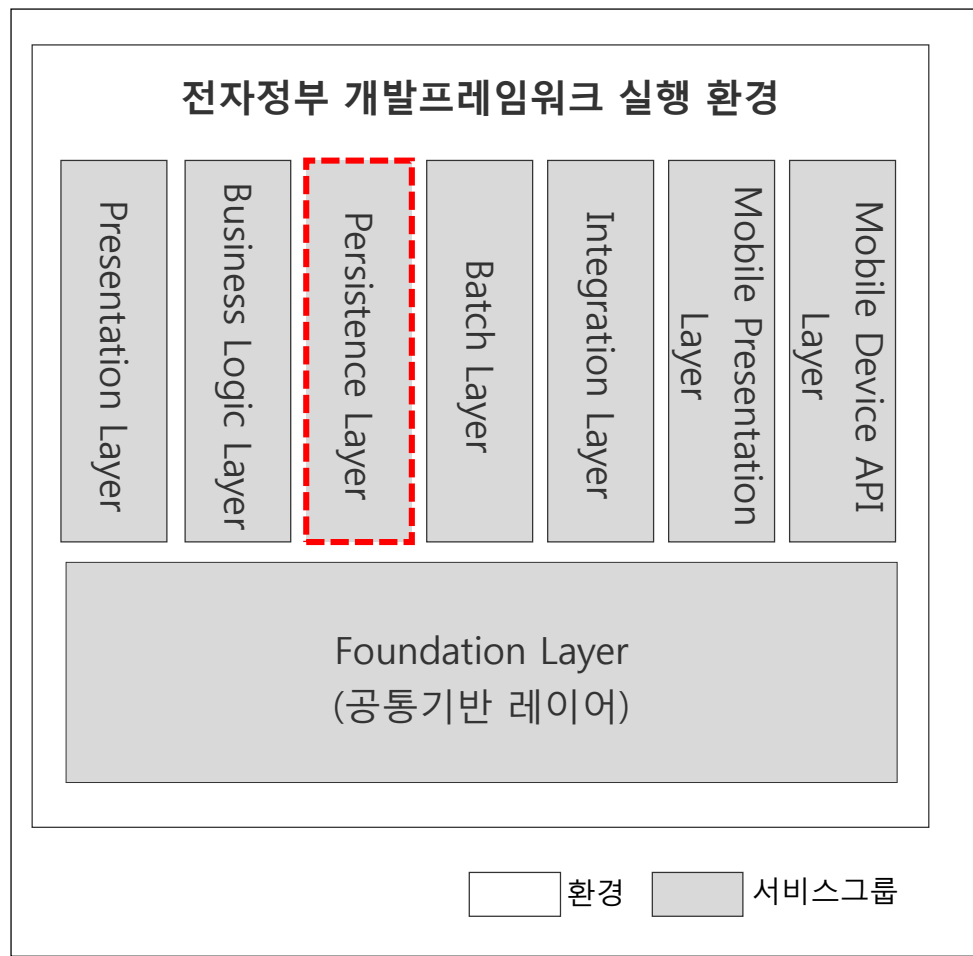


전자정부 표준프레임워크
실행환경(데이터처리)

1. 개요
2. Data Source
3. Data Access
 - iBatis, MyBatis, Spring Data JPA
4. ORM (Hibernate)
5. Transaction

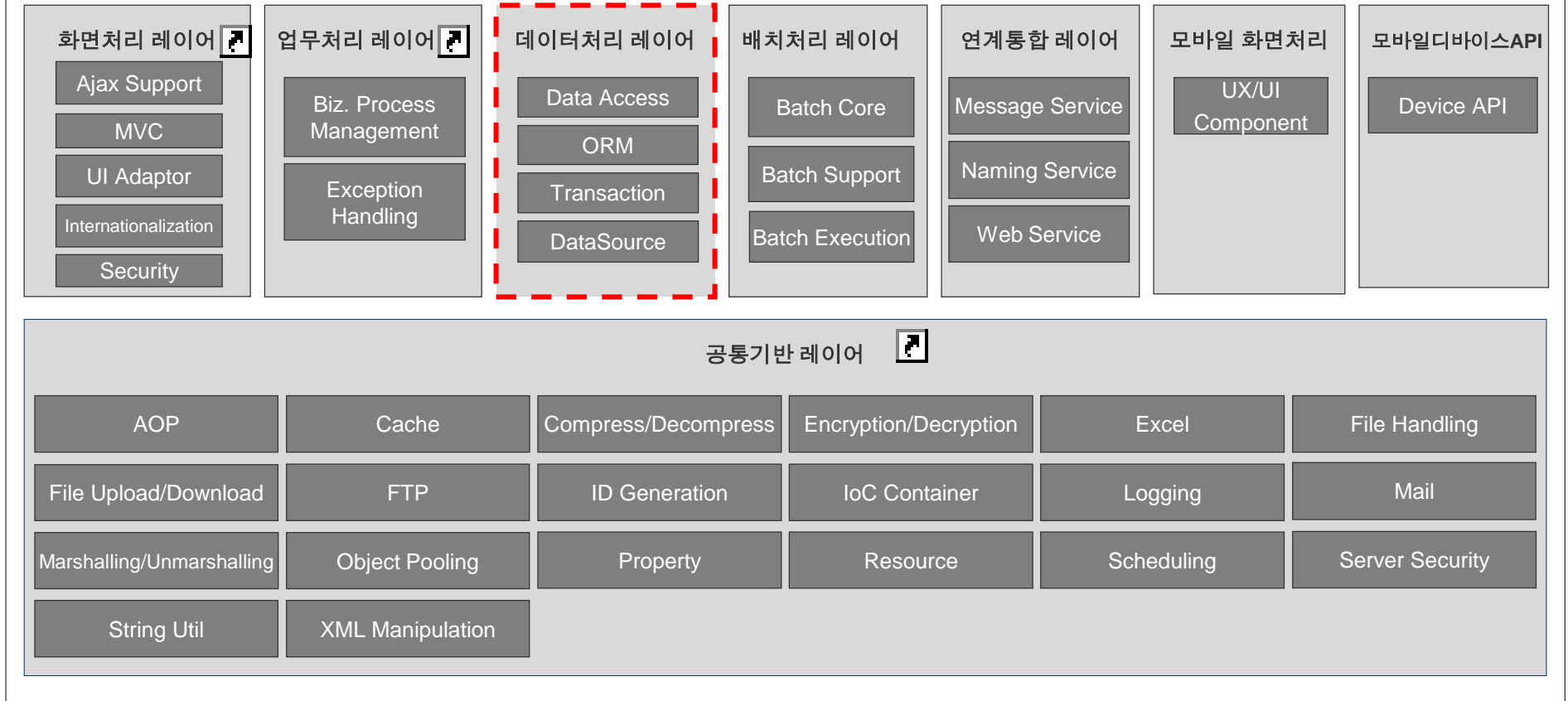
- ❑ 데이터베이스에 대한 연결 및 영속성 처리, 선언적인 트랜잭션 관리를 제공하는 Layer임



서비스 그룹	설명
Presentation Layer	<ul style="list-style-type: none"> 업무 프로그램과 사용자 간의 Interface를 담당하는 Layer로서, 사용자 화면 구성, 사용자 입력 정보 검증 등의 기능을 제공함
Business Logic Layer	<ul style="list-style-type: none"> 업무 프로그램의 업무 로직을 담당하는 Layer로서, 업무 흐름 제어, 에러 처리 등의 기능을 제공함
Persistence Layer	<ul style="list-style-type: none"> 데이터베이스에 대한 연결 및 영속성 처리, 선언적인 트랜잭션 관리를 제공하는 Layer임
Batch Layer	<ul style="list-style-type: none"> 대용량 데이터 처리를 위한 기반 환경을 제공하는 Layer임
Integration Layer	<ul style="list-style-type: none"> 타 시스템과의 연동 기능을 제공하는 Layer임
Foundation Layer	<ul style="list-style-type: none"> 실행 환경의 각 Layer에서 공통적으로 사용하는 공통 기능을 제공함

- ❑ 데이터처리 레이어는 DataSource, Data Access 등 총 4개의 서비스를 제공함

실행 환경



공통기반 레이어 살펴보기

- 데이터처리 레이어는 Spring, iBatis, MyBatis, Hibernate 등 총 4종의 오픈소스 SW를 사용하고 있음

서비스	오픈소스 SW	버전
DataSource	Spring	4.0.9
Data Access	iBatis SQL Maps	2.3.4
	MyBatis	3.2.7
ORM	Hibernate	4.3.5
Transaction	Spring	4.0.9

❑ DataSource 서비스

- 데이터베이스에 대한 연결을 제공하는 서비스이다. 다양한 방식의 데이터베이스 연결을 제공하고, 이에 대한 추상화 계층을 제공함으로써, 업무로직과 데이터베이스 연결방식 간의 종속성을 배제한다.
- DataSource 객체 획득 방법
 - DriverManagerDataSource 빈 등록
 - BasicDataSource 빈 등록
 - ComboPooledDataSource 빈 등록
 - JNDI DataSource

JNDI Lookup을 이용하여 WAS에서 제공되는 JNDI tree로부터 DataSource를 가져옴
- Connection 객체 획득 방법: `dataSource.getConnection();`

❑ Spring DataSource

Spring DataSource 설정

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="${driver}" />
  <property name="url" value="${dburl}" />
  <property name="username" value="${username}" />
  <property name="password" value="${password}" />
</bean>
```

- driverClassName : JDBC driver class name 설정
- url : DataBase 접근하기 위한 JDBC URL
- username : DataBase 접근하기 위한 사용자명
- password : DataBase 접근하기 위한 암호

테스트 코드

```
@Resource(name = "dataSource")
DataSource dataSource;

@Test
public void testJdbcDataSource() throws Exception {
    assertNotNull(dataSource);
    assertEquals("org.springframework.jdbc.datasource.DriverManagerDataSource", dataSource.getClass().getName());
    Connection con = null; Statement stmt = null; ResultSet rs = null;
    try {
        con = dataSource.getConnection();
        assertNotNull(con);
        stmt = con.createStatement();
        rs = stmt.executeQuery("select 'x' as x from dual");
        while (rs.next()) {
            assertEquals("x", rs.getString(1)); .....
        }
    }
}
```

❑ DBCP DataSource

DBCP DataSource 설정

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

- driverClassName : jdbc driver class name 설정
- url : DataBase url 설정
- username : DataBase 접근하기 위한 사용자명
- password : DataBase 접근하기 위한 암호
- defaultAutoCommit : DataBase로 부터 리턴된 connection에 대한 auto-commit 여부를 설정
- poolPreparedStatements : PreparedStatement 사용여부

테스트 코드

```
@Resource(name = "dataSource")
DataSource dataSource;

@Test
public void testDbcpDataSource() throws Exception {
    assertNotNull(dataSource);
    assertEquals("org.apache.commons.dbcp.BasicDataSource", dataSource.getClass().getName());
    Connection con = null;    Statement stmt = null;    ResultSet rs = null;
    try {
        con = dataSource.getConnection();
        assertNotNull(con);
        stmt = con.createStatement();
        rs = stmt.executeQuery("select 'x' as x from dual");
        while (rs.next()) { assertEquals("x", rs.getString(1)); }
    }
}
```


❑ C3P0 DataSource

C3P0 DataSource 설정

```
<bean id= "dataSource" class= "com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method= "close">
  <property name= "driverClass" value= "${driver}" />
  <property name= "jdbcUrl" value= "${dburl}" />
  <property name= "user" value= "${username}" />
  <property name= "password" value= "${password}" />
  <property name= "initialPoolSize" value= "3" />
  <property name= "minPoolSize" value= "3" />
  <property name= "maxPoolSize" value= "50" />
  <property name= "idleConnectionTestPeriod" value= "200" />
  <property name= "acquireIncrement" value= "1" />
  <property name= "maxStatements" value= "0" />
  <property name= "numHelperThreads" value= "3" />
</bean>
```

- driverClass : jdbc driver
- jdbcUrl : DB URL
- user : 사용자명
- password : 암호
- initialPoolSize : 풀 초기값
- maxPoolSize : 풀 최소값
- idleConnectionTestPeriod : idle상태 점검시간
- acquireIncrement : 증가값
- maxStatements : 캐쉬유지여부
- numHelperThreads : HelperThread 개수

테스트 코드

```
@Resource(name = "dataSource")
DataSource dataSource;

@Test
public void testC3p0DataSource() throws Exception {
    assertNotNull(dataSource);    assertEquals("com.mchange.v2.c3p0.ComboPooledDataSource", dataSource.getClass().getName());
    Connection con = null;    Statement stmt = null;    ResultSet rs = null;
    try {
        con = dataSource.getConnection();    assertNotNull(con);
        stmt = con.createStatement();
        rs = stmt.executeQuery("select 'x' as x from dual");
        while (rs.next()) { assertEquals("x", rs.getString(1)); } } }
```

❑ JNDI DataSource

JNDI DataSource 설정 1

```
<jee:jndi-lookup id="dataSource" jndi-name="${jndiName}" resource-ref="true">
  <jee:environment>
    java.naming.factory.initial=${jeus.java.naming.factory.initial}
    java.naming.provider.url=${jeus.java.naming.provider.url}
  </jee:environment>
</jee:jndi-lookup>
```

JNDI DataSource 설정 2

```
<util:properties id="jndiProperties" location="classpath:/META-INF/spring/jndi.properties" />
<jee:jndi-lookup id="dataSource" jndi-name="${jndiName}" resource-ref="true" environment-ref="jndiProperties" />
```

테스트 코드

```
@Resource(name = "dataSource")
DataSource dataSource;

@Test
public void testIndiJeusDataSource() throws Exception {
    assertNotNull(dataSource);
    assertEquals("jeus.jdbc.connectionpool.DataSourceWrapper", dataSource.getClass().getName());
    //assertEquals("weblogic.jdbc.common.internal.RmiDataSource_922_WLStub", dataSource.getClass().getName());

