1. JDBC

JDBC : 자바 언어로 다양한 종류의 관계형 데이터베이스에 접속하고 SQL문을 수행하여 처리하고자 할 때 사용되는 표준 SQL 인터페이스 API

코드를 추가하거나 수정할 때마다 매번 DB연동에 필요한 반복되는 자바 코드를 입력해야되는 번거로움을 처리하기 위해 JdbcTemplate 클래스를 이용한다

- JdbcTemplate을 사용하지 않은 경우

BoardDAO.java

```
private final String BOARD_INSERT =
"insert into board(seq,titile,writer,content)
values((select nvl(max(seq),0)+ 1 from board),?,?,?)";
private Connection conn = null;
private PreparedStatement stmt = null;
private ResultSet rs = null;

public void insertBoard(BoardVO vo){
  try{
    conn = JDBCUtil.getConnection();
    stmt = conn.prepareStatement(BOARD_INSERT);
    stmt.setString(1,vo.getTitle());
    stmt.setString(2,vo.getWriter());
    stmt.setString(3,vo.getContent());
    stmt.executeUpdate();
  } catch ( Exception e) {
     e.printStackTrace();
  } finally {
    JDBCUtil.close(stmt, conn);
  }
}
```

JDBCUtil.java

```
public class JDBCUtil{
  public static Connection getConnection(){
    try{
      Class.forName("org.h2.Driver");
      return DriverManager.getConnection("jdbc:h2:tcp://localhost/~/test","sa","");
  } catch(Exception e){
      e.printStackTrace();
  }
    return null;
  }
```

```
  public static void close(PreparedStatement stmt, Connection conn) {
    if(stmt!=null){
      try{
        if(!stmt.isClosed()) stmt.close();
      } catch ( Exception e) {
         e.printStackTrace();
      } finally {
        stmt = null;
      }
    }
      if(conn!=null){
      try{
        if(!conn.isClosed()) conn.close();
      } catch ( Exception e) {
         e.printStackTrace();
      } finally {
        conn = null;
      }
    }
  }
}
```

- JdbcTemplate 클래스
  - GoF 디자인 패턴 중 템플릿 메소드 패턴이 적용된 클래스
    - 템플릿 메소드 패턴 : 복잡하고 반복되는 알고리즘을 캡슐화해서 재사용하는 패턴
  - 이를 이용하면 JDBC의 DB 연동 로직은 JdbcTemplate 클래스의 템플릿 메소드가 제공하고, 개발자는 달라지는 Sql문과 설정값만 신경쓰면 된다.
- JDBC 설정
  - 라이브러리 추가
    - pom.xml에 DBCP 관련 <dependency> 추가

pom.xml

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${org.springframework-version}</version>
</dependency>
<!-- DBCP -->
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.4</version>
```

```
</dependency>
```

- DataSource 설정

  applicationContext.xml

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
  <property name="driverClassName" value="org.h2.Driver"/>
  <property name="url" value="jdbc:h2:tcp://localhost/~/test"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>
```

  - Property 파일을 활용한 DataSource 설정

  1) resources/config/database.properties

```
jdbc.driver=org.h2.Driver
jdbc.url = jdbc:h2:tcp://localhost/~/test
jdbc.username=sa
jdbc.password=
```

  2) applicationContext.xml

```
<!-- DataSource 설정: connect -->
<context:property-placeholder
location="classpath:config/database.properties"/>

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

- JdbcTemplate 메소드
  - update() 메소드

```
int update(String sql, Object[] args)
```

    - INSERT, UPDATE, DELETE 구문 처리시 사용
    - 사용법(by “?”에 값을 설정하는 방식)
      - “?”수만큼 값을 나열해서 인자로 전달

```
public void updateBoard(BoardVO vo){
  String BOARD_UPDATE="update board set title=?, content=? where seq=?";
  int cnt =
jdbcTemplate.update(BOARD_UPDATE,vo.getTitle(),vo.getContent(),vo.getSeq());
  System.out.println(cnt+ "건 데이터 수정");
}
```

- "?"수만큼의 값을 세팅한 배열 객체를 두번째 인자로 전달

