸다.

**맞춤형 저장소 동작의 혼합**

public class OrderRepositoryImpl implements OrderOperations {  
 @Autowired  
 private MongoOperations mongo;  
  
 @Override  
 public List<Order> findOrdersByType(String t) {  
 String type = t.equals("NET") ? "WEB" : t;  
 Criteria where = Criteria.*where*("type").is(t);  
 Query query = Query.*query*(where);  
  
 return mongo.find(query, Order.class);  
 }  
}

findOrdersByType() 메소드는 생성된 쿼리에 매칭하는 도큐먼트의 데이터베이스에 쿼리하기 위한MongoOperations를 사용한다.

**12.2 Neo4J로 그래프 데이터 사용하기**

도큐먼트 데이터베이스는 대단위의 도큐먼트를 저장하는 반면에 그래프 데이터베이스는 서로간의 관계로 연결된 세밀한 노드 단위로 데이터를 저장한다.

간단히 생각해 보면, 그래프 데이터베이스는 도큐먼트 데이터베이스보다 더 일반적인 목적을 가지고, 관계형 데이터베이스보다는 스키마가 없는 대안을 제공한다.

스프링 데이터 Neo4j는 스프링 데이터 JPA와 스프링 데이터 MongoDB와 동일한 기능을 제공한다. 또한 자바 타입을 노드와 관계성, 템플릿에 기반을 둔 Neo4j 액세스, 구현체 자동 생성에 매핑하기 위한 애너테이션을 제공한다.

**12.2.1 스프링 데이터 Neo4j 설정하기**

**Neo4J 설정**

@Configuration  
@EnableNeo4jRepositories(basePackages = "com.sjb.chapter12")  
public class Neo4jConfig extends Neo4jConfiguration {  
  
 public Neo4jConfig() {  
 setBasePackage("orders");  
 }  
  
 @Bean(destroyMethod = "shutdown")  
 public GraphDatabaseService graphDatabaseService() {  
 return new GraphDatabaseFactory()  
 .newEmbeddedDatabase("/tmp/graphdb");  
 }  
}

@EnableNeo4jRepositories 애너테이션은 스프링 데이터 Neo4j를 활성화하여 자동 Neo4j 저장소 구현체를 생성한다.

graphDatabaseService() 메소드는 임베디드 Neo4j 데이터베이스를 생성하기 위해 GraphDatabaseFactory를 사용한다.

Neo4j에서 임베디드 데이터는 인메모리(in-memory) 데이터베이스와 혼동되어서는 안 된다.

임베디드의 의미는 데이터베이스 엔진이 독립된 서버로 동작되는 것이 아닌 애플리케이션의 일부로서 동일 JVM 내에서 동작하는 것을 의미한다.

**Neo4j XML 설정**

<neo4j:config  
 storeDirectory="/tmp/graphdb"  
 base-package="com.sjb.chapter12"/>  
<neo4j:repositories base-package="com.sjb.chapter12"/>  
  
<neo4j:config base-package="orders"  
 graphDatabaseService="graphDatabaseService"/>  
<bean id="graphDatabaseService" class="org.springframework.data.neo4j.rest.SpringRestGraphDatabase">  
 <constructor-arg value="http://graphdbserver:7474/db/data"/>  
 <constructor-arg value="db.username"/>  
 <constructor-arg value="db.password"/>  
</bean>

**12.2.2 그래프 엔티티 애너테이션하기**

Neo4j는 두 가지 엔티티를 정의한다. 노드(node)와 관계(relationship) 엔티티다.

노드 엔티티는 애플리케이션의 요소들을 나타내고, 관계 엔티티는 요소들이 어떻게 관계를 가지고 있는지를 나타낸다.

@NodeEntity // Order는 노드  
public class Order {  
 @GraphId  
 private Long id; // 그래프 ID  
  
 private String customer;  
  
 private String type;  
  
 @RelatedTo(type = "HAS\_ITEMS") // 아이템의 관계  
 private Set<Item> items = new LinkedHashSet<Item>();

클래스 레벨의 @NodeEntity와 더불어 id 프로퍼티는 @GraphId로 애너테이션된다.

Neo4j의 모든 엔트리들은 그래프 ID를 가진다. 이는 JPA @Entity 또는 MongoDB 도큐먼트의 @Id 애너테이션된 프로퍼티와 거의 동일하다.

Neo4j와 같은 그래프 데이터베이스는 노드와 관계로 표시되는 데이터를 캡처하기에 훌륭하다.

우리가 살고 있는 세상이 서로 관계가 있는 모든 종류의 사물들로 구성된다고 할 때, 그래프 데이터베이스는 넓은 범위의 도메인에 적합하다.

**12.3 Redis에서 키-값 데이터 사용하기**

Redis는 키-값 저장소로 알려진 특별한 종류의 데이터베이스이다.

**12.3.1 Redis에 연결하기**

Redis 연결 팩토리는 Redis 데이터베이스 서버 대상 연결점을 제공한다.

스프링 데이터 Redis는 네 개의 Redis 클라이언트 구현체를 위한 팩토리 연결을 가진다.

-JedisConnectionFactory

-JredisConnectionFactory

-LettuceConnectionFactory

-SrpConnectionFactory

**JedisConnecitonFactory 빈 설정**

// 기본 설정  
@Bean  
public RedisConnectionFactory redisCF() {  
 return new JedisConnectionFactory();  
}  
  
// 다른 호스트  
@Bean  
public RedisConnectionFactory redisCF2() {  
 JedisConnectionFactory cf = new JedisConnectionFactory();  
 cf.setHostName("redis-server");  
 cf.setPort(7379);  
 return cf;  
}  
  
// 인증  
@Bean  
public RedisConnectionFactory redisCF3() {  
 JedisConnectionFactory cf = new JedisConnectionFactory();  
 cf.setHostName("redis-server");  
 cf.setPort(7379);  
 cf.setPassword("foobared");  
 return cf;  
}  
  
// Lettuce  
@Bean  
public RedisConnectionFactory redisCF4() {  
 LettuceConnectionFactory cf = new LettuceConnectionFactory();  
 cf.setHostName("redis-server");  
 cf.setPort(7379);  
 cf.setPassword("foobared");  
 return cf;  
}

모든 Redis 연결 팩토리는 setHostName(), setPort(), set-Password() 메소드를 가진다.

**12.3.2 RedisTemplate을 가지고 작업하기**

public void save() {  
 RedisConnectionFactory cf = config.redisCF();  
  
 // Save  
 RedisConnection conn = cf.getConnection();  
 conn.set("Greeting".getBytes(), "Hello World".getBytes());  
  
 // Get  
 byte[] greetingBytes = conn.get("greeting".getBytes());  
 String greeting = new String(greetingBytes);  
}

이는 문제 없이 잘 동작할 것이다. 하지만 바이트 배열을 사용하길 원하는가?

다른 스프링 데이터 프로젝트와 마찬가지로 Redis는 템플릿을 사용하여 상위 레벨 데이터 액세스 옵션을 제공한다.

**-RedisTemplate**

**-StringRedisTemplate**

RedisTemplate은 Redis 데이터 액세스를 단순화한 클래스이며 키와 여러 타입의 값을 저장하는데, 바이트 배열은 아니다.

키와 같이 String이라는 것을 생각해 볼 때, StringRedisTemplate은 String 포커스를 가지기 위해 RedisTemplate을 확장한다.

다음과 같이 RedisTemplate를 만든다.

RedisConnectionFactory cf = config.redisCF();  
RedisTemplate<String, Product> redis =  
 new RedisTemplate<>();  
redis.setConnectionFactory(cf);

RedisTemplate은 두 가지 타입으로 파라미터화된다. 첫 번째 타입은 키이고, 두 번째는 값이다.

String값과 String 키를 사용할 경우, RedisTemplate 대신 StringRedisTemplate 사용을 고려해야 한다.

**RedisTemplate**은 sub-API를 통해 많은 기능을 제공하고, 컬렉션 값의 단일 값과 차별화를 가진다.

|  |  |  |
| --- | --- | --- |
| **메소드** | **Sub-API** | **인터페이스 설명** |
| opsForValue() | ValueOperations<K,V> | 간단한 값을 가지는 엔트리를 사용하는 동작 |
| opsForList() | ListOperations<K,V> | 리스트 값을 가지는 엔트리를 사용하는 동작 |
| opsForSet() | SetOperations<K,V> | 세트 값을 가지는 엔트리를 사용하는 동작 |
| opsForZSet() | ZSetOperations<K,V> | ZSet(정렬된 세트) 값을 가지는 엔트리를 사용하는 동작 |
| opsForHash() | HashOperations<K, HK, HV> | 해시 값을 가지는 엔트리를 사용하는 동작 |
| boundValueOps(K) | BoundValueOperations<K,V> | 주어진 키에 한정된 간단한 값을 사용하기 위한 동작 |
| boundListOps(K) | BoundListOperations<K,V> | 주어진 키에 한정된 리스트 값을 사용하기 위한 동작 |
| boundSetOps(K) | BoundSetOperations<K,V> | 주어진 키에 한정된 세트 값을 사용하는 동작 |
| boundZSet(K) | BoundZSetOperations<K,V> | 주어진 키에 한정된 ZSet 값을 사용하는 동작 |
| boundHashOps(K) | BoundHashOperations<K,V> | 주어진 키에 한정된 해시 값을 사용하는 동작 |

**간단한 값을 가지고 사용하기**

// Set  
Product product = new Product();  
product.setSku("123456");  
product.setName("심준보");  
redis.opsForValue().set(product.getSku(), product);  
  
// Get  
Product result = redis.opsForValue().get(product.getSku());

주어진 키로 엔트리가 검색이 되지 않을 경우에는 널(null)이 반환된다.