    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        con = dataSource.getConnection();
        assertNotNull(con);
        stmt = con.createStatement();
        rs = stmt.executeQuery("select 'x' as x from dual");
        while (rs.next()) {
            assertEquals("x", rs.getString(1));
        }
    }
}
```

❑ Commons DBCP

- <http://commons.apache.org/dbcp/>

❑ C3P0

- <http://www.mchange.com/projects/c3p0/index.html>

□ 서비스 개요

- JDBC 를 사용한 Data Access를 추상화하여 간편하고 쉽게 사용할 수 있는 Data Mapper framework 인 iBATIS 를 Data Access 기능의 기반 오픈 소스로 채택
- iBATIS 를 사용하면 관계형 데이터베이스에 액세스하기 위해 필요한 일련의 자바 코드 사용을 현저히 줄일 수 있으며 간단한 XML 기술을 사용하여 SQL 문을 JavaBeans (또는 Map) 에 간편하게 매핑할 수 있음
- Data Access 서비스는 다양한 데이터베이스 솔루션 및 데이터베이스 접근 기술에 일관된 방식으로 대응하기 위한 서비스
 - 데이터를 조회하거나 입력, 수정, 삭제하는 기능을 수행하는 메커니즘을 단순화함
 - 데이터베이스 솔루션이나 접근 기술이 변경될 경우에도 데이터를 다루는 시스템 영역의 변경을 최소화할 수 있도록 데이터베이스와의 접점을 추상화함
 - 추상화된 데이터 접근 방식을 템플릿(Template)으로 제공함으로써, 개발자들의 업무 효율을 향상시킴.

❑ iBatis는 단순성이라는 사상을 강조한 퍼시스턴스 프레임워크로, SQL 맵을 이용하여 반복적이고 복잡한 DB 작업 코드를 최소화함

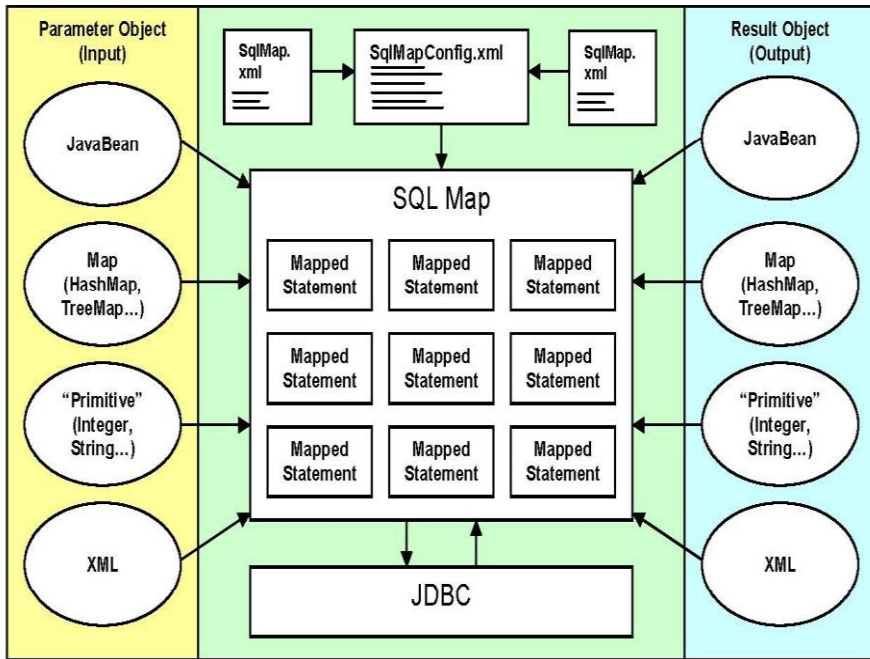
- 단순성이라는 사상을 강조하여, XML을 이용하여 Stored Procedure 혹은 SQL 문과 자바 객체간의 매핑을 지원
- 2001년 Clinton Begin (Apache 소프트웨어 재단)에 의해 개발된 퍼시스턴스 프레임워크

항목	iBatis	Hibernate	비고
응답 지연 시간 (Round Trip Delay Time)	짧다	길다	Hibernate의 경우, 쿼리 자동생성 등의 기능으로 인해 다소 시간 소요 길다.
유연성 (Flexibility)	우수	미흡	
학습 곡선 (Learning Curve)	작다	크다	iBatis가 보다 JDBC와 유사하기 때문
SQL 지식	높아야 함	별로 필요치 않음	

* "Performance Comparison of Persistence Frameworks," Sabu M. Thampi, Ashwin a K. (2007)

❑ iBatis는 소스코드 외부에 정의된 SQL문과 설정 정보를 바탕으로, 객체와 테이블 간의 매핑 기능을 제공

- iBATIS Data Mapper API 는 XML을 사용하여 SQL문과 객체 매핑 정보를 간편하게 기술할 수 있도록 지원
- 자바빈즈 객체와 Map 구현체, 다양한 원시 래퍼 타입(String, Integer..) 등을 PreparedStatement 의 파라미터나 ResultSet에 대한 결과 객체로 쉽게 매핑해 줌



구성요소	설명
SqlMapConfig XML File	- iBatis 동작을 위한 DataSource, Data Mapper 및 Thread Management 등과 같은 공통 설정
SqlMap XML File	- XML 방식으로 실행할 SQL문과 매핑 정보를 설정
SQL Map	- iBatis는 PreparedStatement 인스턴스를 생성하고, 제공된 파라미터 객체를 사용해서 파라미터를 셋팅한 후, Statement를 실행하고 ResultSet으로부터 결과 객체를 생성
Mapped Statement	- Parameter 객체와 Result 객체를 이용하여 SQL Statement로 치환
Parameter Object	- 파라미터 객체는 JavaBean, Map, Primitive 객체로서, 입력값을 셋팅하기 위해 사용되는 객체
Result Object	- 결과 객체는 JavaBean, Map, Primitive 객체로서, 쿼리문의 결과값을 담는 객체

□ 주요 기능

- 추상화된 접근 방식 제공
 - JDBC 데이터 액세스에 대한 추상화된 접근 방식으로, 간편하고 쉬운 API, 자원 연결/해제, 공통 에러 처리 등을 통합 지원함
- **자바 코드로부터 SQL문 분리**
 - 소스코드로부터 SQL문을 분리하여 별도의 repository(의미있는 문법의 XML)에 유지하고 이에 대한 빠른 참조구조를 내부적으로 구현하여 관리/유지보수/튜닝의 용이성을 보장함.
- **SQL문 자동 실행, 입/출력 파라미터 자동 바인딩 지원**
 - 쿼리문의 입력 파라미터에 대한 바인딩과 실행결과 resultset 의 가공(맵핑) 처리시 객체(VO, Map, List) 수준의 자동화를 지원함
- **Dynamic SQL 지원**
 - 코드 작성, API 직접 사용없이 입력 조건에 따른 동적인 쿼리문 변경을 지원함
- 다양한 DB 처리 지원
 - 기본 질의 외에 Batch SQL, Paging, Callable Statement, BLOB/CLOB 등 다양한 DB처리를 지원함

❑ iBatis를 사용한 Persistence Layer 개발 순서

1) [iBatis 설정 1] SQL Mapping XML 파일 작성

- 실행할 SQL문과 관련 정보 설정
- SELECT/INSERT/UPDATE/DELETE, Parameter/Result Object, Dynamic SQL 등

2) [iBatis 설정 2] iBatis Configuration XML 파일 작성

- iBatis 동작에 필요한 옵션을 설정
- <sqlMap>: SQL Mapping XML 파일의 위치

3) [스프링연동 설정] SqlMapClientFactoryBean 정의

- Spring와 iBatis 연동을 위한 설정
- 역할) iBatis 관련 메서드 실행을 위한 SqlMapClient 객체를 생성
- dataSource: DB Connection 생성
- configLocation: iBatis Configuration XML 파일의 위치
- mappingLocations: 모든 SQL Mapping XML 파일을 일괄 지정 가능

4) DAO 클래스 작성

- 실행할 SQL문을 호출하기 위한 메서드 구현: SQL Mapping XML 내에 정의한 각 Statement id를 매개변수로 전달
- 규칙) SqlMapClientDaoSupport를 상속하는 EgovAbstractDAO 클래스를 상속받아 확장/구현

❑ [iBatis 설정 1] SQL Mapping XML 파일 작성 (1/2)

- 실행할 SQL문과 Parameter Object와 Result Object, Dynamic SQL 등을 설정

SQL Mapping XML 설정

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.com//DTD SQL Map 2.0//EN" "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="Dept">
  <typeAlias alias="deptVO" type="x.y.z.DeptVO" />
  <resultMap id="deptResult" class="deptVO">
    <result property="deptNo" column="DEPT_NO" />
    <result property="deptName" column="DEPT_NAME" />
    <result property="loc" column="LOC" />
  </resultMap>
  <select id="selectDept" parameterClass="deptVO" resultMap="deptResult">
    <![CDATA[
      select DEPT_NO, DEPT_NAME, LOC
      from DEPT
      where DEPT_NO = #deptNo#
    ]]>
  </select>
  <!-- 참고) Result Object의 변수명과 Table의 컬럼명이 동일하다면, resultClass 속성을 이용하여 직접 deptVO를 지정해도 된다
       parameterMap 속성 사용은 권장하지 않음 -->

  <insert id="insertDept" parameterClass="deptVO">
    <![CDATA[
      insert into DEPT (DEPT_NO,DEPT_NAME,LOC)
      values (#deptNo#,#deptName#,#loc#)
    ]]>
  </insert>
```

루트요소 <sqlMap>, namespace는 옵션

현 파일 내에서 자주 사용되는 객체에 alias를 지정하여 풀네임 대신 사용

객체 변수명과 테이블 컬럼명이 다른 경우, 두 관계를 정의하기 위한 요소로, SQL문 외부에 정의하고 SQL문에서 resultMap 속성으로 참조

ResultSet에 저장된 쿼리 결과값을 컬럼명=변수명 기준으로, Result Object에 바인딩할 때 사용한다. 이 예제에서는 객체 변수명과 테이블 컬럼명이 다르므로, resultMap 요소를 이용하여 변수명-컬럼명 관계 정보를 제공

데이터 입력을 위해 Parameter Object로 부터 값을 전달받아 SQL문에 바인딩

❑ [iBatis 설정 1] SQL Mapping XML 파일 작성 (2/2)

- 실행할 SQL문과 Parameter Object와 Result Object, Dynamic SQL 등을 설정

SQL Mapping XML 설정

```
<update id="updateDept" parameterClass="deptVO">
  update DEPT
  set DEPT_NAME = #deptName#, LOC = #loc#
  where DEPT_NO = #deptNo#
</update>
```

```
<delete id="deleteDept" parameterClass="deptVO">
  delete from DEPT
  where DEPT_NO = #deptNo#
</delete>
```

```
<sql id="columns">DEPT_NO, DEPT_NAME, LOC</sql>
```

반복되는 구문을 재사용하기 위한 요소로,
다른 Statement에서 <include>의 refid 속성으로 참조

```
<select id="selectDeptList" parameterClass="deptVO" resultMap="deptResult">
  select <include refid="columns" />
  from DEPT
```

```
<dynamic prepend="where">
  <isNotNull prepend="and" property="deptNo">
    DEPT_NO = #deptNo#
  </isNotNull>
  <isNotNull prepend="and" property="deptName">
    DEPT_NAME LIKE '%' || #deptName# || '%'
  </isNotNull>
</dynamic>
</select>
</sqlMap>
```

SQL문 실행 전, 다양한 조건에 따라 동적으로 SQL문을 변환하여 실행하는 요소로,
deptNo 혹은 **deptName** 변수가 NotNull이면, 두 변수값을 파라미터로 WHERE
조건절을 만들고, 모두 Null이면 SELECT절~FROM절까지 수행한다

❑ [iBatis 설정 2] iBatis Configuration XML 파일 (1/3)

- iBatis 공통 설정 파일로, SqlMapClient 설정관련 상세 내역을 제어할 수 있는 메인 설정
- 주로 transaction 관리 관련 설정 및 다양한 옵션 설정, Sql Mapping 파일들에 대한 path 설정 등을 포함한다.

iBatis Configuration XML 설정	
<pre><?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE sqlMapConfig PUBLIC "-//IBATIS.com//DTD SQL Map Config 2.0//EN" "http://www.ibatis.com/dtd/sql-map-config-2.dtd"> <sqlMapConfig></pre>	루트요소 <sqlMapConfig>
<pre> <sqlMap resource="META-INF/sqlmap/mappings/testcase-basic.xml"/> </sqlMap></pre>	SQL Mapping File의 위치 설정 (필수)
<pre> <settings cacheModelsEnabled="true" enhancementEnabled="true" lazyLoadingEnabled="true" maxRequests="128" maxSessions="10" maxTransactions="5" useStatementNamespaces="false" defaultStatementTimeout="1"/></pre>	SqlMapClient 객체 생성 옵션
<pre> <typeHandler javaType="java.util.Calendar" jdbcType="TIMESTAMP" callback="egovframework.rte.psl.dataaccess.typehandler.CalendarTypeHandler"/></pre>	javaType과 jdbcType 간 타입 핸들링
<pre> <properties resource="META-INF/spring/jdbc.properties"/></pre>	placeholder 사용을 위한 Properties 파일 참조 설정
<pre> <transactionManager type="JDBC"> <dataSource type="DBCP"> <property name="driverClassName" value="\${driver}" /> <property name="url" value="\${dburl}" /> <property name="username" value="\${username}" /> <property name="password" value="\${password}" /> <!-- OPTIONAL PROPERTIES BELOW --></pre>	TransactionManager와 DataSource 설정

❑ [iBatis 설정 2] iBatis Configuration XML 파일 (2/3)

```

<property name="maxActive" value="10" />
<property name="maxIdle" value="5" />
<property name="maxWait" value="60000" />
<!-- validation query -->
<!--<property name="validationQuery" value="select * from DUAL" />-->
<property name="logAbandoned" value="false" />
<property name="removeAbandoned" value="false" />
<property name="removeAbandonedTimeout" value="50000" />
<property name="Driver.DriverSpecificProperty" value="SomeValue" />
</dataSource>
</transactionManager>
...
</sqlMapConfig>

```

요소	설명
properties	표준 java properties (key=value 형태)파일에 대한 연결을 지원하며 설정 파일내에서 \${key} 와 같은 properties 형태로 외부화 해놓은 실 제의 값(여기서는 DB 접속 관련 driver, url, id/pw)을 참조할 수 있다. resource 속성으로 classpath 지정 가능, url 속성으로 유효한 URL 상에 있는 자원을 지정 가능
settings	이 설정 파일을 통해 생성된 SqlMapClient instance 에 대하여 다양한 옵션 설정을 통해 최적화할 수 있도록 지원한다. 모든 속성은 선택사 항(optional) 이다.
typeHandler	javaType과 jdbcType 일치를 위해 TypeHandler 구현체를 등록하여 사용할 수 있다.
sqlMap	매핑할 SQL 구문이 정의된 파일을 지정한다.

