

The background features a series of overlapping, wavy, ribbon-like shapes in various shades of green and white, creating a sense of motion and depth. The colors range from a vibrant lime green to a soft, pale green, with some areas appearing as white highlights. The overall effect is a modern, organic, and flowing design.

# **Spring MyBatis**

# MyBatis?

- 객체와 테이블을 매핑하는 ORM과는 다르게 객체와 SQL문을 매핑하는 프레임워크
  - 독자적인 질의 언어를 지원하지 않고 SQL 사용
- 장점
  - 저수준 JDBC 코드의 복잡성 제거
  - 친근한 SQL 기반으로 초기 학습곡선 완만
  - 기존 데이터베이스와 호환성 높음
  - Spring 프레임워크와 통합 기능 제공
  - 성능 우수

# MyBatis – 스프링 연동

- 의존성 패키지 등록 (pom.xml)

- MyBatis 패키지 등록

```
<dependency>  
  <groupId>org.mybatis</groupId>  
  <artifactId>mybatis</artifactId>  
  <version>3.2.8</version>  
</dependency>
```

- MyBatis – Spring 연동 패키지 등록

```
<dependency>  
  <groupId>org.mybatis</groupId>  
  <artifactId>mybatis-spring</artifactId>  
  <version>1.2.2</version>  
</dependency>
```

# MyBatis – 스프링 연동

## ■ 스프링 설정 파일 작성 (/resources/mybatis-config.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>

  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC" />
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/demoweb" />
        <property name="username" value="devadmin" />
        <property name="password" value="mysql" />
      </dataSource>
    </environment>
  </environments>

  <mappers>
    <mapper resource="com/example/springmybatis/BoardMapper.xml" />
  </mappers>

</configuration>
```

# MyBatis – 스프링 연동

## ▪ MyBatis 클래스

종류	설명
SqlSessionFactoryBuilder	<ul style="list-style-type: none"><li>• SqlSessionFactory를 생성하는 객체</li><li>• mybatis-config.xml 설정 파일 사용</li></ul>
SqlSessionFactory	<ul style="list-style-type: none"><li>• SqlSession을 생성하는 객체</li></ul>
SqlSession	<ul style="list-style-type: none"><li>• SQL을 실행하는 객체</li><li>• Mapper 파일의 매핑 정보 사용</li></ul>

## ▪ MyBatis 연동 스프링 클래스

종류	설명
SqlSessionFactoryBean	<ul style="list-style-type: none"><li>• SqlSessionFactoryBuilder를 사용해서 SqlSessionFactory 생성</li><li>• 스프링의 FactoryBean 인터페이스 구현 → getObject 메서드 재정의를 통해 SqlSessionFactory 객체 반환</li></ul>
SqlSessionTemplate	<ul style="list-style-type: none"><li>• SqlSession을 Wrapping한 객체</li></ul>

# MyBatis – 스프링 연동

- MyBatis – 스프링 연동 빈 등록
  - SqlSessionFactory

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
    <property name="dataSource" ref="dataSource" />  
    <property name="configLocation" value="classpath:mybatis-config.xml"/>  
</bean>
```

- SqlSessionTemplate

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">  
    <constructor-arg index="0" ref="sqlSessionFactory" />  
</bean>
```

# MyBatis 설정 파일

## ■ 구성 요소

종류	설명
environments	데이터 소스 및 트랜잭션 관리자 환경 설정
mappers	SQL 매핑 파일 또는 인터페이스 경로 지정
properties	다른 요소에서 재사용 할 수 있도록 설정 정보를 분리 정의
typeAliases	클래스의 전체 경로명 대신 사용할 별명 지정
typeHandlers	Java와 JDBC 타입 사이의 변환을 처리하는 핸들러 설정
settings	MyBatis 행위를 조정하기 위한 값 설정
objectFactory	MyBatis가 결과 객체인 인스턴스를 생성하기 위해 제공하는 objectFactory를 커스터마이징하는 클래스를 사용하는 경우에 설정 커스터마이징 클래스는 DefaultFactory 클래스 상속
plugins	MyBatis가 매핑을 수행할 때 사용하는 메서드 호출을 가로채기 위한 플러그인을 커스터마이징하는 클래스를 사용하는 경우에 설정 커스터마이징 클래스는 Intercept 인터페이스 구현





# MyBatis 매핑 설정 파일

## ■ 매핑 설정 파일 형식

```
<?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="com.ensoa.order.entity.mapper.ProductMapper">

    <resultMap id="ProductResult" type="com.ensoa.order.entity.ProductEntity" >..

