## 1. 스프링 기초

01. 스프링 프레임워크란?

중량급인 J2EE 컨테이너를 대신할 경량 컨테이너로서 스프링을 고안.
경량 컨테이너란 POJO(Plain Old Java Object)로 불리는, 컨테이너와 프레임워크 등에 의존하지 않는 일반 오브젝트의 생애 주기 관리나 오브젝트 간의 의존 관계를 해결하는 아키텍처를 구현한 컨테이너.

02. 스프링 프레임워크 특징

1) 경량 컨테이너.
스프링 컨테이너는 자바 객체의 생성, 소멸과 같은 라이프 사이클을 관리하며, 스프링 컨테이너로부터 필요한 객체를 가져와 사용.

2) DI(Dependency Injection)를 지원.
설정 파일이나 어노테이션을 통해서 객체 간의  의존 관계를 설정

3) AOP(Aspect Oriented Programming) 지원.
트랜잭션이나 로깅, 보안과 같이 여러 모듈에서 공통으로 필요로 하지만 실제 모듈의 핵심은 아닌 기능들을 분리해서 각 모듈에서 적용

4) POJO(Pain Old Java) 지원.

5) 트랙잭션 처리를 위한 일관된 방법 제공.

6) JDBC, MyBatis(iBatis), 하이버네트, JPA 등 데이터베이스 처리를 위한 라이브러리

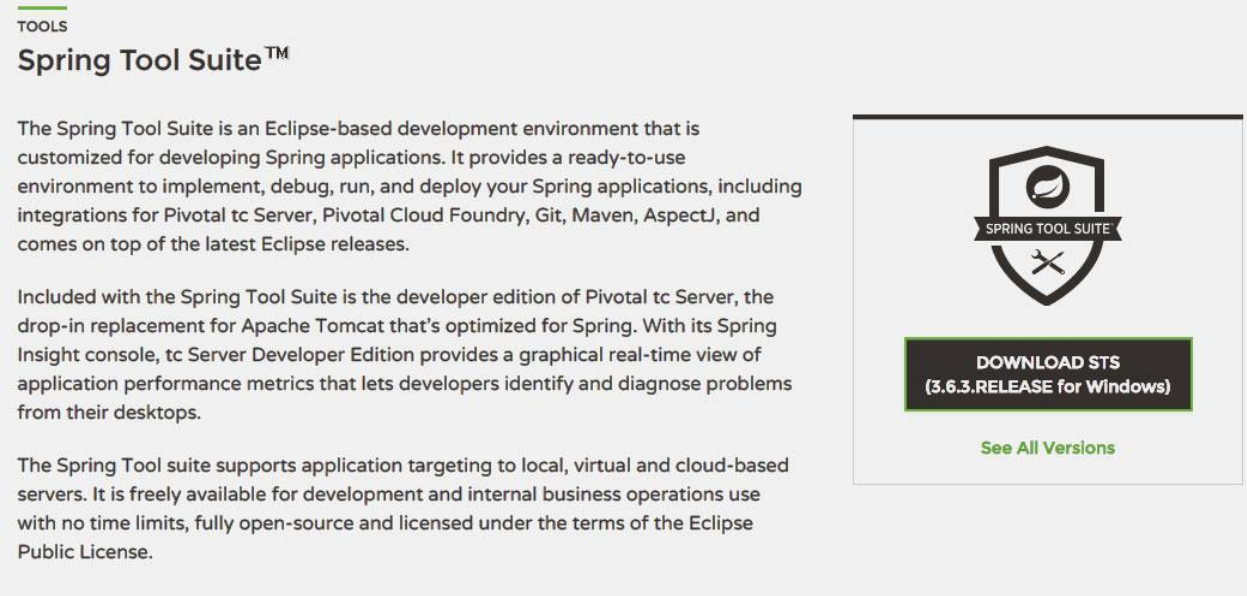
연동

7) JMS, 메일, 스케줄링 등 엔터프라이즈 어플리케이션을 개발하는데 필요한 다양한 API 연동

03. 스프링 프레임워크 설치


스프링 사이트 : http://spring.io/

STS 다운로드 : http://spring.io/tools/sts




04. Dependency Injection

객체간 의존 관계를 형성할 때 의존(dependency)하는 객체를 조립기가 삽입(inject)해 주는 방식. 스프링은 설정 파일과 어노테이션을 이용하여 손쉽게 객체 간의 의존 관계를 설정하는 기능 제공.

05. AOP

AOP(Aspect Oriented Programming)는 공통의 관심사항을 적용해서 발생하는 의존 관계의 복잡성과 코드 중복을 해소해 주는 프로그래밍 기법. 보안이나 트랜잭션과 같은 공통 기능을 별도의 모듈로 구현한 뒤, 각 기능을 필요로 하는 곳에 적용.

## 2. 스프링 DI 와 객체 관리

01. ID 란

DI(Dependency Injection)은 의존 관계 주입이다. 오브젝트 사이의 의존 관계를 만드는

것이다. 오브젝트 사이의 의존 관계를 만든다는 말은 어떤 오브젝트의 프로퍼티(인스턴스 변수를 가리킴)에 그 오브젝트가 이용할 오브젝트를 설정하는 것이다. 이것을 학술적으로 말하면, 어떤 오브젝트가 의존(이용)할 오브젝트를 주입 혹은 인젝션(프로퍼티에 설정)한다는 것이다.

02. 스프링 컨테이너

스프링은 객체를 관리하는 컨테이너를 제공. 스프링은 컨테이너에 객체를 담아 두고, 필요할 때에 컨테이너로부터 객체를 가져와 사용할 수 있도록 하고 있다.

1) BeanFactory 인터페이스

org.springframework.beans.factory.BeanFactory 인터페이스는 빈 객체를 관리하고 각 빈 객체간의 의존 관계를 설정해 주는 기능을 제공하는 가장 단순한 컨테이너.

