**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";  
}