**리스트 사용하기**

// Set - 엔트리 종단에 추가  
Product product = new Product();  
product.setSku("123456");  
product.setName("심준보");  
redis.opsForList().rightPush("cart", product);  
  
// Set - 엔트리 시작점에 추가  
redis.opsForList().leftPush("cart", product);

// Get  
Product first = redis.opsForList().leftPop("cart");  
Product last = redis.opsForList().rightPop("cart");

leftPop, rightPop을 사용하여 양쪽 종단에서 팝한다.

🡺pop 메소드는 리스트에서 팝된 아이템을 제거 할 때 부작용이 있을 수 있다.

// 단순 Get  
List<Product> products = redis.opsForList().range("cart", 2, 12);

Range() 메소드는 리스트 엔트리에서 값을 삭제할 수 없지만, 키와 인덱스 범위 값을 검색한다.

**세트 내 동작 수행**

// Set  
Product product = new Product();  
product.setSku("123456");  
product.setName("심준보");  
redis.opsForSet().add("cart", product);  
  
// Get  
Set<Product> diff = redis.opsForSet().difference("cart1", "cart2");  
Set<Product> union = redis.opsForSet().union("cart1", "cart2");  
Set<Product> isect = redis.opsForSet().intersect("cart1", "cart2");  
  
// Delete  
redis.opsForSet().remove(product.getSku());

세트는 인덱스와 정렬을 하지 않으므로 세트에서 단일 아이템을 핀포인트(pinpoint)하거나 가져올 순 없다.

**키 바인딩하기**

BoundListOperations<String, Product> cart =  
 redis.boundListOps("cart");  
Product popped = cart.rightPop();  
cart.rightPush(new Product("1", "홍길동"));  
cart.rightPush(new Product("2", "임꺽정"));  
cart.rightPush(new Product("3", "전우치"));

**12.3.3. 키와 값의 직렬 변환 설정**

엔트리가 Redis 키-값 저장소에 저장될 때, 키와 값은 Redis 직렬 변환기를 사용하여 직렬화된다.

스프링 데이터 Redis는 다음을 포함하는 직렬 변환기를 지원한다.

-GenericToStringSerializer – 스프링 변환 서비스를 이용한 직렬 변환

-JacksonJsonRedisSerializer – Jackson1을 이용하여 객체를 Json으로 직렬 변환

-Jackson2JsonRedisSerializer – Jackson2을 이용하여 객체를 Json으로 직렬 변환

-JdkSerializationRedisSerializer – 자바 직렬 변환 사용

-OxmSerializer – XML 직렬 변환을 위해 스프링 O/X 매핑의 진행자/비진행자를 사용한 직렬 변환

-StringRedisSerializer – String 키와 값의 직렬 변환

RedisTemplate은 JdkSerializationRedisSerializer를 사용하며, 키와 값은 자바를 통해서 직렬화된다.

StringRedisTemplate은 기본적으로 StringRedisSerializer를 사용한다.

**CHAPTER13**

캐싱(caching)은 자주 필요한 정보를 저장하는 방법으로, 해당 정보가 필요할 때 사용될 수 있도록 한다. 스프링은 캐시 솔루션을 구현하지 않지만, 다양한 캐싱 구현체와 통합된 캐싱을 위한 선언적 지원을 제공한다.

**13.1 캐시 지원하기**

스프링의 캐시 추상화 지원은 다음의 두 가지 형태를 가진다.

-애너테이션 주도 캐싱

-XML 선언 캐싱

스프링 캐시 추상화를 사용하는 가장 일반적인 방법은 @Cacheable과 @cacheEvict 같은 애노테이션을 사용하여 메소드를 애너테이션하는 것이다.

**JavaConfig 방식**

@Configuration  
@EnableCaching  
public class CacheConfig {  
 @Bean  
 public CacheManager cacheManager() {  
 return new ConcurrentMapCacheManager();  
 }  
}

**XMLConfig 방식**

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:cache="http://www.springframework.org/schema/cache"  
 xsi:schemaLocation="  
 http://www.springframework.org/schema/data/jpa  
 http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd  
 http://www.springframework.org/schema/cache  
 http://www.springframework.org/schema/cache/spring-cache.xsd”  
  
 <context:component-scan base-package="com.sjb.api"/>  
  
 <!-- Cache -->  
 <cache:annotation-driven />  
 <bean id ="cacheManager" class="org.springframework.cache.concurrent.ConcurrentMapCacheManager" />  
</beans>

@EnableCaching과 <cache:annotation-driven>은 동일 하게, 스프링의 캐싱 애너테이션을 하는 포인트커트를 가지는 애스펙트를 생성한다.

ConcurrentMapCacheManager는 캐시 저장소로 java.util.concurrent.ConcurrentHashMap을 사용한다.

캐시 저장소는 메모리 기반이고, 애플리케이션 라이프사이클에 엮어 있으므로 큰 규모의 애플리케이션에 이상적인 선택은 아니다.

**13.1.1 캐시 매니저 설정하기**

스프링 3.1은 다섯 개의 캐시 매니저 구현체를 지원한다.

-SimpleCacheManager

-NoOpCacheManager

-ConcurrentMapCacheManager

-CompositeCacheManager

-EhCacheCacheManager

**Ehcache로 캐싱하기**

Ehcache는 가장 인기 있는 캐시 프로바이더 중 하나다. Ehcache 웹사이트는 ‘자바에서 가장 많이 쓰이는 캐시’라고 주장한다.

**ehcache.xml (EhCache Config)**

<ehcache>  
 <cache name="spittleCache"  
 maxBytesLocalHeap="50m"  
 timeToLiveSeconds="100">  
 </cache>  
</ehcache>

**EhCache JavaConfig**

@Configuration  
@EnableCaching  
public class CachingConfig {  
  
 @Bean  
 public EhCacheCacheManager cacheManager(CacheManager cm) { // EhCacheCacheManager 설정  
 return new EhCacheCacheManager(cm);  
 }  
  
 @Bean  
 public EhCacheManagerFactoryBean ehcache() { // EhCacheManagerFactoryBean  
 EhCacheManagerFactoryBean ehCacheFactoryBean =  
 new EhCacheManagerFactoryBean();  
 ehCacheFactoryBean.setConfigLocation(  
 new ClassPathResource("com/sjb/chapter13/cache/ehcache.xml")  
 );  
 return ehCacheFactoryBean;  
 }  
}

**13.2 캐싱을 위한 애너테이션 메소드**

스프링은 캐싱 규칙을 위한 네 개의 애너테이션을 제공한다.

|  |  |
| --- | --- |
| 애너테이션 | 내용 |
| @Cacheable | 스프링 메소드 사용 전에 메소드 반환 값을 위해 캐시를 살펴본다.  값이 존재할 경우, 캐시된 값이 반환된다.  값이 존재하지 않는다면, 메소드가 실행되고 반환된 값이 캐시에 남는다. |
| @CachPut | 스프링이 캐시에 메소드 반환 값을 저장한다. 캐시는 메소드 실행전에 체크되지 않으며, 메소드는 항상 실행된다. |
| @CacheEvict | 스프링이 캐시에서 한 개 이상의 엔트리를 내쫒는다. |
| @Caching | 다른 캐싱 애너테이션을 여러 번 즉시 적용하기 위해 사용할 수 있는 그룹 애너테이션이다. |

모든 애너테이션은 메소드와 클래스에서 모두 사용된다.

단일 메소드에서 사용될 때, 애너테이션에 의해서 미리 정해진 캐싱 동작은 오직 그 메소드에 대해서만 적용된다.

그러나 애너테이션이 클래스 레벨이면 캐싱 동작은 그 클래스의 모든 메소드에 적용된다.

**13.2.1 캐시 채우기**

@Cacheable은 먼저 캐시의 엔트리를 찾고, 매칭되는 엔트리가 발견되면 메소드 호출을 미리 수행한다.

매칭되는 엔트리가 발견되지 않으면, 메소드가 호출되고 반환 값은 캐시 안에 남는다.

@CachePut은 캐시 내에 매칭 값을 체크하지 않으며, 타깃 메소드가 사용될 수 있도록하고 반환 값을 캐시에 추가한다.

@Cacheable("spittleCache")  
Spitter findOne(long id);

인터페이스 메소드를 애너테이션 할 때 @cacheable 애너테이션은 SpittleRepository의 모든 구현체에서 상속되며, 동일한 캐싱 규칙이 적용된다.

**캐시에 값 넣기**

@CachePut 애너테이션된 메소드는 항상 호출되고 반환 값이 캐시에 남는다.

예를 들어 Spittle save() 메소드를 통해서 저장될 때, 곧 요청받을 확률이 매우 높다.

따라서 save()가 호출될 때 캐시로 Spittle을 넣고, findOne()을 호출해두면, 바로 적용할 수 있다.

@CachePut("spittleCache")  
Spitter save(Spitter spitter);

🡺여기서 문제점.. Spitter 객체를 구분할 수 있는 키가 없다.

**맞춤형 캐시 키 만들기**

스프링은 캐시 규칙을 정의하기 위해 특별히 여러 개의 SpEL 표현식을 제공한다.