```
public void updateBoard(BoardVO vo){
  String BOARD_UPDATE="update board set title=?, content=? where seq=?";
  Object[] args = {vo.getTitle(),vo.getContent(),vo.getSeq());
  int cnt = jdbcTemplate.update(BOARD_UPDATE,args);
  System.out.println(cnt+ "건 데이터 수정");
}
```

- queryForInt() 메소드
  - SELECT 구문으로 검색된 정수값을 리턴받기 위해 사용

```
int queryForInt(String sql)
int queryForInt(String sql, Object... args)
int queryForInt(String sql, Object[] args)
```

- queryForObject() 메소드
  - SELECT 구문의 실행 결과를 특정 자바 객체(Value Object)로 매핑하여 리턴받을 때 사용
  - 검색 결과가 두개 이상이거나 없으면 예외(IncorretResultSizeDataAccessException)를 발생시킴

```
VO queryForObject(String sql)
VO queryForObject(String sql, RowMapper<T> rowMapper)
VO queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)
```

  - 사용 예

```
public BoardVO getBoard(BoardVO vo){
  String BOARD_GET = "select * from board where seq=?";
  Object[] args = {vo.getSeq()};
  return jdbcTemplate.queryForObject(BOARD_GET,args,new BoardRowMapper());
}
```

- query() 메소드
  - SELECT문의 실행 결과가 리스트일 때 사용
  - queryForObject()는 SELECT문의 결과가 객체 하나일 때 사용
  - List컬렉션에 저장되어 리턴

```
List[] query(String sql)
List[] query(String sql, RowMapper<T> rowMapper)
List[] query(String sql, Object[] args, RowMapper<T> rowMapper)
```

- JdbcTemplate 클래스 DAO에 적용시키는 법
  - JdbcDaoSupport 상속

```
@Repository
public class BoardDAOSpring extends JdbcDaoSupport{
...
@Autowired
public void setSuperDataSource(DataSource dataSource){
  super.setDataSource(dataSource);
}
 // 글 등록
 public void insertBoard(BoardVO vo){
   Object[] args = {vo.getTitle(), vo.getWriter(), vo.getContent());
   getJdbcTemplate().update(BOARD_INSERT,args);
 }
...
}
class BoardRowMapper implements RowMapper<BoardVO>{
   public BoardVO mapRow(ResultSet rs, int rowNum) throws SQLException{
    BoardVO board = new BoardVO();
    board.setSeq(rs.getInt("SEQ"));
    board.setTitle(rs.getString("TITLE"));
    board.setWriter(rs.getString("WRITER"));
    board.setContent(rs.getString("CONTENT"));
    board.setRegDate(rs.getDate("REGDATE"));
    board.setCnt(rs.getInt("CNT"));
    return board;
  }
}
```

- getJdbcTemplate() 메소드를 호출하면 JdbcTemplate 객체가 반환되어 DAO의 모든 메소드를 JdbcTemplate 객체로 구현할 수 있다.

○ JdbcTemplate 클래스 <bean> 등록, 의존성 주입

- applicationContext.xml에 DataSource 외에 JDBC 설정을 한다.

```
<!-- Spring JDBC 설정 -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="dataSource"/>
</bean>
```

- DAO 클래스에서 @Autowired 어노테이션을 이용하여 JdbcTemplate 타입의 객체를 의존성 주입 처리한다.

BoardDAOSpring.java

```
...
@Repository
public class BoardDAOSpring{
  @Autowired
  private JdbcTemplate jdbcTemplate;
  ...
 jdbcTemplate.update(BOARD_INSERT,args);
```

