

Spring & Ibatis 프레임워크 과정

11.8.27 ~ 11.10.15

한국소프트웨어기술진흥협회(KOSTA)

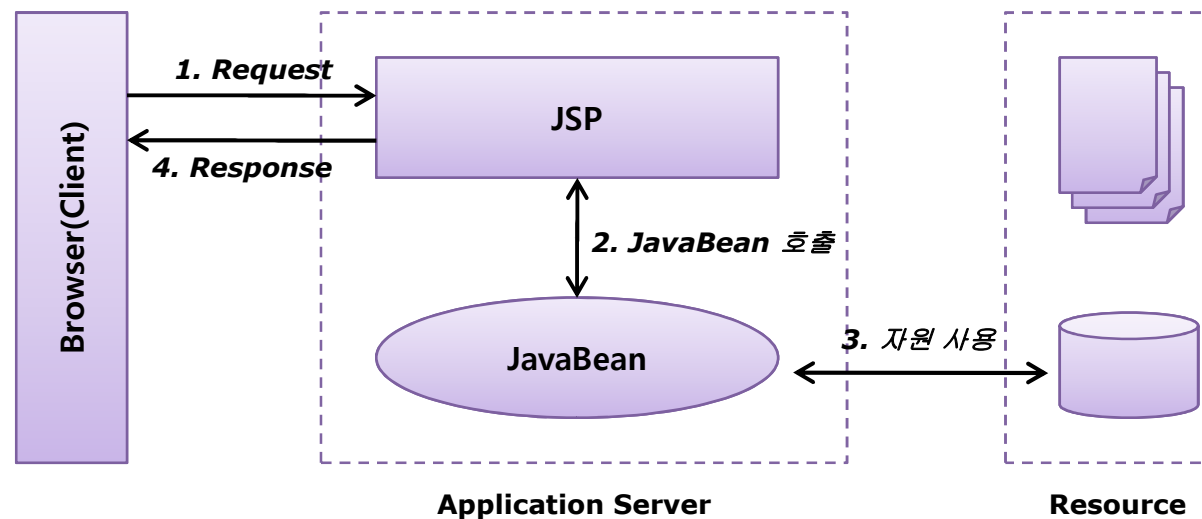


Web Application 설계방식

- 모델1 설계방식
- 모델2 설계방식

모델1 설계방식 (1/2)

- 모델1 개요
 - JSP 만 이용하여 개발하는 경우
 - JSP + Java Bean을 이용하여 개발하는 경우
 - Model2의 Controller 개념이 모호



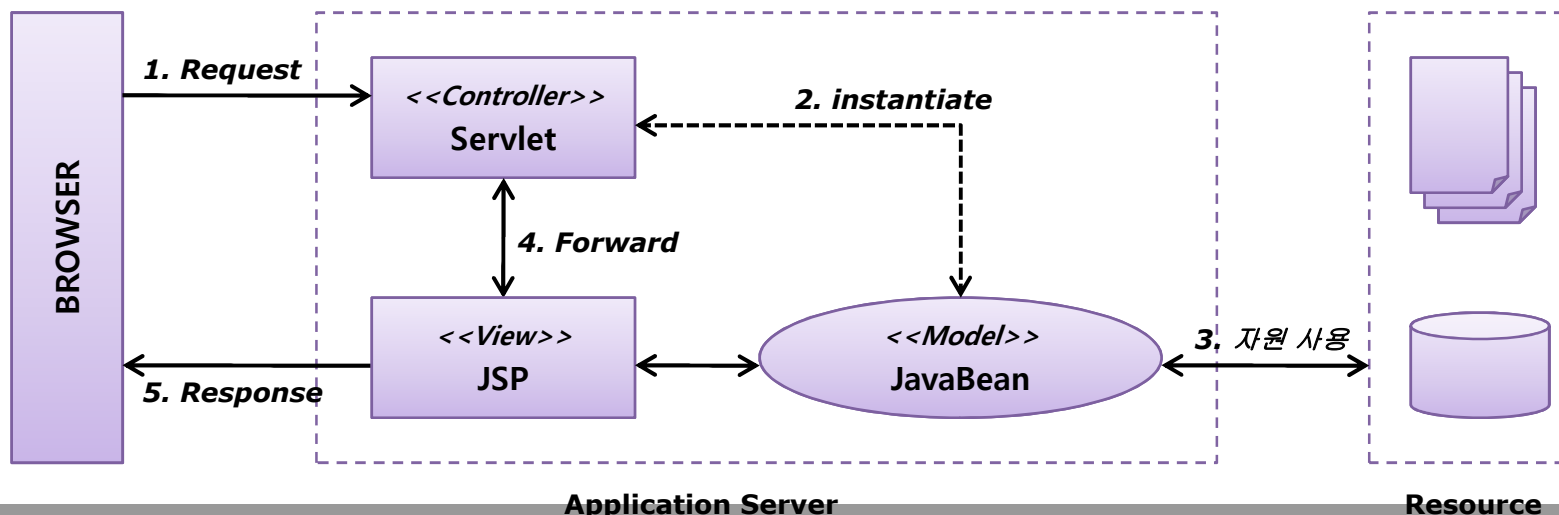


모델1 설계방식 (2/2)

- 모델1의 장단점
 - 장점
 - 개발속도가 빠름
 - 개발자의 기술적인 숙련도가 낮아도 배우기 쉬워 빠르게 적용 가능
 - 단점
 - JSP 페이지에서 프레젠테이션 로직과 비즈니스 로직이 혼재되어 복잡
 - 로직의 혼재로 인해 개발자와 디자이너의 작업 분리가 어려움
 - JSP 코드의 복잡도로 인해 유지보수가 어려워짐
- 웹 애플리케이션이 복잡해지고 사용자 요구가 증가함에 따라 새로운 개발방식을 요구

모델2 설계방식 (1/2)

- 모델2 개요
 - GUI 개발모델인 MVC를 웹 애플리케이션에 적용하여 구현한 방식
 - Application의 역할을 Model - View - Controller로 분리
 - Model: Business Logic을 담당한다. - Java Bean으로 구현
 - Business Service(Manager) - Business Logic의 workflow를 관리
 - DAO (Data Access Object) - Database와 연동하는 Business Logic을 처리.
 - View: Client에게 응답을 처리한다. - JSP로 구현
 - Controller: 클라이언트의 요청을 받아 Model과 View사이에서 일의 흐름을 조정한다. -Servlet으로 구현
 - Client의 요청을 받아 Client가 보낸 Data를 읽고 검사한다.
 - Model에게 Business Logic을 요청한다.
 - Model의 처리 결과에 맞는 View에게 응답을 요청한다.





모델2 설계방식 (2/2)

- 모델2의 장단점
 - 장점
 - 비즈니스 로직과 프리젠테이션의 분리로 인해 어플리케이션이 명료해지며 유지보수와 확장이 용이함
 - 디자이너와 개발자의 작업을 분리해 줌
 - 단점
 - 개발 초기에 아키텍처 디자인을 위한 시간의 소요로 개발 기간이 늘어남
 - MVC 구조에 대한 개발자들의 이해가 필요함

Spring framework



Spring 이란?

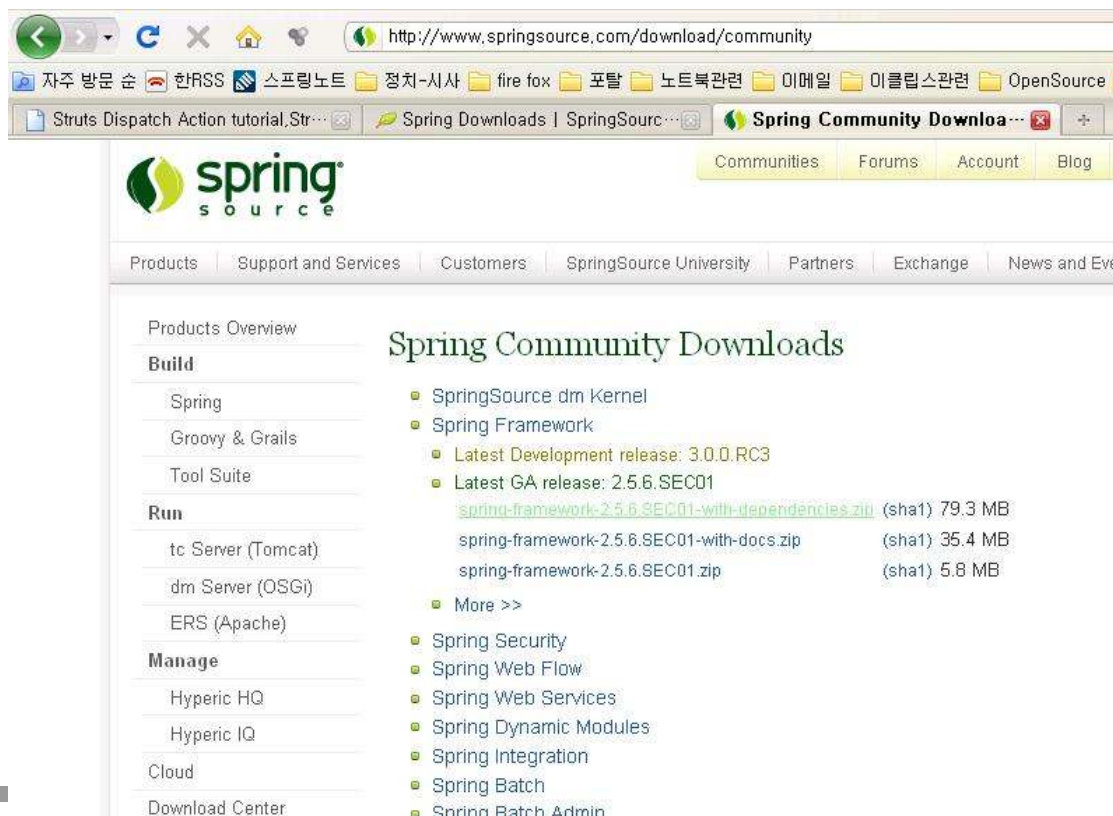
- 오픈 소스 프레임워크
 - Rod Johnson 창시
 - Expert one-on-one J2EE Design - Development, 2002, Wrox
 - Expert one-on-one J2EE Development without EJB, 2004, Wrox
 - 엔터프라이즈 어플리케이션 개발의 복잡성을 줄여주기 위한 목적
 - EJB 사용으로 수행되었던 모든 기능을 일반 POJO(Plain Old Java Object) 를 사용해서 가능하게 함.
 - 경량 컨테이너(light weight container)
 - www.springframework.org
- 주요 개념
 - 의존성 주입(lightweight dependency injection)
 - 관점 지향 컨테이너(aspect-oriented container)

Spring 장점

- 경량 컨테이너 - 객체의 라이프 사이클 관리, JEE 구현을 위한 다양한 API제공
- DI (Dependency Injection) 지원
- AOP (Aspect Oriented Programming) 지원
- POJO (Plain Old Java Object) 지원
- JDBC를 위한 다양한 API 지원
- Transaction 처리를 위한 일관된 방법제공
- 다양한 API와의 연동 지원

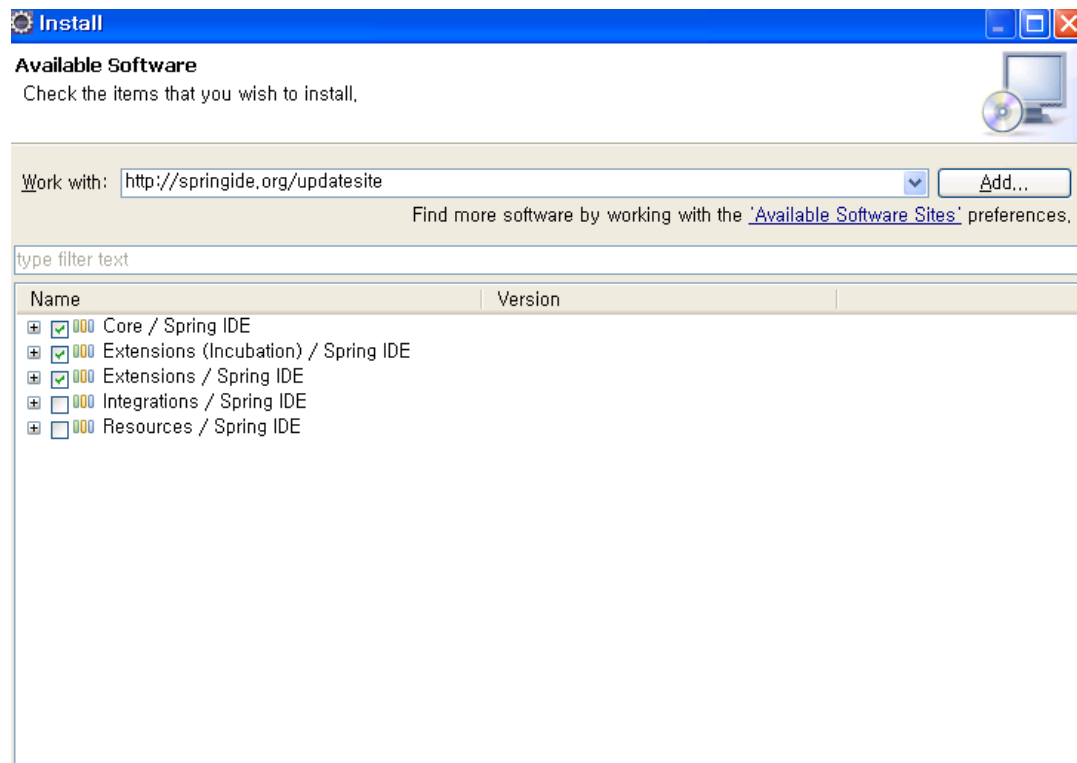
Spring Container 설치

- 스프링 커뮤니티 사이트 <http://www.springsource.org/>
- 다운로드
<http://www.springsource.org/download>
- spring-framework-2.5.6.SEC01-with-dependencies.zip 을 다운 받는다.



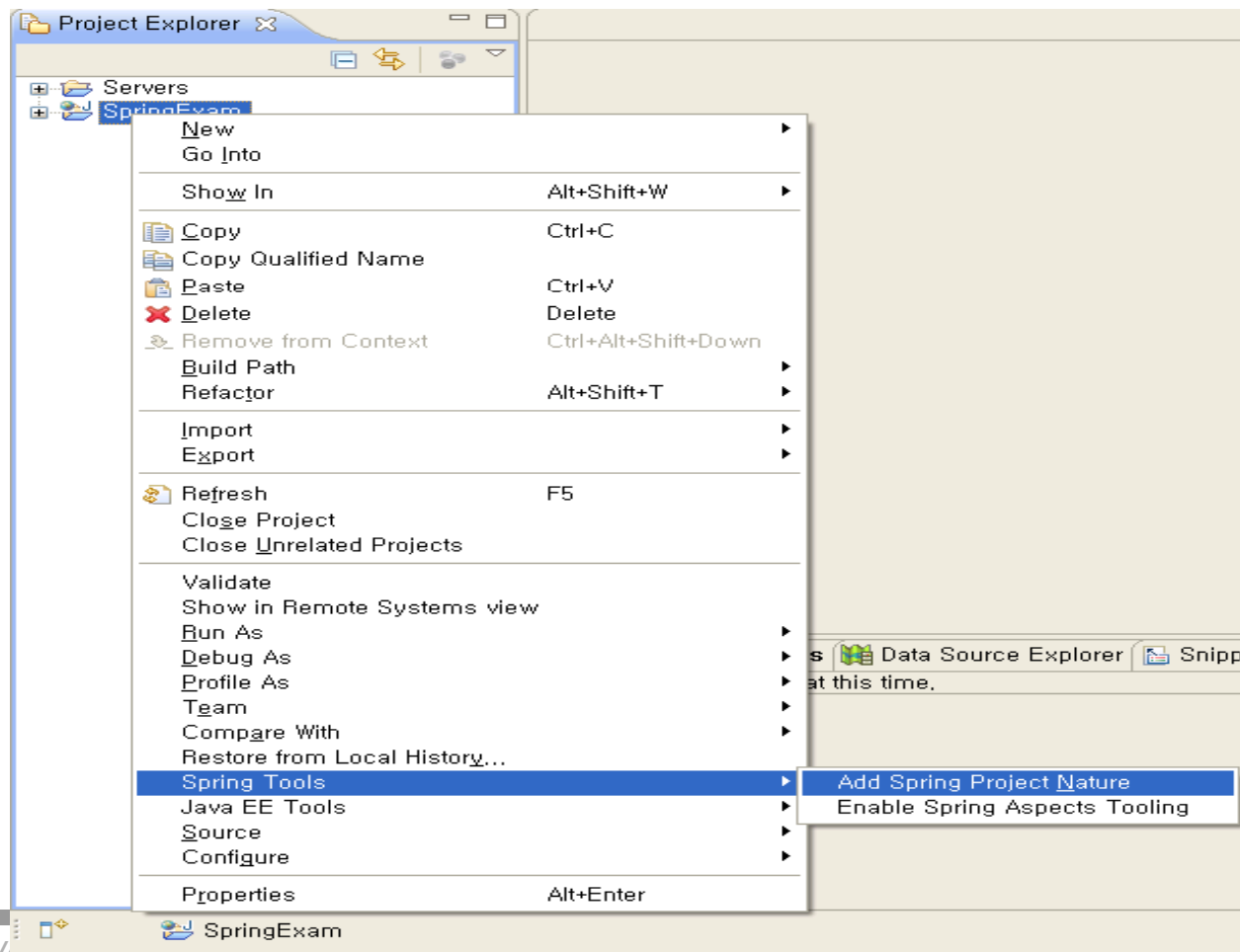
Spring IDE 이클립스에 설정 및 사용 (1/2)

- 상단 메뉴 : help-install new software
 - update site : <http://springide.org/updatesite>
 - core, Extensions 체크한 뒤 설치한다.