@CachePut(value = "spittleCache", key = "#result.id")  
Spitter save(Spitter spitter);

이와 같은 방식으로 지정된 @CachePut을 사용하는 캐시는 save() 메소드에 적용되지 않는다.

그러나 반환되는 Spittle은 Spittle의 id 프로퍼티와 동일한 키를 가지고 캐시에 저장된다.

**조건 캐싱**

@Cacheable과 @CachePut은 조건 캐싱을 위한 두 가지 애트리뷰트를 제공한다.

Unless와 condition으로, 둘 다 SpEL 표현식이다. Unless 애트리뷰트의 SpEL 표현식 확인 결과가 true이면 캐시된 메소드로부터 반환된 데이터가 캐시에 저장되지 않는다.

이와 유사하게, condition 애트리뷰트의 SpEL 표현식이 false로 나오면 캐싱은 메소드에 대해 사용될 수 없다.

메시지 프로퍼티가 텍스트 ‘NoCache’를 포함하는 Spittle 객체를 캐싱하는 것을 원하지 않는다고 가정하자. Spittle이 캐싱되는 것을 막기 위해 unless 애트리뷰트를 다음과 같이 설정한다.

@Cacheable(value = "spittleCache", unless = "#result.message.contains('NoCache')")  
Spittle findOne(long id);

@Cacheable(value = "spittleCache", unless = "#result.message.contains('NoCache')", condition = "#id >= 10")  
Spittle findOne(long id);

findOne()이 파라미터로 10보다 작은 값을 가지고 호출되면 캐시는 검색되지 않으며, 캐시의 반환 Spittle도 검색되지 않는다.

**13.2.2 캐시 엔트리 삭제하기**

@CacheEvict 애노테이션된 메소드가 호출되면, 하나 이상의 엔트리들이 캐시에서 제거된다.

@CacheEvict("spittleCache")  
void remove(long id);

@Cacheable, @CachePut과 달리, @CacheEvict는 void 메소드에서 사용된다.

@Cacheable과 @CachePut은 non-void 반환 값이 필요하며, 캐시 내에 존재할 수 있는 아이템이다.

그러나 @CacheEvict가 캐시에서 아이템을 제거만 하므로, 심지어 void이더라도 다른 메소드에서 존재된다.

**13.3 XML에서 캐싱 선언하기**

왜 XML에서 캐싱을 선언해야 할까

-소스 코드에서 스프링에 특정한 애너테이션을 추가할 때 불편하다.

-소스 코드를 가지지 않은 빈에 캐싱을 적용하고자 할 때 적용 가능하다.

**XML 요소를 사용하여 SpittleRepository 캐싱 규칙을 선언한다.**

<?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:cache="http://www.springframework.org/schema/cache"  
 xmlns:aop="http://www.springframework.org/schema/aop"  
 xsi:schemaLocation="http://www.springframework.org/schema/aop  
 http://www.springframework.org/schema/aop/spring-aop.xsd  
 http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/cache  
 http://www.springframework.org/schema/cache/spring-cache.xsd">  
  
 <aop:config>  
 <aop:advisor advice-ref="cacheAdvice"  
 pointcut="execution(\* com.sjb.chapter13.repository.SpittleRepository.\*(..))"/>  
 </aop:config>  
 <cache:advice id="cacheAdvice">  
 <cache:caching>  
 <cache:cacheable cache="spittleCache" method="findRecent"/>  
 <cache:cacheable cache="spittleCache" method="findOne"/>  
 <cache:cacheable cache="spittleCache" method="findBySpitterId"/>  
 <cache:cache-put cache="spittleCache" method="save" key="#result.id"/>  
 <cache:cache-evict cache="spittleCache" method="remove"/>  
 </cache:caching>  
 </cache:advice>  
 <bean id="cacheManager" class="org.springframework.cache.concurrent.ConcurrentMapCacheManager"/>  
</beans>

**13.4 요약**

캐싱이 애스펙트 지향 액티비티라는 사실을 논의하였다. 실제로 스프링은 애스펙트로서 캐싱을 구현하였다. 따라서 XML에서 캐싱 규칙을 선언할 때 이와 같은 내용이 명백해진다.

사용자는 캐싱 어드바이스를 포인트커트에 바인드해야 한다.

**CHAPTER 14**

**14.1 애너테이션을 사용한 시큐어 메소드**

스프링 시큐리티는 세 가지 다른 종류의 시큐리티 애너테이션을 제공한다.

-스프링 시큐리티의 @Secured

-JSR-250의 @RolesAllowed

-표현식 주도의 애너테이션 (@PreAuthorize, @PostAuthorize, @PreFilter, @PostFilter를 사용)

**14.1.1 @Secure를 이용한 메소드 보안**

@Configuration  
@EnableGlobalMethodSecurity(securedEnabled = true)  
public class SecurityConfig extends GlobalMethodSecurityConfiguration {  
  
 @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth  
 .inMemoryAuthentication()  
 .withUser("user").password("password").roles("USER");  
 }  
}

웹 레이어 시큐리티 설정에서 이미 인증 설정되지 않았다면, GlobalMethodSecurityConfiguration의 configure() 메소드를 오버라이딩해서 사용한다.

securedEnabled가 true일 때 포인트커트가 생성되며, 스프링 시큐리티 애스펙트는 @Secured로 애너테이션되는 빈 메소드를 감싼다.

@Secured("ROLE\_SPITTER")  
public void addSpittle(Spittle spittle) {  
   
}

@Secured 애너테이션은 인자로 String 배열을 사용한다.

마일 @Secured 애너테이션에 인자로 하나 이상의 값이 전달된다면 인증된 사용자는 메소드를 호출하기 위해 인자로 전달된 권한 중 최소 한 가지 권한은 갖고 있어야 한다.

@Secured({"ROLE\_SPITTER". "ROLE\_ADMIN"})  
public void addSpittle(Spittle spittle) {  
   
}

ROLE\_SPITTER나 ROLE\_ADMIN 권한을 가지고 있어야 호출 가능하다.

만일 표준 애너테이션의 사용을 원하면 @RoleAllowed를 대신 사용하는 것도 고려한다.

**14.1.2 JSR-250의 @RolesAllowed 사용**

@RolesAllowed 애너테이션이 @Secured와 다른점은 JSR-250에 정의된 자바 표준 애너테이션 중 하나라는 점이다.

@Configuration  
@EnableGlobalMethodSecurity(jsr250Enabled = true)  
public class JSRConfig extends GlobalMethodSecurityConfiguration {  
  
 @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth  
 .inMemoryAuthentication()  
 .withUser("user").password("password").roles("USER");  
 }  
}

Jsr250Enabled 애트리뷰트만 활성화했지만, jsr250Enabled가 securedEnabled와 상호 배타적인 것은 아니다. 두 가지 애너테이션 방식을 동시에 활성화할수 있다.

Jsr250Enabled가 true로 설정되면, 포인트커트는 영향을 받고 @RolesAllowed로 애너테이션된 메소드는 스프링 시큐리티의 애스펙트로 감쌀 수 있다.(@Secured 방식과 동일)

@RolesAllowed("ROLE\_SPITTER")  
public void addSpittle(Spittle spittle) {

// ...  
}

@RolesAllowed와 @Secured 의 공통적인 단점은 사용자가 특별한 권한을 가지는가의 여부에 따라서 메소드 실행이 제한된다.

메소드가 실행될지 여부를 허용하기 위한 결정 역할을 수행할 다른 팩터(factor)는 없다.

**14.2 메소드 레벨 시큐리티를 위한 표현식 사용하기**

스프링 시큐리티 3.0은 메소드에 보다 흥미로운 보안 제약 사항을 제공하기 위해 SpEL을 사용하는 새로운 애너테이션을 도입했다.

**SpEL 표현식으로 메소드 보안을 적용하기 위해 스프링 시큐리티 3.0에서 제공하는 네 가지 새로운 애너테이션**

|  |  |
| --- | --- |
| **애너테이션** | **설 명** |
| @PreAuthorize | 표현식 평가 결과에 따라 호출 전 메소드 접근을 제한한다. |
| @PostAuthorize | 메소드 호출을 허용하지만, 표현식 평가 결과가 false인 경우 보안 예외를 발생시킨다. |
| @PostFilter | 메소드 호출을 허용하지만, 표현식에 대한 메소드의 결과를 필터링한다. |
| @PreFilter | 메소드 호출을 허용하지만, 메소드 진입 전에 입력 값을 필터링한다. |

**14.2.1 메소드 액세스 규칙 표현하기**

@PreAuthorize는 메소드 수행 전에 평가되며, 표현식이 true 값을 가지지 않는다면 메소드가 실행되지 않는다.

@PostAuthorize는 시큐리티 예외 상황이 발생하거나 하지 않는 경우를 결정하기 전에 메소드 반환될 때까지 대기한다.

**사전권한 부여 메소드 액세스**

@Secured나 @RolesAllowed와 동일해 보이지만, 실제로 @PreAuthorize를 이용하여 인증된 사용자에게 주어진 권한을 기준으로 접근을 제한한다.

@PreAuthorize("hasRole('ROLE\_SPITTER')")  
public void addSpittle(Spittle spittle) {  
 // ...  
}

ROLE\_SPITTER 권한을 갖는 경우 메소드 접근을 승인한다. 그렇지 않은 경우에 시큐리티 예외 상황이 발생하고 메소드는 실행되지 않는다.

@PreAuthorize("hasRole('ROLE\_SPITTER') and #Spittle.text.length() <= 140)" + "or hasRole('ROLE\_PREMIMU')")  
public void addSpittle(Spittle spittle) {  
 // ...  
}

표현식의 #spittle 부분은 같은 이름의 메소드 파라미터를 직접 참조한다는 의미다.