    <select id="findAll" resultType="ProductEntity">..

    <select id="findById" parameterType="long" resultType="ProductEntity">..

    <update id="update" parameterType="ProductEntity">..

    <delete id="delete" parameterType="long">..

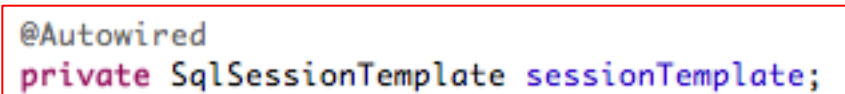
    <insert id="insert" parameterType="ProductEntity" ..

</mapper>
```

# MyBatis ■ 이용한 SQL 실행

## ▪ SqlSessionTemplate 의존성 주입

```
@Repository("productRepository")  
public class ProductRepositoryMyBatis implements ProductRepository {  
    @Autowired  
    private SqlSessionTemplate sessionTemplate;
```



```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">  
    <constructor-arg index="0" ref="sqlSessionFactory" />  
</bean>
```

# MyBatis■ 이용한 SQL 실행

## ■ Mapper Interface 구현 객체 의존성 주입

```
public interface ProductMapper {  
    List<ProductEntity> findAll();  
    List<ProductEntity> findAll(RowBounds rowBounds);  
    ProductEntity findById(long id);  
    void insert(ProductEntity Product);  
    void update(ProductEntity Product);  
    void delete(long id);  
}
```

매핑 설정 파일의 Namespace와 일치하는 패키지.인터페이스를 만들고 <select>, <insert> 등의 요소에 부여된 id와 일치하는 추상메서드 선언

```
<bean id="productMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">  
    <property name="mapperInterface" value="com.ensoa.order.entity.mapper.ProductMapper" />  
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />  
</bean>
```

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
    <property name="dataSource" ref="dataSource" />  
    <property name="configLocation" value="classpath:mybatis-config.xml" />  
</bean>
```

```
@Repository("productRepository")  
public class ProductRepositoryMyBatis implements ProductRepository {  
    @Autowired  
    private ProductMapper mapper;
```

# Select Mapping XML

## ■ 형식 및 속성

```
<select
  id="selectPerson"
  parameterType="int"
  parameterMap="deprecated"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10000"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
```

속성	설명
id	구문을 찾기 위해 사용될 수 있는 명명 공간내 유일한 구분자
parameterType	구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭
resultType	이 구문에 의해 리턴되는 기대타입의 패키지 경로를 포함한 전체 클래스명이나 별칭.
resultMap	외부 resultMap 의 참조명.
flushCache	이 값을 true 로 설정하면, 구문이 호출될 때 마다 캐시 제거
useCache	이 값을 true 로 셋팅하면, 구문의 결과를 캐시
timeout	예외가 던져지기 전에 데이터베이스의 요청 결과를 기다리는 최대시간 설정
fetchSize	지정된 수만큼의 결과를 리턴하도록 하는 드라이버 힌트 값.
statementType	STATEMENT, PREPARED 또는 Callable 중 하나를 선택할 수 있다.
resultSetType	FORWARD_ONLY SCROLL_SENSITIVE SCROLL_INSENSITIVE 중 하나를 선택할 수 있다.