Spring IDE 이클립스에 설정 및 사용 (1/2)

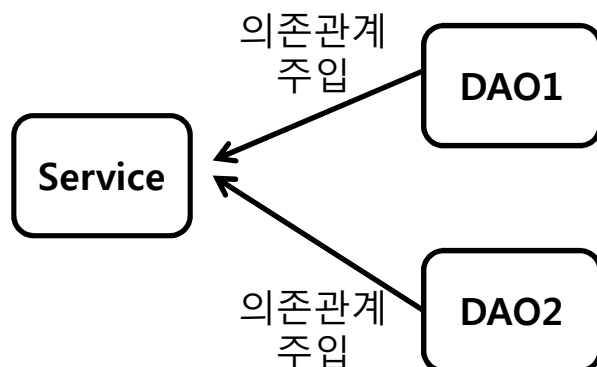
- Project 생성
- 오른 마우스 메뉴-Spring Tools-Add Spring Project Nature 선택



Dependency Injection (의존성 주입)

의존성 주입 (Dependency Injection, DI)

- 의존 관계 주입 (dependency injection)
 - 객체간의 의존관계를 객체 자신이 아닌 외부의 조립기가 수행한다.
 - 제어의 역행 (inversion of control, IoC) 이라는 의미로 사용되었음.
 - Martin Fowler, 2004
 - 제어의 어떠한 부분이 반전되는가라는 질문에 '의존 관계 주입'이라는 용어를 사용
 - 복잡한 어플리케이션은 비즈니스 로직을 수행하기 위해서 두 개 이상의 클래스들이 서로 협업을 하면서 구성됨.
 - 각각의 객체는 협업하고자 하는 객체의 참조를 얻는 것에 책임성이 있음.
 - 이 부분은 **높은 결합도(highly coupling)**와 테스트하기 어려운 코드를 양산함.
 - DI를 통해 시스템에 있는 각 객체를 조정하는 외부 개체가 객체들에게 생성시에 의존관계를 주어짐.
 - 즉, 의존이 객체로 주입됨.
 - 객체가 협업하는 객체의 참조를 어떻게 얻어낼 것인가라는 관점에서 책임성의 역행(inversion of responsibility)임.
 - 느슨한 결합(loose coupling)이 주요 강점
 - 객체는 인터페이스에 의한 의존관계만을 알고 있으며, 이 의존관계는 구현 클래스에 대한 차이를 모르는채 서로 다른 구현으로 대체가 가능



Spring의 DI 지원

- Spring Container가 DI 조립기를 제공
 - 스프링 설정파일을 통하여 객체간의 의존관계를 설정한다.
 - Spring Container가 제공하는 api를 이용해 객체를 사용한다.



Spring 설정파일

- Application에서 사용할 Spring 자원들을 설정하는 파일
- Spring container는 설정파일에 설정된 내용을 읽어 Application에서 필요한 기능들을 제공한다.
- XML 기반으로 작성한다.
- Root tag는 <beans> 이다
- 파일명은 상관없다.

예) applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

</beans>
```




Bean객체 주입 받기 – 설정파일 설정(1/2)

- 주입 할 객체를 설정파일에 설정한다.
 - **<bean>** : 스프링컨테이너가 관리할 Bean객체를 설정
 - 기본 속성
 - **name** : 주입 받을 곳에서 호출 할 이름 설정
 - **id** : 주입 받을 곳에서 호출할 이름 설정 ('/' 값으로 못 가짐)
 - **class** : 주입할 객체의 클래스
 - **factory-method** : Singleton 패턴으로 작성된 객체의 **factory** 메소드 호출 시

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="dao" class="spring.di.model.MemberDAO"/>

</beans>
```



Bean객체 주입 받기 - 설정 Bean 사용(2/2)

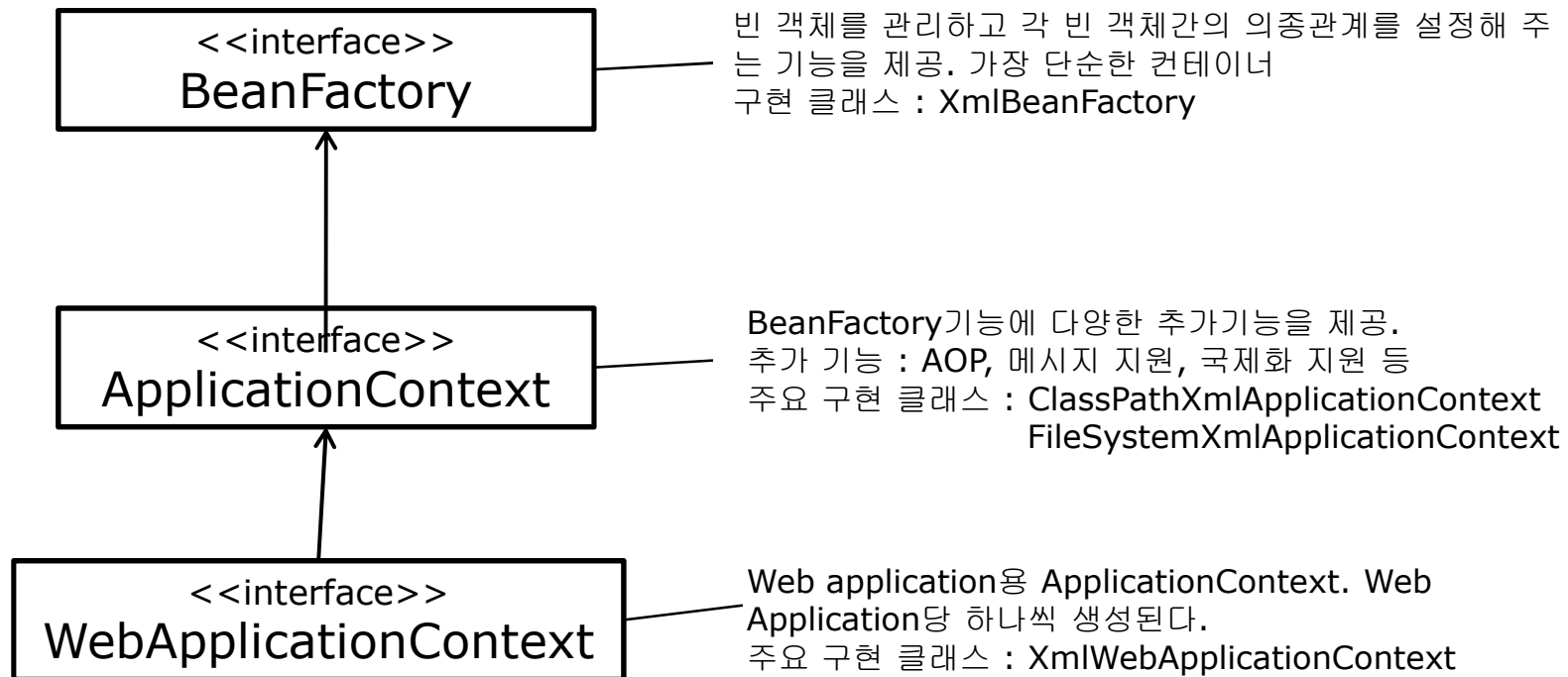
- 설정 파일에 설정한 bean을 Container가 제공하는 주입기 역할 api를 통해 주입 받는다.

```
public static void main(String [] args){  
  
    //설정파일이 어디 있는지를 저장하는 객체  
    Resource resource = new ClassPathResource("applicationContext.xml");  
    //객체를 생성해주는 factory 객체  
    BeanFactory factory = new XmlBeanFactory(resource);  
    //설정파일에 설정한 <bean> 태그의 id/name을 통해 객체를 받아온다.  
    MemberDAO dao = (MemberDAO)factory.getBean("dao");  
  
}
```



DI관련 주요 클래스 (1/2)

- Spring Container : 객체를 관리하는 컨테이너.
 - 다음 아래의 interface들을 구현한다.



DI관련 주요 클래스 (2/2)

- Resource 구현 클래스
 - Resource interface : 다양한 종류의 자원을 동일한 방식으로 통일하여 표현할 수 있게 한다.
 - XmlBeanFactory는 객체 생성시 설정 파일의 위치를 알려 줘야 한다. – Resource를 이용
 - 주요 구현 클래스
 - org.springframework.core.io.FileSystemResource
파일시스템의 특정 파일의 자원을 관리
 - org.springframework.core.io.ClassPathResource
클래스 패스에 있는 자원을 관리
 - org.springframework.web.context.support.ServletContextResource
웹 어플리케이션의 **Root** 경로를 기준으로 지정한 경로의 자원을 관리
 - InputStreamResource, PortletContextResource

예)

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



설정을 통한 객체 주입 – Constructor를 이용(1/4)

- 객체 또는 값을 생성자를 통해 주입 받는다.
- **<constructor-arg>** : **<bean>**의 하위태그로 설정한 **bean** 객체 또는 값을 생성자를 통해 주입하도록 설정
 - 설정 방법 : **<ref>**, **<value>**와 같은 하위태그를 이용하여 설정, 속성을 이용해 설정
 - 하위태그 이용
 - **<ref bean="bean name"/>** - 객체를 주입 시
 - **<value>값</value>** - 문자(String), Primitive data 주입 시
 - **type** 속성 : 값을 1차로 String으로 처리한다. 값의 타입을 명시해야 하는 경우 사용. ex) **<value type="int">10</value>**
 - 속성 이용
 - **ref="bean 이름"**
 - **value="값"**



설정을 통한 객체 주입 – Constructor를 이용(2/4)

값을 주입 받을 객체

```
package vo;
public class Person{
    private String id,
    private String name,
    private int age;

    public Person(String id){...}           //1번 생성자
    public Person(String id, String name){...} //2번 생성자
    public Person(int age){...}           //3번 생성자
}
```

1번 생성자에 주입 예

```
<bean id="person" class="vo.Person">
    <constructor-arg>
        <value>abcde</value>
    </constructor-arg>
</bean>
또는
<bean id="person" class="vo.Person">
    <constructor-arg value="abc"/>
</bean>
```



설정을 통한 객체 주입 – Constructor를 이용(3/4)

2번 생성자에 주입 예

```
<bean id="person" class="vo.Person">
  <constructor-arg>
    <value>abcde</value>
  </constructor-arg>
  <constructor-arg>
    <value>Hong Gil Dong</value>
  </constructor-arg>
</bean>
```

또는

```
<bean id="person" class="vo.Person">
  <constructor-arg value="abc"/>
  <constructor-arg value="Hong Gil Dong"/>
</bean>
```

3번 생성자에 주입 예

```
<bean id="person" class="vo.Person">
  <constructor-arg>
    <value type="int">30</value>
  </constructor-arg>
</bean>
```

또는

```
<bean id="person" class="vo.Person">
  <constructor-arg value="abc" type="int"/>
```



설정을 통한 객체 주입 – Constructor를 이용(4/4)

- bean객체를 주입

값을 주입 받을 객체

```
public class BusinessService{  
    private Dao dao = null;  
    public BusinessService(Dao dao){  
        this.dao = dao;  
    }  
}
```

```
<bean id="dao" class="spring.di.model.OracleDAO"/>  
<bean id="service" class="spring.di.model.service.BusinessService">  
    <constructor-arg>  
        <ref bean = "dao"/>  
    </constructor-arg>  
</bean>  
또는  
<bean id="service" class="spring.di.model.service.BusinessService">  
    <constructor-arg ref="dao">  
</bean>
```




설정을 통한 객체 주입 – Property를 이용(1/5)

- property를 통해 객체 또는 값을 주입 받는다.- setter 메소드
 - 주의 : setter를 통해서만 하나의 값을 받을 수 있다.
- <property> : <bean>의 하위태그로 설정한 bean 객체 또는 값을 property를 통해 주입하도록 설정
 - 속성 : name – 값을 주입할 property 이름 (setter의 이름)
 - 설정 방법
 - <ref>, <value>와 같은 하위태그를 이용하여 설정
 - 속성을 이용해 설정
 - xml namespace를 이용하여 설정



설정을 통한 객체 주입 – Property를 이용(2/5)

- 하위태그를 이용한 설정
 - `<ref bean="bean name"/>` - 객체를 주입 시
 - `<value>값</value>` - 문자(String) Primitive data 주입 시
 - type 속성 : 값의 타입을 명시해야 하는 경우 사용.
- 속성 이용
 - `ref="bean 이름"`
 - `value="값"`
- XML Namespace를 이용
 - `<beans>` 태그의 스키마설정에 namespace등록
 - `xmlns:p="http://www.springframework.org/schema/p"`
 - `<bean>` 태그에 속성으로 설정
 - 기본데이터 주입 : `p:propertyname="value". ex) <bean p:id>`
 - bean 주입 : `p:propertyname-ref="bean_id"`
`ex) <bean p:dao-ref="dao">`

설정을 통한 객체 주입 – Property를 이용(3/5)

Primitive Data Type 주입

값을 주입 받을 객체

```
package spring.vo;  
public class Person{  
    private String id,  
    private String name,  
    private int age;  
  
    public void setId(String id) {...}  
    public void setName(String name) {...}  
    public void setAge(int age) {...}
```

```
<bean id="person" class="vo.Person">  
    <property name="name">  
        <value>hong</value>  
    </property>  
    <property name="id" value="abcde"/>  
    <property name="age" value="20"/>  
</bean>
```

설정을 통한 객체 주입 – Property를 이용(4/5)

Bean 객체 주입

값을 주입 받을 객체

```
public class BusinessService{  
    private Dao dao = null;  
    public setDao(Dao dao){...}  
}
```

```
<bean id="dao" class="spring.di.model.OracleDAO"/>  
<bean id="service" class="spring.di.model.service.BusinessService">  
    <property name="dao">  
        <ref bean ="dao"/>  
    </property>  
</bean>  
또는  
<bean id="service" class="spring.di.model.service.BusinessService">  
    < property name="dao" ref="dao">  
</bean>
```

설정을 통한 객체 주입 – Property를 이용(5/5)

XML Namespace를 이용한 주입

값을 주입 받을 객체

```
public class BusinessService{  
    private Dao dao = null;  
    private int waitingTime = 0;  
    public setDao(Dao dao){...}  
    public setWaitingTime(int wt){...}  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"  
    xmlns:p="http://www.springframework.org/schema/p"  
>  
<bean id="dao" class="spring.di.model.OracleDAO"/>  
<bean id="person" class="vo.Person"  
    p:waitingTime="20"  
    p:dao-ref="dao"/>  
</beans>
```

Collection 객체 주입하기 (1/5)