구현 클래스는 org.springframework.beans.factory.xml.XmlBeanFactory

```
Resource resource = new FileSystemResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(reource);
AgentService agent = (AgentService)factory.getBean("agent");
```

Resource 구현 클래스

| 클래스 | 설명 |
|---|---|
| org.springframework.core.io.FileSystemResource | 파일 시스템의 특정 파일로부터 정보를 읽어 온다. |
| org.springframework.core.io.In | InputStream 으로부터 정보를 읽어 온다. |

| | |
|---|---|
| putStreamResource | |
| org.springframework.core.io.ClassPathResource | 클래스패스에 있는 자원으로부터 정보를 읽어 온다. |
| org.springframework.core.io.UrlResource | 특정 URL 로부터 정보를 읽어 온다. |
| org.springframework.web.context.support.ServletContextResource | 웹 어플리케이션의 루트 디렉터리를 기준으로 지정한 경로에 위치한 자원으로부터 정보를 읽어 온다. |

2) ApplicationContext 인터페이스와 WebApplicationContext 인터페이스

org.springframework.context.ApplicationContext 인터페이스

빈 관리 기능, 빈 객체 라이프 사이클, 파일과 같은 자원 처리 추상화, 메시지 지원 및 국제화 지원, 이벤트 지원, XML 스키마 확장을 통한 편리한 설정 등 추가적인 기능을 제공.

```
String configLocation = "config/applicationContext.xml";
ApplicationContext context = new ClassPathXmlApplicationContext(configLocation);
ParserFactory factory = (ParserFactory)context.getBean("parserFactory");
```

여러 XML 파일 사용시

```
String[] configLocation = new
String[]{"config/applicationContext.xml","config/aop.xml"};
ApplicationContext context = new ClassPathXmlApplicationContext(configLocation);
ParserFactory factory = (ParserFactory)context.getBean("parserFactory");
```

org.springframework.web.context.WebApplicationContext 인터페이스

웹 어플리케이션을 위한 ApplicationContext 로서 하나의 웹 어플리케이션(즉, 하나의 ServletContext) 마다 한 개 이상의 WebApplicationContext 를 가질 수 있다.

웹 어플리케이션에서 web.xml 파일에 설정을 통해 XmlWebApplicationContext 객체를 생성하고 사용.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>WEB-INF/applicationContext.xml </param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

03. 빈(Bean) 생성과 의존 관계 설정

1) 빈 객체 설정 및 컨테이너를 통한 빈 객체 사용

```
<bean id="articleDao" class="com.spring.chap02.MysqlArticleDao"></bean>
or
<bean name="articleDao" class="com.spring.chap02.MysqlArticleDao"></bean>
```

XmlBeanFactory 클래스를 이용해서 빈 객체를 가져와 사용하는 코드

```
Resource resource = new ClassPathResource("applicationContext.xml");
//스프링 컨테이너 생성
BeanFactory beanFactory = new XmlBeanFactory(resource);
//스프링 컨테이너로부터 빈 객체를 가져와 사용
WriteArticleService articleService = (WriteArticleService)
beanFactory.getBean("writeArticleService");
```

스프링 3 버전부터 타입 변환 대신 제네릭(Generic)을 이용해서 원하는 타입으로 빈을 구할 수 있는 getBean() 메서드 제공

```
ArticleDao articleDao = beanFactory.getBean("articleDao",ArticleDao.class);
```

2) 의존 관계 설정

2-1) 생성자 방식

```
<bean name="writeArticleService"
class="madvirus.spring.chap01.WriteArticleServiceImpl">
        <constructor-arg>
                <ref bean="articleDao" />
        </constructor-arg>
</bean>
<bean name="articleDao"
class="madvirus.spring.chap01.MySQLArticleDao"></bean>
```

2-2) 프로퍼티 설정 방식

```
<bean name="writeArticleService"
class="madvirus.spring.chap02.WriteArticleServiceImpl">
        <property name="articleDao">
            <ref bean="mysqlArticleDao" />
         </property>
</bean>
<bean name="mysqlArticleDao" class="madvirus.spring.chap02.MysqlArticleDao"
/>
```

2-3) XML 네임스페이스를 이용한 프로퍼티 설정

```
        <beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:p="http://www.springframework.org/schema/p"
xmlns:aop="http://www.springframework.org/schema/aop"
            xmlns:context="http://www.springframework.org/schema/context"
            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-3.0.xsd
                http://www.springframework.org/schema/aop
                http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
                http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
```

```
<bean id="monitor" class="madvirus.spring.chap02.SystemMonitor"
        p:periodTime="10" p:sender-ref="smsSender" />
```

주요 스키마 명세

| 명칭 | 스키마 파일 | URI | 설명 |
|---|---|---|---|
| bean | spring-beans-3.1.xsd | http://www.springframework.org/schema/beans | Bean(컴포넌트)설정 |
| context | spring-context-3.1.xsd | http://www.springframework.org/schema/context | Bean(컴포넌트)검색과 애노테이션 설정 |
| util | spring-util-3.1.xsd | http://www.springframework.org/schema/util | 정의와 프로퍼티 파일을 불러오는 등의 유틸리티 기능 설정 |
| jee | spring-jee-3.1.xsd | http://www.springframework.org/schema/jee | JNDI 의 lookup 및 EJB 의 lookup 설정 |
| lang | spring-lang-3.1.xsd | http://www.springframework.org/schema/lang | 스크립트 언어를 이용할 경우의 설정 |
| aop | spring-aop-3.1.xsd | http://www.springframework.org/schema/aop | AOP 설정 |
| tx | spring-tx-3.1.xsd | http://www.springframework.org/schema/tx | 트랜잭션 설정 |
| mvc | spring-mvc-3.1.xsd | http://www.springframework.org/schema/mvc | Spring MVC 설정 |

2-4) 룩업 메서드 인젝션 방식

```
package madvirus.spring.chap02;

public abstract class Processor {

	public void process(String commandName) {
		CommandFactory factory = getCommandFactory();
		Command command = factory.createCommand(commandName);
		command.execute();
	}

	   //룩업 메서드를 이용해서 의존 객체를 구함
	protected abstract CommandFactory getCommandFactory();
}
```

- 접근 수식어가 public 이나 protected 여야 한다.
- 리턴 타입이 void 가 아니다.
- 인자를 갖지 않는다.
- 추상 메서드여도 된다.
- final 이 아니다.