❑ [iBatis 설정 2] iBatis Configuration XML 파일 (3/3)

- <settings>에서 사용 가능한 속성들

속성	설명	예시 및 디폴트값
maxRequests	같은 시간대에 SQL 문을 실행할 수 있는 thread 의 최대 갯수 지정	maxRequests="256", 512
maxSessions	주어진 시간에 활성화될 수 있는 session(또는 client) 수 지정	maxSessions="64", 128
maxTransactions	같은 시간대에 SqlMapClient.startTransaction() 에 들어갈 수 있는 최대 갯수 지정	maxTransactions="16", 32
cacheModelsEnabled	SqlMapClient 의 모든 cacheModel 에 대한 사용 여부를 global 하게 지정	cacheModelsEnabled="true", true (enabled)
lazyLoadingEnabled	SqlMapClient 의 모든 lazy loading 에 대한 사용 여부를 global 하게 지정	lazyLoadingEnabled="true", true (enabled)
enhancementEnabled	runtime bytecode enhancement 기술 사용 여부 지정	enhancementEnabled="true", false (disabled)
useStatementNamespaces	Statement 호출 시 namespace값 사용 여부	useStatementNamespaces="false", false (disabled)
defaultStatementTimeout	모든 JDBC 쿼리에 대한 timeout 시간(초) 지정, 각 statement 의 설정으로 override 가능함. 모든 driver가 이 설정을 지원하는 것은 아님에 유의할 것.	지정하지 않는 경우 timeout 없음(cf. 각 statement 설정에 따라)
classInfoCacheEnabled	introspected(java 의 reflection API에 의해 내부 참조된) class의 캐쉬를 유지할지에 대한 설정	classInfoCacheEnabled="true", true (enabled)
statementCachingEnabled	prepared statement 의 local cache 를 유지할지에 대한 설정	statementCachingEnabled="true", true (enabled)

□ [스프링연동 설정] SqlMapClientFactoryBean 정의

- Spring과 iBatis 연동을 위한 설정으로, iBatis 관련 메서드 실행을 위해 SqlMapClient 객체가 필요
- 스프링에서 SqlMapClient 객체를 생성하고 관리할 수 있도록, SqlMapClientFactoryBean을 정의
 - Id와 class는 고정값
 - dataSource : 스프링에서 설정한 Datasource Bean id를 설정하여 iBatis가 DataSource를 사용하게 한다.
 - configLocation : iBatis Configuration XML 파일이 위치하는 곳을 설정한다.
 - mappingLocations : SQL Mapping XML 파일을 일괄 지정할 수 있다. 단, Configuration 파일에 중복 선언할 수 없다.
 - 실행환경 3.5 부터는 Spring 4 변경사항에 의해 org.springframework.orm.ibatis.SqlMapClientFactoryBean 클래스가 egovframework.rte.psl.orm.ibatis.SqlMapClientFactoryBean 로 변경됨.

SqlMapClientFactoryBean 설정

```
<!-- SqlMap setup for iBATIS Database Layer -->
<bean id="sqlMapClient" class="egovframework.rte.psl.orm.ibatis.SqlMapClientFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="configLocation" value="classpath:/egovframework/sqlmap/rte/sql-map-config.xml"/>
  <!-- <property name="mappingLocations" value="classpath:/META-INF/sqlmap/mappings/*.xml" /> -->
</bean>
```

❑ iBatis를 활용한 자바클래스 작성

- EgovAbstractDAO 클래스를 상속받아 DAO 클래스를 작성

DAO class 샘플 코드

```
@Repository("empDAO")
public class EmpMapper extends EgovAbstractDAO {

    public void insertEmp(EmpVO vo) {
        insert("insertEmp", vo);
    }

    public int updateEmp(EmpVO vo) {
        return update("updateEmp", vo);
    }

    public int deleteEmp(EmpVO vo) {
        return delete("deleteEmp", vo);
    }

    public EmpVO selectEmp(EmpVO vo) {
        return (EmpVO) selectByPk("selectEmp", vo);
    }

    public List<EmpVO> selectEmpList(EmpVO searchVO) {
        return list("selectEmpList", searchVO);
    }
}
```

EgovAbstractDAO는 SqlSessionDaoSupport의 하위 클래스로, SqlMapClient 설정과 메소드 호출의 편리함을 제공한다

SQL Mapper XML 내에 정의한 각 Statement id를 이용하여 실행할 SQL문을 호출하도록 작성한다

□ 세부사항 설명

- iBATIS Configuration
 - iBATIS 의 메인 설정 파일인 SQL Map XML Configuration 파일(이하 sql-map-config.xml 설정 파일) 작성과 상세한 옵션 설정
- Data Type
 - 데이터베이스를 이용하여 데이터를 저장하고 조회할 때 Java 어플리케이션에서의 Type 과 DBMS 에서 지원하는 관련 매핑 jdbc Type 의 정확한 사용이 필요
- parameterMap
 - 해당 요소로 SQL 문 외부에 정의한 입력 객체의 속성에 대한 name 및 javaType, jdbcType 을 비롯한 옵션을 설정할 수 있는 매핑 요소
- Inline parameters
 - prepared statement 에 대한 바인드 변수 매핑 처리를 위한 parameterMap 요소(SQL 문 외부에 정의한 입력 객체 property name 및 javaType, jdbcType 을 비롯한 옵션을 설정매핑 요소) 와 동일한 기능을 처리하는 간편한 방법
- resultMap
 - resultMap 은 SQL 문 외부에 정의한 매핑 요소로, result set 으로부터 어떻게 데이터를 뽑아낼지, 어떤 칼럼을 어떤 property로 매핑할지에 대한 상세한 제어를 가능케 해줌
- Dynamic SQL
 - SQL 문의 동적인 변경에 대한 상대적으로 유연한 방법을 제공하는 iBATIS 의 Dynamic 요소

□ 세부사항 설명 - Data Type

- 어플리케이션을 작성할 때 Data Type 에 대한 올바른 사용과 관련 처리는 매우 중요함
- 데이터베이스를 이용하여 데이터를 저장하고 조회할 때 **Java 어플리케이션에서의 Type 과 DBMS 에서 지원하는 관련 매핑 jdbc Type 의 정확한 사용이 필요함**
- **기본 Data Type 사용 방법**
 - iBATIS SQL Mapper 프레임워크는 Java 어플리케이션 영역의 표준 JavaBeans 객체(또는 Map 등)의 각 Attribute 에 대한 Java Type 과 JDBC 드라이버에서 지원하는 각 DBMS의 테이블 칼럼에 대한 Data Type 의 매핑을 기반으로 parameter / result 객체에 대한 바인딩/매핑 을 처리함
 - 각 javaType 에 대한 매칭되는 jdbcType 은 일반적인 Ansi SQL 을 사용한다고 하였을 때 아래에서 대략 확인할 수 있음
 - 특정 DBMS 벤더에 따라 추가적으로 지원/미지원 하는 jdbcType 이 다를 수 있고, 또한 같은 jdbcType 을 사용한다 하더라도 타입에 따른 사용 가능한 경계값(boundary max/min value)은 다를 수 있음

□ 세부사항 설명 - parameterMap

- 해당 요소로 SQL 문 외부에 정의한 입력 객체의 속성에 대한 name 및 javaType, jdbcType 을 비롯한 옵션을 설정할 수 있는 매핑 요소임.
- JavaBeans 객체(또는 Map 등)에 대한 prepared statement 에 대한 바인드 변수 매핑을 처리할 수 있음.
- 유사한 기능을 처리하는 parameterClass 나 Inline Parameter 에 비해 많이 사용되지 않지만 더 기술적인 (descriptive) parameterMap(예를 들어 stored procedure 를 위한) 이 필요함 XML 의 일관된 사용과 순수성을 지키고자 할때 좋은 접근법이 될 수도 있음.
- Dynamic 요소와 함께 사용될 수 없고 바인드 변수의 갯수와 순서를 정확히 맞춰야 하는 불편이 있는 등 일반적으로 사용을 추천하지 않음.

□ 세부사항 설명 - Inline Parameters

- prepared statement 에 대한 바인드 변수 매핑 처리를 위한 **parameterMap** 요소
- SQL 문 외부에 정의한 입력 객체 property name 및 javaType, jdbcType 을 비롯한 옵션을 설정매핑 요소와 동일한 기능을 처리하는 간편한 방법을 **Inline Parameters** 방법으로 제공 보통 parameterClass 로 명시된 입력 객체에 대해 바인드 변수 영역을 간단한 #property# 노테이션으로 나타내는 Inline Parameter 방법은 기존 parameterMap 에서의 '?' 와 이의 순서를 맞춘 외부 parameterMap 선언으로 처리하는 방법에 비해 많이 사용되고 일반적으로 추천하는 방법
- Dynamic 요소와 함께 사용될 수 있고 별도의 외부 매핑 정의없이 바인드 변수 처리가 필요한 위치에 해당 property 를 직접 사용 가능하며, 필요한 경우 jdbcType 이나 nullValue 를 간단한 추가 노테이션과 같이 지정할 수 있음
 - (ex. #empName:VARCHAR:blank#)
 - (ex. #comm,javaType=decimal,jdbcType=NUMERIC,nullValue=-99999#)
,(comma) 로 구분된 필요한 속성=값 을 상세하게 기술할 수도 있음.

□ 세부사항 설명 - resultMap

- resultMap 은 SQL 문 외부에 정의한 매핑 요소
- result set 으로부터 어떻게 데이터를 뽑아낼지, 어떤 칼럼을 어떤 property로 매핑 할 지에 대한 상세한 제어를 가능케 해줌.
- resultMap 은 일반적으로 가장 많이 사용되는 중요한 매핑 요소로 **resultClass 속성**을 이용한 자동 매핑 접근법에 비교하여 칼럼 타입의 지시, null value 대체값, typeHandler 처리, complex property 매핑 (다른 JavaBean, Collections 등을 포함하는 복합 객체) 등을 허용함.

□ 세부사항 설명 - Dynamic SQL

- 일반적으로 JDBC API 를 사용한 코딩에서 한번 정의한 쿼리문을 최대한 재사용하고자 하나 단순 파라미터 변수의 값만 변경하는 것으로 해결하기 어렵고 다양한 조건에 따라 조금씩 다른 쿼리의 실행이 필요한 경우 많은 if~else 조건 분기의 연결이 필요한 문제가 있음.
 - 1-1 Dynamic SQL mapping xml 파일

SQL Mapping XML - Dynamic SQL 설정

```
<typeAlias alias="jobHistVO" type="egovframework.rte.dataaccess.vo.JobHistVO" />

<select id="selectJobHistListUsingDynamicElement" parameterClass="jobHistVO" resultClass="jobHistVO">
<![CDATA[
select EMP_NO    as empNo,
       START_DATE as startDate,
       END_DATE   as endDate,
       JOB        as job,
       SAL        as sal,
       COMM       as comm,
       DEPT_NO    as deptNo
from   JOBHIST
]]>
<dynamic prepend="where">
<isNotNull property="empNo" prepend="and">
EMP_NO = #empNo#
</isNotNull>
</dynamic>
order by EMP_NO, START_DATE
</select>
```

□ 세부사항 설명 - Dynamic SQL

- Unary 비교 연산 (1/2)
 - 1-2 Sample Unary 비교 연산

SQL Mapping XML - Dynamic SQL 설정

```
<typeAlias alias="egovMap" type="egovframework.rte.psl.dataaccess.util.EgovMap" />

<select id="selectDynamicUnary" parameterClass="map" remapResults="true" resultClass="egovMap">
select
<dynamic>
<isEmpty property="testEmptyString">
'empty String' as IS_EMPTY_STRING
</isEmpty>
<isNotEmpty property="testEmptyString">
'not empty String' as IS_EMPTY_STRING
</isNotEmpty>
..
<isPropertyAvailable prepend="," property="testProperty">
'testProperty Available' as TEST_PROPERTY_AVAILABLE
</isPropertyAvailable>
<isNotPropertyAvailable prepend="," property="testProperty">
'testProperty Not Available' as TEST_PROPERTY_AVAILABLE
</isNotPropertyAvailable>
</dynamic>
from dual
</select>
```

□ 세부사항 설명 - Dynamic SQL

- Unary 비교 연산 (2/2)
 - Unary 비교 연산 태그

태그	설명
isEmpty	Collection, String(또는 String.valueOf()) 대상 속성이 null 이거나 empty("") 또는 size() < 1) 인 경우 true
isNotEmpty	Collection, String(또는 String.valueOf()) 대상 속성이 not null 이고 not empty("") 또는 size() < 1) 인 경우 true
isNull	대상 속성이 null 인 경우 true
isNotNull	대상 속성이 not null 인 경우 true
isPropertyAvailable	파라미터 객체에 대상 속성이 존재하는 경우 true
isNotPropertyAvailable	파라미터 객체에 대상 속성이 존재하지 않는 경우 true

- Unary 비교 연산 태그 속성

속성	설명
prepend	동적 구문 앞에 추가되는 override 가능한 SQL 영역.
property	필수. 파라미터 객체의 어떤 property 에 대한 체크인지 지정.
removeFirstPrepend	첫번째로 내포될 내용을 생성하는 태그의 prepend 를 제거할지 여부(true/false)
open	전체 결과 구문에 대한 시작 문자열
close	전체 결과 구문에 대한 닫는 문자열

□ 세부사항 설명 - Dynamic SQL

- Binary 비교 연산 (1/2)
 - 1-3 Sample Binary 비교 연산

SQL Mapping XML - Dynamic SQL 설정

```
<typeAlias alias="egovMap" type="egovframework.rte.psl.dataaccess.util.EgovMap" />

<select id="selectDynamicBinary" parameterClass="map" resultMap="true" resultClass="egovMap">
  select
    <dynamic>
      <isEqual property="testString" compareValue="test">
        '$testString$' as TEST_STRING, 'test : equals' as IS_EQUAL
      </isEqual>
      <isNotEqual property="testString" compareValue="test">
        '$testString$' as TEST_STRING, 'test : not equals' as
        IS_EQUAL
      </isNotEqual>
      ..
      <isLessThan property="testOtherString" prepend="," compareProperty="testString">
        '$testOtherString$' <![CDATA[<]]>
        '$testString$' as COMPARE_PROPERTY_LESS_THAN
      </isLessThan>
    </dynamic>
  from dual
</select>
```

□ 세부사항 설명 - Dynamic SQL

- Binary 비교 연산 (2/2)

- Binary 비교 연산 태그

태그	설명
isEqual	대상 속성이 compareValue 값 또는 compareProperty 로 명시한 대상 속성 값과 같은 경우 true
isNotEqual	대상 속성이 compareValue 값 또는 compareProperty 로 명시한 대상 속성 값과 다른 경우 true
isGreaterEqual	대상 속성이 compareValue 값 또는 compareProperty 로 명시한 대상 속성 값보다 크거나 같은 경우 true
isGreaterThan	대상 속성이 compareValue 값 또는 compareProperty 로 명시한 대상 속성 값보다 큰 경우 true
isLessEqual	대상 속성이 compareValue 값 또는 compareProperty 로 명시한 대상 속성 값보다 작거나 같은 경우 true
isLessThan	대상 속성이 compareValue 값 또는 compareProperty 로 명시한 대상 속성 값보다 작은 경우 true

- Binary 비교 연산 태그 속성

속성	설명
prepend	동적 구문 앞에 추가되는 override 가능한 SQL 영역.
property	필수. 파라미터 객체의 어떤 property 에 대한 비교인지 지정.
compareProperty	파라미터 객체의 다른 property 와 대상 property 값을 비교하고자 할 경우 지정. (compareValue 가 없는 경우 필수)
compareValue	대상 property 와 비교될 값을 지정. (compareProperty 가 없는 경우 필수)
removeFirstPrepend	첫번째로 내포될 내용을 생성하는 태그의 prepend 를 제거할지 여부(true/false)
open	전체 결과 구문에 대한 시작 문자열
close	전체 결과 구문에 대한 닫는 문자열