- <property> 또는 <constructor-arg>의 하위 태그로 Collection 값을 설정하는 태그를 이용해 값 주입 설정

- 설정 태그

태그	Collection종류	설명
<list>	java.util.List	List 계열 컬렉션 값 목록 전달
<set>	java.util.Set	Set 계열 컬렉션 값 목록 전달
<map>	java.util.map	Map 계열 컬렉션 에 key-value 의 값 목록 전달
<props>	java.util.Properties	Properties 에 key(String)-value(String)의 값 목록 전달

- Collection에 값을 설정 하는 태그
 - <ref> : <bean>으로 등록된 객체
 - <value> : 기본데이터
 - <bean> : 임의의 bean
 - <list>,<map>,<props>,<set> : 컬렉션
 - <null> : null

Collection 객체 주입하기 (2/5)

- <list>
 - List 계열 컬렉션이나 배열에 값들을 넣기.
 - <ref>, <value> 태그를 이용해 값 설정
 - <ref bean="bean_id"/> : bean 객체 list에 추가
 - <value [type="type"]>값</value> : 문자열(String), Primitive 값 list에 추가

```
public void setMyList(List list){...}
```

```
<bean id="otherbean" class="vo.OtherBean"/>
<bean id="myBean" class="vo.MyVO">
  <property name="myList">
    <list>
      <value>10</value> ->String으로 저장됨
      <value type="java.lang.Integer">20</value> ->Integer
                                                           로 저장됨
      <ref bean="otherbean"/>
    </list>
  </property>
</bean>
```



Collection 객체 주입하기 (3/5)

- <map>
 - Map 계열의 Collection에 객체들을 넣기
 - 속성 : key-type, value-type : key와 value의 타입을 고정시킬 경우 사용
 - <entry>를 이용해 key-value를 map에 등록
 - 속성
 - key, key-ref : key 설정
 - value, value-ref : 값 설정

```
public void setMyMap(Map map){...}
```

```
<bean id="otherbean" class="vo.OtherBean"/>
<bean id="myBean" class="vo.MyVO">
  <property name="myMap">
    <map>
      <entry key="id" value="abc"/>
      <entry key="other" value-ref="otherbean"/>
    </map>
  </property>
</bean>
```


Collection 객체 주입하기 (4/5)

- `<props>`
 - `java.util.Properties` 값(문자열)을 넣기
 - `<prop>`를 이용해 key-value를 properties에 등록
 - 속성
 - key : key값 설정
 - 값은 태그 사이에 넣는다. : `<prop key="id">abcde</prop>`

```
public void setJdbcProperty (Properties props){...}
```

```
<bean id="myDAO" class="vo.DAO">  
  <property name="jdbcProperty">  
    <props>  
      <prop key="driver">JDBC Driver</prop>  
      <prop key="url">jdbc:url://127.0.0.1/mydb</prop>  
      <prop key="user">dbUser</prop>  
      <prop key="pwd">dbPassword</prop>  
    </props>  
  </property>  
</bean>
```

Collection 객체 주입하기 (5/5)

- <set>
 - java.util.Set에 객체를 넣기
 - 속성 : value-type : value 타입 설정
 - <value>, <ref>를 이용해 값을 넣는다.

```
public void setMySet(Set props){...}
```

```
<bean id="otherbean" class="vo.OtherBean"/>
<bean id="myBean" class="vo.Bean">
  <property name="mySet">
    <set>
      <value>10</value>
      <value>20</value>
      <ref bean="otherbean"/>
    </set>
  </property>
</bean>
```

Bean 객체의 생성 단위 (1/2)

- BeanFactory를 통해 Bean을 요청시 객체생성의 범위(단위)를 설정
- <bean> 의 scope 속성을 이용해 설정
 - scope의 값

값	
singleton	컨테이너는 하나의 빈 객체만 생성한다. - default
prototype	빈을 요청할 때 마다 생성한다.
request	Http 요청마다 빈 객체 생성
session	HttpSession 마다 빈 객체 생성
global-session	글로벌 http세션에 대해 빈 객체 생성- 포틀릿 관련

- request, session은 WebApplicationContext에서만 적용 가능

빈(bean) 생성 제어 (2/2)

- 빈(bean) 범위 지정
 - singleton과 prototype
 - `<bean id="dao" class="dao.OracleDAO" scope="prototype"/>`
 - prototype은 Spring 어플리케이션 컨텍스트에서 `getBean`으로 빈(bean)을 사용시마다 새로운 인스턴스를 생성함.
 - singleton은 Spring 어플리케이션 컨텍스트에서 `getBean` 으로 빈(bean)을 사용시 동일한 인스턴스를 생성함.

빈(bean) 생성 제어 (3/3)

- Factory 메소드로부터 빈(bean) 생성

```
public class OracleDAO{  
    private OracleDAO() {}  
    private static OracleDAO instance;  
    public static OracleDAO getInstance(){  
        if(instance==null)  
            instance = new OracleDAO();  
        return instance;  
    }  
}
```

Singleton 클래스는 static factory 메소드를 통해서 인스턴스 생성이 가능하면 단 하나의 인스턴스만을 생성함.

```
<bean id="dao" class="OracleDAO"  
      factory-method="getInstance"/>
```

* 주 : `getBean()`으로 호출 시 `private` 생성자도 호출 하여 객체를 생성한다.
그러므로 위의 상황에서 `factory` 메소드로만 호출 해야 객체를 얻을 수 있는 것은 아니다.

Spring AOP

Spring AOP 개요 (1/2)

- 핵심 관심 사항(core concern) 과 공통 관심 사항 (cross-cutting concern)
- 기존 OOP 에서는 공통관심사항을 여러 모듈에서 적용하는데 중복된 코드를 양산과 같은 한계가 존재 - 이를 극복하기 위해 AOP 가 등장
- Aspect Oriented Programming은 문제를 해결하기 위한 핵심 관심 사항과 전체에 적용되는 공통 관심 사항을 기준으로 프로그래밍함으로써 공통 모듈을 손쉽게 적용할 수 있게 해준다.

Spring AOP 개요 (2/2)

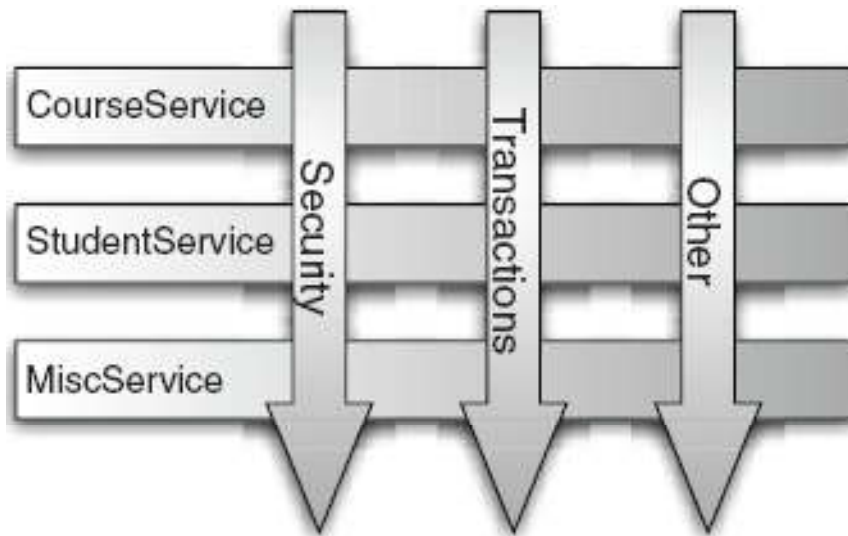


Figure 4.1
Aspects modularize cross-cutting concerns, applying logic that spans multiple application objects.

핵심관심사항 : CourseService, StudentService, MiscService

공통관심사항 : Security, Transactions, Other

- 핵심관심사항에 공통관심사항을 어떻게 적용시킬 것인가...

Spring AOP 용어

- Aspect - 여러 객체에서 공통으로 적용되는 공통관심 사항(ex:트랜잭션, 로깅, 보안)
- JoinPoint - Aspect가 적용 될 수 있는 지점(ex:메소드, 필드)
- Pointcut - 공통관심사항이 적용될 Joinpoint
- Advice - 어느 시점(ex: 메소드 수행 전/후, 예외발생 후 등)에 어떤 공통 관심기능(Asspect)을 적용할지 정의한 것.
- Weaving- 어떤 Advice를 어떤 Pointcut(핵심사항)에 적용시킬 것인지에 대한 설정 (Advisor)



Spring에서 AOP 구현 방법

- AOP 구현의 세가지 방법
 - POJO Class를 이용한 AOP구현
 - 스프링 API를 이용한 AOP구현
 - 어노테이션(Annotation) 을 이용한 AOP 구현

POJO 기반 AOP구현

- XML 스키마 확장기법을 통해 설정파일을 작성한다.
- POJO 기반 Advice 클래스 작성



POJO 기반 AOP구현 - 설정파일 작성 (1/5)

- XML 스키마를 이용한 AOP 설정
 - aop 네임스페이스와 XML 스키마 추가

```
<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:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

</beans>
```



POJO 기반 AOP구현 - 설정파일 작성 (2/5)

- XML 스키마를 이용한 AOP 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:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="writelog" class="org.kosta.spring.LogAspect"/>

  <aop:config>
    <aop:pointcut id="publicmethod" expression="execution(public * org.kosta.spring..*(..))"/>
    <aop:aspect id="loggingAspect" ref="writelog">
      <aop:around pointcut-ref="publicmethod" method="logging"/>
    </aop:aspect>
  </aop:config>

  <bean id="targetclass" class="org.kosta.spring.TargetClass"/>

</beans>
```



POJO 기반 AOP구현 - 설정파일 작성 (3/5)

- AOP 설정 태그

1. <aop:config> : aop설정의 root 태그. - weaving들의 묶음.
2. <aop:aspect> : Aspect 설정 - 하나의 weaving에 대한 설정
3. <aop:pointcut> : Pointcut 설정
4. Advice 설정태그들
 - A. <aop:before> - 메소드 실행 전 실행될 Advice
 - B. <aop:after-returning> - 메소드 정상 실행 후 실행될 Advice
 - C. <aop:after-throwing> - 메소드에서 예외 발생시 실행될 Advice
 - D. <aop:after> - 메소드 정상 또는 예외 발생 상관없이 실행될 Advice - finally
 - E. <aop:around> - 모든 시점에서 적용시킬 수 있는 Advice 구현



POJO 기반 AOP구현 - <aop:aspect> (4/5)

- 한 개의 Aspect (공통 관심기능)을 설정
- ref 속성을 통해 공통기능을 가지고 있는 bean을 연결한다.
- id는 이 태그의 식별자를 설정
- 자식 태그로 <aop:pointcut> advice관련 태그가 올 수 있다.

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="writelog">
    <aop:pointcut id="publicmethod" expression="execution(public * public * org.myspring..*. * (..))"/>
    <aop:around pointcut-ref="publicmethod" method="logging"/>
  </aop:aspect>
</aop:config>
```



POJO 기반 AOP구현 - <aop:pointcut> (5/5)

- Pointcut(공통기능이 적용될 곳)을 지정하는 태그
 - <aop:config>나 <aop:aspect>의 자식 태그
 - <aop:config> 전역적으로 사용
 - <aop:aspect> 내부에서 사용
 - AspectJ 표현식을 통해 pointcut 지정
 - 속성 :
 - id : 식별자로 advice 태그에서 사용됨
 - expression : pointcut 지정
- <aop:pointcut id="publicmethod" expression="execution(public * org.myspring..*.*(..))"/>

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="writelog">
    <aop:pointcut id="publicmethod" expression="execution(public * org.myspring..*.*(..))"/>
    <aop:around pointcut-ref="publicmethod" method="logging"/>
  </aop:aspect>
</aop:config>
```


POJO 기반 AOP구현 - AspectJ 표현식 (1/3)

- AspectJ에서 지원하는 패턴 표현식
- 스프링은 메서드 호출관련 명시자만 지원

명시자(제한자패턴? 리턴타입패턴 패키지패턴?이름패턴(파라미터패턴))
-?는 생략가능

- 명시자
 - **execution** : 실행시킬 메소드 패턴을 직접 입력하는 경우
 - **within** : 메소드가 아닌 특정 타입에 속하는 메서드들을 설정할 경우
 - **bean** : 2.5버전에 추가됨. 설정파일에 지정된 빈의 이름(name 속성)을 이용해 **pointcut**설정

POJO 기반 AOP구현 - AspectJ 표현식 (2/3)

- 표현

명시자(수식어패턴? 리턴타입패턴 패키지패턴? 클래스이름패턴.메소드이름패턴(파라미터패턴))
-?는 생략가능
예) `execution(public * abc.def..*Service.set*(..))`

- 수식어 패턴에는 `public`, `protected` 또는 생략한다.

- `*` : 1개의 모든 값을 표현

- `argument`에서 쓰인 경우 : 1개의 `argument`
 - `package`에 쓰인 경우 : 1개의 하위 `package`

- `..` : 0개 이상

- `argument`에서 쓰인 경우 : 0개 이상의 `argument`
 - `package`에 쓰인 경우 : 0개의 이상의 하위 `package`

- 위 예 설명

적용 하려는 메소드들의 패턴은 `public` 제한자를 가지며 리턴 타입에는 모든 타입이 다 올 수 있다. 이름은 `abc.def` 패키지와 그 하위 패키지에 있는 모든 클래스 중 `Service`로 끝나는 클래스들 에서 `set`으로 시작하는 메소드이며 `argument`는 0개 이상 오며 타입은 상관 없다.



POJO 기반 AOP구현 - AspectJ 표현식 (3/3)

- 예

```
execution(* test.spring.*.*())  
execution(public * test.spring..*.*())  
execution(public * test.*.*.get*(*))  
execution(String test.spring.MemberService.registMember(..))  
execution(* test.spring..*Service.regist*(..))  
execution(public * test.spring..*Service.regist*(String, ..))  
  
within(test.spring.service.MemberService)  
within(test.spring..MemberService)  
within(test.spring.aop..*)  
  
bean(memberService)  
bean(*Service)
```



POJO 기반 AOP구현 - Advice 작성

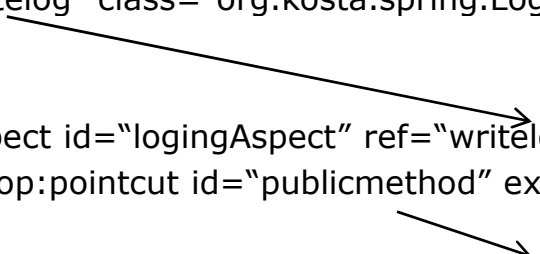
- POJO 기반 Aspect클래스 작성
 - 설정 파일의 advice 관련 태그에 맞게 작성한다.
 - `<bean>`으로 등록 하며 `<aop:aspect>` 의 ref 속성으로 참조한다.
 - 공통 기능 메소드 : advice 관련 태그들의 method 속성의 값이 메소드의 이름이 된다.

POJO 기반 AOP구현 - Advice 정의 관련 태그

- 속성
 - pointcut-ref : <aop:pointcut>태그의 id명을 넣어 pointcut지정
 - pointcut : 직접 pointcut을 설정 한다.
 - method : Aspect bean에서 호출할 메소드명 지정

```
<bean id="writelog" class="org.kosta.spring.LogAspect"/>

<aop:config>
  <aop:aspect id="loggingAspect" ref="writelog">
    <aop:pointcut id="publicmethod" expression="execution(public * org.my.spring..*.*(..))"/>
    <aop:before pointcut-ref="publicmethod" method="logging"/>
  </aop:aspect>
</aop:config>
```





POJO 기반 AOP구현 - Aspect클래스 작성 (1/4)

- POJO 기반의 클래스로 작성한다.
 - 클래스 명이나 메서드 명에 대한 제한은 없다.
 - 메소드 구문은 호출되는 시점에 따라 달라 질 수 있다.
 - 메소드의 이름은 **advice** 태그(<aop:before/>)에서 **method** 속성의 값이 메소드 명이 된다.
- **before** 메소드
 - 대상 객체의 메소드가 실행되기 전에 실행됨
 - return type : void
 - argument : 없거나 JoinPoint 객체를 받는다.
 - ex)

```
public void beforeLogging(JoinPoint jp){  
    }  
}
```

POJO 기반 AOP구현 - Aspect클래스 작성 (2/4)

- After Returning Advice
 - 대상객체의 메소드 실행이 정상적으로 끝난 뒤 실행됨
 - return type : void
 - argument :
 - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용된다.
 - 대상 메소드에서 리턴되는 값을 argument로 받을 수 있다.