```
}
```

2. Mybatis

a. Mybatis란?

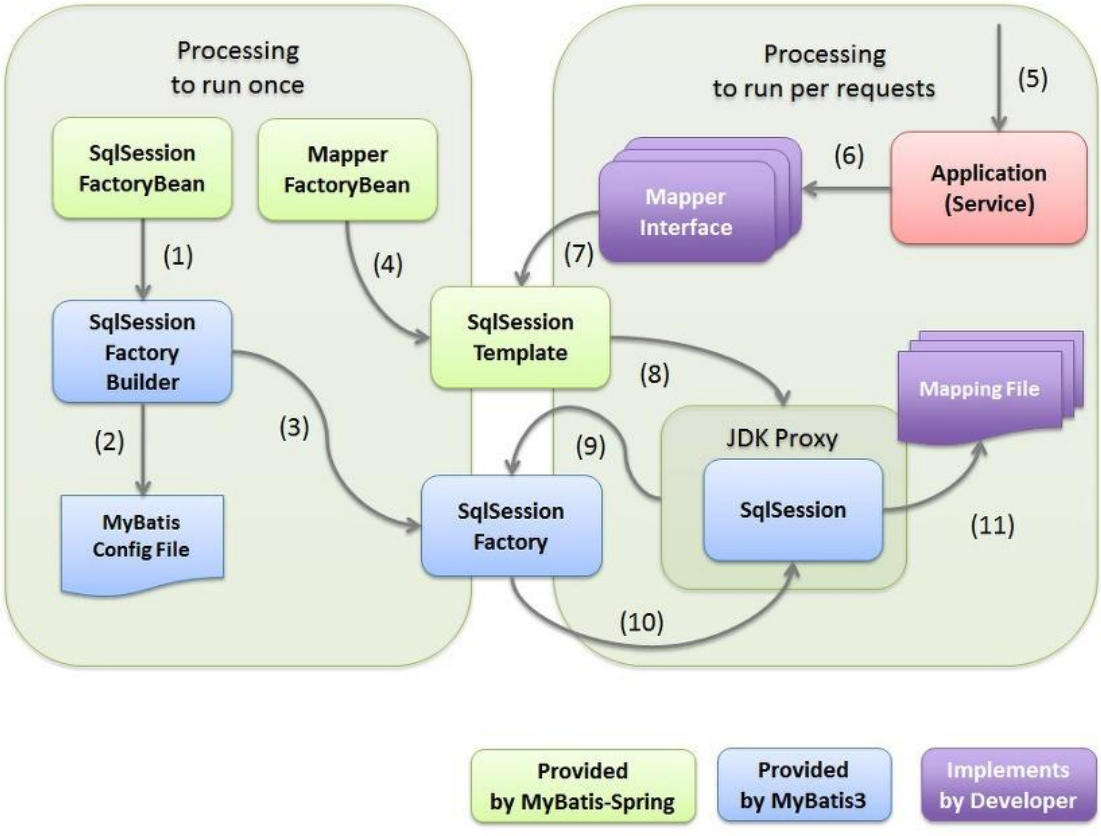
자바의 관계형 데이터베이스 프로그래밍을 좀 더 쉽게 할 수 있게 도와 주는 개발 프레임 워크로서 JDBC를 통해 데이터베이스에 엑세스하는 작업을 캡슐화하고 일반 SQL쿼리, 저장 프로시저 및 고급 매핑을 지원하며 모든 JDBC 코드 및 매개 변수의 중복작업을 제거한다.

b. 특징

i. 한두 줄의 자바 코드로 DB연동을 처리한다.

ii. SQL 명령어를 자바 코드에서 분리하여 XML 파일에 따로 관리한다.

c. Mybatis-Spring 구조



(1)~(4) : 응용 프로그램 시작시 수행되는 프로세스

1	SqlSessionFactoryBean은 SqlSessionFactoryBuilder를 위해 SqlSessionFactory를 빌드하도록 요청합니다.
2	응용 프로그램은 SqlSessionFactoryBuilder를 사용하여 빌드된 SqlSessionFactory에서 SqlSession을 가져옵니다.
3	SqlSessionFactoryBuilder는 MyBatis 구성 파일의 정의에 따라 SqlSessionFactory를 생성합니다. 따라서 생성된 SqlSessionFactory는 Spring DI 컨테이너에 의해 저장됩니다.
4	MapperFactoryBean은 안전한 SqlSession(SqlSessionTemplate) 및 스레드 안전 매퍼 개체(Mapper 인터페이스의 프록시 객체)를 생성합니다. 따라서 생성되는 매퍼 객체는 스프링 DI 컨테이너에 의해 저장되며 서비스 클래스 등에 DI가 적용됩니다. 매퍼 개체는 안전한 SqlSession(SqlSessionTemplate)을 사용하여 스레드 안전 구현을 제공합니다.

(5)~(11) : 클라이언트의 각 요청에 대해 수행되는 프로세스