**사후권한 부여 메소드 액세스**

메소드에 권한을 부여하는 다소 명확하지 않은 방법이 사후권한을 부여하는 방법이다.

사후권한 부여는 보안이 적용된 메소드에서 반환된 객체를 기준으로 보안 적용 여부를 결정한다.

권한을 부여하는 시점만 제외하면 @PostAuthorize와 @PreAuthorize는 동일하게 동작한다.

@PostAuthorize("returnObject.spitter.username == principal.username")  
Spittle getSpittleById(long id);

보안이 적용된 메소드에서 반환된 객체에 쉽게 접근하기 위해 스프링 시큐리티는 SpEL에 returnObject 이름을 제공한다. 예제에서는 반환된 객체가 Spittle 이므로 표현식은 spitter의 username 프로퍼티를 이용한다.

**14.2.2 필터링 메소드의 입출력**

**사후필터링 메소드 반환 값**

@PostFilter는 값 파라미터로 SpEL 표현식을 가진다.

@PostFilter는 컬렉션의 각 멤버에 대한 표현식이 메소드로부터 반환되는지를 검증하고, 표현식에 대한 멤버 검증 결과가 false이면 이를 제거한다.

@PreAuthorize("hasAnyRole({'ROLE\_SPITTER', 'ROLE\_ADMIN'})")  
@PostFilter("hasRole('ROLE\_ADMIN') || " + "filterObject.spitter.username == pricipal.name")  
public List<Spittle> getOffensiveSpittles();

@PostFilter 애너테이션이 그 리스트를 필터링하면 사용자는 볼 수 있도록 허용된 spittle 객체만을 볼 수 있다.

특히 관리자는 모든 오펜시브 Spittle을 볼 수 있고, 비관리자는 단지 주어진 Spittle만을 볼 수 있다.

**사전필터링 메소드 파라미터**

사후필터링된 메소드의 반환 값과 더불어, 메소드로 전달된 값을 사전필터링하는 옵션을 가진다.

이는 덜 보편화된 기술이지만, 특정할 때 도움된다.

@PreAuthorize("hasAnyRole({'ROLE\_SPITTER', 'ROLE\_ADMIN'})")  
@PostFilter("hasRole('ROLE\_ADMIN') || " + "filterObject.spitter.username == pricipal.name")  
void deleteSpittles(List<Spittle> spittles);

표현식은 컬렉션 내 각 아이템에 대해 검증하며, 표현식 검증된 아이템들이 true를 가지면 리스트에 존재한다.

**CHAPTER 15**

**15.1 스프링 리모팅 개요**

리모팅(remoting)은 클라이언트 애플리케이션과 서비스 간의 대화다.

클라이언트는 마치 프록시가 해당 서비스 기능을 제공하고 있는 것처럼 프록시를 호출한다.

그러면 프록시가 클라이언트를 대신해 원격 서비스와 통신한다.

프록시는 연결에 관련된 세부 사항을 처리하고 원격 서비스를 원격 호출한다.

**15.2 RMI 활용**

스프링은 RMI 서비스를 로컬 자바빈즈인 것처럼 스프링 애플리케이션에 연결할 수 있게 해 주는 프록시 팩토리 빈(proxy factory bean)을 제공함으로써 RMI 모델을 단순화 한다.

**15.2.1 RMI 서비스 익스포트**

RMI 서비스를 만드는 과정은 다음과 같다.

1. java.rmi.RemoteException을 던지는 메소드가 포함된 서비스 구현 클래스를 작성한다.

2. java.rmi.Remote를 확장해 서비스 인터페이스를 만든다.

3. RMI 컴파일러(rmic)를 실행해 클라이언트 스텁(stub)과 서버 스켈레톤(skeleton) 클래스를 생성한다.

4. 서비스를 호스트하기 위해 RMI 레지스트리를 시작한다.

5. 서비스를 RMI 레지스트리에 등록한다.

**15.2.2 RMI 서비스 와이어링**

RMI는 원격 서비스와 통신하는 훌륭한 방법이기는 하지만 한계도 있다.

우선 RMI는 방화벽을 넘어 작업하는 데 어려움이 있다. 그 이유는 RMI가 통신을 위해 대체로 방화벽이 허용하지 않을 **임의의** 포트를 사용하기 때문이다.

또 한 가지 고려할 사항은 RMI가 자바 기반이라는 사실이다. 이는 클라이언트와 서비스 둘 다 자바로 작성돼야 한다는 것을 의미한다.

**15.3. Hessian과 Burlap을 이용한 리모트 서비스 노출**

Hessian과 Burlap은 카우초 테크놀로지가 제공하는 솔루션으로, HTTP를 통한 가벼운 원격 서비스를 가능하게 한다.

🡺API와 통신 프로토콜 둘 모두를 가능한 한 간단하게 유지함으로써 웹 서비스를 단순화하는 것이 목표다.

Hessian은 RMI처럼 클라이언트와 서비스 간에 바이너리 메시지를 이용해서 통신한다.

그러나 다른 바이너리 리모팅 기술(RMI)와 같은)과는 달리 이 바이너리 메시지는 PHP, Python, C++, C#등 자바외의 언어에 인식된다.

Burlap은 XML 기반 리모팅 기술로, XML을 파싱할 수 있는 언어라면 어떤 언어로든 자동적으로 이식된다.

그리고 Burlap은 XML이므로 Hessian의 바이너리 포맷에 비해 사람이 훨씬 쉽게 읽을 수 있다.

**Hessian 컨트롤러 구성하기**

RmiServiceExporter와 HessianServiceExporter의 한 가지 주된 차이점은 Hessian이 HTTP 기반이므로 HessianServiceExporter가 스프링 MVC 컨트롤러로 구현된다는 것이다.

그래서 익스포트된 Hessian 서비스를 사용하려면 두 가지 추가적인 설정 단계를 거쳐야 한다.

-web.xml에 스프링 DispatcherServlet을 설정하고 애플리케이션을 웹 애플리케이션으로 적용한다.

-Hessian 서비스 URL을 적절한 Hessian 서비스 빈으로 디스패치하도록 스프링 설정 파일에서 URL 핸들러를 설정한다.

**15.3.2. Hessian/Burlap 서비스에 액세스하기**

RMI 서비스와 Hessian 및 Burlap 서비스들 사이의 구성상 차이가 크지 않은 것에 대해서는

실제로 이점이다.

🡺이런 유사성 덕분에 스프링이 지원하는 여러 가지 리모팅 기술을 새로운 모델을 배울 필요 없이 힘들이지 않고 교체해서 사용할 수 있다.

**15.4. 스프링의 HttpInvoker 사용하기**

-RMI (자바의 직렬화 메커니즘) 자바에 최적화 되었지만 랜덤 포트를 사용하기 때문에 방화벽이 있으면 사용못한다.

-Hessian, Burlap 다른 언어에 인식될 수 있다. 데이터 모델이 복잡한 경우 Hessian/Burlap 직렬화 모델은 충분하지 못할 수도 있다.

🡺이런 장점들을 모아서 HttpInvoker가 탄생했다.

**15.4.1 빈을 HTTP 서비스로 익스포트하기**

HttpInvoker의 제약사항은 오로지 스프링 프레임워크에 의해서만 제공되는 리모팅 솔루션이라는 점이다. 이는 클라이언트와 서비스 양쪽이 모두 스프링을 사용할 수 있는 애플리케이션이어야 한다는 것을 의미한다.

그리고 자바 직렬화가 사용되므로 양쪽 모두 같은 버전의 클래스를 갖고 있어야 한다. (RMI와 유사)

**15.5 웹 서비스의 발행과 소비**

스프링은 일반적으로 알려진 XML 웹 서비스나 JAX-WS를 이용하여 스프링 자체적으로 SOAP 웹 서비스를 발행하고 소비하는 기능을 지원한다.

**15.6 요약**

스프링은 원격 서비스와 관련된 작업을 일반적인 자바빈을 이용한 작업처럼 쉽게 만들어 주는 리모팅 지원을 제공한다.

클라이언트 측에서 스프링은 스프링 애플리케이션에서 원격 서비스를 구성할 수 있게 해주는 프록시 팩토리 빈을 제공한다.

리모팅에 RMI, Hessian, Burlap 등 어떤 것을 사용하든 원격 서비스를 마치 POJO인 것처럼 애플리케이션에 연결한다.

**CHAPTER 16**

**16.1 휴식(REST)을 취한다.**

**16.1.1 REST의 기본 개념**

REST는 SOAP의 수많은 XML 네임스페이스를 이용하지 않고 평범한 HTTP URL을 통해 호출된다.

반대로 REST는 RPC와 거의 관련이 없다. RPC가 서비스 지향적이고 액션과 동사에 초점을 맞추는 데 반해, REST는 리소스 지향적이고 애플리케이션을 표현하는 객체와 명사를 강조한다.

-표현(Representational) – REST 리소스는 XML, JSON, 심지어 HTML을 표함하여 리소스 사용자에게 가장 적합한, 사실상 거의 모든 형식을 표현한다.

-상태(State) – REST와 작업할 경우 리소스에 대해 취할 수 있는 액션보다 리소스의 상태에 대해 더 많은 괌심을 둔다.

-전달(Transfer) – REST는 한 애플리케이션에서 다른 애플리케이션으로 어떤 표현 형식으로 리소스 데이터 전달을 포함한다.