# Select 매핑

## ■ 사용 사례 (다중 행 반환 - 단순 매핑)

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;  
}
```

```
<select id="findAll" resultType="ProductEntity">  
    SELECT PRODUCT_ID AS ID, NAME, PRICE, DESCRIPTION FROM PRODUCT  
</select>
```

```
public List<ProductEntity> findAll() {  
    List<ProductEntity> products =  
        sessionTemplate.selectList("com.ensoa.order.entity.mapper.ProductMapper.findAll");  
    return products;  
}
```

# Select 매핑

## ■ 사용 사례 (다중 행 반환 - 매핑 인터페이스 사용)

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;  
}
```

```
<mapper namespace="com.ensoa.order.entity.mapper.ProductMapper">  
  
    <select id="findAll" resultType="ProductEntity">  
        SELECT PRODUCT_ID AS ID, NAME, PRICE, DESCRIPTION FROM PRODUCT  
    </select>  
</mapper>
```

```
public interface ProductMapper {  
    List<ProductEntity> findAll();  
    List<ProductEntity> findAll(RowBounds rowBounds);  
    ProductEntity findById(long id);  
    void insert(ProductEntity product);  
    void update(ProductEntity product);  
    void delete(long id);  
}
```

```
@Override  
public List<ProductEntity> findAll() {  
    List<ProductEntity> products = mapper.findAll();  
    return products;  
}
```

# Select 매핑

## ■ 사용 사례 (단일 행 반환 - 매핑 인터페이스 사용)

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;  
}
```

```
<select id="findById" parameterType="long" resultType="ProductEntity">  
    SELECT PRODUCT_ID AS ID, NAME, PRICE, DESCRIPTION FROM PRODUCT WHERE PRODUCT_ID = #{id}  
</select>
```

```
public interface ProductMapper {  
    List<ProductEntity> findAll();  
    List<ProductEntity> findAll(RowBounds rowBounds);  
    ProductEntity findById(long id);  
    void insert(ProductEntity product);  
    void update(ProductEntity product);  
    void delete(long id);  
}
```

```
@Override  
public ProductEntity findOne(long id) {  
    ProductEntity product = mapper.findById(id);  
    return product;  
}
```

# Insert, Update, Delete Mapping XML

## ■ Insert, Update, Delete

```
<insert
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  keyProperty=""
  keyColumn=""
  useGeneratedKeys=""
  timeout="20">

<update
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<delete
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">
```

속성	설명
id	구문을 찾기 위해 사용될 수 있는 명명공간내 유일한 구분자
parameterType	구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭
flushCache	이 값을 true 로 셋팅하면 구문이 호출될 때 마다 캐시 제거
timeout	예외가 던져지기 전에 데이터베이스의 요청 결과를 기다리는 최대시간을 설정
statementType	STATEMENT, PREPARED 또는 CALLABLE 중 하나를 선택
useGeneratedKeys	(입력(insert, update)에만 적용) 데이터베이스에서 내부적으로 생성한 키(예를 들어, MySQL 또는 SQL Server 와 같은 RDBMS 의 자동 증가 필드)를 받는 JDBC getGeneratedKeys 메서드를 사용하도록 설정하다. 디폴트는 false 이다.
keyProperty	(입력(insert, update)에만 적용) getGeneratedKeys 메서드나 insert 구문의 selectKey 하위 요소에 의해 리턴된 키를 셋팅할 프로퍼티를 지정..
keyColumn	(입력(insert, update)에만 적용) 생성키를 가진 테이블의 컬럼명을 셋팅. 키 컬럼이 테이블이 첫 번째 컬럼이 아닌 데이터베이스에서만 필요



# Insert 매핑

## ■ 사용 사례 (매핑 인터페이스 사용)

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;  
}
```

```
<insert id="insert" parameterType="ProductEntity"  
    useGenerateKeys="true" keyProperty="id">  
    INSERT INTO PRODUCT(NAME, PRICE, DESCRIPTION) VALUES (#{name}, #{price}, #{description})  
</insert>
```

```
public interface ProductMapper {  
    List<ProductEntity> findAll();  
    List<ProductEntity> findAll(RowBounds rowBounds);  
    ProductEntity findById(long id);  
    void insert(ProductEntity product);  
    void update(ProductEntity product);  
    void delete(long id);  
}
```

```
@Override  
public void save(ProductEntity product) {  
    mapper.insert(product);  
}
```