5	클라이언트가 응용 프로그램에 대한 프로세스를 요청합니다.
6	애플리케이션(서비스)은 DI 컨테이너에서 주입한 매퍼 객체(매퍼 인터페이스를 구현하는 프록시 객체)의 방법을 호출합니다.
7	매퍼 객체는 호출된 메소드에 해당하는 SqlSession (SqlSessionTemplate) 메서드를 호출합니다.
8	SqlSession (SqlSessionTemplate)은 프록시 사용 및 안전한 SqlSession 메서드를 호출합니다.
9	프록시 사용 및 스레드 안전 SqlSession은 트랜잭션에 할당된 MyBatis3 표준 SqlSession을 사용합니다.

	트랜잭션에 할당된 SqlSession이 존재하지 않는 경우 SqlSessionFactory 메서드를 호출하여 표준 MyBatis3의 SqlSession을 가져옵니다.
10	SqlSessionFactory는 MyBatis3 표준 SqlSession을 반환합니다. 반환된 MyBatis3 표준 SqlSession이 트랜잭션에 할당되기 때문에 동일한 트랜잭션 내에 있는 경우 새 SqlSession을 생성하지 않고 동일한 SqlSession을 사용합니다.on 메서드를 호출하고 SQL 실행을 요청합니다.
11	MyBatis3 표준 SqlSession은 매핑 파일에서 실행할 SQL을 가져와 실행합니다.

d. Mybatis JAVA API

SqlSession 객체를 사용해 DAO 구현

i. SqlSessionFactoryBuilder 클래스

1. 해당 클래스의 build() 메소드 이용

   Mybatis 설정 파일을 로딩하여 SqlSessionFactory 객체를 생성

2. 파일 로딩을 위한 입력 스트림 (Reader) 설정

   Mybatis 설정 파일을 로딩하려면 Reader 객체 필요

   Reader 객체는 Resources.getResourceAsReader() 메소드를 사용해서 얻음