2-5) 임의 빈 객체 전달

```
<bean id="processor" class="madvirus.spring.chap02.SystemMonitor"
p:periodTime="10">
```

```
        <property name="sender">
        <!-- 임의 빈 객체 →
        <bean class="madvirus.spring.chap02.SmsSender">
              <constructor-arg value="true"/>
        </bean>
        </property>
</bean>
```

2-6) 콜렉션 타입 프로퍼티 설정

| 태그 | 컬렉션 타입 | 설명 |
|---|---|---|
| <list> | java.util.List<br>자바 배열 | List<br>타입이나 배열에 값 목록을 전달할 때 사용 |
| <map> | java.util.Map | Map 타입에 <키,값> 목록을 전달할 때 사용 |
| <set> | java.util.Set | Set 타입에 값 목록을 전달할 때 사용 |
| <properties> | java.util.Properties | Properties 타입에 <프로퍼티이름,<br>프로퍼티값> 목록을 전달할 때 사용 |

- List 타입과 배열

```
      <bean name="performanceMonitor"
class="madvirus.spring.chap02.PerformanceMonitor">
                  <property name="deviations">
```

```
                <list>
                        <value type="java.lang.Double">0.2</value>
                        <value type="java.lang.Double">0.3</value>
                </list>
        </property>
    </bean>
```

- Map 타입

```
<!-- Map 타입 프로퍼티 설정 -->
    <bean name="protocolHandlerFactory"
class="kr.spring.ch10.ProtocolHandlerFactory">
        <!-- setMap 메서드를 찾아서 객체 전달 -->
        <property name="map">
            <!-- <map> -->
             <map key-type="java.lang.String" value-
type="java.lang.Object">
                    <entry>
                        <key><value>soap</value></key>
                        <ref bean="soapHandler"/>
                    </entry>
                    <entry>
                        <key><value>rest</value></key>
                        <ref bean="restHandler"/>
                    </entry>
            </map>
        </property>
    </bean>
```

- Properties 타입

```
<!-- Properties 타입 프로퍼티 설정 -->
	<bean name="bookClient" class="kr.spring.ch11.BookClient">
		<property name="prop">
			<props>
				<prop key="server">192.168.0.1</prop>
				<prop key="connectionTimeout">5000</prop>
			</props>
		</property>
	</bean>
```

- Set 타입

```
<!-- Set 타입 프로퍼티 설정 -->
	<bean name="bookClient2" class="kr.spring.ch12.BookClient">
		<property name="subSet">
			<set>
				<value>10</value>
				<value>20</value>
			</set>
		</property>
	</bean>
```

3) 의존 관계 자동 설정

| byName | 프로퍼티의 이름과 같은 이름을 갖는 빈 객체를 설정 |
|---|---|

| | |
|---|---|
| byType | 프로퍼티의 타입과 같은 타입을 갖는 빈 객체를 설정 |
| constructor | 생성자 파라미터 타입과 같은 타입을 갖는 빈 객체를 생성자에 전달 |
| autodetect | constructor 방식을 먼저 적용하고, byType<br>방식을 적용하여 의존 객체를 설정 |

04. 빈 범위 설정

스프링 빈의 범위를 설정

| 범위 | 설명 |
|---|---|
| singleton | 스프링 컨테이너에 한 개의 빈 객체만 존재한다.(기본값) |
| prototype | 빈을 사용할 때 마다 객체를 생성 |
| request | HTTP 요청 마다 빈 객체를 생성.<br>WebApplicationContext 에서만 적용 가능 |
| session | HTTP 세션 마다 빈 객체를 생성.<br>WebApplicaitionContext 에서만 적용 가능 |
| global-<br>session | 글로벌 HTTP 세션에 대해 빈 객체를 생성.<br>포틀릿을 지원하는 컨텍스트에 대해서만 적용 가능하다. |

05. 어노테이션 기반 설정

1) @Required 어노테이션을 이용한 필수 프로퍼티 검사

org.springframework.beans.factory.annotation 패키지에 위치한 @Required

어노테이션은 필수 프로퍼티를 명시할 때 사용.

2) @Autowired 어노테이션을 이용한 자동 설정

org.springframework.beans.factory.annotation 패키지에 위치한 @Autowired 어노테이션은 의존 관계를 자동으로 설정할 때 사용

2-1) @Autowired 어노테이션 적용 프로퍼티의 필수 여부 지정

```
@Autowired(required=false)
```

(2) @Qualifier 어노테이션을 이용한 자동 설정 제한

```
@Autowired
@Qualifier("main")
private Recorder recorder;

<bean id="recorder" class="com.spring.ch04.homecontrol.Recorder">
    <qualifier value="main"/>
</bean>
```

1.3 @Resource 어노테이션을 이용한 프로퍼티 설정

```
@Resource(name = "camera1")
private Camera camera1;

@Resource(name = "camera2")
private Camera camera2;
```

1.4 @PostConstruct 어노테이션 및 @PreDestory 어노테이션과 라이프 사이클

@PostConstruct 어노테이션과 @PreDestory 어노테이션은 javax.annotation 패키지에 위치하며, @Resource 어노테이션과 마찬가지로 자바 6 및 JEE 5 에 추가된 어노테이션으로서 라이프 사이클의 초기화 및 제거 과정을 제공한다.

```
@PostConstruct
public void init(){
    //초기화 처리
}
@PreDestory
public void close(){
    //자원 반환 등 종료 처리
}
```

06. 어노테이션을 이용한 자동 스캔

<context:component-scan> 태그를 추가하면 스프링은 지정한 패키지에서 @Component 어노테이션이 (또느 하위 어노테이션이) 적용된 클래스를 검색하여 빈으로 등록하게 된다.

```
<context:component-scan base-package="com.spring.ch06" />
```