```
<aop:after-returning pointcut-ref="publicmethod" method="returnLogging" returning="retValue"/>
```

```
public void returnLogging(Object retValue){  
    //대상객체에서 리턴되는 값을 받을 수는 있지만 수정할 수는 없다.  
}
```

POJO 기반 AOP구현 - Aspect클래스 작성 (3/4)

- After Throwing Advice
 - 대상객체의 메소드 실행 중 예외가 발생한 경우 실행됨
 - return type : void
 - argument :
 - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용된다.
 - 대상메소드에서 전달되는 예외객체를 argument로 받을 수 있다.

```
<aop:after-throwing pointcut-ref="publicmethod" method="returnLogging" throwing="ex"/>
```

```
public void returnLogging(MyException ex){  
    //대상객체에서 리턴되는 값을 받을 수는 있지만 수정할 수는 없다.  
}
```




POJO 기반 AOP구현 - Aspect클래스 작성 (4/4)

- Around Advice

- 위의 네 가지 Advice를 다 구현 할 수 있는 Advice.
- return type : Object
- argument
 - org.aspectj.lang.ProceedingJoinPoint 를 반드시 첫 argument로 지정한 다.

```
<aop:around pointcut-ref="publicmethod" method="returnLogging" />

public Object returnLogging(ProceedingJoinPoint joinPoint) throws Throwable{
    //대상 객체의 메소드 호출 전 해야 할 전 처리 코드
    try{
        Object retValue = joinPoint.proceed(); //대상객체의 메소드 호출
        //대상 객체 처리 이후 해야 할 후처리 코드
        return retValue; //호출 한 곳으로 리턴 값 넘긴다. - 넘기기 전 수정 가능
    }catch(Throwable e){
        throw e; //예외 처리
    }
}
```

JoinPoint

- 대상객체에 대한 정보를 가지고 있는 객체로 Spring container로부터 받는다.
- org.aspectj.lang 패키지에 있음
- 받듯이 Aspect 메소드의 첫 argument로 와야한다.
- 메소드들

Object getTarget() : 대상객체를 리턴

Object[] getArgs() : 파라미터로 넘겨진 값들을 배열로 리턴. 넘어온 값이 없으면 빈 배열객체가 return 됨.

Signature getSignature () : 호출 되는 메소드의 정보

Signature : 호출 되는 메소드에 대한 정보를 가진 객체

String getName() : 메소드 명

String toLongString() : 메서드 전체 syntax를 리턴

String toShortString() : 메소드를 축약해서 return - 기본은 메소드 이름만 리턴

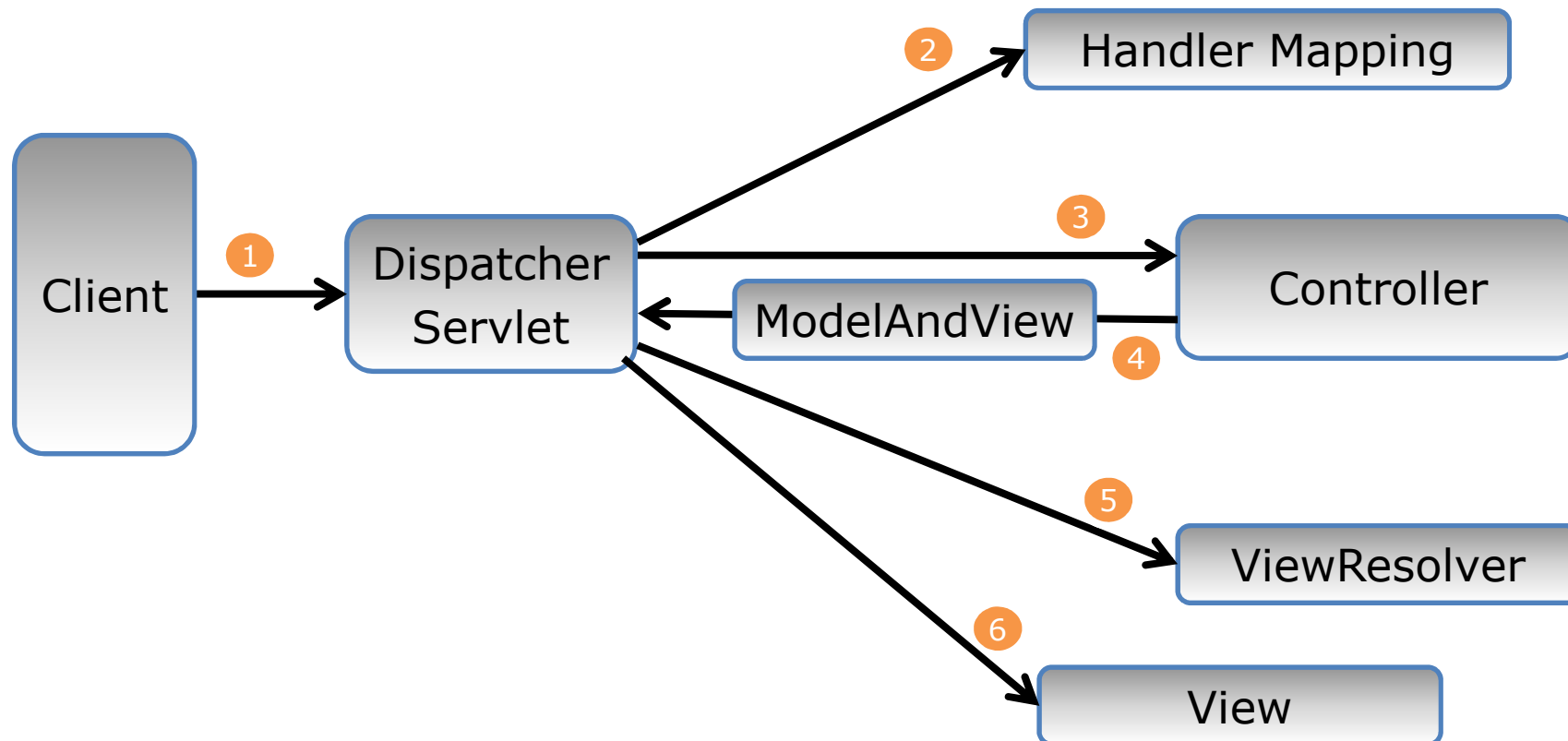
@Aspect 어노테이션을 이용한 AOP

- @Aspect 어노테이션을 이용하여 Aspect 클래스에 직접 Advice 및 Pointcut등을 직접 설정
- 설정파일에 <aop:aspectj-autoproxy/> 를 추가 필요
- Aspect class를 <bean>으로 등록
- 어노테이션(Annotation)
 - @Aspect : Aspect 클래스 선언
 - @Before("pointcut")
 - @AfterReturning(pointcut="", returning="")
 - @AfterThrowing(pointcut="", throwing="")
 - @After("pointcut")
 - @Around("pointcut")
- Around를 제외한 나머지 메소드들은 첫 argument로 JoinPoint를 가질 수 있다.
- Around 메소드는 argument로 ProceedingJoinPoint를 가질 수 있다.

Spring MVC

Spring MVC 흐름 (1/2)

- Spring MVC
 - MVC 패턴 기반 웹 개발 프레임워크





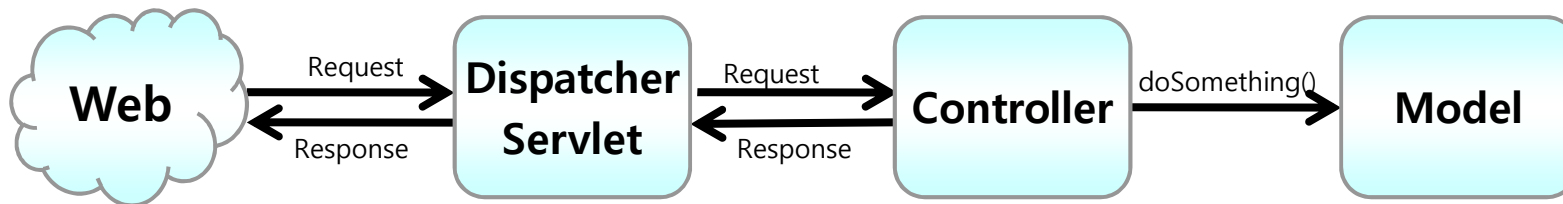
Spring MVC 흐름 (2/2)

- 요청 처리 순서
 - ① DispatcherServlet이 요청을 수신
 - 단일 front controller servlet
 - 요청을 수신하여 처리를 다른 컴포넌트에 위임
 - 어느 컨트롤러에 요청을 전송할지 결정
 - ② DispatcherServlet은 HandlerMapping에 어느 컨트롤러를 사용할 것인지 문의
 - URL과 매핑
 - ③ DispatcherServlet은 요청을 컨트롤러에게 전송하고 컨트롤러는 요청을 처리한 후 결과 리턴
 - 비즈니스 로직 수행 후 결과 정보(Model)가 생성되어 JSP와 같은 뷰에서 사용됨
 - ④ ModelAndView 오브젝트에 수행결과가 포함되어 DispatcherServlet에 리턴
 - ⑤ ModelAndView는 실제 JSP정보를 갖고 있지 않으며, ViewResolver가 논리적 이름을 실제 JSP이름으로 변환
 - ⑥ View는 결과정보를 사용하여 화면을 표현함.



Spring MVC 구현 Step

- Spring MVC를 이용한 어플리케이션 작성 스텝
 1. web.xml에 DispatcherServlet 등록 및 Spring 설정파일 등록
 2. 설정파일에 HandlerMapping 설정
 3. 컨트롤러 구현 및 Spring 설정파일에 등록
 4. 컨트롤러와 JSP의 연결 위해 View Resolver 설정
 5. JSP 코드 작성



DispatcherServlet 설정과 ApplicationContext (1/3)

- DispatcherServlet 설정
 - web.xml에 등록
 - 스프링 설정파일 : "<servlet-name>-servlet.xml" 이고 WEB-INF\아래 추가한다.
 - <url-pattern>은 DispatcherServlet이 처리하는 URL 매핑 패턴을 정의

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

- **Spring Container**는 설정파일의 내용을 읽어 **ApplicationContext** 객체를 생성한다.
- 설정 파일명 : dispatcher-servlet.xml – MVC 구성 요소 (HandlerMapping, Controller, ViewResolver, View) 설정과 bean, aop 설정들을 한다.

DispatcherServlet 설정과 ApplicationContext (2/3)

- Spring 설정파일 등록하기
 - <servlet>의 하위태그인 <init-param>에 contextConfigLocation 이름으로 등록
 - 경로는 Application Root부터 절대경로로 표시
 - 여러 개의 경우 , 또는 공백으로 구분

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/server-service.xml, dao-service.xml</param-value>
  </init-param>
</servlet>
```

DispatcherServlet 설정과 ApplicationContext (3/3)

- DispatcherServlet 여러 개 설정시 공통 Spring 설정파일 등록
 - 컨텍스트 설정 파일(스프링 설정파일)들을 로드하기 위해 리스너 (ContextLoaderListener) 설정
 - 설정파일 <context-param>으로 등록

```
<listener>  
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>
```

```
<context-param>  
  <param-name>contextConfigLocation</param-name>  
  <param-value>/WEB-INF/service-service.xml  
               /WEB-INF/dao-data.xml  
</param-value>  
</context-param>
```



HandlerMapping

- Client요청과 Controller를 연결을 설정
- 다양한 HandlerMapping 클래스를 Springframework가 제공 하며 Spring 설정파일 에 <bean name="HandlerMapping">으로 등록하여 설정한다.
- BeanNameUrlHandlerMapping
 - bean의 이름과 url을 mapping
- SimpleUrlHandlerMapping
 - url pattern들을 properties로 등록해 처리

HandlerMapping

BeanNameUrlHandlerMapping 설정

```
<bean id="HandlerMapping"  
      class="org.springframework.web.servlet.Handler.BeanNameUrlHandlerMapping"/>  
<bean name="/hello.do" class="controller.HelloController"/>  
<bean name="/welcome.do" class="controller.WelcomeController"/>
```

SimpleUrlHandlerMapping 설정

```
<bean id="HandlerMapping"  
      class="org.springframework.web.servlet.Handler.SimpleUrlHandlerMapping">  
<property name="mappings">  
  <props>  
    <prop key="/register.do">registerContoller</prop>  
    <prop key="/delete.do">deleteController</prop>  
  </props>  
</property>  
</bean>  
<!--컨트롤러 bean으로 등록-->  
<bean name="registerContoller" ...../>  
<bean name="deleteController" ..../>
```



Controller 작성

- Controller 종류
 - Controller (interface)
 - AbstractController
 - AbstractComandController
 - MultiActionController
- 위의 interface/class를 상속하여 Controller 작성한다.



AbstractController (1/2)

- 가장 기본이 되는 Controller
- 작성
 - AbstractController 상속한다.
 - public ModelAndView HandlerrequestInternal
(HttpServletRequest request,
HttpServletResponse response)
throws Exception
오버라이딩 하여 코드 구현
 - ModelAndView에 view가 사용할 객체와 view에 대한
id값을 넣어 생성 후 return

AbstractController (2/2)

```
public class HelloworldAbstractController extends AbstractController{  
    protected ModelAndView HandlerrequestInternal(  
        HttpServletRequest request,  
        HttpServletResponse response)  
        throws Exception {  
        //Model 호출 - Business Logic 처리  
        //ModelAndView를 통해 view로 수행 넘김  
        return new ModelAndView("hello","message",helloVO);  
    }  
}
```

AbstractCommandController (1/2)

- HttpServletRequest로 받아온 form data(parameter)를 동적으로 특정 데이터 객체(VO-Command지칭)로 바인드 하는 controller.
- Struts의 ActionForm의 기능과 유사
 - Spring의 경우 Struts 처럼 데이터객체가 framework-specific 인터페이스를 implement하지 않아도 된다.
 - 생성된 Command객체는 requestScope에 속성으로 binding된다.
- 작성
 - 데이터 객체는 java beans 규약에 맞게 생성 (setter). parameter의 이름과 데이터객체의 property이름이 동일해야 한다.
 - AbstractCommandController를 상속하여 클래스 작성
 - Data객체에 설정을 위해아래 두 메소드 생성자에서 호출한다.
 - setCommandClass(Class commandClass) : Command클래스
 - setCommandName(String commandName) : requestScope에 Command객체를 넣을 때 사용할 이름, 생략가능
 - protected ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object command, BindException be) throws Exception{ }

메소드를 overriding하여 구현



AbstractCommandController (2/2)

Command class(VO)

```
public class PersonCommand{  
    private String id;  
    private String name;  
    ....  
    setter/getter  
}
```

Controller (VO)

```
public class RegisterController extends AbstractCommandController{  
    public RegisterController(){  
        setCommandClass(PersonCommand.class);  
        setCommandName("personCommand");  
    }  
    protected ModelAndView handle(HttpServletRequest req, HttpServletResponse res,  
                                Object command, BindException error){  
        PersonCommand cmd = (PersonCommand)command;  
        //Business Logic 처리  
        ModelAndView mv = new ModelAndView("show_content");  
        return mv;  
    }  
}
```

MultiActionController (1/3)

- 하나의 Controller에서 여러 개의 요청을 수행할 경우 사용
 - struts의 DispatcherAction과 동일한 역할
- 연관된 request를 하나의 controller로 묶을 경우 사용.
- 작성
 - MultiActionController 상속
 - client의 요청을 처리할 메소드 구현

```
public [ModelAndView|Map|void] 메소드이름(  
    HttpServletRequest req, HttpServletResponse res  
    [HttpSession|Command]) [throws Exception]{}
```

 - return type : ModelAndView, Map, void 중 하나
 - argument :
 - 1번 - HttpServletRequest, 2번 - HttpServletResponse
 - 3번 - 선택적이며 HttpSession 또는 Command



MultiActionController (2/3)

– MethodNameResolver 등록

- 역할 : 어떤 메소드가 클라이언트의 요청을 처리할 것인지 결정
- Spring 설정파일에 <bean>으로 등록
- controller에서는 property로 주입 받는다.
- 종류
 - ParameterMethodNameResolver : parameter로 메소드 이름 전송
 - InternalPathMethodNameResolver : url 마지막 경로 메소드이름으로 사용
 - PropertiesMethodNameResolver : URL과 메소드 이름 mapping을 property로 설정