□ 세부사항 설명 - Dynamic SQL

- ParameterPresent 비교(1/2)
 - 1-4 Sample ParameterPresent 비교

SQL Mapping XML - Dynamic SQL 설정

```
<typeAlias alias="egovMap" type="egovframework.rte.psl.dataaccess.util.EgovMap" />

<select id="selectDynamicParameterPresent" parameterClass="map" resultMap="true" resultClass="egovMap">
  select
    <isParameterPresent>
      'parameter object exist' as IS_PARAMETER_PRESENT
    </isParameterPresent>
    <isNotParameterPresent>
      'parameter object not exist' as IS_PARAMETER_PRESENT
    </isNotParameterPresent>
  from dual
</select>
```

□ 세부사항 설명 - Dynamic SQL

- ParameterPresent 비교 (2/2)
 - ParameterPresent 비교 태그

태그	설명
isParameterPresent	파라미터 객체가 전달된(not null) 경우 true
isNotParameterPresent	파라미터 객체가 전달되지 않은(null) 경우 true

- ParameterPresent 비교 태그 속성

속성	설명
prepend	동적 구문 앞에 추가되는 override 가능한 SQL 영역.
removeFirstPrepend	필수. 파라미터 객체의 어떤 property 에 대한 비교인지 지정.
open	전체 결과 구문에 대한 시작 문자열
close	전체 결과 구문에 대한 닫는 문자열

□ 세부사항 설명 - Dynamic SQL

- iterate 연산 (1/2)
 - 1-5 Sample iterate 연산

SQL Mapping XML - Dynamic SQL 설정

```
<typeAlias alias="jobHistVO" type="egovframework.rte.psl.dataaccess.vo.JobHistVO" />
<typeAlias alias="empIncludesEmpListVO" type="egovframework.rte.psl.dataaccess.vo.EmpIncludesEmpListVO" />

<select id="selectJobHistListUsingDynamicIterate" parameterClass="empIncludesEmpListVO" resultClass="jobHistVO">
<![CDATA[
select EMP_NO    as empNo,
       START_DATE as startDate,
       END_DATE   as endDate,
       JOB        as job,
       SAL        as sal,
       COMM       as comm,
       DEPT_NO    as deptNo
from   JOBHIST
]]>
<dynamic prepend="where">
  <iterate property="empList" open="EMP_NO in (" conjunction="," close=")">
    #empList[].empNo#
  </iterate>
</dynamic>
order by EMP_NO, START_DATE
</select>
```

□ 세부사항 설명 - Dynamic SQL

- iterate 연산 (2/2)
 - iterate 연산 태그

태그	설명
iterate	collection 형태의 대상 객체에 대하여 포함하고 있는 각 개별 요소만큼 반복 루프를 돌며 해당 내용을 수행함

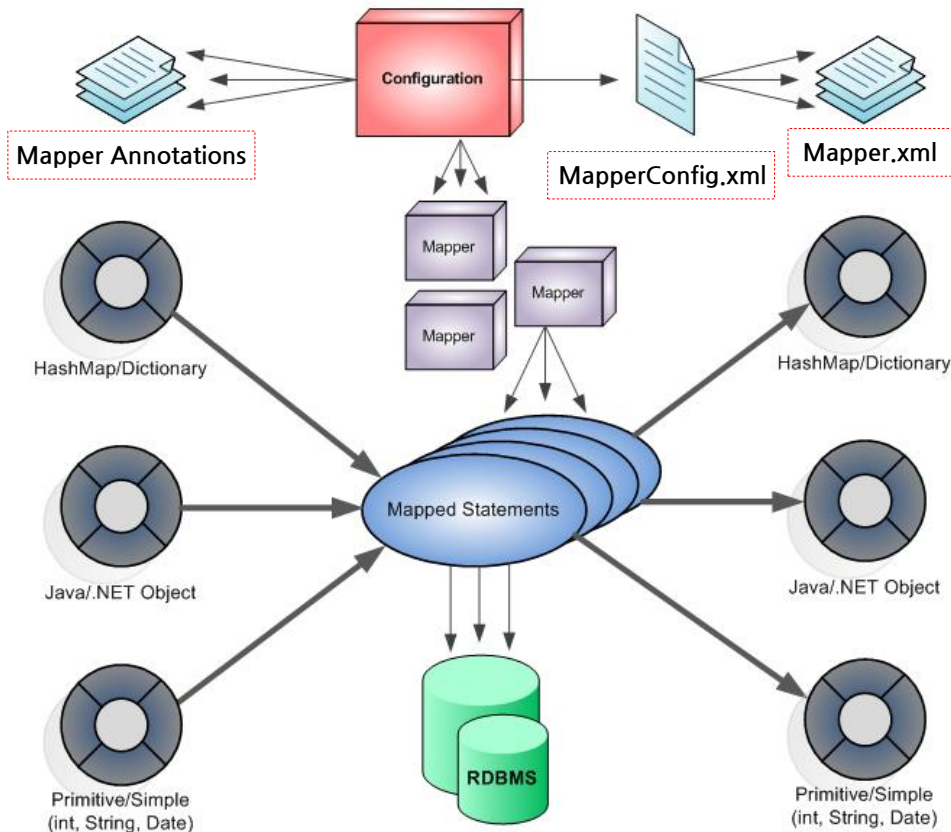
- iterate 연산 태그 속성

속성	설명
prepend	동적 구문 앞에 추가되는 override 가능한 SQL 영역.
property	필수. 파라미터 객체의 어떤 property 에 대한 비교인지 지정.
removeFirstPrepend	첫번째로 내포될 내용을 생성하는 태그의 prepend 를 제거할지 여부(true/false/iterate)
open	전체 결과 구문에 대한 시작 문자열
Close	전체 결과 구문에 대한 닫는 문자열
conjunction	각 iteration 사이에 적용될 문자열. AND, OR 연산자나 ',' 등의 구분자 필요시 유용함

LAB 204-iBatis 실습

❑ MyBatis 데이터 매퍼 서비스

- 개발자가 작성한 SQL문 혹은 저장프로시저 결과값을 자바 오브젝트에 자동 매핑하는 서비스
- 수동적인 JDBC 방식의 데이터 처리 작업 코드와는 달리 쿼리결과와 오브젝트 간 자동 매핑을 지원
- SQL문과 저장프로시저는 **XML 혹은 어노테이션 방식으로 작성 가능**



구성요소	설명
MapperConfig XML File	- MyBatis 동작을 위한 기본적인 설정을 공통으로 정의
Mapper XML File	- 실행할 SQL문 및 매핑 정보를 XML 방식으로 정의
Mapper Annotations	- 자바 코드 내에서 실행할 SQL문 및 매핑 정보를 어노테이션을 이용하여 정의
Parameter Object	- SQL문의 조건절에서 값을 비교하거나 INSERT, UPDATE절 등에서 필요한 입력값을 받아오기 위한 오브젝트
Result Object	- 쿼리 결과를 담아 리턴하기 위한 오브젝트

□ 주요 변경 사항

- iBatis의 SqlMapClient → SqlSession 변경

- SqlSession 인터페이스
 - MyBatis를 사용하기 위한 기본적인 인터페이스로, SQL문 처리를 위한 메서드를 제공
 - 구문 실행 메서드, 트랜잭션 제어 메서드 등 포함
 - selectList(), selectOne(), insert(), update(), delete(), commit(), rollback(), ...
 - SqlSessionFactory 클래스를 통해 MyBatis Configuration 정보에 해당 SqlSession 인스턴스를 생성

- 어노테이션 방식 설정 도입

- MyBatis는 본래 XML 기반의 프레임워크였으나, Mybatis 3.x 부터 어노테이션 방식의 설정을 지원
- Mapper XML File 내 SQL문 및 매핑 정보를, 자바 코드 내에서 어노테이션으로 그대로 적용 가능

- iBatis의 RowHandler → ResultHandler 변경

- ResultHandler 인터페이스
 - Result Object에 담겨 리턴된 쿼리 결과를 핸들링할 수 있도록 메서드 제공
 - 사용 예시) 대량의 데이터 처리 시, 처리 결과를 File로 출력하고자 할 때 혹은 Result Object의 형태를 Map 형태로 가져올 때

❑ Migrating from iBatis (1/2)

변경 또는 추가사항			iBatis 사용 시	MyBatis 사용 시
소스	패키지	변경	com.ibatis.*	org.apache.ibatis.*
	클래스	변경	SqlMapClient	SqlSession
		변경	SqlMapClientFactory	SqlSessionFactory
		변경	RowHandler	ResultHandler
Configuration XML File	DTD	변경	<!DOCTYPE sqlMapConfig PUBLIC "-//iBatis.com//DTD SQL Map Config 2.0//EN" "http://www.ibatis.com/dtd/sql-map-config-2.dtd">	<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
	요소와 속성	변경	< sqlMapConfig >	< configuration >
		변경	<settings ... />	<settings> <setting /> </settings>
		변경	<typeHandler callback="..." />	<typeHandlers> <typeHandler handler="..." /> </typeHandlers>
		추가	<transactionManager type="..."> <dataSource type="..." /> </transactionManager>	<environment id="..."> <transactionManager type="..." /> <dataSource type="..." /> </environment>
		추가		< typeAliases > <typeAlias alias="..." type="..." /> </typeAliases>
		변경	< sqlMap resource="..." />	< mappers > < mapper resource="..." /> </mappers>

❑ Migrating from iBatis (2/2)

변경 또는 추가사항			iBatis 사용 시	MyBatis 사용 시
Mapper XML File	DTD	변경	<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN" "http://www.ibatis.com/dtd/sql-map-2.dtd">	<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
	요소와 속성	변경	< sqlMap namespace="...">	< mapper namespace="...">
		변경	<cacheModel />	<cache />
		변경	-- SQL Statement <select id="..." resultClass resultMap ="..."> SELECT * FROM EMP </select> <insert id="..." parameterClass parameterMap ="..."> INSERT INTO EMP VALUES (#empNo# , #empName#); </insert>	<select id="..." resultType resultMap ="..."> SELECT * FROM EMP </select> <insert id="..." parameterType parameterMap ="..."> INSERT INTO EMP VALUES (#{empNo} , #{empName}); </insert>
		변경/추가	< dynamic > <isEqual /> <isNotNull /> ... </dynamic>	<if test="..." /> <choose> <when /> ... <otherwise /> <trim prefixOverrides suffixOverrides="..." /> <foreach collection="..." item="..." />
스프링연동	빈생성	변경	<bean id="sqlMapClient" class=" egovframework.rte.psl.orm.ibatis.SqlMapClientFactoryBean ">	<bean id="sqlSession" class=" org.mybatis.spring.SqlSessionFactoryBean ">
쿼리 호출	파라미터	추가	-- Statement ID로 실행할 SQL문을 호출 List list = ...selectList(queryId, parameterObject);	-- Statement ID로 실행할 SQL문을 호출 List list = selectList(queryId, parameterObject); -- 메서드명으로 실행할 쿼리호출 List list = ...selectList(parameterObject);

❑ MyBatis를 활용한 Persistence Layer 개발

1) [MyBatis 설정 1] SQL Mapper XML 파일 작성 설정

- 실행할 SQL문과 관련 정보 설정
- SELECT/INSERT/UPDATE/DELETE, Parameter/Result Object, Dynamic SQL 등

2) [MyBatis 설정 2] MyBatis Configuration XML 파일 작성

- MyBatis 동작에 필요한 옵션을 설정
- <mapper>: SQL Mapper XML 파일의 위치

3) [스프링연동 설정] SqlSessionFactoryBean 정의

- Spring와 MyBatis 연동을 위한 설정
- 역할) MyBatis 관련 메서드 실행을 위한 SqlSession 객체를 생성
- dataSource, configLocation, mapperLocations 속성 설정

4) DAO 클래스 작성

- 방법1) SqlSessionDaoSupport를 상속하는 EgovAbstractMapper 클래스를 상속받아 확장/구현
 - 실행할 SQL문을 호출하기 위한 메서드 구현: SQL Mapping XML 내에 정의한 각 Statement id를 매개변수로 전달
- 방법2) DAO 클래스를 Interface로 작성하고, 각 Statement id와 메서드명을 동일하게 작성 (Mapper Interface 방식)
 - Annotation을 이용한 SQL문 작성 가능
 - 메서드명을 Statement id로 사용하기 때문에, 코드 최소화 가능

□ [MyBatis 설정 1] SQL Mapper XML 파일 작성 (1/5)

- 실행할 SQL문과 Parameter Object와 Result Object 정보 등을 설정

SQL Mapper XML 설정

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="Dept">
  <resultMap id="deptResult" type="deptVO">
    <result property="deptNo" column="DEPT_NO" />
    <result property="deptName" column="DEPT_NAME" />
    <result property="loc" column="LOC" />
  </resultMap>

  <sql id="columns"> DEPT_NO, DEPT_NAME, LOC</sql>

  <select id="selectDept" parameterType="deptVO" resultMap="deptResult">
    <![CDATA[
      select <include refid="columns" />
      from DEPT
      where DEPT_NO = #{deptNo}
    ]]>
  </select>
</mapper>

```

루트요소 <sqlMap>, namespace는 각 mapper 파일을 구분하기 위한 속성

"deptVO"에 대한 Alias 설정은 Configuration 파일에 정의되어 있음

parameter/resultClass → parameter/resultType 변경됨

파라미터 바인딩 시, #property# → #{property} 변경됨

□ [MyBatis 설정 1] SQL Mapper XML 파일 작성 (2/5) - Dynamic SQL

- If

- if는 가장 많이 사용되는 Dynamic 요소로, test문의 true, false값에 따라 다양한 조건 설정이 가능
- SQL문의 다양한 위치에서 사용 가능하고, 선언된 if 조건에 따라 순서대로 test문을 수행

SQL Mapper XML 설정

```
<select id="selectEmpList" parameterType="empVO" resultType="empVO">
  <![CDATA[
    select EMP_NO as empNo,
           EMP_NAME as empName
    from EMP
    where JOB = 'Engineer'
  ]]>
  <if test="empNo != null">
    AND EMP_NO = #{empNo}
  </if>
  <if test="empName != null">
    AND EMP_NAME LIKE '%' || #{empName} || '%'
  </if>
</select>
```

❑ [MyBatis 설정 1] SQL Mapper XML 파일 작성 (3/5) - Dynamic SQL

- choose (when, otherwise)
 - 모든 조건을 적용하는 대신 한 가지 조건 만을 적용해야 할 필요가 있는 경우, choose 요소를 사용하며 이는 자바의 switch 구문과 유사한 개념임

SQL Mapper XML 설정

```
<select id="selectEmpList" parameterType="empVO" resultType="empVO">
  select *
  from EMP
  where JOB = 'Engineer'
  <choose>
    <when test="mgr != null">
      AND MGR like #{mgr}
    </when>
    <when test="empNo != null and empName != null">
      AND EMP_NAME like #{empName}
    </when>
    <otherwise>
      AND HIRE_STATUS = 'Y'
    </otherwise>
  </choose>
</select>
```

❑ [MyBatis 설정 1] SQL Mapper XML 파일 작성 (4/5) - Dynamic SQL

- trim (where, set)

- AND, OR, '와 같이 반복되는 문자를 자동적으로 trim(제거)
- 아래 예제의 <trim prefix="WHERE" prefixOverrides="AND|OR">은 <where>와 동일하게 동작

SQL Mapper XML 설정