# Update 매핑

## ■ 사용 사례 (매핑 인터페이스 사용)

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;  
}
```

```
<update id="update" parameterType="ProductEntity">  
    UPDATE PRODUCT  
    SET NAME = #{name}, PRICE = #{price}, DESCRIPTION = #{description}  
    WHERE PRODUCT_ID = #{id}  
</update>
```

```
public interface ProductMapper {  
    List<ProductEntity> findAll();  
    List<ProductEntity> findAll(RowBounds rowBounds);  
    ProductEntity findById(long id);  
    void insert(ProductEntity product);  
    void update(ProductEntity product);  
    void delete(long id);  
}
```

```
@Override  
public void update(ProductEntity product) {  
    mapper.update(product);  
}
```

# Delete 매핑

## ■ 사용 사례 (매핑 인터페이스 사용)

```
<delete id="delete" parameterType="long">  
    DELETE FROM PRODUCT WHERE PRODUCT_ID = #{id}  
</delete>
```

```
public interface ProductMapper {  
    List<ProductEntity> findAll();  
    List<ProductEntity> findAll(RowBounds rowBounds);  
    ProductEntity findById(long id);  
    void insert(ProductEntity product);  
    void update(ProductEntity product);  
    void delete(long id);  
}
```

```
@Override  
public void update(ProductEntity product) {  
    mapper.update(product);  
}
```

# 자동 증가 컬럼 처리

## ▪ Generate Key 설정 : 컬럼 값 자동 생성 처리

자동 증가 컬럼

```
<insert id="insertAuthor" useGeneratedKeys="true"
      keyProperty="id">
  insert into Author (username,password,email,bio)
  values ({username},{password},{email},{bio})
</insert>
```

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author
    (id, username, password, email,bio, favourite_section)
  values
    ({id}, {username}, {password}, {email}, {bio}, {favouriteSection,jdbcType=VARCHAR})
</insert>
```

생성된 데이터로 설정

# ResultMap

- 매핑 설정 파일에 작성한 SQL과 객체의 필드 사이의 이름 불일치와 같은 문제를 해결하기 위해 사용자 정의 매핑 설정인 ResultMap 사용

```
<resultMap id="ProductResult" type="com.ensoa.order.entity.ProductEntity" >
  <id property="id" column="product_id"/>
  <result property="name" column="name"/>
  <result property="price" column="price"/>
  <result property="description" column="description"/>
</resultMap>

<select id="findAll" resultType="ProductEntity">
  SELECT PRODUCT_ID, NAME, PRICE, DESCRIPTION FROM PRODUCT
</select>
```

# Parameter 형식

## ▪ 단일 값 전달인자

```
<select id="selectUsers" resultType="User">
  select id, username, password
  from users
  where id = #{id}
</select>
```

## ▪ 객체 전달인자

```
<insert id="insertUser" parameterType="User">
  insert into users (id, username, password)
  values (#{id}, #{username}, #{password})
</insert>
```

## ▪ HashMap 전달인자

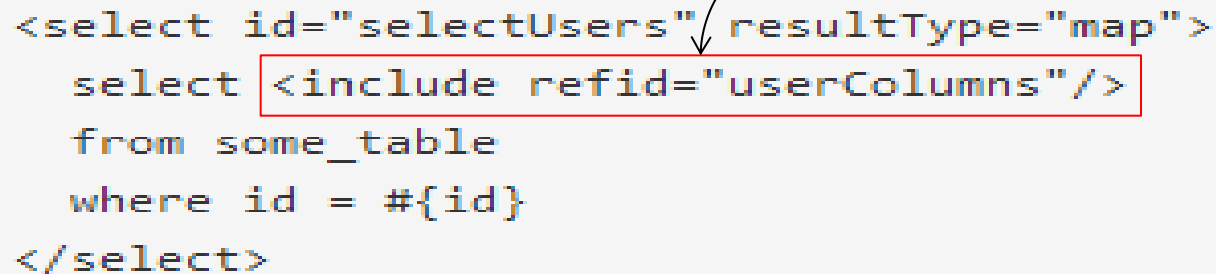
```
<insert id="insertUser" parameterType="hashMap">
  INSERT INTO user (id, username, password)
  VALUES (#{id}, #{username}, #{password})
</insert>
```