MultiActionController (3/3)

Controller class

```
public class MemberController extends MultiActionController{  
  
    public ModelAndView registerMember(HttpServletRequest request,  
                                         HttpServletResponse response) throws Exception{  
        //Business Logic 구현  
        ModelAndView mv = new ModelAndView();  
        mv.setViewName("register_ok");  
        return new ModelAndView();  
    }  
}
```

```
<bean id="methodNameResolver"  
      class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">  
    <property name="paramName" value="mode"> </property>  
</bean>  
  
<bean id="memberController" class="controller.multiaction.MemberController">  
    <property name="methodNameResolver">  
        <ref bean="methodNameResolver"/>  
    </property>  
</bean>
```

호출 : <http://ip:port/applName/member?mode=registerMember>

ModelAndView (1/2)

- Controller 처리 결과 후 응답할 view와 veiw에 전달할 값을 저장.
- 생성자
 - ModelAndView(String viewName) : 응답할 view설정
 - ModelAndView(String viewName, Map values) : 응답할 view와 view로 전달할 값들을 저장 한 Map 객체
 - ModelAndView(String viewName, String name, Object value) : 응답할 view이름, view로 넘길 객체의 name-value
- 주요 메소드
 - setViewName(String view) : 응답할 view이름을 설정
 - addObject(String name, Object value) : view에 전달할 값을 설정
 - requestScope에 설정됨
 - addAllObjects(Map values) : view에 전달할 값을 Map에 name-value로 저장하여 한번에 설정 - requestScope에 설정됨
- Redirect 방식 전송
 - view이름에 redirect: 접두어 붙인다.
ex) mv.setViewName("redirect:/welcome.html");



ModelAndView (2/2)

```
protected ModelAndView HandlerrequestInternal(HttpServletRequest req,  
                                              HttpServletResponse res) throws Exception{  
    //Business Logic 처리  
    ModelAndView mv = new ModelAndView();  
    mv.setViewName("/hello.jsp");  
    mv.addObject("greeting", "hello world");  
    return mv;  
}
```

ViewResolver

- Controller가 넘긴 view이름을 통해 알맞은 view를 찾는 역할
 1. Controller는 ModelAndView 객체에 응답할 view이름을 넣어 return.
 2. DispatcherServlet은 ViewResolver에게 응답할 view를 요청한다.
 3. ViewResolver는 View 이름을 이용해 알맞는 view 객체를 찾는다.
- 다양한 ViewResolver를 SpringFramework는 제공한다.
 - InternalResourceViewResolver : 뷰의 이름을 JSP, HTML등과 연동한 View를 return
- ViewResolver – Spring 설정파일에 등록한다.

ViewResolver

Spring 설정파일에 설정

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

Controller

```
ModelAndView mv = new ModelAndView();  
mv.setViewName("hello");
```

위의 경우
/WEB-INF/jsp/hello.jsp 를 찾는다.

Spring 과 DB



Spring의 데이터 접근 방식 (1/4)

- Spring에서의 JDBC 프로그래밍
 - 데이터 접근 템플릿 제공 - JDBC 프로그래밍을 쉽게 할 수 있는 템플릿
 - DAO 지원 클래스 제공 - Template을 쉽게 사용할 수 있도록 지원하는 class
- 데이터 접근 템플릿
 - 템플릿 메소드(template method) 패턴
 - 템플릿 메소드는 프로세스에 대한 틀(skeleton)을 정의함.
 - 전반적인 프로세스(의 흐름)는 고정되어 있고, 변하지 않음.
 - 특정 시점의 프로세스는 특정 상황에 따라 세부적인 구현 내용이 달라짐.
 - Spring의 데이터 접근은 템플릿 메소드 패턴을 적용
 - 데이터 접근 단계에서 연결을 얻고, 자원을 해제하는 부분은 고정
 - 데이터 접근 방법은 각각 다름.
 - 두가지 클래스
 - 템플릿(template)
 - 프로세스의 고정된 부분을 담당

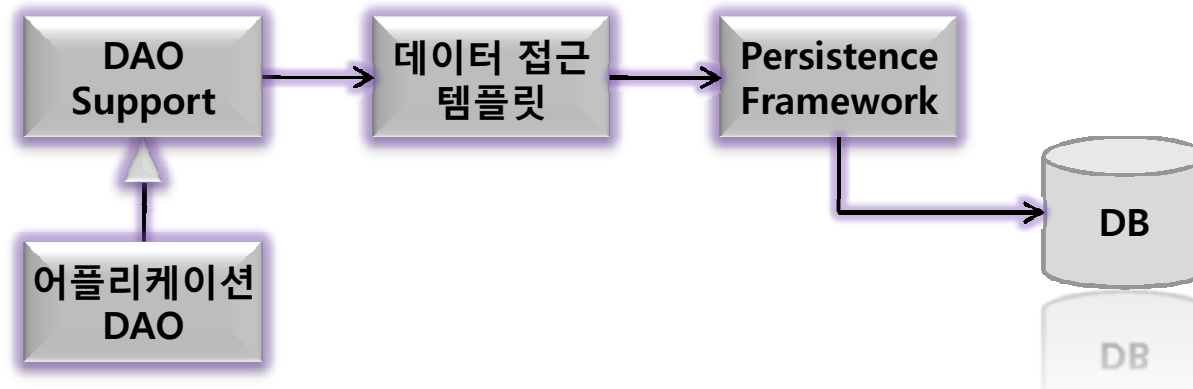
Spring의 데이터 접근 방식 (2/4)

- 데이터 접근 템플릿화
 - Spring의 템플릿 클래스

템플릿 클래스 (org.springframework.*)	사용 목적
<code>jdbc.core.JdbcTemplate</code>	JDBC 연결
<code>jdbc.core.namedparam.NamedParameterJdbcTemplate</code>	명명된(named) 파라미터에 대한 지원과 함께 사용하는 JDBC 연결
<code>jdbc.core.simple.SimpleJdbcTemplate</code>	Java 5 구조와 함께 단순화된 JDBC 연결
<code>org.ibatis.SqlMapClientTemplate</code>	iBATIS SqlMap 클라이언트
<code>org.hibernate.HibernateTemplate</code>	Hibernate 2.x 세션
<code>org.hibernate3.HibernateTemplate</code>	Hibernate 3.x 세션
<code>orm.jdo.JdoTemplate</code>	Java Data Object 구현
<code>orm.jpa.JpaTemplate</code>	Java Persistence API 엔티티 관리자
<code>orm.toplink.TopLinkTemplate</code>	오라클의 TopLink
<code>jca.cci.core.CciTemplate</code>	JCA CCI 연결

Spring의 데이터 접근 방식 (3/4)

- DAO 지원 클래스 사용
 - Spring의 DAO 지원 클래스



- 어플리케이션에 대한 DAO 구현시 Spring의 DAO 지원 클래스를 상속받음.
 - 내부의 데이터 접근 템플릿에 바로 접근하려면 템플릿 조회 메소드를 호출
 - JdbcDaoSupport 클래스를 상속받고, getJdbcTemplate()을 호출해서 작업 수행
- 저장 플랫폼을 접근하려면 DB와 통신하기 위해 사용되는 클래스를 접근함.
 - JdbcDaoSupport의 getConnection() 메소드

Spring의 데이터 접근 방식 (4/4)

- DAO 지원 클래스 사용
 - Spring의 DAO 지원 클래스

DAO 지원 클래스 (org.springframework.*)	사용 목적
<code>jdbc.core.support.JdbcDaoSupport</code>	JDBC 연결
<code>jdbc.core.namedparam.NamedParameterJdbcDaoSupport</code>	명명된(named) 파라미터에 대한 지원과 함께 사용하는 JDBC 연결
<code>jdbc.core.simple.SimpleJdbcDaoSupport</code>	Java 5 구조와 함께 단순화된 JDBC 연결
<code>org.ibatis.support.SqlMapClientDaoSupport</code>	iBATIS SqlMap 클라이언트
<code>org.hibernate.support.HibernateDaoSupport</code>	Hibernate 2.x 세션
<code>org.hibernate3.support.HibernateDaoSupport</code>	Hibernate 3.x 세션
<code>orm.jdo.support.JdoDaoSupport</code>	Java Data Object 구현
<code>orm.jpa.support.JpaDaoSupport</code>	Java Persistence API 엔티티 관리자
<code>orm.toplink.support.TopLinkDaoSupport</code>	오라클의 TopLink
<code>jca.cci.support.CciDaoSupport</code>	JCA CCI 연결

데이터 소스 설정 (1/4)-JNDI 데이터 소스 사용(1)

- JNDI 데이터 소스 사용
 - Spring 어플리케이션이 JEE 어플리케이션 서버 내에 배포될 때 사용
 - 장점
 - 어플리케이션 외부에 전적으로 관리를 맡김.
 - 어플리케이션 서버는 우수한 성능으로 데이터 소스를 관리하고 시스템 관리자는 데이터 소스를 제어할 수 있음.
 - JndiObjectFactoryBean 이용

```
<bean id="dataSource"  
      class="org.springframework.jndi.JndiObjectFactoryBean" scope="singleton">  
  <property name="jndiName" value="/jdbc/oracle"/>  
  <property name="resourceRef" value="true"/>  
</bean>
```

- jndiName 속성은 JNDI에서 자원의 이름을 지정
- 어플리케이션이 자바 어플리케이션 서버 내에서 구동하면 resourceRef 값을 true 로 세팅
 - 원래의 jndiName 값에 'java:comp/env' 가 붙음.
 - 'java:/comp/env/jdbc/oracle'

데이터 소스 설정 (2/4)-JNDI 데이터 소스 사용(2)

- JNDI 데이터 소스 사용 - 2
 - Spring 2.0 이상에서의 JNDI 데이터 소스
 - Spring 설정파일에 **jee** 네임스페이스 등록하여 처리

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
  <jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/oracle"
    resource-ref="true"/>
</beans>
```

데이터 소스 설정 (3/4)-DBCP 사용

- DBCP 사용
 - Spring이 기본으로 제공하는 DBCP 사용 할 수 있다.
 - Jakarta Commons Database Connection Pools (DBCP)
 - <http://commons.apache.org/dbcp/>
 - BasicDataSource 사용
 - DataSource를 bean으로 등록한다.

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521/XE"/>
  <property name="username" value="hr"/>
  <property name="password" value="hr"/>
  <property name="initialSize" value="5"/>
  <property name="maxActive" value="10"/>
</bean>
```

- 설정항목
 - <http://commons.apache.org/dbcp/configuration.html> 참고

데이터 소스 설정 (4/4)-JDBC DataSource 사용

- JDBC 드라이버 기반한 데이터 소스
 - DriverManagerDataSource
 - 요청된 연결에 대해서 매번 새로운 연결을 제공.
 - DBCP의 BasicDataSource와 달리 DriverManagerDataSource에 의해 제공되는 연결은 풀링되지 않음.
 - SingleConnectionDataSource
 - 요청된 연결에 대해서 매번 동일한 연결을 제공.
 - 정확하게 풀링된 데이터 소스는 아니지만, 단 하나의 연결을 풀링하는 데이터 소스라고 보면 됨.

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521/XE"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>
```

- 멀티쓰레드 환경에서 사용이 제약됨.

Spring에서 JDBC 사용 예 (1/2)

- 기본 JDBC 코드 - DataSource를 주입 받아 JDBC 코딩

```
public class MemberDAO{
    private DataSource dataSource;
    public void insertMember(MemberVO mvo) {
        Connection conn = null;
        PreparedStatement pstmt = null;
        String sql = "insert ...";
        try {
            conn = dataSource.getConnection();
            pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, mvo.getId());
            pstmt.setString(2, mvo.getPassword());
            pstmt.setString(3, mvo.getName());
            pstmt.setString(4, mvo.getAddress());
            pstmt.executeUpdate();
        }
```

```
    } catch (SQLException e) {
        //오류처리
    } finally {
        try {
            if (stmt != null) pstmt.close();
            if (conn != null) conn.close();
        } catch (SQLException e) {}
    }
}

//주입받는다.
public void setDataSource(DataSource ds) {
    this.dataSource = ds;
}
}
```

Spring에서 JDBC 사용 (2/2)

- JDBC 코드
 - 저장 – Spring 설정 파일

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:XE"/>
  <property name="username" value="hr"/>
  <property name="password" value="hr"/>
</bean>

<bean id="memberDAO" class="dao.MemberDAO">
  <property name="dataSource" ref="dataSource"/>
</bean>
```



JDBC Template을 이용(1/4)

- JDBC 템플릿 사용
 - 세가지 템플릿 클래스
 - JdbcTemplate
 - Spring의 JDBC 템플릿 중에 가장 기본 클래스로 JDBC를 통한 데이터베이스에 대한 단순한 연결과 단순한 인덱싱 파라미터 쿼리를 제공
 - NamedParameterJdbcTemplate
 - 인덱싱된 파라미터가 아닌 SQL에 명명된(named) 파라미터로 값을 세팅하는 쿼리를 수행하게 함.
 - SimpleJdbcTemplate
 - Autoboxing 이나, generics, 가변(variable) 파라미터 리스트와 같은 Java 5 기능을 사용하여 JDBC 템플릿 사용방법을 단순화시켜서 제공

JDBC Template을 이용(2/4)

- JDBC 템플릿 사용 – update 계열
 - JdbcTemplate를 사용한 데이터 접근

```
public interface DAO {  
    void insert(MemberVO mvo);  
}
```

```
public class MemberDAO implements DAO {  
    private JdbcTemplate jdbcTemplate;  
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;  
    }  
    ...  
    @Override  
    public void insert(MemberVO mvo) {  
        jdbcTemplate.update(MEMBER_INSERT,  
            new Object[]{mvo.getId(), mvo.getPassword(),  
                mvo.getName(), mvo.getAddress()});  
    }  
}
```

Spring 설정
파일에서 주입

JdbcTemplate을
사용한 실행 구문

쿼리의 인덱싱된 파라미터 값

실행할 SQL 구문



Spring에서 JDBC 사용 (5/6)

- JDBC 템플릿 사용
 - JdbcTemplate를 사용한 데이터 접근

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

- DB 연결, statement 생성, SQL 실행 등은 JdbcTemplate 이 대신 수행함.
- SQLException은 JdbcTemplate에서 catch 하여 throw 함.
 - 일반적인 SQLException을 세부적인 데이터 접근 예외로 변경하여 throw 함.
 - Spring의 템플릿 메소드들이 제공하는 예외는 Runtime계열의 메소드 이므로 따로 예외를 잡을 필요 없다.

Spring에서 JDBC 사용 (6/6)

- JDBC 템플릿 사용 - select
 - JdbcTemplate를 사용한 데이터 접근

```
public Motorist selectMemberById(String id) {  
    List matches = jdbcTemplate.query(MEMBER_SELECT_BY_ID,  
        new Object[] {id},  
        new RowMapper() {  
            @Override  
            public Object mapRow(ResultSet rs, int rowNum) throws SQLException {  
                return new Motorist(rs.getInt(1), rs.getString(2), rs.getString(3),  
                    rs.getString(4), rs.getString(5));  
            }  
        });  
    return matches.size() > 0 ? (Motorist)matches.get(0) : null;  
}
```

실행할 SQL 구문

쿼리의 인덱싱된 파라미터 값

ResultSet에서 값을 추출하여 도메인 객체를 만들어 냄.

쿼리 실행으로부터 결과인 모든 행(row)에 대해서 JdbcTemplate은 RowMapper의 mapRow() 메소드를 호출

RowMapper 내에서 VO(DTO)객체를 생성하고 ResultSet에서 값을 세팅함.