```
Reader reader =
   Resources.getResourceAsReader("sql-map-config.xml");
SqlSessionFactory sessionFactory =
   new SqlSessionFactoryBuilder().build(reader);
```

ii. SqlSessionFactory 클래스

1. openSession() 메소드를 통해 SqlSession 객체 생성

```
SqlSession session = sessionFactory.openSession();
session.insert("BoardDAO.insertBoard",vo);
```

iii. SqlSession 클래스

Mapper XML에 등록된 SQL을 실행하기 위한 API 제공

1. selectOne()

```
public Object selectOne(String statement[,Object parameter])
```

   a. 오직 하나의 데이터를 검색하는 SQL문을 실행할 때 사용

   b. 두 개 이상의 레코드가 리턴되면 예외 발생

2. selectList()

```
public List selectList(String statement[,Object parameter])
```

   a. 여러 개의 데이터를 검색하는 SQL문을 실행할 때 사용

3. insert(), update(), delete()

   각각 INSERT, UPDATE, DELETE 구문 실행할 때 사용

   몇 건의 데이터가 처리되었는지를 리턴

```
public int insert(String statement, Object parameter)
```

```
public int update(String statementId, Object parameterObject) throws
SQLException
```

```
public int delete(String statementId, Object parameterObject) throws
SQLException
```

e. 연동을 위해 설정해야 할 File

i. pom.xml

```
<!-- Mybatis -->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.3.1</version>
</dependency>

<!-- Mybatis Spring -->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.2.4</version>
</dependency>
```

ii. applicationContext.xml -> dataSource, SessionFactory 클래스 bean 설정

```
<context:property-placeholder location="classpath:database.properties"/>

<bean id="ds" class="org.apache.commons.dbcp.BasicDataSource" destory-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>

<!-- Spring과 Mybatis 연동 설정 -->
<bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="ds"/>
  <property name="configLocation" value="classpath:mybatis-config.xml"/>
</bean>

<bean class="org.mybatis.spring.SqlSessionTemplate">
  <constructor-arg ref="sessionFactory"></constructor-arg>
</bean>
```

iii. Mybatis 환경설정 파일 작성 (Mybatis configuration file : sql-map-config.xml)

```xml
<?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>

  <!-- Properties 파일 설정 -->
  <properties resource ="db.properties"/>

  <!--Alias 설정-->
  <typeAliases>
   <typeAlias alias="board" type="com.springbook.biz.board.BoardVO"/>
  </typeAliases>

  <!--DataSource 설정-->
  <environments default="development">
   <environment id="development">

     <transactionManager type="JDBC"/>

     <dataSource type="POOLED">
        <property name="driver" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
     </dataSource>

   </environment>
  </environments>

  <!--Sql Mapper 설정-->
  <mappers>
    <mapper resource="mappings/board-mapping.xml"/>
  </mappers>
</configuration>
```

iv. SQL Mapper XML 파일 작성 (Mapping file : board-mapping.xml)

-> Mapper Config 대신 Mapper Interface로 구현하는 법도 있음

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

```
<mapper namespace="BoardDAO">
  <insert id="insertBoard">
    insert into board(seq,title,writer,content) values((select
coalesce(max(seq),0)+ 1
    from board),#{title},#{writer},#{content})</insert>
  <update id="updateBoard">
    update board set title=#{title}, content=#{content} where seq=#{seq}
  </update>
  <delete id="deleteBoard">
    delete board where seq=#{seq}
  </delete>
  <select id="getBoard" resultType="board">
    select * from board where seq=#{seq}
  </select>
  <select id="getBoardList" resultType="board">
    select * from board where title like '%'||#{searchKeyword}||'%'
    order by seq desc
  </select>
</mapper>
```

+ sql-map-config.xml 에서 typeAlias 를 설정하면 board-mapping.xml(Mapper)에서 parameterType, resultType을 쉽게 설정할 수 있다.

```
<typeAliases>
  <typeAlias alias="board" type="com.springbook.biz.board.BoardVO"/>
</typeAliases>
```

```
<select id="getBoard" parameterType="board" resultType="board">
  select * from board where seq = #{seq}
</select>
```

v. DAO에서 할당받은 SqlSession(SqlSessionTemplate) 객체의 메서드를 통해 CRUD를 구현할 수 있다.

vi. insertBoard(BoardVO vo)

```
public void insertBoard(BoardVO vo){
            sqlSession.insert("BoardDAO.insertBoard",vo);
            mybatis.commit();
}
```

f. element

i. <mapper> : root element, namespace속성 사용 -> DAO에서 해당 value로 접근 가능

ii. <select> : select query문 사용시, id 속성은 필수, parameterType, resultType 사용 가능

(resultType : select문으로 ResultSet 객체가 나오면 그 객체를 어떤 타입과 매핑할지 결정시켜줌)

iii. <insert> : INSERT구문 작성시, parameterType, resultType 사용 가능

1. 자식 element로 <selectKey> 사용해서 생성된 키를 쉽게 가져올 수 있는 방법을 제공
2. "BOARD_SEQ"라는 시퀀스로부터 유일한 킷값을 얻어내어 글등록에서 일련번호(seq)값으로 사용하는 설정