1) 자동 검색된 빈의 이름과 범위

```
@Component
public class HomeController{
```

```
}
```

특정한 이름을 명시해 주고 싶다면 다음과 같이 어노테이션의 속성에 빈의 이름을 입력

```
@Component("homeControl")
public class HomeController{
}
```

2) @Component 의 확장

| 어노테이션 | 설명 |
|---|---|
| @Controller | 프레젠테이션 층 스프링 MVC 용 애노테이션 |
| @Service | 비즈니스 로직 층 Service 용 애노테이션. |
| @Repository | 데이터 액세스 층의 DAO 용 애놑이션.예외를 모두 DataAccessException 으로 변환한다.(PersistenceExceptionTranslationPostProcessor 를 Bean 정의 파일에 등록함으로써 유효해진다.) |

3) 빈의 범위 설정

스프링은 기본적으로 빈의 범위를 "singleton"으로 설정하지만 빈의 범위를 변경할 때는 @Scope 를 이용

```
@Component
@Scope("prototype")
```

```
public class HomeController{
}
```

4) @Scope 의 주요 Value 속성

| Value 속성 | 설명 |
|---|---|
| singleton | 인스턴스를 싱글톤으로 한다. |
| prototype | 이용할 때마다 인스턴스화한다. |
| request | Servlet API 의 Request 스코프인 동안만 인스턴스가 생존* |
| session | Servlet API 의 Session 스코프인 동안만 인스턴스가 생존* |

* Servlet2.3 에서는 RequestContextFilter, Servlet2.4 에서는 RequestContextLinstener 설정이 web.xml 에 필요

07. 자바 코드 기반 설정

Spring JavaConfig 프로젝트는 XML 이 아닌 자바 코드를 이용해서 컨테이너를 설정할 수 있는 기능을 제공하는 프로젝트

```
@Configuration
public class SpringConfig{

  @Bean
  public AlarmDevice alarmDevice(){
    return new SmsAlarmDevice();
 }
}
```

**3. 스프링 AOP**

로깅과 같은 기본적인 기능에서부터 트랜잭션이나 보안과 같은 기능에 이르기까지 어플리케이션 전반에 걸쳐 적용되는 공통기능이 존재하고 이런 공통 기능들은 어플리케이션의 핵심 비즈니스 로직과는 구분되는 기능이다. 핵심 비즈니스 기능과 구분하기 위해 공통 기능을 공통 관심 사항(cross-cutting concern)이라고 표현하며, 핵심 로직을 핵심 관심 사항(core concern)이라고 표현.

01 AOP 소개

AOP(Aspect Oriented Programming)는 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법. 공통 관심 사항을 구현한 코드를 핵심 로직을 구현한 코드 안에 삽입하는 것을 의미.

1) AOP 용어

| | |
|---|---|
| Advice | 언제 공통 관심 기능을 핵심 로직에 적용할 지를 정의. |
| Joinpoint | Advice 를 적용 가능한 지점을 의미. 메서드 호출,<br>필드 값 변경 등이 Joinpoint 에 해당 |
| Pointcut | Joinpoint 의 부분 집합으로서 실제로 Advice 가 적용되는 Joinpoint 를 나타냄. |
| Weaving | Advice 를 핵심 로직 코드에 적용하는 것을 weaving 이라고 함. |
| Aspect | 여러 객체에 공통으로 적용되는 공통 관심 사항 |

2) 세 가지 Weaving 방식

- 컴파일 시에 Weaving 하기
- 클래스 로딩 시에 Weaving 하기
- 런타임 시에 Weaving 하기

3) 스프링에서의 AOP

- XML 스키마 기반의 POJO 클래스를 이용한 AOP 구현
- @Aspect 어노테이션 기반의 AOP 구현

4) 구현 가능한 Advice 종류

| 종류 | 설명 |
|---|---|
| Before Advice | 대상 객체의 메서드 호출 전에 공통 기능을 실행 |
| After Returning Advice | 대상 객체의 메서드가 예외 없이 실행한 이후에 공통 기능을 실행 |
| After Throwing Advice | 대상 객체의 메서드를 실행하는 도중 예외가 발생한 경우에 공통 기능을 실행 |
| After Advice | 대상 객체의 메서드를 실행하는 도중에 예외가 발생했는지의 여부와 상관없이 메서드 실행 후 공통 기능을 실행. |
| Around Advice | 대상 객체의 메서드 실행 전,<br>후 또는 예외 발생 시점에 공통 기능을 실행하는데 사용 |

02. XML 스키마를 이용한 AOP 설정

```
<!-- Aspect 설정 : Advice 를 어떤 Pointcut 에 적용할지 설정 -->
<aop:config>
	<aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
		<aop:pointcut expression="execution(public * com.spring.board..*(..))"
id="publicMethod"/>
		<aop:around method="trace" pointcut-ref="publicMethod"/>
	</aop:aspect>
</aop:config>

<bean id="writeArticleService"
class="com.spring.board.service.WriteArticleServiceImpl">
	<constructor-arg>
		<ref bean="articleDao"/>
	</constructor-arg>
</bean>
<bean id="articleDao"  class="com.spring.board.dao.WriteArticleDao"/>
```

<aop:config> : AOP 설정 정보임을 나타냄

<aop:aspect> : Aspect 를 설정함

<aop:pointcut> : Pointcut 을 설정함

<aop:around> : Around Advice 를 설정함

Advice 정의 관련 태그

| 종류 | 설명 |
|---|---|
| <aop:before> | 메서드 실행 전에 적용되는 Advice 를 정의한다. |

| | |
|---|---|
| <aop:after-returning> | 메서드 정상적으로 실행된 후에 적용되는 Advice 를 정의한다. |
| <aop:after-throwing> | 메서드가 예외를 발생시킬 때 적용되는 Advice 를 정의한다. try-catch 블록에서 catch 블록과 비슷하다. |
| <aop:after> | 메서드가 적상적으로 실행되느니지 또는 예외를 발생시키는지 여부에 상관없이 적용되는 Advice 를 정의한다. try-catch-finally 에서 finally 블록과 비슷하다. |
| <aop:around> | 메서드 호출 이전, 이후, 예외 발생 등 모든 시점에 적용 가능한 Advice 를 정의한다. |