Struts와 Spring 통합

통합의 두가지 방법

- contextLoaderPlugin를 사용하여 Spring이 Action들을 bean들로 관리하도록 설정하고 Action들의 의존성을 Spring 컨텍스트 파일에 세팅하는 방법
- Spring의 ActionSupport 클래스를 상속해서 *getWebApplicationContext()* 메소드를 사용하여 Spring 관리되는 bean들을 명시적으로 가로채는 방법

ContextLoaderPlugin 이용한 연동 (1/3)

- ContextLoaderPlugin : Struts ActionServlet을 위해 Spring 설정 파일(context file)을 읽도록 하는 Plugin.
- 읽어 들인 (loading) 설정파일은 ContextLoaderListener에 의해 로드된 WebApplicationContext를 부모 클래스로 참조
- Spring 설정 파일 명 : [ActionServlet 이름] -servlet.xml
 - web.xml 설정에서 ActionServlet의 name 이 action인 경우 action-servlet.xml 이 Spring 설정 파일 명
- 작성
 - *struts-config.xml*에 plug-in 설정

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">  
  <set-property property="contextConfigLocation"  
    value="/WEB-INF/action-servlet.xml.xml,/WEB-INF/applicationContext.xml"/>  
</plug-in>
```



ContextLoaderPlugin 이용한 연동 (2/3)

- Spring 설정을 사용할 Action들은 <action-mapping>의 type 속성에 DelegatingActionProxy 클래스를 설정.
- DelegatingActionProxy 역할 : Struts 액션에 대한 요청이 들어오면 Spring 설정파일에 등록 된 Action을 실행

```
<action path="/searchMember"  
        type="org.springframework.web.struts.DelegatingActionProxy"  
        name="memberForm">  
    <forward name="success" path="/WEB-INF/pages/detail.jsp"/>  
    <forward name="failure" path="/WEB-INF/pages/search.jsp"/>  
</action>
```



ContextLoaderPlugin 이용한 연동 (3/3)

- Struts의 Action class를 Spring 설정 파일에 <bean>으로 등록

```
<bean name="/searchMember" class="member.actions.SearchMemberAction">  
  <property name="memberService">  
    <ref bean="memberService"/>  
  </property>  
</bean>
```



ActionSupport을 이용한 연동 (1/2)

- ActionSupport : Spring에서 Struts의 Action을 지원하기 위해 제공 되는 Action Class
 - ActionSupport를 상속받은 struts Action은 WebApplicationContext 객체를 사용할 수 있다.
 - 종류
 - ActionSupport
 - DispatchActionSupport
- Struts에서 Action class들을 만들때 Action이 아닌 ActionSupport를 상속 받도록 한다.
- ActionSupport 제공 메소드
 - getWebApplicationContext() – WebApplicationContext return함



ActionSupport을 이용한 연동 (2/2)

```
public class MemberSearchAction extends ActionSupport {  
    public ActionForward execute(ActionMapping mapping,  
                                ActionForm form,  
                                HttpServletRequest request,  
                                HttpServletResponse response)  
        throws Exception {  
        WebApplicationContext ctx = getWebApplicationContext();  
        MemberService ms = (MemberService) ctx.getBean("memberService");  
        //Business Logic 처리  
        return mapping.findForward("success");  
    }  
}
```

iBATIS

개요

- JDBC 코드의 패턴
 - Connection -> Statement -> 쿼리전송->연결 close
 - 모든 JDBC 코드는 위의 패턴을 가진다.
 - 이 패턴을 캡슐화 하여 JDBC 코드를 간편하게 사용할 수 있도록 Framework화 가능
- iBATIS 란
 - SQL 실행 결과를 자바빈즈 혹은 Map 객체에 매핑해 주는 Persistence 솔루션으로 SQL을 소스코드가 아닌 XML로 따로 분리해 관리하도록 지원
- 장점
 - SQL 문장과 프로그래밍 코드의 분리
 - JDBC 라이브러리를 통해 매개변수를 전달하고 결과를 추출하는 일을 간단히 처리가능
 - 자주 쓰이는 데이터를 변경되지 않는 동안에 임시 보관 (Cache) 가능
 - 트랜잭션처리 제공



iBATIS 설치

- <http://ibatis.apache.org/java.cgi> 에서 받는다.
 - 압축을 풀면 lib 디렉토리에 api가 있다.
 - API를 Application에 설치
 - Standalone Application
 - 어플리케이션 start 스크립트에 클래스 패스 정의
 - `java -cp ibatis-2.3.0.677.jar:... MyMainClass`
 - Web Application
 - WEB-INF/lib에 추가
- 주) 다른 경로에 추가하면 설정파일을 찾지 못하는 문제가 발생할 수 있다.

iBATIS Quick Start (1/3)

- iBATIS SqlMap Config(sqlmap-config.xml) 작성 – iBATIS framework에 대한 설정

```
<?xml version="1.0" encoding="EUC-KR"?>
<!DOCTYPE sqlMapConfig
  PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
  "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
  <transactionManager type="JDBC">
    <dataSource type="SIMPLE">
      <property name="JDBC.Driver"
        value="oracle.jdbc.driver.OracleDriver" />
      <property name="JDBC.ConnectionURL"
        value="jdbc:oracle:thin:@127.0.0.1:1521:XE" />
      <property name="JDBC.Username"
        value="hr" />
      <property name="JDBC.Password"
        value="hr" />
    </dataSource>
  </transactionManager>

  <sqlMap resource="User.xml" />
</sqlMapConfig>
```

iBATIS Quick Start (2/3)

- SqlMap 파일 작성(User.xml) – SQL 문을 등록하는 설정 파일

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
  PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN" "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="User">
  <typeAlias alias="User" type="myibatis.domain.User"/>

  <resultMap id="userMap" class="User">
    <result property="userId" column="userId"/>
    <result property="password" column="password"/>
    <result property="name" column="name"/>
    <result property="email" column="email"/>
    <result property="userType" column="userType"/>
  </resultMap>

  <select id="selectAllUsers" resultMap="userMap">
    select
      USER_ID as userId,
      PASSWORD as password,
      NAME as name,
      EMAIL as email
    from USER
    order by USER_ID ASC
  </select>
</sqlMap>
```



iBATIS Quick Start (3/3)

- iBATIS Data Access Object 작성
 - UserIBatisDao.java

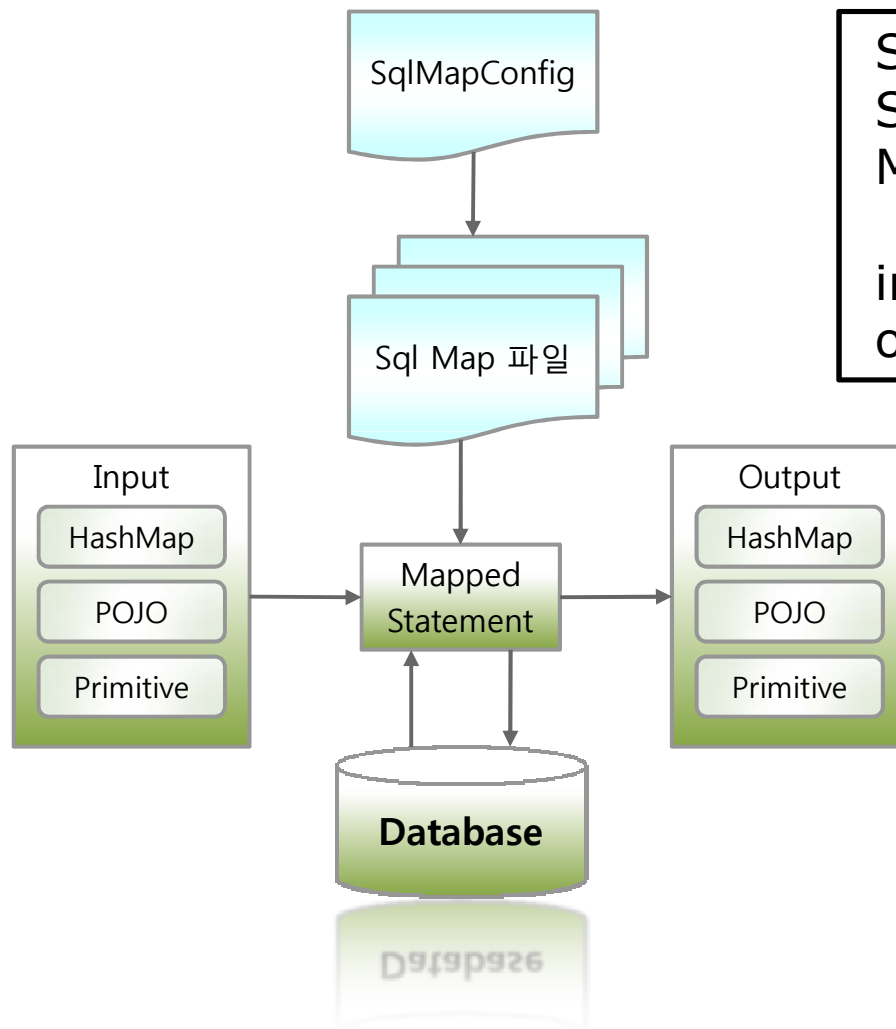
```
package myibatis.dao;
import java.io.IOException;
import java.io.Reader;
import java.sql.SQLException;
import java.util.List;
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import myibatis.domain.User;

public class UserIBatisDao {
    private static final String resource = "sqlmap-config.xml";
    private SqlMapClient client;

    public UserIBatisDao() {
        try {
            Reader reader = Resources.getResourceAsReader(resource);
            this.client = SqlMapClientBuilder.buildSqlMapClient(reader);
        } catch (IOException e) {
            throw new RuntimeException("SqlMapClient 생성중 오류발생", e);
        }
    }

    public List<User> findAllUsers() {
        try {
            List<User> users = client.queryForList("selectAllUsers", null);
            return users;
        } catch (SQLException e) {
            throw new RuntimeException("사용자 조회중 오류 발생", e);
        }
    }
}
```

iBATIS 실행 흐름도



SqlMapConfig : 전역정보설정파일
 SqlMaps : SQL문 설정 파일
 MappedStatement : 설정된 쿼리를
 실행하는 iBATIS 객체
 input : SQL문 실행 시 필요한 값
 output : select 실행 결과



iBATIS 설정 파일

- 설정파일은 XML기반으로 작성
- SqlMapConfig
 - 전역 설정 위한 파일 : iBATIS에 JDBC 처리를 하기 위해 필요한 사항들을 설정한다.
 - Transaction 관리 정보, DataSource Factory를 위한 설정 정보, SqlMap 파일의 위치 등
- SqlMap
 - SQL문을 등록
 - SQL문을 실행하기 위해 필요한 input Data와 Output Data에 대한 설정을 한다.
- 설정 파일들은 작성 후 classpath내 저장한다.



SqlMapConfig 설정 (1/4)

- **SQLMapConfig** 설정 파일
(SqlMapConfig.xml)

프러퍼티파일설정

전역 설정 옵션

트랜잭션 관리

SqlMap 파일 참조

```
<?xml version="1.0" encoding="EUC-KR"?>
<!DOCTYPE sqlMapConfig
  PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
  "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
  <properties resource="sqlmap-config.properties"/>
  <settings useStatementNamespaces="false" cacheModelsEnabled="true"
    enhancementEnabled="true" lazyLoadingEnabled="true"
    maxRequests="${maxRequests}" maxSessions="${maxSessions}"
    maxTransactions="${maxTransactions}" />

  <transactionManager type="${transactionManager}">
    <dataSource type="${dataSource}">
      <property name="DataSource" value="${dsName}" />
      <property name="JDBC.Driver" value="${driver}" />
      <property name="JDBC.ConnectionURL" value="${url}" />
      <property name="JDBC.Username" value="${user}" />
      <property name="JDBC.Password" value="${pword}" />
    </dataSource>
  </transactionManager>

  <sqlMap resource="sample/config/User.xml" />
</sqlMapConfig>
```

SqlMapConfig 설정(2/4)

- **<typeAlias>** 요소
 - 설정파일에서 사용할 클래스의 별칭(alias) 설정
 - iBATIS는 정의된 alias로 언제든지 원래 type에 접근 가능
 - SqlMapConfig에 설정하면 모든 SqlMap에서 사용가능

```
<typeAlias alias="Category" type="my.ibatis.Category"/>
```

- 미리 정의된 typeAlias
 - Transaction manager : JDBC, JTA, EXTERNAL
 - Data types : string, int, long, double, boolean, hashmap, arraylist, object 등
 - Data source factory : SIMPLE, DBCP, JNDI
 - Cache controller : FIFO, LRU, MEMORY, OSCACHE
 - XML result : Dom, domCollection, Xml, XmlCollection

SqlMapConfig 설정 (3/4)

- <transactionManager> 요소
 - Transaction Manager 타입 설정
 - JDBC : 단순 JDBC 기반의 Transaction Manager를 제공함
 - JTA : application이 동작하는 컨테이너 기반의 Transaction Manager를 제공함
 - EXTERNAL : 트랜잭션 관리를 iBATIS에서 하지 않음
 - **<dataSource>** : iBATIS에서 사용할 DataSource를 생성하는 DataSource Factory 지정
 - SIMPLE : iBATIS 자체 제공 하는 DataSourceFactory 사용
 - DBCP : Jakarta Commons Database Connection Pool 구현함
 - JNDI : Naming서버에 등록된 DataSource를 사용함
보통 Container가 제공하는 것을 사용
 - 하위 태그를 이용하여 필요한 property들을 설정한다.
 - driver, url, 계정정보 등

SqlMapConfig 설정 (4/4)

- <sqlMap> 요소
 - SQL문을 가지고 있는 설정파일인 SQL Map파일의 위치 지정
 - resource 속성 : class path 상의 SQL Map 파일

iBATIS를 통한 SQL문 실행

- iBATIS의 JavaBean
- select
- insert
- update
- delete
- Parameter 매핑
- Result 매핑

JavaBeans 기초

- iBATIS의 Java property 접근
 - Property 구성
 - private instance variable과 public setter, getter 메소드

```
private String name;
public String getName();
public void setName(String name);
```

- property 타입이 boolean일 경우

```
private boolean isStudent;
public boolean isStudent();
public void setStudent(boolean isStudent);
```

- Beans 탐색

Java code	Dot 표기법
anOrder.getAccount().getUserName()	anOrder.account.userName
anOrder.getOrderItem().get(0).getProductId()	anOrder.orderItem[0].productId
anObject.getID()	anObject.ID
anObject.getNAME()	anObject.NAME



iBATIS를 통한 SQL 실행 - 개요

- Sql Map – SQL문 설정, Input Data, Output Data를 설정하는 xml 기반 설정파일
 - Sql Map 파일은 SqlConfigMap에 등록한다.
- SqlMapClient interface : SQL문 실행 메소드를 정의
 - package : com.ibatis.sqlmap.client
 - SqlMapClientBuilder를 통해 얻어온다.

```
//iBATIS Framework가 실행할때 필요한 SqlMapConfig
Reader reader = Resources.getResourceAsReader("config/SqlMapConfig.xml");
//SqlMapConfig의 내용을 적용하여 실행할 SqlMapClient 생성
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
```

- Sql Map에 SQL 문을 등록하고 프로그램에서는 SqlMapClient의 메소드를 이용해 등록된 쿼리를 실행 시킨다.