```
<insert id="insertBoard" parameterType="board">
   <selectKey keyProperty="seq" resultType="int">
      select board_seq.nextval as seq from dual
   </selectKey>
   insert into board(seq, title, writer, content)
   values(#{seq},#{title},#{writer}, #{content})
</insert>
```

iv. <update>, <delete>

v. resultMap 속성 사용

1. 쿼리가 단순 조회가 아닌 JOIN 구문을 포함하거나 검색된 테이블의 칼럼 이름과 매핑에 사용될 자바 객체의 변수 이름이 다를 때에 정확하게 자바 객체로 매핑시키기 위해서는 resultType을 사용하지 않고 resultMap을 사용한다.
2. Ex

```
<mapper namespace="Board">
```

```
 <resultMap id="boardResult" type="board">
   <id property="seq" column="SEQ"/>
   <result property="title" column="TITLE"/>
   <result property="writer" column="WRITER"/>
   <result property="content" column="CONTENT"/>
   <result property="regDate" column="REGDATE"/>
   <result property="cnt" column="CNT"/>
 </resultMap>

 <select id="getBoardList" parameterType="board" resultMap="boardResult">
 select * from board where title like '%' ||
         # {searchKeyword} || '%' order by seq desc
</mapper>
```

3. 1) 위 설정은 boardResult라는 아이디로 <resultMap>을 설정

   2) PK인 SEQ 칼럼 에만 <id> 사용하고

   3) 나머지 칼럼에는 <result>를 사용해서 쿼리로 얻어낸 칼럼의 값과 BoardVO 객체의 변수를 매핑

   4) 설정된 <resultMap>을 getBoardList로 등록된 쿼리에서 resultMap 속성으로 참조

vi. CDATA Section 사용

1. SQL 구문 내에 '<'나 '>'를 사용하면 XML은 tag의 시작으로 인식해서 에러를 발생
2. CDATA Section으로 SQL 구문을 감싸주면 XML문법에 의해 '<'나 '>'도 단순 문자 데이터(Character DATA)로 인식해서 XML parser가 해석하지 않게 해줌
3. Ex

```
<select id="getBoard" resultType="board">
  <![CDATA[
    select * from board where seq <= #{seq}
  ]]>
</select>
```

vii. 가독성을 위해 쿼리문은 대문자로 작성

g. **동적 SQL 처리 방법**

분기처리를 수정을 할 때 직접 java코드를 손대지 않고 별도의 XML파일로 수정할 수

있어서 유지보수에 이점을 보인다.

i. if

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
   <if test="title != null">
     AND title LIKE #{title}
   </if>
   <if test="author != null and author.name != null ">
      AND author_name LIKE #{author.name}
    </if>
</select>
```

ii. choose (when, otherwise) - Switch 문과 동일

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

iii. trim (where, set)

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

where element는 태그에 의해 컨텐츠가 리턴되면 “WHERE”만을 추가한다.

“AND”나 “OR”로 시작한다면 해당 “AND”나 “OR”을 지워버린다.

위의 동작을 trim을 이용해서 사용자 정의도 아래와 같이 가능하다.

1. <WHERE> 이 앞에 있을 때 컨텐츠가 AND나 OR로 시작하면

   AND나 OR 을 지움

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

2. <SET>에서 “,”가 컨텐츠의 마지막에 있으면 “,”를 지움

```
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

iv. foreach

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
```

```
  <foreach item="item" index="index" collection="list"
      open="(" separator="," close=")">
        #{item}
  </foreach>
</select>
```

시작, 종료 문자열을 open, close 속성으로 정할 수 있고 separator로 구분자를 지정할 수 있다.

collection을 받아 각 요소의 value와 index를 이용하기 위한 이름을 item, index 속성으로 명시한다.