```

<select id="selectEmpList" parameterType="empVO" resultType="empVO">
  select *
  from EMP
  where
  <if test="empNo != null">
    EMP_NO = #{empNo}
  </if>
  <if test="empName != null">
    AND EMP_NAME LIKE '%' || #{empName} || '%'
  </if>
</select>

<update id="updateEmp" parameterType="empVO">
  update EMP
  <trim prefix="SET" suffixOverrides=","> -- <set> 요소로 대체 가능
    <if test="empNo != null"> EMP_NO = #{empNo}, </if>
    <if test="empName != null"> EMP_NAME = #{empName} </if>
  </trim>
  ...
</update>

```

첫 번째 조건이 false, 두 번째 조건이 true일 경우, SQL Syntax Error!!

변경

```

<trim prefix="WHERE" prefixOverrides="AND|OR ">...</trim>
or
<where>

```


❑ [MyBatis 설정 1] SQL Mapper XML 파일 작성 (5/5) - Dynamic SQL

- foreach

- Map, List, Array에 담아 넘긴 값을 꺼낼 때 사용하는 요소

MyBatis Configuration XML 설정

```
<select id="selectforForeachFromList" parameterType="map" resultMap="egovMap">
  select
    <!-- List를 넘긴 경우, collection="list" / Array를 넘긴 경우, collection="array" 지정 -->
    <foreach collection="list" item="item" separator=", ">
      '${item}' as ${item}
    </foreach>

    <!-- Map 객체에 "collection"이라는 키로 List를 넘긴 경우 -->
    <foreach collection="collection" item="item" separator=", ">
      '${item}' as ${item}
    </foreach>
  from dual
</select>
```

□ [MyBatis 설정 2] MyBatis Configuration XML 파일 작성

- MyBatis 공통 설정 파일로, SqlSession 설정관련 상세 내역을 제어할 수 있는 메인 설정

MyBatis Configuration XML 설정

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>
  <typeAliases>
    <typeAlias alias="deptVO" type="x.y.z.service.DeptVO" />
    <typeAlias alias="empVO" type="x.y.z.service.EmpVO" />
  </typeAliases>

  <mappers>
    <mapper resource="META-INF/sqlmap/mappers/lab-dept.xml" />
    <mapper resource="META-INF/sqlmap/mappers/lab-emp.xml" />
  </mappers>
</configuration>
```

요소	설명
properties	설정 파일내에서 \${key} 와 같은 형태로 외부 properties 파일을 참조할 수 있다.
settings	런타임시 MyBatis의 행위를 조정하기 위한 옵션 설정을 통해 최적화할 수 있도록 지원한다
typeAliases	타입 별칭을 통해 자바타입에 대한 좀더 짧은 이름을 사용할 수 있다. 오직 XML 설정에서만 사용되며, 타이핑을 줄이기 위해 사용된다.
typeHandlers	javaType과 jdbcType 일치를 위해 TypeHandler 구현체를 등록하여 사용할 수 있다.
environments	환경에 따라 MyBatis 설정을 달리 적용할 수 있도록 지원한다.
mappers	매핑할 SQL 구문이 정의된 파일을 지정한다.

□ [스프링연동 설정] SqlSessionFactoryBean 정의

- Spring와 MyBatis 연동을 위한 설정으로, MyBatis 관련 메서드 실행을 위해 SqlSession 객체가 필요
- 스프링에서 SqlSession 객체를 생성하고 관리할 수 있도록, SqlSessionFactoryBean을 정의
 - id와 class는 고정값
 - dataSource : 스프링에서 설정한 DataSource Bean id를 설정하여 MyBatis가 DataSource를 사용하게 한다.
 - configLocation : MyBatis Configuration XML 파일이 위치하는 곳을 설정한다.
 - mapperLocations : SQL Mapper XML 파일을 일괄 지정할 수 있다. 단, Configuration 파일에 중복 선언할 수 없다.

SqlSessionFactoryBean 설정

```
<!-- SqlSession setup for MyBatis Database Layer -->
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="configLocation" value="classpath:/META-INF/sqlmap/config/sql-mapper-config.xml" />
  <!-- <property name="mapperLocations" value="classpath:/META-INF/sqlmap/mappers/*.xml" /> -->
</bean>
```

❑ MyBatis를 활용한 자바클래스 작성 1

- EgovAbstractMapper 클래스를 상속받아 DAO 클래스를 작성

DAO class 샘플 코드

```
@Repository("empMapper")
public class EmpMapper extends EgovAbstractMapper {

    public void insertEmp(EmpVO vo) {
        insert("insertEmp", vo);
    }

    public int updateEmp(EmpVO vo) {
        return update("updateEmp", vo);
    }

    public int deleteEmp(EmpVO vo) {
        return delete("deleteEmp", vo);
    }

    public EmpVO selectEmp(EmpVO vo) {
        return (EmpVO) selectByPk("selectEmp", vo);
    }

    public List<EmpVO> selectEmpList(EmpVO searchVO) {
        return list("selectEmpList", searchVO);
    }
}
```

EgovAbstractMapper는 SqlSessionDaoSupport의 하위 클래스로, SqlSession 설정과 메소드 호출의 편리함을 제공한다

SQL Mapper XML 내에 정의한 각 Statement id를 이용하여 실행할 SQL문을 호출하도록 작성한다

❑ MyBatis를 활용한 자바클래스 작성 2

- DAO 클래스 대신 Interface 작성 (Mapper Interface 방식) (1/4)

- 기존 DAO 클래스의 MyBatis 메소드 호출 코드를 최소화시킨 방법으로, 각 Statement id와 메서드명을 동일하게 작성하면 MyBatis가 자동으로 SQL문을 호출한다.
- 실제 내부적으로 MyBatis는 풀네임을 포함한 메서드명을 Statement id로 사용한다.

Mapper Interface 샘플 코드

```
package x.y.z.mapper;
```

```
@Mapper("deptMapper")  
public interface DeptMapper {
```

```
    public void insertDept(DeptVO vo);
```

```
    public int updateDept(DeptVO vo);
```

```
    public int deleteDept(DeptVO vo);
```

```
    public DeptVO selectDept(DeptVO vo);
```

```
    public List<DeptVO> selectDeptList(DeptVO vo);
```

```
}
```

Mapper Interface를 자동 스캔하기 위한 기능으로,
MapperConfigurer 빈설정과 함께 사용한다

Service 클래스에서 메소드를 호출하면,
메서드명과 일치하는 SQL문을 자동 실행한다

❑ MyBatis를 활용한 자바클래스 작성 2

- DAO 클래스 대신 Interface 작성 (Mapper Interface 방식) (2/4)

- 이 때 SQL Mapper XML 파일의 namespace값을 해당 Mapper의 풀네임으로 설정해야 한다.
- MyBatis는 해당 Mapper의 풀네임과 일치하는 namespace에서 메서드명과 동일한 id를 가진 Statement를 호출한다.
- namespace : 각 SQL Mapper XML을 구분

SQL Mapper XML 설정 1

```
<mapper namespace="x.y.z.mapper.DeptMapper">
  <insert id="insertDept" parameterType="deptVO">...</insert>
  <update id="updateDept" parameterType="deptVO">...</update>
  <delete id="deleteDept" parameterType="deptVO">...</delete>
  <select id="selectDept" parameterType="deptVO" resultMap="deptResult">...</select>
  <select id="selectDeptList" parameterType="deptVO" resultMap="deptResult">...</select>
</mapper>
```

SQL Mapper XML 설정 2

```
<mapper namespace="x.y.z.mapper.EmpMapper">
  <insert id="insertEmp" parameterType="empVO">...</insert>
  <update id="updateEmp" parameterType="empVO">...</update>
  <delete id="deleteEmp" parameterType="empVO">...</delete>
  <select id="selectEmp" parameterType="empVO" resultMap="empResult">...</select>
  <select id="selectEmpList" parameterType="empVO" resultMap="empResult">...</select>
</mapper>
```

❑ MyBatis를 활용한 자바클래스 작성 2

- DAO 클래스 대신 Interface 작성 (Mapper Interface 방식) (3/4)

- @Mapper를 사용하여 Mapper Interface가 동작하도록 하려면, MapperConfigurer 클래스를 빈으로 등록한다.
- MapperConfigurer는 @Mapper를 자동 스캔하고, MyBatis 설정의 편리함을 제공한다.
- basePackage : 스캔 대상에 포함시킬 Mapper Interface가 속한 패키지를 지정

MapperConfigurer 설정

```
<!-- MapperConfigurer setup for MyBatis Database Layer -->
<bean class="egovframework.rte.psl.dataaccess.mapper.MapperConfigurer">
  <property name="basePackage" value="x.y.z.mapper" />
</bean>
```

❑ MyBatis를 활용한 자바클래스 작성 2

- DAO 클래스 대신 Interface 작성 (Mapper Interface 방식) (4/4) - 어노테이션을 이용한 SQL문 작성
 - 인터페이스 메소드 위에 @Statement(Select, Insert, Update, Delete ...)를 선언하여 쿼리를 작성한다.
 - SQL Mapper XML을 작성할 필요가 없으나, Dynamic 쿼리를 사용하지 못하고 쿼리의 유연성이 떨어진다.

Mapper Interface 샘플 코드

```
@Mapper("deptMapper")
public interface DeptMapper {

    @Select("select DEPT_NO as deptNo, DEPT_NAME as deptName, LOC as loc from DEPT where DEPT_NO = #{deptNo}")
    public DeptVO selectDept(BigDecimal deptNo);

    @Insert("insert into DEPT(DEPT_NO, DEPT_NAME, LOC) values (#{deptNo}, #{deptName}, #{loc})")
    public void insertDept(DeptVO vo);

    @Update("update DEPT set DEPT_NAME = #{deptName}, LOC = #{loc} WHERE DEPT_NO = #{deptNo}")
    public int updateDept(DeptVO vo);

    @Delete("delete from DEPT WHERE DEPT_NO = #{deptNo}")
    public int deleteDept(BigDecimal deptNo);

}
```


LAB 205-MyBatis 실습

❑ Spring Data

- 데이터베이스 관련 많은 하위 프로젝트를 포함하는 오픈 소스 프로젝트
- non-relational databases, map-reduce frameworks, and cloud based data services 등의 새로운 데이터 액세스 기술을 보다 쉽게 사용 할 수 있는 기능 제공

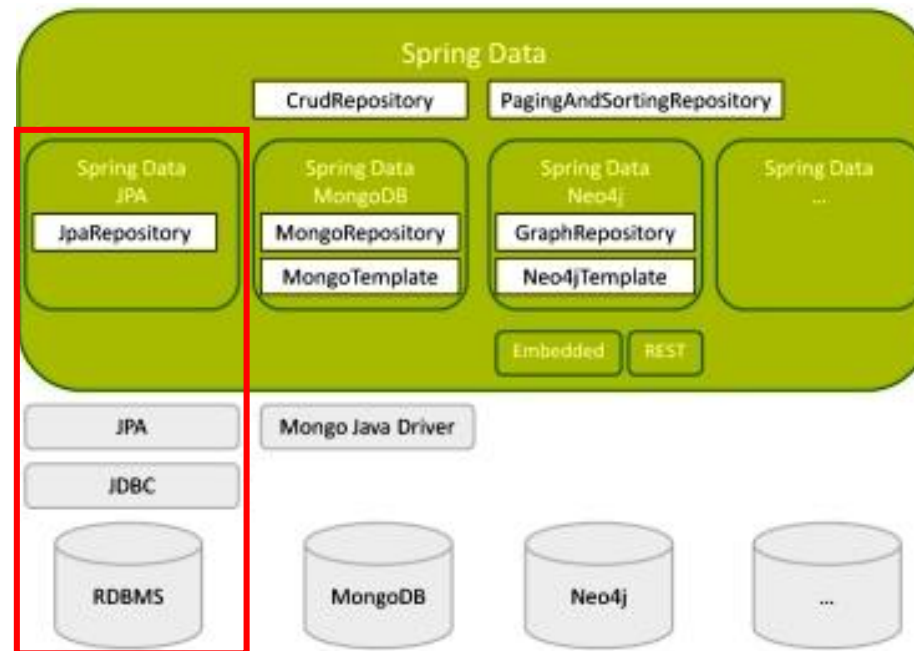
❑ Spring Data Project

Category	Sub project	Description
Relational Databases	JPA	Spring Data JPA - Simplifies the development of creating a JPA-based data access layer
	JDBC Extensions	Support for Oracle RAC, Advanced Queuing, and Advanced datatypes. Support for using QueryDSL with JdbcTemplate.
Big Data	Apache Hadoop	Apache Hadoop project is an open-source implementation of frameworks for reliable, scalable, distributed computing and data storage.
Data-Grid	GemFire	VMware vFabric GemFire is a distributed data management platform providing dynamic scalability, high performance, and database-like persistence. It blends advanced techniques like replication, partitioning, data-aware routing, and continuous querying.
HTTP	REST	Spring Data REST - Perform CRUD operations of your persistence model using HTTP and Spring Data Repositories.
Key Value Stores	Redis	Redis is an open source, advanced key-value store.
Document Stores	MongoDB	MongoDB is a scalable, high-performance, open source, document-oriented database.
Graph Databases	Neo4j	Neo4j is a graph database, a fully transactional database that stores data structured as graphs.
Column Stores	HBase	Apache HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable. HBase functionality is part of the Spring for Apache Hadoop project.
Common Infrastructure	Commons	Provides shared infrastructure for use across various data access projects. General support for cross-database persistence is located here

❑ Spring Data 특징

- DB 종류에 관계 없이 데이터 처리가 가능하도록 Persistence Layer를 추상화
- Repository라는 인터페이스를 통해 데이터 처리에 기본적으로 사용되는 CRUD나 pagination, sorting 과 같은 오퍼레이션을 제공
- 개발자는 기본 Repository Interface를 상속받아 정해진 키워드로 오퍼레이션을 쉽게 작성 가능 (Spring Data가 런타임 시에 구현체를 생성)

❑ Spring Data Architecture



❑ CrudRepository

- CRUD 오퍼레이션을 제공하는 인터페이스

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {  
    <S extends T> S save(S entity);-----전달받은 엔터티 저장  
    T findOne(ID primaryKey);-----전달된 ID로 식별한 엔터티 리턴  
    Iterable<T> findAll();-----모든 엔터티를 리턴  
    Long count();-----엔터티의 갯수를 리턴  
    void delete(T entity);-----전달받은 엔터티를 삭제  
    boolean exists(ID primaryKey);----- 전달된 ID에 해당하는 엔터티의 존재여부를 리턴  
}
```

❑ PagingAndSortingRepository

- Pagination과 Sorting 오퍼레이션을 제공하는 인터페이스

```
public interface PagingAndSortingRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {  
    Iterable<T> findAll(Sort sort);  
    Page<T> findAll(Pageable pageable);  
}
```

❑ Repository Interface 정의

- Repository, CrudRepository, PagingAndSortingRepository 중 하나를 상속받아 인터페이스 작성
- @RepositoryDefinition를 사용하면 제공되는 Repository를 상속받지 않고도 정의 가능
- 인터페이스 내 오퍼레이션은 정해진 키워드에 따라 작성 = 쿼리 메서드