**REST는 가장 적합한 형식이 무엇이든 간에(JSON, XML, HTML 등) 서버에서 클라이언트로(또는 반대로) 리소스의 상태를 전달하는 것이다.**

-생성 (POST)

-읽기 (GET)

-업데이트 (PUT or PATCH)

-삭제 (DELETE)

**16.1.2. 스프링이 REST를 지원하는 방법**

스프링 3부터 스프링 MVC가 REST를 최고 수준으로 지원할 수 있도록 몇 가지 개선이 이루어졌다.

스프링 4.0이 REST 리소스의 개발을 지원한는 방법은 다음과 같다

-컨트롤러는 REST의 네 가지 주요 메소드인 GET, PUT, DELETE, POST를 포함하여 모든 HTTP 메소드에 대한 요청을 처리한다.

-새로운 @PathVariable 애너테이션은 컨트롤러가 파라미터화된 URL(경로의 일부분에 변수 입력이 있는 URL)에 대한 요청을 처리할 수 있도록 한다.

-리소스는 XML, JSON, Atom 그리고 RSS 같은 데이터 모델 렌더링을 위한 새로운 뷰 구현을 포함하여 스프링의 뷰와 뷰 리졸버를 이용해 다양한 방식으로 표현한다.

-클라이언트에 대한 가장 적합한 표현은 새로운 ContentNegotiatingViewResolver를 이용해 선택한다.

-뷰 기반의 렌더링은 새로운 @ResponseBody 애너테이션과 다양한 HttpMethodConverter 구현체를 이용해 모두 무시한다.

-마찬가지로 새로운 @RequestBody 애너테이션은 HttpMethodConverter 구현체와 함께 인바운드 HTTP 데이터를 컨트롤러의 핸들러 메소드에 전달하는 자바 객체로 변환한다.

-RestTemplate은 클라이언트 측의 REST 리소스 사용을 간소화한다.

**16.2 첫 번째 REST 엔드포인트 만들기**

@Controller  
@RequestMapping("/spittles")  
public class SpittleController {  
 private static final String *MAX\_LONG\_AS\_STRING* = "9223372036854775807";  
 @Autowired  
 private SpittleRepository spittleRepository;  
  
 @RequestMapping(method = RequestMethod.*GET*)  
 public List<Spittle> spittles(  
 @RequestParam(value = "max", defaultValue = *MAX\_LONG\_AS\_STRING*) long max,  
 @RequestParam(value = "count", defaultValue = "20") int count) {  
 return spittleRepository.findSpittles(max, count);  
 }  
}

앞의 예제에서 사용한 이 컨트롤러는 전혀 RESTful 하지 않고, 리소스를 제공하는 컨트롤러도 아니다.

지금 우리는 REST API를 만드는 방버에 대해 이야기하고 있다. HTML은 데이터의 적절한 표현이 아니다.

리소스의 사용자가 JSON을 선호하면 리소스는 JSON 포맷으로 표현된다.

만약 사용자가 꺾쇠괄호를 좋아하면 동일한 리소스는 XML로 표현된다.

REST 엔드포인트를 실행하는 다른 애플리케이션이나 코드처럼 사람 외의 사용자가 사용할 리소스 표현 방식 중 가장 알맞은 것은 XML과 JSON이다.

스프링은 리소스의 자바 표현을 클라이언트에 전달될 표현으로 변환하는 두 가지 방법을 제공한다.

-콘텐츠 협상 – 모델이 클라이언트에 제공되는 표현으로 렌더링될 수 있도록 뷰는 선택된다.

-메시지 변환 – 메시지 변환기는 컨트롤러에서 반환된 객체를 클라이언트에 제공되는 표현으로 변경한다.

**16.2.1 리소스 표현 협상**

스프링의 ContentNegotiatingViewResolver는 클라이언트가 고려하려는 콘텐츠 타입을 선택하는 특별한 뷰 리졸버다.

ContentNegotiatingViewResolver의 동작 방식을 이해하려면 콘텐츠 협상의 두 단계를 알아야 한다.

1. 요청 미디어 타입 결정

2. 요청 미디어 타입에 대해 최적의 뷰 검색

**요청 미디어 타입 결정**

ContentNegotiatingViewResolver는 먼저 URL의 파일 확장자를 살펴본 후에 Accept 헤더를 살펴보고 요청하는 모든 미디어 타입을 사용한다.

만일 URL의 끝에 파일 확장자가 있으면 ContentNegotiatingViewResolver는 확장자에 기반한 원하는 타입을 만들어 낸다.

ex) 확장자 .json (application/json), .xml (application/xml), .html (text/html)

결론적으로 Accept 헤더가 없고 확장자가 별 도움이 안된다면 ContentNegotiatingViewResolver

는 기본 콘텐츠 타입으로 되돌아가고, 클라이언트는 서버가 전달하는 표현이 무엇이든지 사용한다.

**미디어 타입 선택 방식이 미치는 영향**

ContentNegotiationManager를 통해 할 수 있는 몇 가지 사항들은 다음과 같다.

-콘텐츠 타입을 요청으로부터 얻지 못할 경우의 기본 콘텐츠 타입을 지정한다.

-요청 파라미터를 통해서 콘텐츠 타입을 지정한다.

-요청의 Accept 헤더를 무시한다.

-특정 미디어 타입에 대한 요청 확장자를 매핑한다.

-확장자로부터 미디어 타입을 검색하기 위한 fallback 옵션으로 JAF(Java Activation Framework)를 사용한다.

-타입이 ContentNegotiationManager인 빈을 직접 선언한다.

-ContentNegotiationManagerFactoryBean을 통해서 간접적으로 빈을 생성한다.

-WebMvcConfigurerAdapter의 configureContentNegotiation() 메소드를 오버라이드 한다.

**16.2.2 HTTP 메시지 변환기 사용**

**응답 바디에서 리소스 상태 반환**

일반적으로 핸들러 메소드는 String 이외의 다른 자바 객체를 반환하면 이 객체는 결국 뷰에 렌더링할 모델이 된다.

하지만 메시지 변환을 적용하면 스프링을 통해 일반 모델/뷰를 스킵할 수 있고, 대신 메시지 변환기를 사용한다.

가장 간단한 방법은 @ResponseBody를 사용하여 컨트롤러 메소드를 애너테이션 하는 것이다.

@RequestMapping(method = RequestMethod.*GET*, produces = "application/json")  
public @ResponseBody List<Spittle> spittles2(  
 @RequestParam(value = "max", defaultValue = *MAX\_LONG\_AS\_STRING*) long max,  
 @RequestParam(value = "count", defaultValue = "20") int count) {  
 return spittleRepository.findSpittles(max, count);  
}

**Jackson은 기본적으로 리플렉션을 사용한다.**

기본적으로 Jackson JSON 라이브러리는 반환 객체로부터 JSON 리소스 표현을 만들 때 리플렉션을 사용한다.

그러나 만약 자바 타입을 추가, 제거, 재명명하여 재구성할 때, 만들어진 JSON도 변경된다.

그러나 JSON은 자바 타입에 대해 Jackson의 매핑 애너테이션을 적용하여 만들어지는 방법에 영향을 미칠 수 있다. 이를 통해 결과 JSON이 어떠한 모습을 가질지 제어할 수 있으며 API와 클라이언트에 대해서는 적용을 멈추고 변경하지 않는다.

**요청 바디에서 리소스 상태받기**

@ResponseBody 를 이용하면 자바 객체를 HTTP 응답 몸체로 전송할 수 있다.

@RequestBody 를 이용하면 HTTP 요청 몸체를 자바 객체로 전달받을 수 있다.

클라이언트가 저장될 새 Spittle을 제공할 방법이 필요하다고 가정하자. 다음과 같은 요청 처리를 위한 컨트롤러 메소드를 작성한다.

@RequestMapping(method = RequestMethod.*POST*, consumes = "application/json")  
public  
@ResponseBody  
Spittle saveSpittle(@RequestBody Spittle spittle) {  
 return spittleRepository.save(spittle);  
}

-produces(@ResponseBody) : accept의 접근 허용을 설정한다.

-consumes(@RequestBody) : content-type의 접근 허용을 설정한다.

**메시지 변환을 위한 기본 컨트롤러**

@ResponseBody와 @RequestBody 애너테이션은 요청을 처리할 때 스프링 메시지 변환기를 사용하기 위한 간결하지만 강력한 방법을 제공한다.

그러나 여러 메소드를 지원한다면 애너테이션들은 반복적으로 사용된다.

스프링4.0은 이를 지원하기 위한 @RestController 애너테이션을 도입한다.

@Controller 대신에 @RestController를 사용하여 컨트롤러 클래스를 애너테이션한다면, 스프링은 메시지 변환을 컨트롤러의 모든 핸들러 메소드에 적용한다.

**16.3. 더 많은 리소스 사용하기**

**16.3.1 클라이언트와 에러 처리하기**

@RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
public @ResponseBody Spittle spittleById(@PathVariable long id) {  
 return spittleRepository.findOne(id);  
}

주어진 ID에 매칭되는 ID를 가진 Spittle이 없고 findOne() 수행 결과가 널이라면 어떤 일이 발생하겠는가?

이는 클라이언트에게 요청한 것을 찾을 수 없다는 것을 말하기 위해 404(Not Found)가 되어야 한다.

-스프링은 해당 시나리오를 수행하기 위해 몇 개의 옵션을 제공한다.