3. Spring의 Transaction

a. 트랜잭션이란?

i. 데이터베이스의 상태를 변환시키는(DML문) 하나의 논리적 기능을 수행하기 위한 작업의 단위

ii. 또는 한꺼번에 모두 수행되어야 할 일련의 연산

iii. 하나의 쿼리에 대한 commit/rollback을 넘어서, 업무처리같이 단위는 하나지만 그 안에 insert/update/delete (+ select) 쿼리 여러개가 순차적으로 모두 에러없이 실행되어야 처리되었다고 말할 수 있을때, 이렇게 안에 세부 순서를 가진 업무 단위 혹은 이를 처리하기 위한 기술

b. 트랜잭션의 특징

i. 트랜잭션은 데이터베이스 시스템에서 병행 제어 및 회복 작업 시 처리되는 작업의 논리적 단위이다.

ii. 사용자가 시스템에 대한 서비스 요구 시 시스템이 응답하기 위한 상태 변환 과정의 작업단위이다.

iii. 하나의 트랜잭션은 Commit되거나 Rollback된다.

c. 트랜잭션의 성질(ACID)

i. Atomicity(원자성)

1. 트랜잭션의 연산은 데이터베이스에 모두 반영되든지 아니면 전혀 반영되지 않아야 한다.
2. 트랜잭션 내의 모든 명령은 반드시 완벽히 수행되어야 하며, 모두가 완벽히 수행되지 않고 어느하나라도 오류가 발생하면 트랜잭션 전부가 취소되어야 한다.

ii. Consistency(일관성)

1. 트랜잭션이 그 실행을 성공적으로 완료하면 언제나 일관성 있는 데이터베이스 상태로 변환한다.
2. 시스템이 가지고 있는 고정요소는 트랜잭션 수행 전과 트랜잭션 수행 완료 후의 상태가 같아야 한다.

iii. Isolation(독립성,격리성)

1. 둘 이상의 트랜잭션이 동시에 병행 실행되는 경우 어느 하나의 트랜잭션 실행중에 다른 트랜잭션의 연산이 끼어들 수 없다.
2. 수행중인 트랜잭션은 완전히 완료될 때까지 다른 트랜잭션에서 수행 결과를 참조할 수 없다.

iv. Durability(영속성,지속성)

1. 성공적으로 완료된 트랜잭션의 결과는 시스템이 고장나더라도 영구적으로 반영되어야 한다.
2. 트랜잭션을 성공적으로 마치면 결과가 항상 저장되어야 한다.

d. 연산

i. Commit연산

Commit 연산은 한 개의 논리적 단위(트랜잭션)에 대한 작업이 성공적으로 끝났고 데이터베이스가 다시 일관된 상태에 있을 때, 이

트랜잭션이 행한 갱신 연산이 완료된 것을 트랜잭션 관리자에게 알려주는 연산이다.

ii. Rollback연산

1. Rollback 연산은 하나의 트랜잭션 처리가 비정상적으로 종료되어 데이터베이스의 일관성을 깨뜨렸을 때, 이 트랜잭션의 일부가 정상적으로 처리되었더라도 트랜잭션의 원자성을 구현하기 위해 트랜잭션이 행한 모든 연산을 취소(Undo)하는 연산이다.
2. Rollback 시에는 해당 트랜잭션을 재시작하거나 폐기한다.

e. 트랜잭션의 상태

i. 활동(Active) : 트랜잭션이 실행중인 상태

ii. 실패(Failed) : 트랜잭션 실행에 오류가 발생하여 중단된 상태

iii. 철회(Aborted) : 트랜잭션이 비정상적으로 종료되어 Rollback 연산을 수행한 상태

iv. 부분 완료(Partially Committed):트랜잭션의 마지막 연산까지 실행했지만, Commit연산이 실행되기 직전의 상태

v. 완료(Committed) : 트랜잭션이 성공적으로 종료되어 Commit 연산을 실행한 후의 상태

f. 프록시 객체

g. 트랜잭션이 필요한 상황

i. 여러 쿼리문이 전부 다 정상적으로 실행되어야 완료되는 작업이 있을 때, 해당 작업을 수행하는 중 오류가 생겨 일부 쿼리문만 처리가 완료가 되면 의도치 않은 데이터 값이 데이터베이스에 남아있게 될 수 있으므로 처음 작업을 **수행하기 전 상태로 아예 roll-back 해야되는 경우**가 발생한다. 이 때 transaction 처리를 하면 개발자가 직접 이전 데이터 상태로 복구하지 않아도 되서 편리성을 준다.