```
public interface UserRepository extends Repository<User, Long> {  
    List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);  
}
```

- 위의 JPA 표준 API는 정의된 메소드를 다음 Query로 변경

```
select u from User u where u.emailAddress = ?1 and u.lastname = ?2
```

❑ Query Method 정의

- 정해진 키워드에 따라 작성된 메서드명으로부터 쿼리를 생성
- 지원하는 키워드와 예제

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	... where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false

❑ @Query

- @Query 어노테이션을 사용하여 쿼리를 직접 선언하는 방법
- 메서드와 쿼리문이 결합하여 동작하는 구조
- 메서드를 호출하고 파라미터를 바인딩하는 즉시 선언한 쿼리가 실행
- 파라미터 바인딩은 '?숫자' 혹은 @Param을 통해 가능

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
}
```

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")  
    User findByLastnameOrFirstname(@Param("lastname") String lastname,  
                                   @Param("firstname") String firstname);  
}
```

❑ iBatis

- <http://ibatis.apache.org>

❑ iBATIS-SqlMaps-2 Developer Guide

- http://svn.apache.org/repos/asf/ibatis/trunk/java/ibatis-2/ibatis-2-docs/en/iBATIS-SqlMaps-2_en.pdf

❑ iBATIS-SqlMaps-2 개발자 가이드

- http://kldp.net/frs/download.php/5035/iBATIS-SqlMaps-2_ko.pdf

❑ Spring Framework - Reference Documentation

- <http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/htmlsingle/#orm-ibatis>

❑ MyBatis - 개발자 가이드

- <http://mybatis.github.io/mybatis-3/>

□ 서비스 개요

- 객체 모델과 관계형 데이터베이스 간의 매핑 기능인 ORM(Object-Relational Mapping) 기능을 제공함으로써, SQL이 아닌 객체를 이용한 업무 로직의 작성이 가능하도록 지원

□ 주요 기능

- 객체와 관계형 데이터베이스 테이블 간의 매핑
 - 프레임워크 설정정보에 저장된 **ORM 매핑정보를 이용하여** 객체와 관계형 데이터베이스 테이블간의 매핑 지원
- 객체 로딩
 - 객체와 매핑되는 관계형 데이터베이스의 값을 읽어와 객체의 속성 값으로 설정함
- 객체 저장
 - 저장하고자 하는 객체의 속성 값을 객체와 매핑되는 관계형 데이터베이스에 저장
- 다양한 연관 관계 지원
 - 객체와 객체 간의 1:1, 1:n, n:n 등의 다양한 연관 관계를 지원
 - 객체의 로딩 및 저장 시, 연관 관계를 맺고 있는 객체도 로딩 및 저장 지원
- Caching
 - 객체에 대한 Cache 기능을 지원하여 성능을 향상시킴

- ❑ **Hibernate는 자바 객체와 관계형 데이터 모델간의 매핑을 위한 도구이며 쿼리 서비스를 지원하는 강력한 고성능의 퍼시스턴스 프레임워크임**

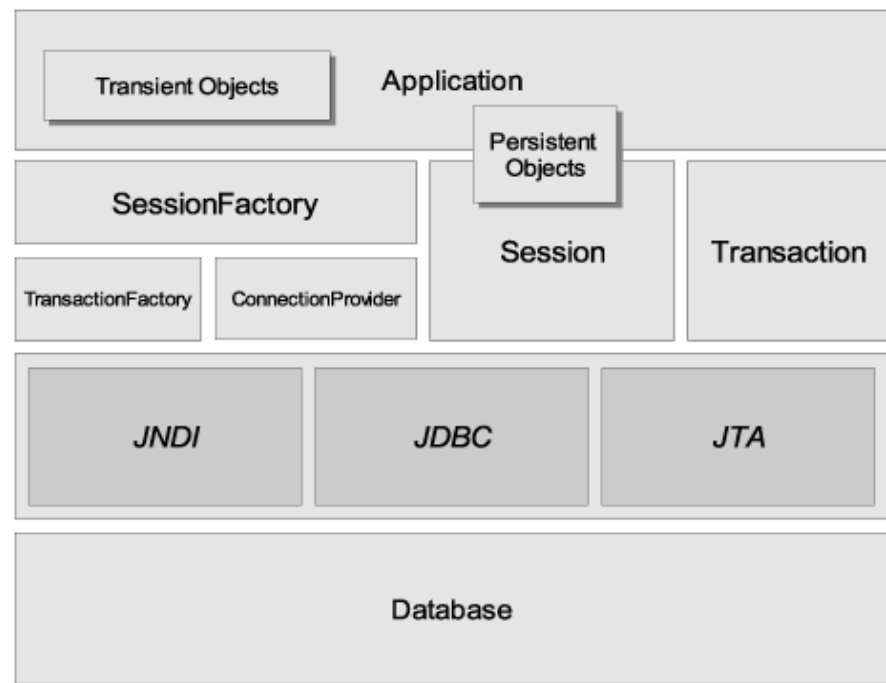


- 관계형 데이터 모델에 대한 객체지향 관점을 제공하는 객체/관계 매핑(Object Relational Mapping) 프레임워크
- Gavin King (JBoss, 현재 Red Hat)을 중심으로 한 소프트웨어 개발팀에 의해 개발됨.

분류 및 성숙도 평가*	설명
라이선스	LGPL (Lesser GNU Public License)
기능성 (Functionality)	✓✓✓✓ (중대형 규모의 기업의 기능적인 요구사항을 충족시킴)
커뮤니티 (Community)	*** (개발, 오류 보고, 수정 등의 활발한 커뮤니티 활동이 있음)
성숙도 (Maturity)	★★★★ (강력하며 높은 품질의 안정적이며 우수한 성능을 충족함)
적용성 (ER-Rating)	◆◆◆ (프레임워크가 성숙하여 기업 환경에 즉시 반영 가능함)
트렌드 (Trend)	↗ (평가 Criteria 전반적으로 발전하고 있으며, 중요도가 커지고 있음)

* Open Source Catalogue 2007, Optaros (Hibernate 3.2 기준)

- ❑ Hibernate는 J2EE 표준인 JNDI, JDBC, JTA를 기반으로 객체 관계형 매핑(OR Mapping), 데이터베이스 연결 및 트랜잭션 관리 기능 등을 제공함



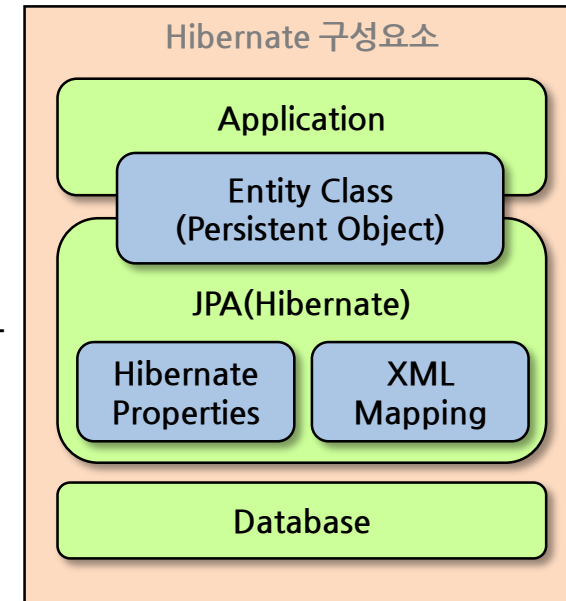
아키텍처 구성요소	설명
SessionFactory	<ul style="list-style-type: none"> 단일 데이터베이스에 대한 캐시로서, Session에 대한 팩토리 기능을 제공한다.
Session	<ul style="list-style-type: none"> 어플리케이션과 영속 저장소 사이의 연결을 표현하는 객체로서, JDBC 커넥션을 Wrapping한다. Transaction에 대한 팩토리 기능을 제공한다.
Persistent Objects and Collections	<ul style="list-style-type: none"> Session과 연관되어 있는 영속(persistent) 상태의 객체로서, 일반적인 JavaBeans/POJO이다. Session이 달하면, Session과 분리되어 Application 내에서 자유롭게 사용할 수 있게 된다.
Transient and Detached Objects and Collections	<ul style="list-style-type: none"> Session과 연관되어 있지 않은 영속 클래스들의 인스턴스로서, 어플리케이션에 의해 초기화된 후 영속화 되지 않았거나, 달려진 Session에 의해 초기화 되었을 수도 있다.
Transaction	<ul style="list-style-type: none"> 작업의 완전성을 보장하기 위한 어플리케이션에 의해 사용되는 객체이다.
ConnectionProvider	<ul style="list-style-type: none"> JDBC 커넥션들에 대한 팩토리 기능을 제공한다.
TransactionFactory	<ul style="list-style-type: none"> Transaction 인스턴스들에 대한 팩토리 기능을 제공한다.

□ Hibernate 특징

- 객체 모델링(Object Oriented Modeling)과 관계형 데이터 모델링(Relational Data Modeling) 사이의 불일치를 해결해주는 OR Mapping 서비스
- 특정 DBMS에 영향을 받지 않으므로 DBMS가 변경되더라도 데이터 액세스 처리 코드에 대한 변경 없이 설정 정보의 변경만으로도 동작 가능
- 별도의 XML 파일로 매핑을 관리하지 않고 Entity Class에 최소한의 Annotation으로 정의함으로써 작업이 용이함
- SQL 실행 결과로부터 전달하고자 하는 객체로 변경하는 코드 작성 시간 감소 (필요 시 SQL 사용도 가능)
- 기본적으로 필요 시점에만 DBMS에 접근하는 Lazy Loading 전략을 채택하고 Cache활용을 통해 DBMS에 대한 접근 횟수를 줄여나가 어플리케이션의 성능을 향상 시킴
- Entity Class가 일반 클래스로 정의됨으로써 상속이나 다양성, 캡슐화 같은 것들을 그대로 적용하면서 퍼시스턴스 오브젝트로 사용 가능
- 자바 표준이므로 많은 벤더들에서 구현체를 지원하고 개발을 편리하게 할 수 있는 JPA툴(Dali)을 지원함
- SQL을 이용하여 처리하는 방식에 익숙한 개발자가 사용하려면 학습이 필요하고 이에 따른 장벽이 존재함

❑ Hibernate 구성요소

- Entity Class (Persistenct Object)
 - 객체의 변수와 테이블의 컬럼 간 관계 정보가 정의된 클래스
 - Annotation으로 컬럼명, 테이블 관계 등의 정보를 Entity 클래스에서 직접 정의
 - 어플리케이션 실행 여부와 상관없이 물리적으로 존재하는 데이터들을 다룸
 - 데이터 처리시 Entity Class를 중심으로 어플리케이션의 데이터와 DBMS 연동함
 - XML기반으로 매핑하는 경우, 별도의 XML 파일이 필요
- Hibernate Properties File
 - 파일명: hibernate.cfg.xml or hibernate.properties
 - Hibernate 실행과 관련한 공통설정 파일
 - 구현체에 대한 선언 및 대상 엔티티 클래스 지정 구현체 별 프로퍼티 지정 등을 할 수 있는 설정파일
- Hibernate(JPA 구현체)
 - Hibernate 모듈은 Hibernate Core, Hibernate Annotations, Hibernate EntityManager 등으로 구성
 - JPA 구성에 필요한 Entity Manager등 구현 클래스를 포함하고 있음



□ Entity 클래스 작성

- 네 개의 Attribute와 각각의 getter • setter 메소드로 구성되어 있는 간단한 Entity 클래스를 생성

Entity 클래스 샘플

```
@Entity  
public class Department implements Serializable {
```

Department가 Entity 클래스임을 명시, 영속성 처리에 사용 가능한 클래스

```
    private static final long serialVersionUID = 1L;
```

```
    @Id  
    private String deptId;
```

테이블의 Primary Key 컬럼과 매핑할 변수에 지정

```
    private String deptName;
```

```
    private Date createDate;
```

```
    private BigDecimal empCount;
```

```
    public String getDeptId() {  
        return deptId;  
    }
```

```
    public void setDeptId(String deptId) {  
        this.deptId = deptId;  
    }
```

```
    ...
```

```
}
```

❑ persistence.xml 작성

- Entity 클래스를 가지고 JPA 수행하기 위한 프로퍼티 파일 작성
- 구현체 제공 클래스정보, 엔티티 클래스 정보, DB 접속 정보, 로깅 정보, 테이블 자동 생성 정보 등을 정의함

```
<persistence-unit name="PersistUnit" transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>egovframework.Department</class>
    <exclude-unlisted-classes />

    <properties>
        <property name="hibernate.connection.driver_class"
            value="org.hsqldb.jdbcDriver" />
        <property name="hibernate.connection.url" value="jdbc:hsqldb:mem:testdb" />
        <property name="hibernate.connection.username" value="sa" />
        <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />

        <property name="hibernate.connection.autocommit" value="false" />
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>

</persistence-unit>
```

□ 테스트 클래스 작성 (1/2)

- Department를 사용하여 입력,수정,조회,삭제 처리를 하는 것을 테스트 케이스로 작성하여 시험

Entity 클래스 테스트 코드

```
@Test
public void testDepartment() throws Exception {

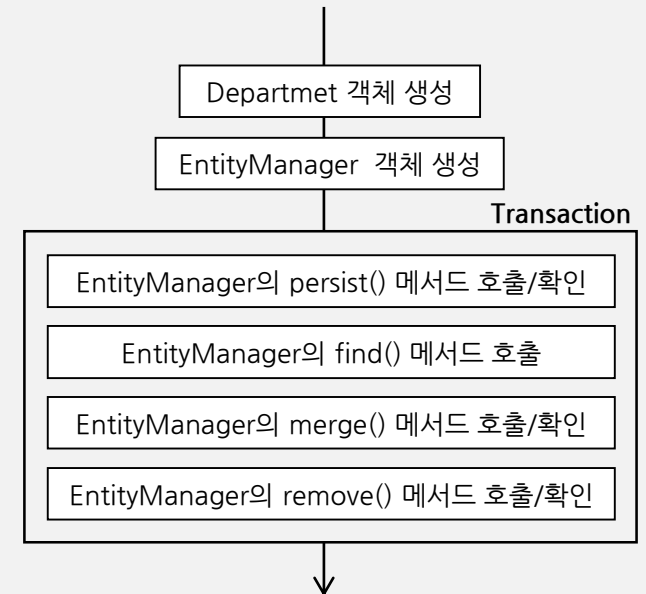
    String modifyName = "Marketing Department";
    String deptId = "DEPT-0001";
    Department department = makeDepartment(deptId);

    // EntityManager 생성
    entityManagerFactory = Persistence.createEntityManagerFactory("PersistUnit");
    entityManager = entityManagerFactory.createEntityManager();

    // 입력
    entityManager.getTransaction().begin();
    entityManager.persist(department);
    entityManager.getTransaction().commit();

    entityManager.getTransaction().begin();
    Department departmentAfterInsert = entityManager.find(Department.class, deptId);