AspectJ 의 Pointcut 표현식

```
<aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
		<aop:pointcut expression="execution(public * 
com.spring.board..*(..))" id="publicMethod"/>
		<aop:around method="trace" pointcut-ref="publicMethod"/>
	</aop:aspect>
```

execution 명시자는 Advice 를 적용할 메서드를 명시할 때 사용됨

```
execution(식어패턴 리턴타입패턴 클래스이름패턴 이름패턴(파라미터패턴)
```

예)

```
execution(public void set*(..))
```

| |
|---|
| 리턴 타입이 void 이고 메서드 이름이 set 으로 시작하고, 파라미터가 0 개 이상인 메서드 호출 |
| **execution(* com.spring.ch01.*.*())**<br>com.spring.ch01 패키지의 파라미터가 없는 모든 메서드 호출 |
| **execution(* com.spring.ch01..*.*(..))**<br>com.spring.ch01 패키지 및 하위 패키지에 있는 파라미터가 0 개 이상인 메서드 호출 |
| **execution(Integer com.spring.ch01..WriteArticleService.write(..))**<br>리턴 타입이 Integer 인 WriteArticleService 인터페이스의 write() 메서드 호출 |
| **execution(* get*(*))**<br>이름이 get 으로 시작하고 1 개의 파라미터를 갖는 메서드 호출 |
| **execution(* get*(*,*))**<br>이름이 get 으로 시작하고 2 개의 파라미터를 갖는 메서드 호출 |
| **execution(* read*(Integer,*))**<br>메서드 이름이 read 로 시작하고, 첫 번째 파라미터 타입이 Integer 이며, 1 개 이상의 파라미터를 갖는 메서드 호출 |

03) @Aspect 어노테이션을 이용한 AOP

```
@Aspect
public class ProfilingAspect {

	@Pointcut("execution(public * com.spring.board..*(..))")
	private void profileTarget(){}

	@Around("profileTarget()")
	public Object trace(ProceedingJoinPoint joinPoint)
```

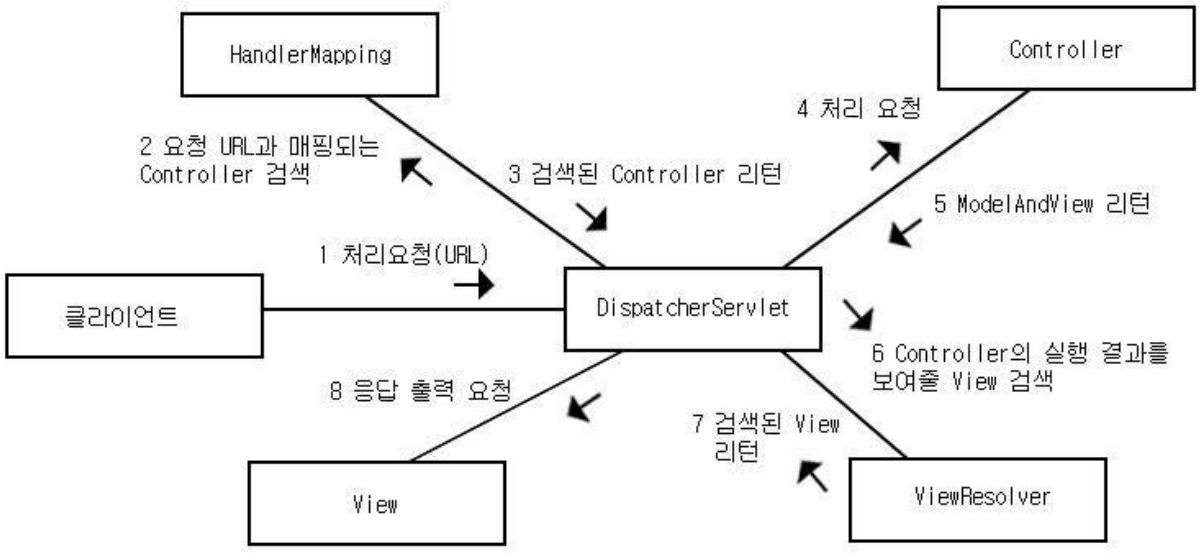
```
                                                throws Throwable{
            String signatureString =
                        joinPoint.getSignature().toShortString();
            System.out.println(signatureString + " 시작");
            long start = System.currentTimeMillis();
            Object result = null;
            try{
                  //핵심 비지니스 로직을 수행
                  result = joinPoint.proceed();
            }catch(Exception e){
                  e.printStackTrace();
            }finally{
                  //메서드 실행 직후의 시간을 저장
                  long finish = System.currentTimeMillis();
                  System.out.println(signatureString + "종료");
                  System.out.println(signatureString + " 실행 시간 : " + (finish-
start)+"ms");
            }
            return result;
      }
}
```

**4. 스프링 MVC**

스프링 MC 의 주요 구성 요소

| 구성요소 | 설명 |
| --- | --- |

| | |
|---|---|
| DispatcherServlet | 클라이언트의 요청을 전달받는다. 컨트롤러에게 클라이언트의 요청을 전달하고, 컨트롤러가 리턴한 결과값을 View 에 전달하여 알맞은 응답을 생성하도록 한다. |
| HandlerMapping | 클라이언트의 요청 URL 을 어떤 컨트롤러가 처리할지를 결정한다. |
| 컨트롤러(Controller) | 클라이언트의 요청을 처리한 뒤, 그 결과를 DispatcherServlet 에 알려준다. 스트럿츠의 Action 과 동일한 역할을 수행한다. |
| ModelAndView | 컨트롤러가 처리한 결과 정보 및 뷰 선택에 필요한 정보를 담는다. |
| ViewResolver | 컨트롤러의 처리 결과를 생성할 뷰를 결정한다. |
| 뷰(View) | 컨트롤러의 처리 결과 화면을 생성한다. JSP 나 Velocity 템플릿 파일 등을 뷰로 사용한다. |