h. Spring에서 AOP로 선언적 transaction Ex

- applicationContext.xml

```
<beans xmlns=..
      xmlns:tx="http://www.springframework.org/schema/tx"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/tx
                  http://www.springframework.org/schema/tx/spring-tx.xsd">

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

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:pointcut id="userPointcut" expression="execution(*
  com.spring.biz.usercomponent.. *Impl. *(..))"/>

  <aop:advisor advice-ref="txAdvice" pointcut-ref="userPointcut"/>
</aop:config>
```

i. annotation으로 transaction 설정 Ex

- applicationContext.xml

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

- UserServiceImpl.java (Service)

```
@Transactional(isolation=isolation.DEFAULT,propagation=Propagation.REQUIRES_NEW)
public void registerUser(UserVO vo) {
  boolean isOverlap = this.isOverlapUser(vo);
  if(isOverlap == false){
```

```
        dao.insertUser(vo);
    }
}
```

- 인터페이스, 클래스, 메소드에 전부 @Transactional 적용이 가능
- 인터페이스에 @Transactional 적용을 한 경우, 해당 인터페이스의 모든 메소드에 @Transactional이 적용 됨
- 인터페이스 - 클래스 - 메소드 모두에 @Transactional이 설정되있다면 스프링은 메소드 어노테이션을 제일 먼저 고려, 그 다음으로 클래스, 인터페이스 순서로 적용함

4. DAO 생성 및 Mapper 구현, MySQL과 데이터 주고받기

a. github Link
https://github.com/HyunSeokJung-student/schoolpoint

b. Mapper

```xml
<!-- DTD 선언 -->
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- SQL Mapping -->
<mapper namespace="UserDAO">

    <insert id="registerUser" parameterType="userVO">
        <![CDATA[
            INSERT INTO USER(
                ID,
                PASSWORD,
                NAME,
                SCHOOLNUMBER
            ) VALUES (
                #{id},
                #{password},
                #{name},
                #{schoolNumber}
            )
        ]]>
    </insert>

    <select id="getUser" parameterType="userVO" resultType="userVO">
        <![CDATA[
            SELECT * FROM USER WHERE ID=#{id}
        ]]>
    </select>
```

```xml
    <select id="loginUser" parameterType="userDTO" resultType="userVO">
        <![CDATA[
            SELECT * FROM USER WHERE ID = #{id} AND PASSWORD = #{password};
        ]]>
    </select>

    <select id="getUserList" resultType="userVO">
        <![CDATA[
            SELECT * FROM USER
        ]]>
    </select>

</mapper>
```

c. DAO (mybatis의 SqlSessionTemplate 적용)

```java
package com.calculation.schoolpoint.model;

import org.mybatis.spring.SqlSessionTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public class UserDAO {

    @Autowired
    private SqlSessionTemplate sqlSessionTemplate;

    public void registerUser(UserVO vo){
        sqlSessionTemplate.insert( statement: "UserDAO.registerUser",vo);
    }

    public UserVO getUser(UserVO vo) { return (UserVO)sqlSessionTemplate.selectOne( statement: "UserDAO.getUser",vo); }

    public UserVO login(UserDTO dto){
        return (UserVO)sqlSessionTemplate.selectOne( statement: "UserDAO.loginUser",dto);
    }

    public List<UserVO> getListUser(UserVO vo) { return sqlSessionTemplate.selectList( statement: "UserDAO.getUserList"); }

}
```