    // 입력 확인
    assertEquals("Department Name Compare!", department.getDeptName(), departmentAfterInsert.getDeptName());
```



□ 테스트 클래스 작성 (2/2)

- Department를 사용하여 입력,수정,조회,삭제 처리를 하는 것을 테스트 케이스로 작성하여 시험

```
// 수정
departmentAfterInsert.setDeptName(modifyName);
entityManager.merge(departmentAfterInsert);
entityManager.getTransaction().commit();

em.getTransaction().begin();
Department departmentAfterUpdate = em.find(Department.class, deptId );

// 수정 확인
assertEquals("Department Modify Name Compare!",modifyName,departmentAfterUpdate.getDeptName());

// 삭제
entityManager.remove(departmentAfterUpdate);
entityManager.getTransaction().commit();

// 삭제 확인
Department departmentAfterDelete = em.find(Department.class, deptId );
assertNull("Department is Deleted!",departmentAfterDelete);

entityManager.close();
}
```

□ Entities (1/3)

- ORM 서비스를 구성하는 가장 기초적인 클래스로 어플리케이션에서 다루고자 하는 테이블에 대응하여 구성할 수 있으며 테이블이 포함하는 컬럼에 대응한 속성들을 가지고 있음

```
@Entity  
public class User implements Serializable {  
    private static final long serialVersionUID = -8077677670915867738L;  
  
    public User(){  
  
    }  
    @Id  
    private String userId;  
  
    private String userName;  
  
    public String getUserName() {  
        return userName;  
    }  
  
    public void setUserName(String userName) {  
        this.userName = userName;  
    }  
}
```

Entity annotation 선언

Serializable 인터페이스 구현

argument 없는 생성자 선언

Primary Key 선언

□ Entities (2/3)

- @Entity
 - 해당 클래스가 Entity 클래스임을 표시하는 것으로 클래스 선언문 위에 기재
 - 테이블명과 Entity명이 다를 때에는 name에 해당 테이블명을 기재

```
@Entity(name="USER_TB")  
public class User {  
}
```

- @Id
 - 해당 Attribute가 Key임을 표시하는 것으로 Attribute 위에 기재

```
@Id  
private String userId;
```

- @Column
 - 해당 Attribute와 매핑되는 컬럼정보를 입력하기 위한 것으로 Attribute위에 기재
 - 컬럼명과 Attribute명이 일치할 경우는 기재하지 않아도 됨

```
@Column(name = "DEPT_NAME", length = 30)  
private String deptName;
```

□ Entities (3/3)

- @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
 - 테이블간 관계를 구성하기 위한 것으로 정의되는 Attribute위에 기재
 - 각각은 1:1, 1:N, N:1, N:N의 관계를 표현함.

```
@ManyToMany
private Set<Role> roles = new HashSet(0);
```

- @Transient
 - 테이블의 컬럼과 매핑되지 않고 쓰이는 Attribute를 정의하고자 할때 Attribute위에 기재

```
@Transient
private String roleName;
```

□ Entity Status

- New(transient) : 단순히 Entity 객체가 초기화되어 있는 상태
- Managed(persistent) : Entity Manager에 의해 Entity가 관리되는 상태
- Detached : Entity 객체가 더 이상 Persistence Context와 연관이 없는 상태
- Removed : Managed 되어 있는 Entity 객체가 삭제된 상태

□ Entity Operation (1/3)

- 특정 DB에 데이터를 입력,수정,조회,삭제,배치 입력 등의 작업을 수행하는 오퍼레이션
- 입력
 - EntityManager의 persist()메소드를 호출하여 DB에 단건의 데이터 추가

```
Department department = new Department();  
String DepartmentId = "DEPT-0001";  
  
entityManager.persist(department);
```

- 수정
 - EntityManager의 merge() 메소드 호출
 - 특정 객체가 Persistent 상태이고, 동일한 트랜잭션 내에서 해당 객체의 속성 값에 변경이 발생한 경우, merge() 메소드를 직접적으로 호출하지 않아도 트랜잭션 종료 시점에 변경 여부가 체크되어 변경 사항이 DB에 반영됨

```
// 2. update a Department information  
department.setDeptName("Purchase Dept");  
  
// 3. 명시적인 메소드 호출  
entityManager.merge(department);
```

□ Entity Operation (2/3)

- 조회

- EntityManager의 find()메소드를 호출하여 DB에서 원하는 한건의 데이터를 조회할 수 있음
- find() 메소드 호출시 대상이 되는 Entity의 Id를 입력 인자로 전달해야 함

```
Department result = (Department) entityManager.find(Department.class, departmentId);
```

- 삭제

- EntityManager의 remove() 메소드 사용
- 삭제 할 객체가 동일한 경우 remove() 메소드 호출시 대상이 되는 Entity를 입력 인자로 전달하여 삭제함

```
// 1. insert a new Department information  
Department department = addDepartment();
```

```
// 2. delete a Department information  
entityManager.remove(department);
```

- 삭제 할 객체가 동일한 객체가 아닐 경우 getReference 메소드를 호출하여 Entity의 Id에 해당하는 객체 정보를 추출하여 그정보를 입력인자로 해서 remove를 호출하여 삭제함

```
Department department = new Department();  
department.setDeptId = "DEPT_1";  
  
// 2. delete a Department information  
entityManager.remove(em.getReference(Department.class, department.getDeptId()));
```

❑ Entity Operation (3/3)

- 배치입력
 - EntityManager의 persist()메소드를 호출하여 DB에 입력하고 loop를 통해 반복적으로 수행
 - OutOfMemoryException 방지를 위해서 일정한 term을 두고 flush(),clear()을 호출하여 메모리에 있는 사항을 삭제

```
public void testMultiSave() throws Exception {  
  
    for (int i = 0; i < 900 ; i++) {  
  
        Department department = new Department();  
        String DeptId = "DEPT-000" + i;  
        department.setDeptId(DeptId);  
        department.setDeptName("Sale" + i);  
        department.setDesc("판매부" + i);  
  
        em.persist(department);  
        logger.debug("=== DEPT-000"+i+" ===");  
  
        // OutOfMemoryException 피하기 위해서  
        if (i != 0 && i % 9 == 0) {  
            em.flush();  
            em.clear();  
        }  
    }  
}
```

❑ Callback Methods (1/2)

- 엔티티 Operation 직전 직후에 비즈니스 로직 체크 등의 로직을 별도 분리하여 처리하도록 지원함
- Callback Methods 종류
 - PrePersist : Persist이전 시점에 수행
 - PostPersist : Persist이후 시점에 수행
 - PreRemove : Remove이전 시점에 수행
 - PostRemove : Remove이후 시점에 수행
 - PreUpdate : Merge이전 시점에 수행
 - PostUpdate : Merge이후 시점에 수행
 - PostLoad : Find 이후 시점에 수행

❑ Callback Methods (2/2)

- Callback Methods 정의 방식
 - 엔티티 클래스에 내부 정의
 - EntityListener를 지정하여 콜백 함수 정의
- 엔티티 클래스에 내부 정의 예제
 - salary가 2000000 이하로 설정되어 Update가 실행될 경우 Exception 이 발생함

```
@Entity
public class User {
    @PrePersist
    @PreUpdate
    protected void validateCreate() throws Exception {
        if (getSalary() < 2000000 )
            throw new Exception("Insufficient Salary !");
    }
}
```

❑ Association Mapping (1/5)

- 두 클래스 사이의 연관관계 유형에 따라 매핑 관계를 선언함
 - One To One Mapping
 - One To Many Mapping
 - Many To Many Mapping
- One To One Mapping 예제
 - Employee 와 TravelProfile가 각각 OneToOne이라는 Annotation을 기재하여 매핑 선언

```
@Entity
public class Employee {
    @OneToOne
    private TravelProfile profile;
}

@Entity
public class TravelProfile {
    @OneToOne
    private Employee employee;
}
```

❑ Association Mapping (2/5)

- One To Many Mapping
 - Department:User = 1:N 의 관계가 있으며 그 관계에 대해서 Department 클래스에서 OneToMany로 표시하고 User 클래스에서 ManyToOne으로 표시하여 관계를 나타냈다.

```
@Entity
public class Department{
    @OneToMany(targetEntity=User.class)
    private Set<User> users = new HashSet(0);
}

@Entity
public class User{
    @ManyToOne
    private Department department;
}
```

❑ Association Mapping (3/5)

- Collection Type
 - Many관계에서 Collection Type은 Set 이외에도 List, Map를 사용할 수 있음
 - Set 타입 : java.util.Set 타입으로 <set>을 이용하여 정의
 - List 타입 : java.util.List 타입으로 <list>를 이용하여 정의
 - Map 타입 : java.util.map 타입으로 <map>을 이용하여 (키,값)을 쌍으로 정의

```
// Set 예제
@OneToMany(targetEntity=User.class)
private Set<User> users = new HashSet(0);

// List 예제
@OneToMany(targetEntity=User.class )
private List<User> users = new ArrayList(0);

// Map 예제
@OneToMany(targetEntity=User.class)
@MapKey(name="userId")
private Map<String,User> users ;
```

❑ Association Mapping (4/5)

- 단방향/양방향 관계 속성
 - 1:N(부모:자식)관계 지정에 있어서 자식쪽에서 부모에 대한 참조 관계를 가지고 있느냐 없느냐에 따라서 참조관계가 있으면 양방향 관계, 없으면 단방향 관계로 정의
 - 단방향/ 양방향 예제

```
//단방향 예제
@Entity
public class Department {
    @OneToMany(targetEntity=User.class)
    private Set<User> users = new HashSet(0);
}

@Entity
public class User {
    @Column(name="DEPT_ID")
    private String deptId;
}
```

```
//양방향 예제
@Entity
public class Department {
    @OneToMany(targetEntity=User.class)
    private Set<User> users = new HashSet(0);
}

@Entity
public class User {
    @ManyToOne
    private Department department;
}
```

❑ Association Mapping (5/5)

- Many To Many Mapping
 - Role:User = M:N 의 관계가 있다면 그 관계에 대해서 Role클래스에서 ManyToMany로 표시하고 User 클래스에서 ManyToMany로 표시하여 관계를 나타내면서 User-Role 관계 테이블을 정의
 - ROLE과 USER를 연결하는 관계 테이블로 AUTHORITY가 사용되었음을 선언

```
@Entity
public class Role {
    @ManyToMany(targetEntity=User.class)
    private Set<User> users = new HashSet(0);
}

@Entity
public class User {
    @ManyToMany
    @JoinTable(name="AUTHORITY",
        joinColumns=@JoinColumn(name="USER_ID"),
        inverseJoinColumns=@JoinColumn(name="ROLE_ID"))
    private Set<Role> roles = new HashSet(0);
}
```

❑ Spring Integration (1/7)

- Spring에서는 JPA 기반에서 DAO 클래스를 쉽게 구현할 수 있도록 하기 위해 JdbcTemplate, HibernateTemplate 등처럼 JpaTemplate 클래스를 제공함
- JPA에서 정의한 Entity Manager의 Method를 직접 이용하는 방식도 제공함
- 기본설정
 - persistence.xml 설정 (persistHSQLMemDB.xml 파일)

```
<persistence-unit name="HSQLMUnit" transaction-type="RESOURCE_LOCAL">
// 구현체는 Hibernate
<provider>org.hibernate.ejb.HibernatePersistence</provider>

// Entity Class List
<class>egovframework.sample.model.bidirection.User</class>
<class>egovframework.sample.model.bidirection.Role</class>
<class>egovframework.sample.model.bidirection.Department</class>
<exclude-unlisted-classes/>

<properties>
// DBMS별 다른 설정 여기는 HSQL 설정.
<property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.hbm2ddl.auto" value="create-drop"/>
</properties>
</persistence-unit>
```

❑ Spring Integration (2/7)

- 기본설정
 - Application Context 설정 (1/2)

// 1.Transaction Manager 설정

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">  
  <property name="entityManagerFactory" ref="entityManagerFactory" />  
</bean>
```

// 2.Entity Manager Factory 설정

```
<bean id="entityManagerFactory"  
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">  
  <property name="persistenceUnitName" value="HSQLMUnit" />  
  <property name="persistenceXmlLocation" value="classpath:META-INF/persistHSQLMemDB.xml" />  
  <property name="dataSource" ref="dataSource" />  
</bean>
```

persistence.xml 파일의
persistence-unit name 속성값과
파일 위치를 지정

// 3.DataSource 설정

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"  
  destroy-method="close">  
  <property name="driverClassName" value="net.sf.log4jdbc.DriverSpy" />  
  <property name="url" value="jdbc:log4jdbc:hsqldb:mem:testdb" />  
  <property name="username" value="sa" />  
  <property name="password" value="" />  
  <property name="defaultAutoCommit" value="false" />  
</bean>
```


❑ Spring Integration (3/7)

- 기본설정

• Application Context 설정 (2/2)

// 4. JPA Annotation 사용 설정

```
<context:annotation-config />
```

// 5. JPA 예외변환 후처리기 설정

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
```

// 6.Annotation 기반의 Transaction 활성화 설정

```
<tx:annotation-driven />
```

❑ Spring Integration (4/7)

- Plain JPA 이용 (1/2)
 - JPA에서 정의한 Entity Manager의 Entity Method를 호출 작업할 수 있음
 - Entity Manager를 통해 작업함으로써 Spring 환경하에서 Spring에 대한 의존성을 최소화 할 수 있음

```
public class RoleDAO {  
    // Application Context 설정의 4 .JPA Annotation 사용 설정에 의해서 가능한 것으로,  
    JPA의 @PersistenceContext를 사용하면 EntityManager 객체를 바로 사용할 수 있다.  
    @PersistenceContext  
    private EntityManager em;  
  
    // EntityManager를 통한 입력  
    public void createRole(Role role) throws Exception {  
        em.persist(role);  
    }  
    // EntityManager를 통한 조회  
    public Role findRole(String roleId) throws Exception {  
        return (Role) em.find(Role.class, roleId);  
    }  
    // EntityManager를 통한 삭제  
    public void removeRole(Role role) throws Exception {  
        em.remove(em.getReference(Role.class, role.getRoleId()));  
    }  
    // EntityManager를 통한 수정  
    public void updateRole(Role role) throws Exception {  
        em.merge(role);  
    }  
}
```

❑ Spring Integration (5/7)

- Plain JPA 이용 (2/2)
 - Entity 클래스

```
@Entity
public class Role implements Serializable {

    private static final long serialVersionUID = 1042037005623082102L;

    @Id
    @Column(name = "ROLE_ID", length=10)
    private String roleId;

    @Column(name = "ROLE_NAME", length=20)
    private String roleName;

    @Column(name = "DESC", length=50)
    private String desc;

    ...
}
```

❑ Spring Integration (6/7)