-상태 코드는 @ResponseStatus 애너테이션으로 지정된다.

-컨트롤러 메소드는 응답에 관한 많은 메타데이터를 가지는ResponseEntitiy를 반환한다.

**ResponseEntity 사용하기**

ResponseBody 대안으로 ResponseEntity를 반환한다.

ResponseEntity는 리소스 표현으로 변환되는 객체와 더불어 응답에 대한 메타데이터(헤더와 상태 코드) 를 가지는 객체다.

@RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {  
 Spittle spittle = spittleRepository.findOne(id);  
 HttpStatus status = spittle != null ?  
 HttpStatus.*OK* : HttpStatus.*NOT\_FOUND*;  
 return new ResponseEntity<Spittle>(spittle, status);  
}

spittleById()는 @ResponseBody로 애너테이션되지 않음을 주목한다.

응답 헤더, 상태 코드, 페이로드를 전달하는 것과 더불어, ResponseEntity는 ResponseBody의 의미를 암시하므로 페이로드 메소드가 @ResponseBody로 애너테이션된다면 응답 바디로 렌더링된다.

@RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
public ResponseEntity<?> spittleById(@PathVariable long id) {  
 Spittle spittle = spittleRepository.findOne(id);  
 if (spittle == null) {  
 com.sjb.chapter16.model.Error error = new com.sjb.chapter16.model.Error(4, "not found");  
 return new ResponseEntity<com.sjb.chapter16.model.Error> (error, HttpStatus.*NOT\_FOUND*);  
 }  
 return new ResponseEntity<Spittle>(spittle, HttpStatus.*OK*);  
}

이제 Spittle이 발견되면, 반환되고 상태 코드 200(OK)를 가지는 ResponseEntitiy로 감싸진다.

다른 측면에서 볼 때 null을 반환하면, Error 객체를 처리하고 상태 코드 404(Not Found)를 가지는 ResponseEntitiy로 감싸진다.

**에러 처리**

@ExceptionHandler(SpittleNotFoundException.class)  
public ResponseEntity<Error> spittleNotFound(SpittleNotFoundException e) {  
 long spittledId = e.getSpittleId();  
 Error error = new Error(4, "Spittle [" + spittledId + "] not found");  
 return new ResponseEntity<Error>(error, HttpStatus.*NOT\_FOUND*);  
}

@ExceptionHandler 애너테이션은 특정 예외 상황을 다루기 위해 컨트롤러 메소드에 적용된다.

SpittleNotFoundException은 완전히 기본적인 예외 클래스다.

public class SpittleNotFoundException extends RuntimeException {  
 private long spittleId;  
  
 public SpittleNotFoundException(long spittleId) {  
 this.spittleId = spittleId;  
 }  
  
 public long getSpittleId() {  
 return spittleId;  
 }  
}

spittleById() 메소드에서 에러 처리 부분을 제거했다.

@RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
public ResponseEntity<?> spittleById(@PathVariable long id) {  
 Spittle spittle = spittleRepository.findOne(id);  
 if (spittle == null) {  
 throw new SpittleNotFoundException(id);  
 }  
 return new ResponseEntity<Spittle>(spittle, HttpStatus.*OK*);  
}

HTTP 상태가 항상 200OK이고, spittleById()가 Spittle을 반환할 것이라고 알고 있으므로

더 이상, ResponseEntity를 사용할 필요가 없다.

@RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
public Spittle spittleById(@PathVariable long id) {  
 Spittle spittle = spittleRepository.findOne(id);  
 if (spittle == null) {  
 throw new SpittleNotFoundException(id);  
 }  
 return spittle;  
}

@ExceptionHandler(SpittleNotFoundException.class)  
@ResponseStatus(HttpStatus.*NOT\_FOUND*)  
public Error spittleNotFound(SpittleNotFoundException e) {  
 long spittledId = e.getSpittleId();  
 return new Error(4, "Spittle [" + spittledId + "] not found");  
}

응답 상태 코드를 설정하기 위해서는 ResponseEntity를 사용한다. 그리고 ResponseEntitiy의 필요성을 제거하고 코드를 강화하기 위해 예외 핸드러(ExceptionHandler)와 ResponseStatus를 사용한다.

**16.3.2 응답 시의 헤더 설정하기**

**ResponseEntity 반환 시의 응답 내 헤더 설정**

@RequestMapping(value = "/test2", method = RequestMethod.*POST*, consumes = "application/json")  
public ResponseEntity<Spittle> saveSpittl2(@RequestBody Spittle spittle) {  
 Spittle result = spittleRepository.save(spittle);  
  
 HttpHeaders headers = new HttpHeaders();  
 URI locationUri = URI.*create*(  
 "http://localhost:8080/spitter/spittles/" + result.getId());  
 headers.setLocation(locationUri);  
  
 ResponseEntity<Spittle> responseEntity =  
 new ResponseEntity<Spittle>(  
 result, headers, HttpStatus.*CREATED*);  
 return responseEntity;  
}

HttpHeaders는 일반 HTTP 헤더를 설저하기 위해 몇 가지 편리한 세터(setter) 메소드를 사용하여 특별한 MultiValueMap<String, String>을 만든다.

URI를 직접 구성하기보다는 스프링은 UriComponentsBuilder 라는 형태의 몇 가지 지원을 제공한다.

**위치 URI를 생성하기 위해 UriComponentsBuilder 사용하기**

// ResponseEntity 사용하여 응답 내 헤더 설정  
@RequestMapping(value = "/test2", method = RequestMethod.*POST*, consumes = "application/json")  
public ResponseEntity<Spittle> saveSpittl2(@RequestBody Spittle spittle, UriComponentsBuilder ucb) {  
 Spittle result = spittleRepository.save(spittle);  
  
 HttpHeaders headers = new HttpHeaders();  
 URI locationUri =  
 ucb.path("/spittles/")  
 .path(String.*valueOf*(spittle.getId()))  
 .build()  
 .toUri();  
 headers.setLocation(locationUri);  
  
 ResponseEntity<Spittle> responseEntity =  
 new ResponseEntity<Spittle>(  
 result, headers, HttpStatus.*CREATED*);  
 return responseEntity;  
}

**16.4 REST 리소스 사용하기**

**16.4.1 RestTemplate 동작 살펴보기**

RestTemplate은 11개의 고유 동작을 정의하고, 각각은 총 36개의 메소드에 오버로드 된다.

각각의 작업은 세 가지 메소드 형태로 오버로드 된다.

-파라미터화된 URL에 대한 지원 없이 URL 명세로 java.net.URI를 취하는 형태

-가변적인 인자 목록으로 지정된 URL 파라미터로 String URL 스펙을 취하는 형태

-Map으로 지정된 URL 파라미터로 String URL 스펙을 취하는 형태

**16.4.2 리소스 GET하기**

GET 요청을 수행하는 두 종류의 메소드인 getForObject()와 getForEntity()를 볼 수 있다.

**16.4.3 리소스 조회**

public Profile fetchFacebookProfile(String id) {  
 RestTemplate rest = new RestTemplate();  
 return rest.getForObject("http://graph.facebook.com/{soitter}", Profile.class, id);  
}

RestTemplate을 사용하면 몇 줄의 코드로 감소한다.

// 가변 방식  
public Profile fetchFacebookProfile(String id) {  
 RestTemplate rest = new RestTemplate();  
 return rest.getForObject("http://graph.facebook.com/{soitter}", Profile.class, id);  
}  
  
// Map 방식  
public Profile fetchFacebookProfile2(String id) {  
 Map<String, String> urlVariables = new HashMap<>();  
 urlVariables.put("id", id);  
 RestTemplate rest = new RestTemplate();  
 return rest.getForObject("http://graph.facebook.com/{spitter}", Profile.class, urlVariables);  
}

가변인자(첫번째)나 Map(두번째) 을 getForObject()의 마지막 파라미터로 전달할 수 있다.

getForObject() 메소드는 응답 보디를 객체로 변환한다. 이 작업은 스프링 MVC가 @ResponseBody 애너테이션이 적용된 핸들러 메소드를 위해 사용한 변한기에 의존한다.

만일 getForObject()에 문제가 발생하면 비검사형 RestClientException이 던져진다.

**16.4.4 응답 메타데이터 추출**

getForEntity() 메소드는 getForObject() 메소드와 거의 동일하게 동작한다.

하지만 getForEntity()는 HTTP 메시지 변환기에 의해 자바 객체로 변환된 리소스를 반환하는데 반해, getForEntity()는 ResponseEntity 내에 전달되는 동일한 객체를 반환한다.

ResponseEntity는 HTTP 상태 코드와 응답 헤더와 같은 응답에 대한 추가적인 정보도 전달한다.

**16.4.5. 리소스 PUT 하기**

Put의 URI 기반 버전을 이용하여 서버에 있는 Spittle 리소스를 업데이트하는 방법은 다음과 같다.

public void updateSpittle(Spittle spittle) throws SpittleNotFoundException {  
 RestTemplate rest = new RestTemplate();  
 String url = "http://localhost:8080/spitter-api/spittles/" + spittle.getId();  
 rest.put(URI.*create*(url), spittle);  
}

getForObject()와 getForEntity()에서 본 것처럼 String 기반의 put() 메소드 중 하나를 사용하면 예외 처리를 포함해 URI 생성과 관련된 불편한 작업을 대부분 줄일 수 있다.