Sql Map (1/3)

- 주요 태그
 - SQL 문 등록 태그

태그명	속성	하위 요소	용도
<select>	id, parameterClass, parameterMap, resultClass, resultMap, cacheModel	모든 dynamic 요소	조회
<insert>	id, parameterClass, parameterMap	모든 dynamic 요소, selectKey	입력
<update>	id, parameterClass, parameterMap	모든 dynamic 요소	수정
<delete>	id, parameterClass, parameterMap	모든 dynamic 요소	삭제

– 외부 Parameter, Result 매핑 설정

태그명	속성	하위 요소	용도
<parameterMap>	id, class	<parameter>	외부 Parameter Map 설정의 Root 태그
<parameter>	property, javaType, jdbcType, nullValue	N/A	하나의 Parameter Map 설정
<resultMap>	id, class	<result>	외부 Result Map 설정
<result>	property, column, jdbcType, javaType, nullValue	N/A	하나의 Result Map 설정

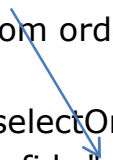
Sql Map (2/3)

- 주요 태그
 - SQL 문 생성 태그

태그명	속성	하위 요소	용도
<sql>	id	모든 dynamic 요소	재사용가능한 SQL문 등록
<include>	refid	모든 dynamic 요소	SQL등록 태그에서 sql에 등록한 태그를 재사용

```

<sql id="select">
  select * from order
</sql>
<select id="selectOrderByid resultClass="map">
  <include refid="select" />
  where order_id = #orderId#
</select>
  
```



Sql Map (3/3)

member.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="member">
  <insert id="insertMember" parameterClass="mvo">
    INSERT INTO MEMBER (ID, PASSWORD, NAME, ADDRESS, MILEAGE)
    VALUES(#id#, #password#, #name#, #address#, #mileage#)
  </insert>
  <update id="updateMember" parameterClass="mvo">
    UPDATE MEMBER
    SET   PASSWORD=#password#, NAME=#name#,
          ADDRESS=#address#, MILEAGE=#mileage#
    WHERE ID=#id#
  </update>
  <delete id="deleteMember">
    DELETE FROM MEMBER WHERE ID=#value#
  </delete>
  <select id="getMemberByID" resultClass="mvo">
    SELECT ID, PASSWORD, NAME, ADDRESS, MILEAGE FROM MEMBER
    WHERE ID=#value#
  </select>
</sqlMap>
```




Parameter mapping (1/3)

- Sql Map에 쿼리 등록시 세가지 요소 설정 필요
 - SQL 문
 - Parameter : Input Data
 - Select문의 경우 Result : Output Data
- Parameter Mapping
 - Parameter : SQL 문에 넣을 값
 - 인 라인 Parameter , 외부 Parameter 매핑 두 가지 방식이 있다.
 - 인 라인 - 값과 값을 넣을 위치를 SQL 안에 설정
 - 외부 parameter mapping - SQL 문에는 ? 로 설정하고 넣을 parameter는 SQL문 밖에서 설정
 - SqlMapClient의 메소드를 통해 받은 값과 SQL 문에 값이 들어갈 곳을 Mapping 한다.
 - mapping 방법
 - #parameter name# } 인라인 파라미터
 - \$parameter name# }
 - ? - 외부 파라미터 매핑
 - parameter name
 - VO : getter와 matching
 - Map : key와 matching



Parameter mapping (2/3)

- `#` 지시자로 인라인 parameter 사용
 - ?로 바꾼 뒤 값을 치환한다. - PreparedStatement 형식
 - String일 경우 ` ` 생성(LIKE Query 문에서 적용 어려움)

```
<select id="selectUser" resultMap="userMap">
  select
    USER_ID as userId,
    PASSWORD as password,
    NAME as name,
    EMPLOYEE_NO as employeeNo,
    EMAIL as email,
    PHONE_NUM as phoneNumber,
    ZIP_CODE as zipCode,
    ADDRESS,
    GRADE as grade
  from USER
  where USER_ID = #value#
</select>
```

...

```
where USER_ID = 'abc';
```

```
User user = (User)client.queryForObject("selectUser", "abc");
```

Parameter mapping (3/3)

- '\$' 지시자로 인라인 parameter 사용
 - 전달받은 값을 바로 치환한다. - copy & paste 개념
 - String일 경우 ''로 감싸 주어야 한다.

```
<select id="selectUser" resultMap="userMap">
  select
    USER_ID as userId,
    PASSWORD as password,
    NAME as name,
    EMPLOYEE_NO as employeeNo,
    EMAIL as email,
    PHONE_NUM as phoneNumber,
    ZIP_CODE as zipCode,
    ADDRESS,
    GRADE as grade
  from USER
  where USER_ID = '%$values$%'
</select>
```

...

where user.NAME like '%송%';

```
List<User> allUsers = client.queryForList("selectAllUsers", "송");
```

Parameter 매핑 – 외부 매핑 (1/2)

- Parameter 매핑의 두가지 방안
 - 인라인 매핑
 - SqlMap Mapping문 내부에서 바로 기술함
 - 매핑이 복잡한 경우 명시적이지 못함
 - 외부 매핑
 - 외부에서 정의됨
 - 보다 명시적임
 - 인라인 매핑과 외부매핑은 같이 사용할 수 없다.
 - ? 와 매칭되어 값이 할당된다.

인라인 매핑의 예

```
<insert id="createCategory" parameterClass="Category">
  insert into CATEGORY_TB (
    CAT_ID,
    CAT_NAME,
    CAT_DESC
  )
  values (
    #id#,
    #name#,
    #description#
  )
</insert>
```

외부 매핑의 예

```
<parameterMap class="dvo" id="deptParamMap">
  <parameter property="deptId"/>
  <parameter property="deptName"/>
  <parameter property="managerId"/>
  <parameter property="loc.locationId"/>
</parameterMap>
<insert id="insert" parameterMap="deptParamMap" >
  INSERT INTO DEPARTMENTS VALUES(?, ?, ?, ?)
</insert>
```

Parameter 매핑 – 외부 매핑 (1/2)

- 외부 Parameter 매핑

속성	설명
property	매핑문에 전달할 JavaBean property 명칭 또는 Map value의 key. 들어갈 이름을 매핑문에 필요한 만큼 반복해서 정의할 수 있음. 예) update 문에서 set 절과 where절에 동일한 이름이 반복적으로 들어갈 수 있음.
javaType	세팅될 parameter의 Java property 타입을 명시적으로 정의
jdbcType	세팅될 parameter의 데이터베이스 타입을 명시적으로 정의 예) Java 타입이 Date(java.util.Date)인 경우 jdbcType을 DATE인지 DATETIME인지 명시할 필요가 있음
nullValue	nullValue에 명시된 값이 JavaBean property에서 넘어오면 null을 대체할 Default 값
typeHandler	javaType 및 jdbcType이 일치되는 경우 typeHandler가 적용되나 그것 없이 직접 typeHandler를 지정할 수 있음

- 외부 Parameter 매핑이 유용한 경우
 - 인라인 Parameter 매핑이 잘 동작하지 않을 경우
 - 성능을 향상시킬 경우
 - 명시적 매핑을 할 경우

Parameter 매핑 - 정리

- 인라인 Parameter 매핑
 - 인라인 Parameter에 데이터베이스 타입 명시
 - #value:[jdbcType]# (#id:[VARCHAR])
- Primitive Parameter
 - int 또는 long 같은 Primitive 타입은 iBATIS에서 자동으로 Integer, Long으로 Wrapping 됨
- JavaBean과 Map Parameter의 차이점
 - JavaBean : property가 정해져 있으므로 load time에 매핑 오류 발견 가능
 - Map : property가 Runtime에 정해지므로 load time에 매핑 오류 발견 불가

Result 매핑 (1/3)

- 명시적 Result 매핑

속성	설명
property	결과 객체의 JavaBean property 또는 Map 내용
column	JDBC ResultSet의 column
columnIndex	ResultSet의 column명 대신 index를 주어 결과 값 매핑이 가능. 약간의 성능 향상이 있으며 필수 사항은 아님.
jdbcType	ResultSet column의 타입을 명시함. java.util.Date의 경우 해당되는 jdbcType이 여러가지(DATE, DATETIME)이므로 정확한 매핑을 위해 명시하는 것이 좋음. JDBC driver에 따라 정의할 필요가 없는 것도 있음.
javaType	세팅될 Java 타입을 명시함.
nullValue	데이터베이스에서 null이 넘어온 경우 null을 대신할 default 값 지정
select	객체 관계를 표현하며 복잡한 property 타입을 로드 할 수 있음 들어갈 값은 반드시 다른 매핑문의 이름이어야 함.

Result 매핑 (2/3)

- Primitive result
 - iBATIS는 primitive 형태의 결과를 허용하지 않음(int, long, double, ..)
 - Wrapping 된 형태의 타입을 리턴(Integer, Long, Double, ..)

```
<select id="getAllOrderCount" resultClass="int">
  select count(*) as value
  from order
</select>
```

```
Integer allCount =
  client.queryForObject("getAllOrderCount", null);
```

- JavaBean 타입으로 Wrapping 할 경우 primitive 형태의 결과를 얻을 수 있음

```
public class PrimitiveResult {
  private int orderCount;
  public int getOrderCount() {
    return orderCount;
  }
  public void setOrderCount(int orderCount) {
    this.orderCount = orderCount;
  }
}
```

```
<resultMap id="primitiveResultMapExample" class="PrimitiveResult">
  <result property="orderCount" column="orderCount" />
</resultMap>
```


Result 매핑 (3/3)

- JavaBean과 Map Result의 장 단점

방식	장점	단점
Bean	성능 향상 컴파일 타임의 이름 체크 IDE에서 Refactoring 지원 가능 Type casting이 적음	get, set메소드 필요
Map	코드량이 적음	느리다 컴파일 타임 체크 불가 런타임 오류가 많음 Refactoring 지원이 없음

SELECT 구문 실행 (1/4)

- **SqlMap : <select> 태그 이용**
 - 속성
 - id : 프로그램에서 호출할 이름
 - resultClass, resultMap : SELECT 문 실행 결과를 담은 객체
 - parameterClass, parameterMap : PARAMETER를 받아올 객체
- **SqlMapClient – queryForObject() : 0또는 1개의 row를 가져올 때 사용 – unique한 값으로 조회 시**

Object queryForObject(String id, Object parameter) throws SQLException;
Object queryForObject(String id, Object parameter, Object result) throws SQLException;

- 첫 번째 메소드 : SQL Map에 정의된 select문 id와 해당 파라미터로 사용
- 두 번째 메소드 : 결과 결과를 담은 객체를 넣음. no-argument 생성자가 없는 resultClass를 사용해야 하는 경우 사용
- 1개 이상의 row가 리턴 된 경우 Exception 발생



SELECT 구문 실행 (2/4)

- SqlMapClient – queryForList() 메소드
 - 1개 이상의 row가 리턴될 경우 사용
 - 결과를 List로 리턴함
 - select 결과의 1개의 row – resultClass의 객체
 - resultClass 객체 들 – List

List queryForList(String id, Object parameter) throws SQLException;

List queryForList(String id, Object parameter, int skip, int max) throws SQLException;

- 첫 번째 메소드 : 사용법이 앞과 동일
- 두 번째 메소드 : DB에서 가져온 전체 row중 일부만 사용하고 싶은 경우 사용
 - 예) DB에서 100건을 가져올 때 이 중 앞의 10건만 취하고 싶으면 skip=0, max=10



SELECT 구문 실행 (3/4)

- queryForMap() 메소드
 - 여러 건을 Map 형태로 리턴함

Map queryForMap(String id, Object parameter, String key) throws SQLException;
Map queryForMap(String id, Object parameter, String key, String value) throws SQLException;

- 첫 번째 메소드 :
 - key – map의 key값으로 들어갈 property 또는 key를 입력
 - 결과 Map의 value는 resultClass에 지정한 클래스의 객체가 들어간다.
- 두 번째 메소드 :
 - key – map의 key값으로 들어갈 property 또는 key를 입력
 - 결과 Map의 value는 지정한 property 또는 key의 데이터만 들어간다.

queryForMap() 사용예

```
Map accountMap = sqlMap.queryForMap("Account.getAll",null,"accountId");  
Map accountMap = sqlMap.queryForMap("Account.getAll",null,"accountId","username");
```

SELECT 구문 실행 - 결과 매핑 (4/4)

- 자동 Result Map
 - select문 실행 결과를 자동으로 객체에 매핑 가능함
 - 단일 column 매핑
 - count(id) 값이 Integer 타입으로 자동 매핑된다.

```
<select id="getAllAccountIdValues" resultClass="int">
  select count(id)
  from Account
</select>
```

```
Integer cnt = (Integer)
client.queryForObject("getAllAccountIdValues", null);
```

- 다중 column 매핑
 - select 문의 column명과 결과 객체의 Bean property 명이 일치해야 함

```
<select id="selectAllCategories" resultClass="Category">
  select
    cat.CAT_ID as id,
    cat.CAT_NAME as name,
    cat.CAT_DESC as description
  from CATEGORY cat
</select>
```

```
public class Category{
  ...
  setId(String id){}
  setname(String name){}
  setDescription(String desc){}
  ...
}
```

INSERT

- SqlMap : <insert> 태그 사용
 - 속성
 - id : 프로그램에서 호출 할 이름
 - parameterClass, parameterMap : PARAMETER를 받아 올 객체
 - Sub tag : <selectKey> - PK 객체 return 시

- SqlMapClient – insert() 사용

Object insert(String id) throws SQLException;
Object insert(String id, Object parameter) throws SQLException;

- return value : PK 객체 - <selectKey>를 사용한 경우

UPDATE

- SqlMap : <update> 태그 사용
 - 속성
 - id : 프로그램에서 호출 할 이름
 - parameterClass, parameterMap : PARAMETER를 받아 올 객체
- SqlMapClient – update() 사용

```
int update (String id) throws SQLException;  
int update(String id, Object parameter) throws SQLException;
```

- return value : int – update가 적용된 record 개수



DELETE

- SqlMap : <delete> 태그 사용
 - 속성
 - id : 프로그램에서 호출 할 이름
 - parameterClass, parameterMap : PARAMETER를 받아올 객체
- SqlMapClient – delete() 사용

```
int delete(String id) throws SQLException;  
int delete(String id, Object parameter) throws SQLException;
```

- return value : int – 삭제가 적용된 record 개수



Dynamic SQL

- 동적 WHERE 절 처리
- Dynamic 태그

동적 WHERE 절 처리

- 언제 사용하는가?
 - where 절의 비교구문이 경우에 따라 달라질 경우
 - 예) 카테고리 조회시 부모카테고리ID가 null일 수도 있는데 null인 경우 조회 방식이 바뀐다.
parentCategoryId=#parentCategoryId# 또는 parentCategoryId IS NULL

```
<select id="getChildCategories" parameterClass="Category"
resultClass="Category">
SELECT * FROM category
<dynamic prepend="WHERE ">
  <isNull property="parentCategoryId">
    parentCategoryId IS NULL
  </isNull>
  <isNotNull property="parentCategoryId">
    parentCategoryId=#parentCategoryId#
  </isNotNull>
</dynamic>
</select>
```

파라미터 Category의 부모 아이디가 null인 경우

파라미터 Category의 부모 아이디가 null이 아닌 경우

SQL 재 활용성 증가

Dynamic 태그

- Dynamic 태그에 대하여
 - 5가지 카테고리
 - dynamic, binary, unary, parameter, iterate
 - 공통 속성
 - prepend, open, close

```

<dynamic prepend="WHERE ">
  ...
  <isEmpty property="y">y=#y#</isEmpty>
  <NotNull property="x" removeFirstPrepend="true" prepend="AND"
    open="(" close=")">
    <isEmpty property="x.a" prepend="OR">a=#x.a#</isEmpty>
    <isEmpty property="x.b" prepend="OR">a=#x.b#</isEmpty>
    <isEmpty property="x.c" prepend="OR">a=#x.c#</isEmpty>
  </NotNull>
  ...
</dynamic>

```

앞뒤를 ()으로 감싼다.

WHERE 구문을 앞에 추가

첫번째 하위 요소의 prepend인 "OR" 을 제거한다.

Dynamic 태그

- <dynamic> 태그
 - 가장 상위의 태그
 - 다른 태그에 포함될 수 없음
 - 접두/접미 구문을 위한 수단 제공(prepend, open, close)

prepend (옵션)	맨 앞에 접두 구문을 붙인다. (예: "WHERE") Body에 내용이 없을 경우 접두 구문도 생략된다.
open (옵션)	Body의 내용 앞에 붙여짐. Body의 내용이 없을 경우 생략된다. prepend와 동시에 정의될 경우 prepend 다음에 나타난다.
close (옵션)	Body의 내용 뒤에 붙여짐. Body의 내용이 없을 경우 생략된다.

Dynamic 태그

- Binary 태그
 - 주어진 Parameter 객체간의 property 값을 비교
 - 태그의 조건이 충족되는 경우 Body내용이 나타남
 - Binary 태그 속성

property(필수)	Parameter 객체의 property. compareValue 또는 compareProperty에서 비교하는데 쓰인다.
prepend(옵션)	맨 앞에 접두 구문을 붙인다. 이 구문이 생략되는 경우 1. Body의 내용이 없는 경우 2. 부모 태그 속성이 removeFirstPrepend="true" 이고 현재 태그가 부모 Body의 첫 번째 요소인 경우
open(옵션)	Body의 내용 앞에 붙여짐. Body의 내용이 없을 경우 생략된다. prepend와 동시에 정의될 경우 prepend 다음에 나타난다.
close(옵션)	Body의 내용 뒤에 붙여짐. Body의 내용이 없을 경우 생략된다.
removeFirstPrepend (옵션)	첫번째 자식 태그의 prepend를 생략시킴
compareProperty	property 와 비교할 property 이름
compareValue	property 속성에 의해 비교되는 값(상수)

Dynamic 태그

- Binary 태그(계속)
 - Binary 태그 종류

<isEqual>	property속성과 compareProperty/compareValue속성이 같은지 비교
<isNotEqual>	property속성과 compareProperty/compareValue속성이 다른지 비교
<isGreaterThan>	property속성이 compareProperty/compareValue속성보다 큰지 비교
<isGreaterEqual>	property속성이 compareProperty/compareValue속성보다 크거나 같은지 비교
<isLessThan>	property속성이 compareProperty/compareValue속성보다 작은지 비교
<isLessEqual>	property속성이 compareProperty/compareValue속성보다 작거나 같은지 비교



Dynamic 태그

- Binary 태그(계속)
 - Binary 태그 예제