# SQL 재사용

- <sql> 요소로 재사용 가능한 SQL 선언

```
<sql id="userColumns"> id,username,password </sql>
```

```
<select id="selectUsers" resultType="map">  
  select <include refid="userColumns"/>  
  from some_table  
  where id = #{id}  
</select>
```



# 1:1 관계 매핑

- 데이터베이스 테이블 간의 1:1 관계를 객체에 매핑하기 위해 다음과 같은 방법 사용

방식	설명
포함 방식	<ul style="list-style-type: none"><li>결과 맵에 포함하는 객체의 필드를 모두 포함하는 방식</li></ul>
중첩 결과 방식	<ul style="list-style-type: none"><li>〈association〉요소를 사용</li><li>포함하는 객체에 대한 별도의 ResultMap을 참조하는 방식</li></ul>
중첩 SELECT 방식	<ul style="list-style-type: none"><li>〈association〉요소를 사용</li><li>포함하는 객체를 조회하는 〈select〉 요소를 참조하는 방식</li></ul>



# 1:1 관계 매핑

## 포함 방식

```
public class OrderItemEntity {  
    private long id;  
    private int amount;  
    private ProductEntity product;
```

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;
```

```
<resultMap id="OrderItemResultEmbedded" type="OrderItemEntity">  
    <id property="id" column="order_item_id" />  
    <result property="amount" column="amount"/>  
    <result property="product.id" column="product_id"/>  
    <result property="product.name" column="name"/>  
    <result property="product.price" column="price"/>  
    <result property="product.description" column="description"/>  
</resultMap>  
  
<select id="findAll" resultMap="OrderItemResultEmbedded">  
    SELECT ORDER_ITEM_ID, AMOUN, ORDER_ID,  
           PRODUCT.PRODUCT_ID, NAME, PRICE, DESCRIPTION  
    FROM ORDER_ITEM  
    LEFT INNER JOIN PRODUCT  
    ON ORDER_ITEM.PRODUCT_ID = PRODUCT_PRODUCT_ID  
</select>
```

# 1:1 관계 매핑

## 중첩 결과 방식

```
public class OrderItemEntity {  
    private long id;  
    private int amount;  
    private ProductEntity product;
```

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;
```

```
<resultMap id="OrderItemResultNested" type="OrderItemEntity">  
    <id property="id" column="order_item_id" />  
    <result property="amount" column="amount"/>  
    <association property="product" column="product_id"  
        resultMap="com.ensoa.order.entity.mapper.ProductMapper.ProductResult"/>  
</resultMap>  
<select id="findAll" resultMap="OrderItemResultNested">  
    SELECT ORDER_ITEM_ID, AMOUN, ORDER_ID,  
           PRODUCT.PRODUCT_ID, NAME, PRICE, DESCRIPTION  
    FROM ORDER_ITEM  
    LEFT INNER JOIN PRODUCT  
    ON ORDER_ITEM.PRODUCT_ID = PRODUCT.PRODUCT_ID  
</select>
```

```
<mapper namespace="com.ensoa.order.entity.mapper.ProductMapper">  
    <resultMap id="ProductResult" type="com.ensoa.order.entity.ProductEntity">  
        <id property="id" column="product_id"/>  
        <result property="name" column="name"/>  
        <result property="price" column="price"/>  
        <result property="description" column="description"/>  
    </resultMap>
```