01. 스프링 MVC 설정

1) DispatcherServlet 설정 및 스프링 컨텍스트 설정

DispatcherServlet 의 설정은 웹 어플리케이션의 /WEB-INF/web.xml 파일에 추가

```
<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>*.do</url-pattern>
</servlet-mapping>
```

2) 컨트롤러 구현 및 설정 추가

```
@Controller
public class HelloController {

        @RequestMapping("/hello.do")
        public ModelAndView hello() {
                ModelAndView mav = new ModelAndView();
                //뷰 이름 지정
```

```
                mav.setViewName("hello");
                //뷰에서 사용할 데이터 셋팅
                mav.addObject("greeting", "Hello World!!");
                return mav;
        }
}
```

3) servlet-context.xml 설정

bean 설정 및 viewResolver 설정

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

        <!-- bean 설정 -->
        <beans:bean id="helloController"
class="kr.spring.ch01.controller.HelloController" />

        <!-- viewResolver 설정 -->
```

```
        <beans:bean id="viewResolver"

        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <beans:property name="prefix" value="/WEB-INF/views/" />
                <beans:property name="suffix" value=".jsp" />
        </beans:bean>
</beans:beans>
```

4) 뷰 코드 구현

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>인사</title>
</head>
<body>
인사말: <strong>${greeting}</strong>
</body>
</html>
```

02. Validator 인터페이스를 이용한 폼 값 검증

```
package kr.spring.ch08.validator;

import kr.spring.ch08.model.Address;
import kr.spring.ch08.model.MemberInfo;
```

```
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public class MemberInfoValidator implements Validator {

	@Override
	//Validator 가 해당 클래스에 대한 값 검증을 지원하는 지의 여부를 리턴
	public boolean supports(Class<?> clazz) {
		return MemberInfo.class.isAssignableFrom(clazz);
	}

	@Override
	//target 객체에 대한 검증을 실행한다. 검증 결과 문제가 있을 경우 errors
객체에
	//어떤 문제인지에 대한 정보를 저장한다.
	public void validate(Object target, Errors errors) {
		MemberInfo memberInfo = (MemberInfo) target;
		if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {
			errors.rejectValue("id", "required");
		}
		if (memberInfo.getName() == null
				|| memberInfo.getName().trim().isEmpty()) {
			errors.rejectValue("name", "required");
		}
		Address address = memberInfo.getAddress();
		if (address == null) {
			errors.rejectValue("address", "required");
		}
	}
}
```

03. 메시지 처리

1). MessageSource 를 이용한 메시지 국제화 처리

국제화 지언을 위해 org.springframework.context.MessageSource 인터페이스를 제공

```
        <bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">
          <property name="basenames">
                    <value>message.greeting</value>
          </property>
    </bean>
```

- ResourceBundle 은 프로퍼티 파일의 이름을 이용해 언어 및 지역에 따른 메시지 로딩

```
message.properties : 기본 메시지
message_ko.properties : 한글 메시지
message_en_US.properties : 미국을 위한 영어 메시지
```

2) 에러 코드 호출시 메서드 사용

reject() 메서드를 이용하여 커맨드 객체 자체에 대한 에러 코드를 입력한 경우, 다음의 순서로 메시지 코드를 생성

```
    1)  에러코드 + "." + 커맨드 객체 이름
```

2) 에러코드

reject() 메서드를 이용하여 특정 필드에 대한 에러 코드를 입력했다면, 다음의 순서로 메시지 코드를 생성

1) 에러코드 + "." + 커맨드 객체 이름 + "." + 필드명
2) 에러코드 + "." + 필드명
3) 에러코드 + "." + 필드 타입
4) 에러코드

예)
1) required.loginCommand.userId
2) required.userId
3) required.java.lang.String
4) required

04. 파일 업로드 처리

1) MulipartResolver 설정

```
<beans:bean id="multipartResolver"

        class="org.springframework.web.multipart.commons.CommonsMultipartResol
ver">
                <beans:property name="maxUploadSize" value="52428800" /><!--
50M -->
                <beans:property name="defaultEncoding" value="UTF-8" />
        </beans:bean>
```

CommonsMultipartResolver 클래스의 프로퍼티

| 프로퍼티 | 타입 | 설명 |
|---|---|---|
| maxUploadSize | long | 최대 업로드 가능한 바이트 크기, -1 은 제한이 없음을 의미한다. 기본값은 -1 이다. |
| maxInMemorySize | int | 디스크에 임시 파일을 생성하기 전에 메모리에 보관할 수 있는 최대 바이트 크기. 기본값은 10240 바이트이다. |
| defaultEncoding | String | 요청을 파싱할 때 사용할 캐릭터 인코딩. 지정하지 않을 경우, HttpServletRequest.setCharacterEncoding() 메서드로 지정한 캐릭터 셋이 사용된다. 아무 값도 없을 경우 ISO-8859-1 을 사용한다. |

2) 구현 예

```
try {
	File file = new File(path + "/" +
command.getReportFile().getOriginalFilename());
                                command.getReportFile().transferTo(file);
} catch (IllegalStateException e) {
	e.printStackTrace();
} catch (IOException e) {
	e.printStackTrace();
}
```