```
<select id="getShippingType" parameterClass="Cart"
  resultClass="Shipping">
  SELECT * FROM Shipping
  <dynamic prepend="WHERE ">
    <isGreaterEqual property="weight" compareValue="100">
      shippingType='FREIGHT'
    </isGreaterEqual>
    <isLessThan property="weight" compareValue="100">
      shippingType='ST-ARD'
    </isLessThan>
  </dynamic>
</select>
```

*Cart.getWeight() >= 100 인
지 체크*

*Cart.getWeight() < 100 인
지 체크*

Dynamic 태그

- Unary 태그
 - 비교를 수행하지 않고 Bean property 상태를 체크한다.
 - 태그의 조건이 충족되는 경우 Body내용이 나타남
 - Unary 태그 속성

property(필수)	상태를 체크하기 위한 Parameter 객체의 property
prepend(옵션)	<p>맨 앞에 접두 구문을 붙인다. 이 구문이 생략되는 경우</p> <ol style="list-style-type: none"> 1. Body의 내용이 없는 경우 2. 부모 태그 속성이 removeFirstPrepend="true" 이고 현재 태그가 부모 Body의 첫번째 요소인 경우
open(옵션)	Body의 내용 앞에 붙여짐. Body의 내용이 없을 경우 생략된다. prepend와 동시에 정의될 경우 prepend 다음에 나타난다.
close(옵션)	Body의 내용 뒤에 붙여짐. Body의 내용이 없을 경우 생략된다.
removeFirstPrepend (옵션)	첫번째 자식 태그의 prepend를 생략시킴

Dynamic 태그

- Unary 태그(계속)
 - Unary 태그 종류

<isPropertyAvailable>	특정 property가 존재하는지 체크. Map의 경우 key가 존재하는지 체크함.
<isNotPropertyAvailable>	특정 property가 존재하지 않는지 체크. Map의 경우 key가 존재하지 않는지 체크함.
<isNull>	특정 property가 null인지 체크.
<isNotNull>	특정 property가 null이 아닌지 체크.
<isEmpty>	특정 property가 null이거나 비어있는지 체크.
<isNotEmpty>	특정 property가 null이아니고 비어있지 않는지 체크.

```

<select id="getProducts" parameterClass="Product"
    resultClass="Product">
    SELECT * FROM Products
    <dynamic prepend="WHERE ">
        <isNotEmpty property="productType">
            productType=#productType#
        </isNotEmpty>
    </dynamic>
</select>
    
```

*Product.getProductType != null 이고
 Product.getProductType.length() > 0
 인지 체크*

Dynamic 태그

- Parameter 태그
 - Parameter 가 매핑문에 들어왔는지 체크하는 태그
 - 태그 속성 : prepend, open, close, removeFirstPrepend
 - 태그 종류

<isParameterPresent>	Parameter가 존재하는지 체크
<isNotParameterPresent>	Parameter가 존재하지 않는지 체크

```

<select id="getProducts" resultClass="Product">
  SELECT * FROM Products
  <isParameterPresent prepend="WHERE ">
    <isNotEmpty property="productType">
      productType=#productType#
    </isNotEmpty>
  </isParameterPresent>
</select>
  
```

이 select 매핑문에 들어오는 어떤 Parameter든지 존재하는지 체크

Dynamic 태그

- <iterate> 태그
 - Collection 형태의 property로 SQL문의 반복적인 구간 생성
 - 각각의 구간은 "conjunction" 속성으로 분리됨
 - 태그 속성 : property(필수), prepend, open, close, conjunction, removeFirstPrepend

conjunction	반복되는 SQL 구문 사이에 구분자로 들어감 (예: ', ')
-------------	------------------------------------

```

<select id="getProducts" parameterClass="Product"
  resultClass="Product">
  SELECT * FROM Products
  <dynamic prepend="WHERE">
    <iterate property="productType" open="(" close=")"
      conjunction="OR" >
      productType=#productType[]#
    </iterate>
  </dynamic>
</select>
  
```

Product의 Collection 형태의 property

반복 구간이 'OR' 로 구분됨

Transaction 관리

iBATIS Transaction 관리

- Auto Transaction
- Local Transaction



Auto Commit

- iBATIS의 자동 트랜잭션 지원
 - Statement를 수행하기만 해도 적용됨
 - 별다른 설정이 없음

```
public void runStatementsUsingAutomaticTransactions() {  
    SqlMapClient sqlMapClient =  
        SqlMapClientConfig.getSqlMapClient();  
    Person p = (Person) sqlMapClient.queryForObject("getPerson",  
        new Integer(9));  
    p.setLastName("Smith");  
    sqlMapClient.update("updatePerson", p);  
}
```

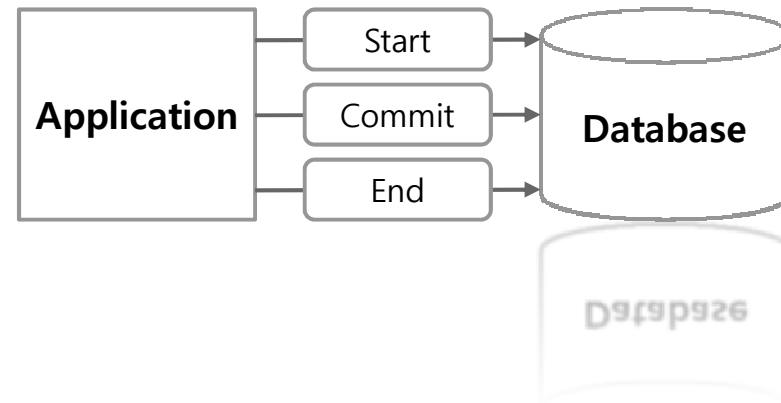


Local Transaction

- 가장 일반적인 형태의 Transaction
- JDBC Transaction Manager 사용
 - sqlmap-config.xml에 설정

```
<transactionManager type="JDBC">  
  <dataSource type="SIMPLE">  
    <property .../>  
    <property .../>  
    <property .../>  
  </dataSource>  
</transactionManager>
```

Local 트랜잭션 Manager 설정





Local Transaction

- Local Transaction 적용 예

```
public void runStatementsUsingLocalTransactions() {  
    SqlMapClient sqlMapClient = SqlMapClientConfig.getSqlMapClient();  
    try {  
        sqlMapClient.startTransaction();  
        Person p = (Person) sqlMapClient.queryForObject("getPerson",  
            new Integer(9));  
        p.setLastName("Smith");  
        sqlMapClient.update("updatePerson", p);  
        Department d = (Department) sqlMapClient.queryForObject("getDept",  
            new Integer(3));  
        p.setDepartment(d);  
        sqlMapClient.update("updatePersonDept", p);  
        sqlMapClient.commitTransaction();  
    } finally {  
        sqlMapClient.endTransaction();  
    }  
}
```

Transaction 시작

Commit

Transaction 종료



Spring에서 Transaction 관리

- Spring의 트랜잭션 관리에 대한 지원
 - 프로그램적인 관리와 선언적인 관리 지원
 - Spring의 프로그램적인 트랜잭션 관리는 EJB와 다름.
 - EJB는 Java Transaction API(JTA) 구현에 종속됨.
 - Spring은 트랜잭션을 필요로 하는 코드로부터 실질적인 트랜잭션 구현을 추상화하는 콜백(callback) 메커니즘을 사용.
 - Spring은 JTA 구현이 필요하지 않음.
 - 단일 저장 리소스인 경우, 저장 메커니즘(JDBC, Hibernate, JDO, OJB 등)에서 제공하는 트랜잭션 지원 사용이 가능
 - 다중 리소스인 경우, 3rd party JTA 구현 도구를 사용해서 분산 트랜잭션 (XA) 지원 가능
 - 프로그램적인 트랜잭션은 코드에서 트랜잭션 영역을 세밀하게 정의하는데 유연성을 제공
 - 선언적인 트랜잭션은 트랜잭션 규칙과 오퍼레이션의 결합도를 없애줌.
 - Spring의 선언적인 트랜잭션에 대한 지원은 EJB의 CMT(container-managed transaction)과 비슷함.
 - Spring의 선언적인 트랜잭션은 CMT 에서 지원하지 않는 isolation level과 timeout과 같은 추가적인 속성 선언이 가능
 - 프로그램적인 트랜잭션과 선언적인 트랜잭션 관리에 대한 선택은 세밀한 통제 (fine-grained control)와 편의성(convenience)에 대한 결정임.
 - Spring 트랜잭션 manager는 플랫폼에 특화된 트랜잭션 구현과 인터페이싱이 가능.

트랜잭션 Manager (1/3)

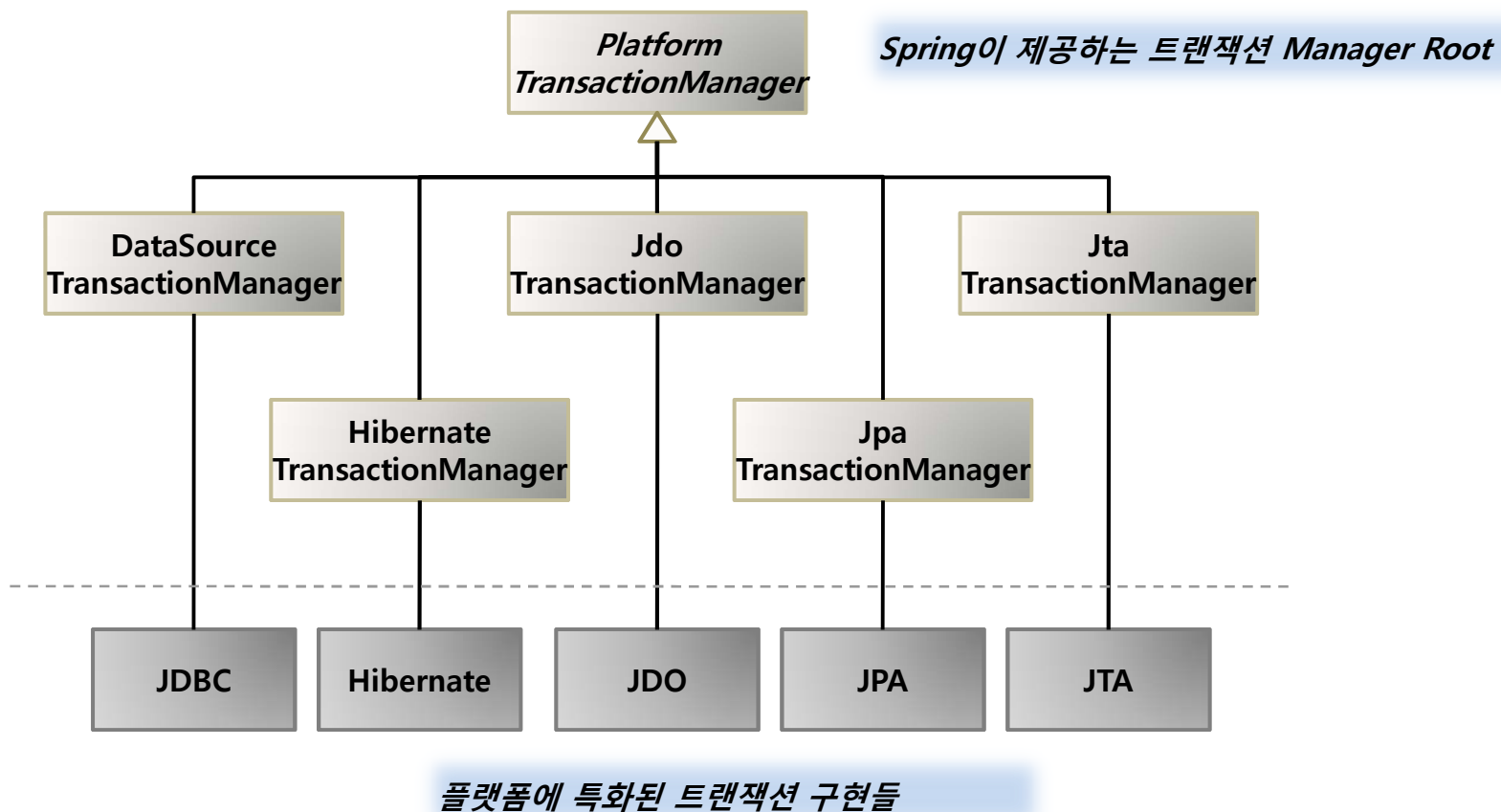
• Spring제공 트랜잭션 Manager

트랜잭션 Manager (org.springframework.*)	사용 목적
jdbc.datasource. DataSourceTransactionManager	Spring의 JDBC 추상화 지원. iBATIS 사용시에도 사용 가능
jca.cci.connection.CciLocalTransactionManager	J2EE Connector Architecture (JCA)와 Common Client Interface(CCI)를 위한 Spring 지원
jms.connection.JmsTransactionManager	JMS 1.1+ 사용시
jms.connection.JmsTransactionManager102	JMS 1.0.2 사용시
orm.hibernate.HibernateTransactionManager	Hibernate 2 사용시
orm.hibernate3.HibernateTransactionManager	Hibernate 3 사용시
orm.jdo.JdoTransactionManager	JDO 사용시
orm.jpa.JpaTransactionManager	Java Persistence API(JPA) 사용시
orm.toplink.TopLinkTransactionManager	오라클의 TopLink 사용시
transaction.jta.JtaTransactionManager	분산 트랜잭션이 필요하거나 맞는 트랜잭션 manager가 없는 경우
transaction.jta.OC4JtaTransactionManager	오라클의 OC4J JEE 컨테이너 사용시
transaction.jta. WebLogicJtaTransactionManager	분산 트랜잭션이 필요하거나 WebLogic 내에서 어플리케이션이 실행되는 경우

트랜잭션 Manager 선택 (2/3)

Spring의 트랜잭션 manager들은 플랫폼에 특화된 트랜잭션 구현들에게 트랜잭션 관리에 대한 책임을 위임함.

- Spring 제공 트랜잭션 Manager





트랜잭션 Manager (3/3)

- JDBC 트랜잭션
 - DataSourceTransactionManager 사용

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

- dataSource 속성은 javax.sql.DataSource 빈(<bean>)
- 내부적으로 DataSource의 java.sql.Connection을 호출하여, 트랜잭션 성공시 commit()을, 실패시 rollback() 메소드를 사용하여 Transaction을 처리

Spring과 iBATIS 연동



Spring과 iBATIS (1/2)

- iBATIS 클라이언트 템플릿 설정
 - SqlMapClientTemplate 설정

```
<bean id="sqlMapClient"
  class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="configLocation" value="sqlMapConfig.xml"/>
</bean>

<bean id="sqlMapClientTemplate"
  class="org.springframework.orm.ibatis.SqlMapClientTemplate">
  <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
```

iBATIS 설정 파일



Spring과 iBATIS (2/2)

- iBATIS 클라이언트 템플릿 설정
 - DAO에서 템플릿 사용

```
public class IbatisUserDao implements UserDao {  
    public IbatisUserDao() {}  
    private SqlMapClientTemplate sqlMapClientTemplate;  
    public void setSqlMapClientTemplate(SqlMapClientTemplate sqlMapClientTemplate) {  
        this.sqlMapClientTemplate = sqlMapClientTemplate;  
    }  
    @Override  
    public User selectMemberById(String id) {  
        return (MemberVO)sqlMapClientTemplate.queryForObject("SELECT_MEMBER_BY_ID", id);  
    }  
    @Override  
    public void insertMember(MemberVO mvo) {  
        sqlMapClientTemplate.insert("INSERT_MEMBER", mvo);  
    }  
}
```

```
<bean id="memberDao" class="dao.MemberDAO">  
    <property name="sqlMapClientTemplate" ref="sqlMapClientTemplate"/>  
</bean>
```