# 1:1 관계 매핑

## ▪ 중첩 select 방식

```
public class OrderItemEntity {  
    private long id;  
    private int amount;  
    private ProductEntity product;  
}
```

```
public class ProductEntity {  
    private long id;  
    private String name;  
    private int price;  
    private String description;  
}
```

```
<resultMap id="OrderItemResult" type="OrderItemEntity">  
    <id property="id" column="order_item_id" />  
    <result property="amount" column="amount" />  
    <association property="product" column="product_id"  
        select="com.ensoa.order.entity.mapper.ProductMapper.findById" />  
</resultMap>  
<select id="findAll" resultMap="OrderItemResult">  
    SELECT * FROM ORDER_ITEM  
</select>
```

```
<mapper namespace="com.ensoa.order.entity.mapper.ProductMapper">  
    <select id="findById" parameterType="long" resultType="ProductEntity">  
        SELECT PRODUCT_ID AS ID, NAME, PRICE, DESCRIPTION FROM PRODUCT WHERE PRODUCT_ID = #{id}  
    </select>
```

# 1 : Many 관계 매핑

- 데이터베이스 테이블 간의 1 : Many 관계를 객체에 매핑하기 위해 다음과 같은 방법 사용

방식	설명
중첩 결과 방식	<ul style="list-style-type: none"><li>• &lt;collection&gt;요소를 사용</li><li>• 포함하는 객체에 대한 별도의 ResultMap을 참조하는 방식</li></ul>
중첩 SELECT 방식	<ul style="list-style-type: none"><li>• &lt;collection&gt;요소를 사용</li><li>• 포함하는 객체를 조회하는 &lt;select&gt; 요소를 참조하는 방식</li></ul>

# 동적 SQL

## ■ 애플리케이션 실행 시점에 수행할 SQL문을 결정하는 방식

- 검색 조건을 사용자 선택하는 경우
- Update 수행 시 선택적으로 특정 컬럼만 변경하는 경우

## ■ 종류

종류	구문
if	<code>&lt;if test= "name != null" &gt;</code> <code>&lt;/if&gt;</code>
choose when otherwi se	<code>&lt;choose&gt;</code> <code>&lt;when test= "condition == 'one' " &gt;</code>  <code>&lt;/when&gt;</code> <code>&lt;when test= "condition == 'one' " &gt;</code>  <code>&lt;/when&gt;</code> <code>&lt;otherwise&gt;</code>  <code>&lt;/otherwise&gt;</code> <code>&lt;/choose&gt;</code>

종류	구문
where	<code>&lt;where&gt;</code> <code>&lt;if test= "name != null" &gt;&lt;/if&gt;</code> <code>&lt;/where&gt;</code>
trim	생략
foreach	<code>&lt;foreach item= "item" collection= "items"</code> <code>open= "( "seperator= ", " close= ")" &gt;</code> <code>&lt;/foreach&gt;</code>
set	<code>&lt;set&gt;</code> <code>&lt;if test= "name != null" &gt;&lt;/if&gt;</code> <code>&lt;/set&gt;</code>

# 동적 SQL 예제

```
<select id="find"
  parameterType="com.ensoa.order.domain.CustomerSearch" resultMap="CustomerResult">
  SELECT * FROM CUSTOMER
  <trim prefix="WHERE" prefixOverrides="AND | OR">
    <where>
      <if test="name != null">
        NAME LIKE #{name}
      </if>
      <if test="address != null">
        AND ADDRESS LIKE #{address}
      </if>
      <if test="email != null">
        AND EMAIL LIKE #{email}
      </if>
    </where>
  </trim>
</select>
<update id="update" parameterType="CustomerEntity">
  UPDATE CUSTOMER
  <set>
    <if test="name != null">NAME = #{name}, </if>
    <if test="address != null">ADDRESS = #{address}, </if>
    <if test="email != null">EMAIL = #{email} </if>
  </set>
  WHERE CUSTOMER_ID = #{id}
</update>
```