- JpaTemplate 이용 (1/2)
 - Spring에서 정의한 JpaDaoSupport를 상속받아 getJpaTemplate()를 통해서 Entity Method 등을 호출 작업할 수 있음

```
public class UserDao extends JpaDaoSupport {  
    // Application Context 에서 설정한 Entity Manager Factory 명을 지정하여 부모의 EntityManagerFactory를 설정한다.  
    @Resource(name="entityManagerFactory")  
    public void setEMF(EntityManagerFactory entityManagerFactory) {  
        super.setEntityManagerFactory(entityManagerFactory);  
    }  
    // getTemplate()에 의한 입력  
    public void createUser(User user) throws Exception {  
        this.getJpaTemplate().persist(user);  
    }  
    // getTemplate()에 의한 조회  
    public User findUser(String userId) throws Exception {  
        return (User) this.getJpaTemplate().find(User.class, userId);  
    }  
    // getTemplate()에 의한 삭제  
    public void removeUser(User user) throws Exception {  
        this.getJpaTemplate().remove(this.getJpaTemplate().getReference(User.class, user.getUserId()));  
    }  
}
```

❑ Spring Integration (7/7)

- JpaTemplate 이용 (2/2)
 - Entity 클래스

```
@Entity
public class User implements Serializable {

    private static final long serialVersionUID = -8077677670915867738L;

    @Id
    @Column(name = "USER_ID", length=10)
    private String userId;

    @Column(name = "USER_NAME", length=20)
    private String userName;

    @Column(length=20)
    private String password;

    ...
}
```

❑ Hibernate 공식 사이트

- www.hibernate.org

❑ Spring JPA

- <http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/htmlsingle/#orm-jpa>
- 이전 버전 참조
 - <http://static.springsource.org/spring/docs/2.5.x/reference/orm.html#orm-jpa>
 - <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/orm.html#orm-jpa>
 - <http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/htmlsingle/#orm-jpa>

□ 서비스 개요

- 트랜잭션 서비스는 **Spring 트랜잭션 서비스를** 채택
- 트랜잭션 서비스 종류
 - DataSource Transaction Service
 - JTA Transaction Service
 - JPA Transaction Service에 대해서 설명한다.
- 트랜잭션 활용 방법
 - XML 설정 및 Annotation을 통해 활용할 수 있는 Declaration Transaction Management
 - 프로그램에서 직접 API를 호출하여 쓸 수 있도록 하는 Programmatic Transaction Management

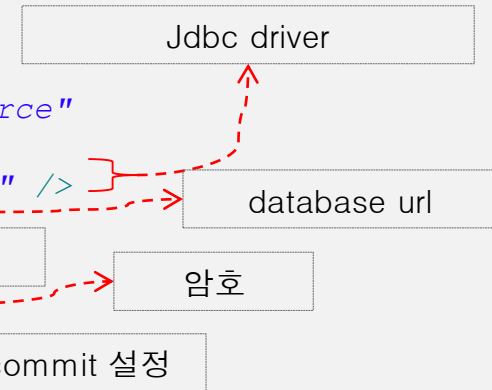
❑ Transaction Service

– DataSource Transaction Service

- DataSource를 사용하여 Local Transaction을 관리 할 수 있다.
- Configuration

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

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="dbc:mysql://db2:1621/rte" />
  <property name="username" value="rte" />
  <property name="password" value="xxx" />
  <property name="defaultAutoCommit" value="false" />
</bean>
```



- Sample Source

```
@Resource(name="transactionManager")
PlatformTransactionManager transactionManager;
...
TransactionStatus txStatus = transactionManager.getTransaction(txDefinition);
```

- JTA Transaction Service
 - JTA를 이용하여 Global Transation관리를 할 수 있도록 지원한다.
 - Configuration

```
<tx:jta-transaction-manager />
<jee:jndi-lookup id="dataSource" jndi-name="dbmsXADS"
  resource-ref="true">
  <jee:environment>
    java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
    java.naming.provider.url=t3://was:7002
  </jee:environment>
</jee:jndi-lookup>
```

위의 설정예에서 jndi-name 과 java.naming.factory.initial,java.naming.provider.url은 사이트 환경에 맞추어 변경해야 한다. DataSource Transaction Service와는 달리 transationManager에 대해서 따로 bean 정의하지 않아도 된다.

– JPA Transaction Service

- JPA Transaction 서비스는 JPA EntityManagerFactory를 이용하여 트랜잭션을 관리한다. JpaTransactionManager는 EntityManagerFactory에 의존성을 가지고 있으므로 반드시 EntityManagerFactory 설정과 함께 정의되어야 한다. 아래에서 예를 들어서 설정 방법을 설명한다. 사용법은 DataSource Transaction Service와 동일하다.

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="OraUnit" />
  <property name="persistenceXmlLocation" value="classpath:META-INF/persistence.xml" />
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="dbc:mysql://db2:1621/rte" />
  <property name="username" value="rte" />
  <property name="password" value="xxx" />
  <property name="defaultAutoCommit" value="false" />
</bean>
```

위의 설정을 보면 transactionManager의 property로 entityManagerFactory로 지정하고 entityManagerFactory의 property로 dataSource를 지정하고 그에 필요한 driver정보, url정보등을 지정한 것을 확인 할 수 있다. 설정한 dataSource 기반하에서 트랜잭션 서비스를 제공한다. 사이트 환경에 맞추어 driverClassName, url, username, password는 변경해서 적용한다. 또한 persistenceUnitName과 persistenceXmlLocation 정보를 지정하는 것을 알수 있다

❑ Declarative Transaction Management

- 코드에서 직접적으로 Transaction 처리하지 않고, 선언적으로 Transaction을 관리할 수 있다. Annotation을 이용한 Transaction 관리, XML 정의를 이용한 Transaction 관리를 지원한다.

• Configuration

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

`@Transactional`
public void removeRole(Role role) throws Exception {
 this.roleDAO.removeRole(role);
}

트랜잭션 처리하고자 하는
메소드위에 기재하여 트랜잭션관리

속 성	설 명	사 용 예
isolation	Transaction의 isolation Level 정의하는 요소. 별도로 정의하지 않으면 DB의 Isolation Level을 따름.	@Transactional(isolation=Isolation.DEFAULT)
noRollbackFor	정의된 Exception 목록에 대해서는 rollback을 수행하지 않음.	@Transactional(noRollbackFor=NoRoleBackTx.class)
noRollbackForClassName	Class 객체가 아닌 문자열을 이용하여 rollback을 수행하지 않아야 할 Exception 목록 정의	@Transactional (noRollbackForClassName="NoRoleBackTx")
propagation	Transaction의 propagation 유형을 정의하기 위한 요소	@Transactional(propagation=Propagation.REQUIRED)
readOnly	해당 Transaction을 읽기 전용 모드로 처리 (Default = false)	@Transactional(readOnly = true)
rollbackFor	정의된 Exception 목록에 대해서는 rollback 수행	@Transactional(rollbackFor=RoleBackTx.class)
rollbackForClassName	Class 객체가 아닌 문자열을 이용하여 rollback을 수행해야 할 Exception 목록 정의	@Transactional(rollbackForClassName="RoleBackTx")
timeout	지정한 시간 내에 해당 메소드 수행이 완료되지 않은 경우 rollback 수행. -1일 경우 no timeout (Default = -1)	@Transactional(timeout=10)

- Configuration Transaction Management
 - XML 정의 설정을 이용해서 Transaction을 관리할 수 있다.
 - Configuration

```
<aop:config>
  <aop:pointcut id="requiredTx"
    expression="execution(* egovframework.sample..impl.*Impl.*(..))" />
  <aop:advisor advice-ref="txAdvice" pointcut-ref="requiredTx" />
</aop:config>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="find*" read-only="true" />
    <tx:method name="createNoRBRole" no-rollback-for="NoRoleBackTx" />
    <tx:method name="createRBRole" rollback-for="RoleBackTx" />
    <tx:method name="create*" />
  </tx:attributes>
</tx:advice>
```

aop:pointcut를 이용하여 실행되어 Catch해야 하는 Method를 지정하고 tx:advice를 통해서 각각에 대한 룰을 정의하고 있다. 이렇게 정의하면 프로그램 내에서는 별도의 트랜잭션 관련한 사항에 대해 기술하지 않아도 트랜잭션관리가 된다.

- <tx:method> 상세 속성 정보

속 성	설 명	사 용 예
name	메소드명 기술. 와일드카드 사용 가능함	Name="find*"
isolation	Transaction의 isolation Level 정의하는 요소	Isolation="DEFAULT"
no-rollback-for	정의된 Exception 목록에 대해서는 rollback을 수행하지 않음	No-rollback-for="NoRolBackTx"
propagation	Transaction의 propagation 유형을 정의하기 위한 요소	propagation="REQUIRED"
read-only	해당 Transaction을 읽기 전용 모드로 처리(Default=false)	read-only="true"
rollback-for	정의된 Exception 목록에 대해서는 rollback 수행	rollback-for=RoleBackTx"
timeout	지정한 시간 내에 해당 메소드 수행이 완료되지 않은 경우 rollback 수행.	timeout="10"

- Propagation Behavior, Isolation Level(두가지 Transaction Management 공통적으로 사용되는 항목)

➤ Propagation Behavior

속 성 명	설 명
PROPAGATION_MADATORY	반드시 Transaction 내에서 메소드가 실행되어야 한다. 없으면 예외발생
PROPAGATION_NESTED	Transaction에 있는 경우, 기존 Transaction 내의 nested transaction 형태로 메소드를 실행하고, nested transaction 자체적으로 commit, rollback이 가능하다. Transaction이 없는 경우, PROPAGATION_REQUIRED 속성으로 행동한다. nested transaction 형태로 실행될 때는 수행되는 변경사항이 커밋이 되기 전에는 기존 Transaction에서 보이지 않는다.
PROPAGATION_NEVER	Manatory와 반대로 Transaction 없이 실행되어야 하며 Transaction이 있으면 예외를 발생시킨다.
PROPAGATION_NOT_SUPPORTED	Transaction 없이 메소드를 실행하며, 기존의 Transaction이 있는 경우에는 이 Transaction을 호출된 메소드가 끝날 때까지 잠시 보류한다
PROPAGATION_REQUIRED	기존 Transaction이 있는 경우에는 기존 Transaction 내에서 실행하고, 기존 Transaction이 없는 경우에는 새로운 Transaction을 생성한다.
PROPAGATION_REQUIRED_NEW	호출되는 메소드는 자신 만의 Transaction을 가지고 실행하고, 기존의 Transaction들은 보류된다
PROPAGATION_SUPPORTS	새로운 Transaction을 필요로 하지는 않지만, 기존의 Transaction이 있는 경우에는 Transaction 내에서 메소드를 실행한다.

➤ Isolation Level

속 성 명	설 명
ISOLATION_DEFAULT	개별적인 PlatformTransactionManager를 위한 격리 레벨
ISOLATION_READ_COMMITTED	이 격리수준을 사용하는 메소드는 commit 되지 않은 데이터를 읽을 수 없다. 쓰기 락은 다른 Transaction에 의해 이미 변경된 데이터는 읽을 수 없다. 따라서 조회 중인 commit 되지 않은 데이터는 불가능하다. 대개의 데이터베이스에서의 디폴트로 지원하는 격리 수준이다.
ISOLATION_READ_UNCOMMITTED	가장 낮은 Transaction 수준이다. 이 격리수준을 사용하는 메소드는 commit 되지 않은 데이터를 읽을 수 있다. 그러나 이 격리수준은 새로운 레코드가 추가되었는지 알 수 없다.
ISOLATION_REPEATABLE_READ	ISOLATION_READ_COMMITTED 보다는 다소 조금 더 엄격한 격리 수준이다. 이 격리 수준은 다른 Transaction이 새로운 데이터를 입력했다면, 새롭게 입력된 데이터를 조회할 수 있다는 것을 의미한다.
ISOLATION_SERIALIZABLE	가장 높은 격리수준이다. 모든 Transaction(조회를 포함하여)은 각 라인이 실행될 때마다 기다려야 하기 때문에 매우 느리다. 이 격리수준을 사용하는 메소드는 데이터 상에 배타적 쓰기를 락을 얻음으로써 Transaction이 종료될 때까지 조회, 수정, 입력 데이터로부터 다른 Transaction의 처리를 막는다. 가장 많은 비용이 들지만 신뢰할만한 격리 수준을 제공하는 것이 가능하다.

❑ Programmatic Transaction Management

- 프로그램에서 직접 트랜잭션을 관리하고자 할 때 사용할 수 있는 방법에 대해서 설명하고자 한다.
TransactionTemplate를 사용하는 방법과 TransactionManager를 사용하는 방법 두 가지가 있다.
- TransactionTemplate Configuration

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

TransactionTemplate를 정의하고 property로 transactionManager를 정의한다.

- TransactionTemplate 를 이용한 Sample Source

```
@Test
public void testInsertCommit() throws Exception {
    transactionTemplate.execute(new TransactionCallbackWithoutResult() {
        public void doInTransactionWithoutResult(TransactionStatus status) {
            try {
                Role role = new Role();
                role.setRoleId("ROLE-001");
                role.setRoleName("ROLE-001");
                role.setRoleDesc(new Integer(1000));
                roleService.createRole(role);
            } catch (Exception e) {
                status.setRollbackOnly();
            }
        }
    });
    Role retRole = roleService.findRole("ROLE-001");
    assertEquals("roleName Compare OK", retRole.getRoleName(), "ROLE-001");
}
```

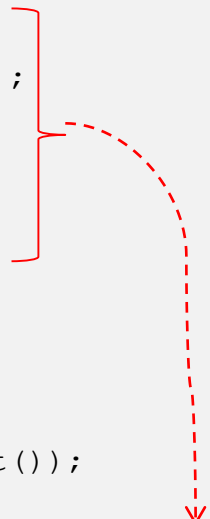
transactionTemplate.execute에
TransactionCallbackWithoutResult를 정의하여
Transaction 관리를 하는 것을 확인할 수 있다.

- Transaction Manager Configuration

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

- Transaction Manager를 이용한 Sample Source

```
@Test
public void testInsertRollback() throws Exception {
    int prevCommitCount = roleService.getCommitCount();
    int prevRollbackCount = roleService.getRollbackCount();
    DefaultTransactionDefinition txDefinition = new DefaultTransactionDefinition();
    txDefinition.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);
    TransactionStatus txStatus = transactionManager.getTransaction(txDefinition);
    try {
        Role role = new Role();
        role.setRoleId(Thread.currentThread().getName() + "-roleId");
        role.setRoleName(Thread.currentThread().getName() + "-roleName");
        role.setRoleDesc(new Integer(1000));
        roleService.createRole(role);
        roleService.createRole(role);
        transactionManager.commit(txStatus);
    }
    catch (Exception e) {
        transactionManager.rollback(txStatus);
    }
    finally {
        assertEquals(prevCommitCount, roleService.getCommitCount());
        assertEquals(prevRollbackCount + 2, roleService.getRollbackCount());
    }
}
```



Transaction 서비스를 직접 얻어온 후에 위와 같이 try~catch 구문 내에서 Transaction 서비스를 이용하여, 적절히 begin, commit, rollback을 수행한다. 이 때, TransactionDefinition와 TransactionStatus 객체를 적절히 이용하면 된다.

❑ Spring Transaction Management

- <http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/transaction.html>
- 이전 버전 참조
 - <http://static.springsource.org/spring/docs/2.5.x/reference/transaction.html>
 - <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/transaction.html>
 - <http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/transaction.html>

Q&A

감사합니다.