**5. JDBC**

01. 커넥션 풀을 이용한  DataSource 설정

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
            <!-- 최대 커넥션 개수 -->
            <property name="maxActive" value="50"/>
            <!-- 접속이 없을 경우 최대 유지 커넥션 개수 -->
            <property name="maxIdle" value="30"/>
            <!-- 접속이 없을 경우 최소 유지 커넥션 개수 -->
            <property name="minIdle" value="20"/>
            <!-- 최대 대기시간(초) : 초과시 연결실패 오류 발생 -->
            <property name="maxWait" value="5"/>
    </bean>
```

BasciDataSoute 클래스의 주요 프로퍼티

| 프로퍼티 | 설명 |
|---|---|
| initialSize | 초기에 풀에 생성되는 커넥션의 개수 |
| maxActive | 커넥션 풀이 제공할 최대 커넥션의 개수 |
| maxIdle | 사용되지 않고 풀에 저장될 수 있는 최대 커넥션 개수. 음수일 경우 제한이 없다. |

| | |
|---|---|
| minIdle | 사용되지 않고 풀에 저장될 수 있는 최소 커넥션 개수 |
| maxWait | 풀에 커넥션이 존재하지 않을 때, 커넥션이 다시 풀에 리턴될 때까지 대기하는 시간. 단위는 1/1000 초이며, -1 일 경우 무한히 대기한다. |
| minEvictableIdleTimeMillis | 사용되지 않는 커넥션을 추출할 때 이 속성에서 지정한 시간 이상 비활성화 상태인 커넥션만 추출한다. 양수가 아닌 경우 비활성화된 시간으로 풀에서 제거되지 않는다. 시간 단위는 1/1000 초이다. |
| timeBetweenEvictionRunsMillis | 사용되지 않은 커넥션을 추출하는 쓰레드의 실행 주기를 지정한다. 양수가 아닐 경우 실행되지 않는다. 단위는 1/1000 초이다. |
| numTestsPerEvictionRun | 사용되지 않는 커넥션을 몇 개 검사할지 지정한다. |
| testOnBorrow | true 일 경우 커넥션 풀에서 커넥션을 가져올 때 커넥션이 유효한지의 여부를 검사한다. |
| testOnReturn | true 일 경우 커넥션 풀에 커넥션을 반환할 때 커넥션이 유효한지의 여부를 검사한다. |
| testWhileIdle | true 일 경우 비활성화 커넥션을 추출할 때 커넥션이 유효한지의 여부를 검사해서 유효하지 않은 커넥션은 풀에서 제거한다. |

02. JdbcTemplate 클래스를 이용한 JDBC 프로그래밍

1) JdbcTemplate 빈 설정

```
<!-- JdbcTemplate 객체 생성 -->
<bean name="jdbcTemplate"
```

```
class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
 </bean>
```

2) Dao 클래스에서의 JdbTemplate 객체 사용

```
        @Autowired
        private JdbcTemplate jdbcTemplate;

        //등록
        public void insertMember(MemberCommand member){
                jdbcTemplate.update(INSERT_SQL,
                new
Object[]{member.getId(),member.getPasswd(),member.getName()});
        }
        //총 데이터 수
        public int getMemberCount(){
                return jdbcTemplate.queryForObject(SELECT_COUNT_SQL,
Integer.class);
        }
        //목록
        public List<MemberCommand> getMemberList(int startRow,int endRow){
                List<MemberCommand> list =
jdbcTemplate.query(SELECT_LIST_SQL,
                                new Object[]{startRow,endRow},
                                new RowMapper<MemberCommand>(){
                                        public MemberCommand mapRow(ResultSet rs,
int rowNum)
                                                        throws SQLException{
```

```
                                        MemberCommand member = new
MemberCommand();
                                        member.setId(rs.getString("id"));

        member.setPasswd(rs.getString("passwd"));
                                        member.setName(rs.getString("name"));

        member.setReg_date(rs.getDate("reg_date"));

                                        return member;
                                    }
                });
                return list;
        }
```

03. NamedParameterJdbcTemplate 클래스를 이용한 JDBC 프로그래밍

1) NamedParameterJdbcTemplate 빈 설정

```
<!-- NamedParameterJdbcTemplate 객체 생성 -->
        <bean name="namedParameterJdbcTemplate"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate"
>
                <constructor-arg ref="dataSource"/>
        </bean>
```

2) Dao 클래스에서의 NamedParameterJdbcTemplate 객체 사용

```
    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    //등록
    public void insertMember(MemberCommand member){
        BeanPropertySqlParameterSource beanProps = new
BeanPropertySqlParameterSource(member);
        namedParameterJdbcTemplate.update(INSERT_SQL,beanProps);
    }
    //총 데이터 수
    public int getMemberCount(){
        return
namedParameterJdbcTemplate.queryForObject(SELECT_COUNT_SQL, new
MapSqlParameterSource(),Integer.class);
    }
    //목록
    public List<MemberCommand> getMemberList(int startRow,int endRow){
        List<MemberCommand> list =
namedParameterJdbcTemplate.query(SELECT_LIST_SQL,
                        new MapSqlParameterSource()

    .addValue("start",startRow)

    .addValue("end",endRow),
                    new RowMapper<MemberCommand>(){
                        public MemberCommand mapRow(ResultSet rs,
int rowNum)
                                throws SQLException{
                            MemberCommand member = new
MemberCommand();
```

```
                                        member.setId(rs.getString("id"));

    member.setPasswd(rs.getString("passwd"));
                                        member.setName(rs.getString("name"));

    member.setReg_date(rs.getDate("reg_date"));

                                        return member;
                                    }
            });
            return list;
    }
```

**6. 트랜잭션**

01. 트랜잭션에는 지켜야 할 ACID 특성

| ACID | 의미 | 설명 |
|---|---|---|
| Atomicity | 트랜잭션의 원자성 | 트랜색션 내의 모든 처리는 전부 실행했거나 혹은 아무것도 실행되지 않았거나 둘 중 하나뿐이다. |
| Consistency | 테이터의 일관성 | 데이터에 일관성이 있어야 한다. 일관성을 지키지 않은 예:상위 테이블이 없는데 하위 테이블이 있는 경우 |
| Isolation | 트랜잭션의 독립성 | 변행해서 달리는 트랜잭션이 서로 독립된 것 |
| Durability | 데이터의 영속성 | 데이터가 영속화 된 것 |

02. JDBC 기반 트랜잭션 관리자 설정

1) 빈 설정

```
<!-- JDBC 기반 트랜잭션 관리자 설정 -->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
p:dataSource-ref="dataSource"/>
```

2) TransactionTemplate 을 이용한 트랜잭션 처리

```
public void updateMember(final MemberCommand member){
                transactionTemplate.execute(new TransactionCallback<Object>(){

                        @Override
                        public Object doInTransaction(TransactionStatus status) {