좀더 간단하게 사용할 수 있다.

public void updateSpittle2(Spittle spittle) throws SpittleNotFoundException {  
 RestTemplate rest = new RestTemplate();  
 rest.put("http://localhost:8080/spitter-api/spittles/{id}", spittle, spittle.getId());  
}

여기서는 Spittle 객체를 전달했으므로 임의의 객체와 작업할 수 있는 메시지 변환기가 필요하다.

만일 jackson2 라이브러리가 클래스패스에 있다면 MappingJacksonHttpMessageConverter는

application/json으로 요청에 대한 Spittle을 작성한다.

**16.4.6 리소스 DELETE하기**

delete() 메소드가 RestTemplate 메소드 중 가장 간단하다. 삭제할 리소스의 URI만 제공하면 된다.

public void deleteSpittle(long id) {  
 RestTemplate rest = new RestTemplate();  
 rest.delete("http://localhost:8080/spitter-api/spittles/{id}", id);  
}

**16.4.7 리소스 데이터 POST 하기**

postForObject()와 postForEntity() 메소드는 GET 요청 전송을 위해 getForObject()와 getForEntitY()

메소드가 작업하는 방식과 유사한 방식으로 POST 요청을 처리한다.

**16.4.8 POST 요청으로부터 객체 응답 수신하기**

public Spitter postSpitterForObject(Spitter spitter) {  
 RestTemplate rest = new RestTemplate();  
 return rest.postForObject("http://localhost:8080/spitter-api/spitters", spitter, Spitter.class);  
}

postSpitterForObject() 메소드에는 새롭게 생성된 Spitter 객체가 주어지고 postForObject() 메소드를 이용하여 서버에 객체를 보낸다. 그리고 응답으로 Spitter 객체를 받고 호출자에게 반환한다.

public Spitter postSpitterForObject2(Spitter spitter) {  
 RestTemplate rest = new RestTemplate();  
 ResponseEntity<Spitter> response = rest.postForEntity("http://localhost:8080/spitter-api/spitters", spitter, Spitter.class);  
 Spitter result = response.getBody();  
 URI url = response.getHeaders().getLocation();  
 return result;  
}

getForEntity() 메소드와 마찬가지로 postForEntity()는 ResponseEntity<T> 객체를 반환한다.

이 객체에서 getBody()를 호출하여 리소스 객체를 얻을 수 있다.

**16.4.9 POST 요청 후 리소스 위치 수신하기**

만일 Location 헤더의 값이 실제 알고 싶은 전부라면, RestTemplate의 PostForLocation 메소드를 사용하면 더 쉽게 알 수 있다.

public String postSpitter(Spitter spitter) {  
 RestTemplate rest = new RestTemplate();  
 return rest.postForLocation(  
 "http://localhost:8080/spitter-api/spitters", spitter).toString();  
}

만일 리소스 생성 후에 서버가 응답의 Location 헤더에 있는 새로운 리소스 URL로 응답한다면,

postForLocation()은 String으로 URL을 반환한다.

**16.4.10 리소스 교환**

서버에 전송하는 요청에 헤더를 설정하고 싶다면?

public Spitter exchange(String spitterId) {  
 RestTemplate rest = new RestTemplate();  
 ResponseEntity<Spitter> response = rest.exchange(  
 "http://localhost:8080/spitter-api/spitters/{spitter}", HttpMethod.*GET*, null, Spitter.class, spitterId);  
 Spitter spitter = response.getBody();  
 return spitter;  
}

HTTP 동사로 HttpMethod.GET을 전달하여 exchange()가 GET 요청을 보내도록 요구한다.

🡺getForEntity() 메소드와 사실상 동일해진다.

하지만 exchange()는 전송할 요청에 헤더를 설정할 수 있게 해준다.

요청 헤더 설정은 원하는 헤더가 로드된 MultiValueMap을 이용하여 exchange()에 전송하려는

HttpEntity를 생성하는 간단한 문제다.

public Spitter exchange2(String spitterId) {  
 MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();  
 headers.add("Accept", "application/json");  
 HttpEntity<Object> requestEntity = new HttpEntity<Object>(headers);  
  
 RestTemplate rest = new RestTemplate();  
 ResponseEntity<Spitter> response = rest.exchange(  
 "http://localhost:8080/spitter-api/spitters/{spitter}", HttpMethod.*GET*, requestEntity, Spitter.class, spitterId);  
 Spitter spitter = response.getBody();  
 return spitter;  
}

**CHAPTER 17**

스프링은 비동기 메시징을 위한 몇 가지 옵션이 있다.

JMS(Java Message Service)와 AMQP(Advanced Message Queuing Protocol)를 사용하여 스프링에서의 메시지를 전송/수신하는 방법을 살펴본다.

**17.1. 비동기 메시징에 대한 간단한 소개**

**17.1.1 메시지 보내기**

비동기 메시징의 핵심은 간접성이다.

어떤 애플리케이션이 정보를 다른 곳으로 전달한다면 두 애플리케이션 사이의 직접적인 연결은 없다.

대신 전송 측 애플리케이션은 수신 측 애플리케이션으로 메시지를 반드시 전달해 주는 대행 서비스에 메시지를 건내줄 뿐이다.

메시징 시스템에는 큐(queue)와 토픽(topic) 이라는 두 종류의 목적지가 있다.

큐에는 지점 대 지점(point-to-point) 모델이 사용되고,

토픽에는 발행-구독(publish-subscribe) 모델이 사용된다.

**지점 대 지점 메시징 모델 (은행 창구 직원)**

지점 대 지점 모델에서 각 메시지는 정확히 하나의 전송자와 수신자를 갖는다.

메시지가 큐에서 전달되면 큐의 메시지가 삭제되므로 메시지는 반드시 하나의 수신자에게만 전달된다.

**발행-구독 메시징 모델 (publish-subscribe)**

큐와 마찬가지로 여러 구독자는 토픽에 도착할 메시지를 기다린다. 하지만 메시지가 정확히 하나의 수진자에게만 전달되는 큐와 달리, 토픽의 모든 구독자가 같은 메시지의 사본을 수신한다.

발행자(publisher)는 구독자(subscriber)가 누군지를 전혀 모른다.

**17.1.2 비동기 메시징의 장점**

동기식 통신의 제약

-동기식 통신은 대기(wating)가 필요하다.(클라이언트는 원격 서비스에 있는 메소드를 호출하고 나면 원격 메소드가 종료할 때까지 대기해야 한다.)

-클라이언트와 서비스의 인터페이스와 결합된다.(서비스의 인터페이스가 변경되면 이 서비스의 모든 클라이언트 또한 적절하게 변경되어야 한다.)

-클라이언트와 서비스의 위치와 결합된다. (클라이언트는 서비스를 연결하려면 반드시 서비스의 네트워크상의 위치를 알고 있어야 한다.)

-클라이언트는 서비스의 이용 가능성(availability)에 영향을 받는다. (서비스가 갑자기 이용부락 상태가 되면 클라이언트도 결국 무력화 된다.)

**대기 없음**

비동기적으로 메시지를 전송할 때, 클라이언트는 메시지가 처리될 때까지 또는 심지어 전달될 때까지도 기다릴 필요가 없다.

클라이언트는 대기할 필요가 없으므로 자유롭게 다른 작업을 수행할 수 있고, 이러한 자유 시간을 활용하면 클라이언트의 성능이 극적으로 향상된다.

**메시지 기반과 결합도 제거**

메소드 호출을 기반으로 하는 RPC 통신과는 달리, 비동기적으로 전송하는 메시지는 데이터 중심(data-centric)이다.

클라이언트에게서 받은 데이터를 처리할 능력만 있다면 어떤 큐 수신자나 토픽 구독자라도 해당 메시지를 처리할 수 있고, 클라이언트는 그것이 구체적으로 무엇인지 또 무엇을 하는지 알 필요가 없다.

**위치 독립성**

동기식 RPC 서비스는 보통 네트워크상의 위치를 알아야 접근 가능하다. 따라서 이 서비스의 위치를 알고 이용해야 하는 클라이언트는 네트워크 구성 변경에 취약하다.

🡺 서비스의 IP 주소나 포트가 바뀌면 클라이언트도 이를 따라서 변경해야만 서비스를 계속 이용할 수 있다.

메시징 클라이언트는 이 서비스가 어디에 위치하는지 알지 못한다.

클라이언트는 단지 메시지를 전송할 큐나 토픽만 알뿐이다.

토픽은 메시지의 사본을 수신하게 되고, 각 서비스의 해당 메시지를 서로 다르게 처리할 수도 있다.

토픽에서 수신한 데이터는 같아도 각 서비스는 독립적으로 서로 다른 처리를 수행한다.

**전달 보증**

메시지를 비동기적으로 전송하는 경우에는 메시지가 언젠가는 전달될 것이므로 클라이언트는 안심해도 된다.

**17.2 JMS(Java Message Service)로 메시지 보내기**

JMS는 메시지 브로커를 사용하여 작업하기 위한 일반 API를 정의하는 자바 표준이다.

JMS를 사용함으로써 JDBC가 일반적인 인터페이스로 데이터베이스 동작을 지원하는 것처럼

모든 호환 구현체는 일반 공통 인터페이스를 통해 동작이 가능해졌다.

스프링은 JmsTemplate이라는 템플릿 기반 추상화를 통해서 JMS를 지원한다.

**17.2.1 스프링에서 메시지 브로커 셋업하기**

ActiveMQ에는 JMS 커넥션 팩토리로 ActiveMQConnectionFactory가 제공되는데, 다음과 같이 설정하면 된다.

<bean id="connectionFactory" class="org.apache.activemq.spring.ActiveMQConnectionFactory"/>

<amq:connectinoFacotry> 요소를 이용하여 커넥션 팩토리를 선언할 수 있다.

<amq:connectionFactory id="connectionFactory" brokerURL="tcp://localhost:61616"/>

<amq:connectionFactory> 요소는 분명히 ActiveMQ에 특화되어 있다.

**ActiveMQ 메시지 목적지 선언**

커넥션 팩토리와 마찬가지로 ActiveMQ 네임스페이스는 큐와 토픽응ㄹ 선언하는데 대안적 방법을 제공한다. 큐의 경우 <amq:queue> 요소도 사용한다.

또는 JMS 토픽이 적합한 경우는 <amq:topic>을 사용한다.

<amq:queue id="spittleQueue" physicalName="spittle.alert.queue" />  
<amq:queue id="spittleTopic" physicalName="spittle.alert.topic" />

**17.2.2 스프링의 JMS 템플릿 사용**

**지저분한 JMS 코드 공략**

전통적인(비스프링) JMS 방식을 이용한 메시지 전송

ConnectionFactory cf =  
 new ActiveMQConnectionFactory("tcp://localhost:61616");  
Connection conn = null;  
Session session = null;  
try {  
 conn = cf.createConnection();  
 session = conn.createSession(false, Session.*AUTO\_ACKNOWLEDGE*);  
 Destination destination = new ActiveMQQueue("spitter.queue");  
 MessageProducer producer = session.createProducer(destination);  
 TextMessage message = session.createTextMessage();  
  
 message.setText("Hello world!");  
 producer.send(message);  
} catch (JMSException e) {  
  
} finally {  
 try {  
 if (session != null) {  
 session.close();  
 }  
 if (conn != null) {  
 conn.close();  
 }  
 } catch (JMSException ex) {  
  
 }  
}

**전통적인(비스프링) JMS 방식을 이용한 메시지 수신**

ConnectionFactory cf =  
 new ActiveMQConnectionFactory("tcp://localhost:61616");  
Connection conn = null;  
Session session = null;  
try {  
 conn = cf.createConnection();  
 session = conn.createSession(false, Session.*AUTO\_ACKNOWLEDGE*);  
 Destination destination = new ActiveMQQueue("spitter.queue");  
 MessageConsumer consumer = session.createConsumer(destination);  
 Message message = consumer.receive();  
 TextMessage textMessage = (TextMessage) message;  
 System.*out*.println("GOT A MESSAGE: " + textMessage.getText());  
 conn.start();  
} catch (JMSException e) {  
  
} finally {  
 try {  
 if (session != null) {  
 session.close();  
 }  
 if (conn != null) {  
 conn.close();  
 }  
 } catch (JMSException ex) {  
  
 }  
}

**JMS 템플릿 활용**

JmsTemplate은 커넥션을 생성하고 세션을 획득하고, 메시지의 실제 전송과 수신 작업을 도맡는다.

더욱이 JmsTemplate은 불편하기 짝이 없는 JMSException을 알아서 처리한다.(비검사형)

JmsTemplate을 사용하려면 우선 스프링 설정 파일에 빈으로 선언해야 한다.

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate"  
 c:\_0-ref="connectionFactory"/>

**메시지 전송**

메시지 큐나 토픽에 메시지를 전송하는 시간은 특히 다른 사용자에게 알림을 보내는 데 걸리는 시간과 비교했을 때 무시할 수 있을 정도로 미미하다.

**JmsTemplate을 이용한 Spittle 전송**

@Autowired  
public AlertServiceImpl(JmsOperations jmsOperations) {  
 this.jmsOperations = jmsOperations;  
}  
  
@Override  
public void sendSpittleAlert(Spittle spittle) {  
 jmsOperations.send(  
 "spittle.alert.queue",  
 new MessageCreator() {  
 @Override  
 public Message createMessage(Session session) throws JMSException {  
 return session.createObjectMessage(spittle);  
 }  
 }  
 );  
}

MessageCreator의 createMessage() 메소드에서 단순히 세션으로부터 객체 메시지를 요청하는데,

이때 세션으로부터 객체 메시지를 만들기 위해 Spittle 객체를 제공한다.

🡺이게 전부다! sendSpittleAlert() 메소드는 메시지를 조립하고 전송하는 데만 주력하고 있다.

**기본 목적지 설정**

<amq:queue id="spittleTopic" physicalName="spittle.alert.topic"/>  
  
  
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate"  
 c:\_0-ref="connectionFactory"  
 p:defaultDestination-ref="spittleTopic"  
/>

**XML 설정을 통해 send 메소드에 대한 호출은 좀 더 간단해 질 수 있다.**

jmsOperations.send(  
 new MessageCreator() {  
 @Override  
 public Message createMessage(Session session) throws JMSException {  
 return session.createObjectMessage(spittle);  
 }  
 }  
);

**전송 시 메시지 변환**

send()와 달리 convertAndSend() 메소드는 인자로 messagecreator를 사용하지 않는다.

사용자를 위한 메시지를 생성하기 위해 빌트인 메시지 변환기를 사용하기 때문이다.

public void sendSpittleAlert2(Spittle spittle) {  
 jmsOperations.convertAndSend(spittle);  
}

🡺기본적으로 SimpleMessageConverter를 사용한다.

객체를 메시지로 변환하기 위해서는 MessageConverter 구현체를 사용한다.

public interface MessageConverter {  
 Object fromMessage(Message<?> var1, Class<?> var2);  
  
 Message<?> toMessage(Object var1, MessageHeaders var2);  
}

**JmsTemplate으로 와이어링한다.**

<!-- messageConverter 를 jackson으로 사용할 경우-->  
<bean id="messageConverter" class="org.springframework.jms.support.converter.MarshallingMessageConverter"/>  
  
<!-- jms connectionFactory-->  
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate"  
 c:\_0-ref="connectionFactory"  
 p:defaultDestination-ref="spittleTopic"  
 p:messageConverter-ref="messageConverter"  
/>

**메시지 소비**

JmsTemplate을 메시지 수신에도 사용할 수 있을까?

**JmsTemplate을 이용한 메시지 수신**

public Spittle receiveSpittleAlert() {  
 try {  
 ObjectMessage receivedMessage =  
 (ObjectMessage) jmsOperations.receive();  
 return (Spittle) receivedMessage.getObject();  
 } catch (JMSException jmsExcetion) {  
 throw JmsUtils.*convertJmsAccessException*(jmsExcetion);  
 }  
}

🡺훨씬 더 간단하다.

JmsTemplate의 receive() 메소드가 호출되면 메시지 브로커로부터 메시지 조회를 시도한다.

도착한 메시지가 없으면 receive() 메소드는 메시지가 올 때까지 대기한다.

JmsTemplate의 receiveAndconvert()를 사용하여 수신단에서 사용된다.

public Spittle receiveSpittleAlert2() {  
 return (Spittle) jmsOperations.receiveAndConvert();  
}

🡺깔끔하다.

**17.2.3 메시지 드리픈 POJO 작성**

이 절에는 스프링에서 메시지 드리븐 POJO를 통해 어떻게 비동기식 메시지 소비를 지원하는지 알아보겠다.

EJB2 명세에서 두드러진 특징 중 하나는 메시지 드리븐 빈(MDB, Message-Driven Bean)이 포함되었다는 사실이다. MDB는 메시지를 비동기식으로 처리하는 EJB다. 바꿔 말하면, MDB는 JMS 목적지에 메시지가 도착했다는 이벤트에 반응해서 해당 메시지를 처리한다.

이는 메시지가 도착할 때까지 블로킹되는 동기식 메시지 수신자와는 반대다.

**메시지 리스너 생성**

@MessageDriven(mappedName = "jms/spittle.alert.queue")  
public class SpittleAlertHandler implements MessageListener {  
 public void handleSpittleAlert(Spittle spittle) {  
 System.*out*.println(spittle.getMessage());  
 }  
  
 @Override  
 public void onMessage(Message message) {  
 // ...  
 }  
}

현재 SpittleAlertHandler의 EJB3구현체는 EJB의 메시지 드리븐 API들과의 결합도가 너무 높고, 원하는 만큼 POJO 스럽지도 않다.

스프링은 JMS 큐나 토픽에서 메시지를 처리하기 위해 POJO에 메소드를 위한 기능을 제공한다.

아래와 같은 SpittleAlertHandler의 POJO 구현이면 충분하다.

public class SpittleAlertHandler {  
 public void handleSpittleAlert(Spittle spittle) {  
 System.*out*.println(spittle.getMessage());  
 }  
}

**메시지 리스너 설정**

메시지 수신 기능을 POJO에 부여하는 방법은 스프링에 메시지 리스너로 설정하는 것이다.

<bean id="spittleHandler" class="com.sjb.chapter17.alerts.SpittleAlertHandler"/>

메시지 드리븐 POJO로 변환하기 위해서 빈을 메시지 리스너가 되도록 선언한다.

<jms:listener-container connection-factory="connectionFactory">  
 <jms:listener destination="spittle.alert.queue"  
 ref="spittleHandler"  
 method="handleSpittleAlert"/>  
</jms:listener-container>

메시지 리스너 컨테이너 안에 포함된 메시지 리스너가 생겼다.

메시지 리스너 컨테이너는 JMS 목적지를 감시하는 특별한 빈으로, 메시지 도착을 기다린다.

**17.2.4 메시지 기반의 RPC 활용**

**JMS 기반의 서비스 익스포팅**