                                memberDao.updateMember(member);
                                //트랜잭션 테스트를 위해서 호출
                                memberDao.insertMember(member);
                                return new Object();
                        }
                });
        }
```

03. 선언적 트랜잭션 처리

1) 빈 설정

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="update*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
```

2) <tx:method> 태그의 속성

| 속성이름 | 설명 |
|---|---|
| name | 트랜잭션이 적용될 메서드 이름을 명시한다. '*'을 사용한 설정이 가능하다. 예를 들어. 'get*'으로 설정할 경우 이름이 get 으로 시작하는 메서드를 의미한다. |
| propagation | 트랜잭션 전파 규칙을 설정한다. |
| isolation | 트랜잭션 격리 레벨을 설정한다 |
| read-only | 읽기 전용 여부를 설정 |
| no-rollback-for | 트랜잭션을 롤백하지 않을 예외 타입을 설정 |
| rollback-for | 트랜잭션을 롤백할 예외 타입을 설정 |
| timeout | 트랜잭션을 타임 아웃 시간을 초 단위로 설정 |

3) <tx:method> 태그의 propagation 속성에 설정 가능한 값

| 속성 값 | 설명 |
|---|---|

| | |
|---|---|
| REQUIRED<br>(기본값) | 메서드를 수행하는 데 트랜잭션이 필요하다는 것을 의미한다. 현재 진행 중인 트랜잭션이 존재하면, 해당 트랜잭션을 사용한다. 존재하지 않는다면 새로운 트랜잭션을 생성한다. |
| MANDATORY | 메서드를 수행하는 데 트랜잭션이 필요하다는 것을 의미한다. 하지만, REQUIRED 와 달리, 진행 중인 트랜잭션이 존재하지 않을 경우 예외를 발생시킨다. |
| REQUIRES_NEW | 항상 새로운 트랜잭션을 시작한다. 기존 트랜잭션이 존재하면 기존 트랜잭션을 일시 중지하고 새로운 트랜잭션을 시작한다. 새로 시작된 트랜잭션이 종료된 뒤에 기존 트랜잭션이 계속된다. |
| SUPPORS | 메서드가 트랜잭션을 필요로 하지는 않지만, 기존 트랜잭션이 존재할 경우 트랜잭션을 사용한다는 것을 의미한다. 진행 중인 트랜잭션이 존재하지 않더라도 메서드는 정상적으로 동작한다. |
| NOT_SUPPORTED | 메서드가 트랜잭션을 필요로 하지 않음을 의미한다.<br>SUPPORTS 와 달리 진행 중인 트랜잭션이 존재할 경우 메서드가 실행되는 동안 트랜잭션은 일시 중지되며, 메서드 실행이 종료된 후에 트랜잭션을 계속 진행한다. |
| NEVER | 메서드가 트랜잭션을 필요로 하지 않으며, 만약 진행 중인 트랜잭션이 존재하면 예외를 발생시킨다. |
| NESTED | 기존 트랜잭션이 존재하면, 기존 트랜잭션에 중첩된 트랜잭션에서 메서드를 실행한다. 기존 트랜잭션이 존재하지 않으면 REQUIRED 와 동일하게 동작한다. 이 기능은 JDBC3.0 드라이버를 사용할 때에만 적용된다. (JTA Provider 가 이 기능을 지원할 경우에도 사용 가능하다.) |

4) <tx:method> 태그의 isolation 속성에 설정 가능한 값

| 속성 값 | 설명 |
| --- | --- |
| DEFAULT | 기본 설정을 사용한다. |
| READ_UNCOMMITTED | 다른 트랜잭션에서 커밋하지 않은 데이터를 읽을 수 있다. |
| READ_COMMITTED | 다른 트랜잭션에 의해 커밋된 데이터를 읽을 수 있다. |
| REPEATABLE_READ | 처음에 읽어 온 데이터와 두 번째 읽어 온 데이터가 동일한 값을 갖는다. |
| SERIALIZABLE | 동일한 데이터에 대해서 동시에 두 개 이상의 트랜잭션이 수행될 수 없다. |

04. 어노테이션 기반 트랜잭션 설정

1) 메서드에 직접 명시

```
@Transactional
public void updateMember(MemberCommand member){

	memberDao.updateMember(member);
	memberDao.insertMember(member);
}
```

2) @Transactional 어노테이션의 주요 속성

| 속성 | 설명 |
| --- | --- |
| propagation | 트랜잭션 전파 규칙을 설정한다. Propagation 열거형 타입에 값이 정의되어 있다. 기본값은 Propagation.REQUIRED 이다. |

| | |
|---|---|
| isolation | 트랜잭션 격리 레벨을 설정한다. Isolation 열거형 타입에 값이 정의되어 있다. |
| readOnly | 읽기 전용 여부를 설정한다. boolean 값을 설정하며, 기본값은 false 이다. |
| rollbackFor | 트랜잭션을 롤백할 예외 타입을 설정한다.<br>예, rollbackFor={Exception.class} |
| noRollbackFor | 트랜잭션을 롤백하지 않을 예외 타입을 설정한다.<br>예, noRollbackFor={ItemNotFoundException.class} |
| timeout | 트랜잭션의 타임아웃 시간을 초 단위로 설정한다. |

@Transactional 어노테이션이 적용된 스프링 빈에 실제로 트랜잭션을 적용하려면 다음과 같이 <tx:annotation-driven> 태그를 설정하면 된다.

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

3) <tx:annotation-driven> 태그의 속성

| 속성 | 설명 | 기본값 |
|---|---|---|
| transaction-manager | 사용할 PlatformTransactionManager 빈의 이름 | transactionManager |
| proxy-target-class | 클래스에 대해서 프록시를 생성할 지의 여부. true 일 경우 CGLIB 를 이용해서 프록시를 생성하며, false 인 경우 자바 다이나믹 프록시를 이용해서 프록시를 생성한다. | false |

| | | |
|---|---|---|
| order | Advice 적용 순서 | int 의 최대값<br>(가장 낮은 순위) |