## 기타 구문

### ■ 저장 프로시저 호출

- 형식 : { CALL 저장프로시저이름(전달인자목록)

```
<select id="findBySp" parameterType="string" resultMap="CustomerResult"
        statementType="CALLABLE">
    { CALL GET_CUSTOMERS("#{name, mode=IN, jdbcType=VARCHAR}) }
</select>
```

### ■ 페이징 조회

```
@Override
public List<CustomerEntity> findAll(Pageable page) {
    RowBounds rowBounds = new RowBounds(page.getIndex(), page.getSize());
    List<CustomerEntity> customers = mapper.findAll(rowBounds);
    return customers;
}
```

# 어노테이션을 이용한 매핑

- 매퍼 인터페이스에 적용
- 종류

종류	매핑 구문
@Select	<code>&lt;select&gt;[/select&gt;</code>
@Insert	<code>&lt;insert&gt;[/insert&gt;</code>
@Update	<code>&lt;update&gt;[/update&gt;</code>
@Delete	<code>&lt;delete&gt;[/delete&gt;</code>
@Options	<code>&lt;insert useGeneratedKeys= "true" keyProperty= "prop" &gt;[/insert&gt;</code> 저장프로시저 호출
@SelectKey	<code>&lt;selectKey&gt;[/selectKey&gt;</code>
@Results	<code>&lt;resultMap&gt;[/resultMap&gt;</code>
@Result	<code>&lt;id&gt;[/id&gt;</code> 또는 <code>&lt;result&gt;[/result&gt;</code>
@One, @Many	1:1 또는 1:Many 관계 매핑
@SelectProvider @InsertProvider @UpdateProvider @DeleteProvider	동적 SQL 매핑



## ▪ Insert, update, delete, select

```
@Select("SELECT PRODUCT_ID AS ID, NAME, PRICE, DESCRIPTION FROM PRODUCT")
List<ProductEntity> findAll();

@Insert("INSERT INTO PRODUCT(NAME, PRICE, DESCRIPTION) " +
        "VALUES ({name}, {price}, {description})")
@Options(useGeneratedKeys=true, keyProperty="prodId")
void insert(ProductEntity Product);

@Update("UPDATE PRODUCT " +
        "SET NAME = {name}, PRICE = {price}, DESCRIPTION = {description} " +
        "WHERE PRODUCT_ID = {id}")
void update(ProductEntity Product);

@Delete("DELETE FROM PRODUCT WHERE PRODUCT_ID = {id}")
void delete(long id);
```

자동증가컬럼

## ▪ ResultMap

```
@Select("SELECT PRODUCT_ID AS ID, NAME, PRICE, DESCRIPTION " +
        "FROM PRODUCT WHERE PRODUCT_ID={id}")
@Results( {
    @Result(id=true, property="id", column="product_id"),
    @Result(property="name", column="name"),
    @Result(property="price", column="price"),
    @Result(property="description", column="description")
})
ProductEntity findById(long id);
```

## ▪ resultMap

```
@Select("SELECT PRODUCT_ID AS ID, NAME, PRICE, DESCRIPTION " +  
        "FROM PRODUCT  WHERE PRODUCT_ID=#{id}")  
@Results( {  
    @Result(id=true, property="id", column="product_id"),  
    @Result(property="name", column="name"),  
    @Result(property="price", column="price"),  
    @Result(property="description", column="description")  
})  
ProductEntity findById(long id);
```

## ▪ 관계 Cardinality

```
@Select("SELECT * FROM ORDERS WHERE ORDERS.ORDER_ID = #{orderId}")  
@Results( {  
    @Result(id=true, property="id", column="order_id" ),  
    @Result(property="orderDate", column="order_date"),  
    @Result(property="customer", column="customer_id",  
            one=@One(select="com.ensoa.order.entity.mapper.CustomerMapper.findById")),  
    @Result(property="items", column="order_id",  
            many=@Many(select="com.ensoa.order.entity.mapper.OrderItemMapper..findByOrderId"))  
})  
OrderEntity findById(long id);
```

## ▪ 동적 SQL

```
public String find(final CustomerSearch customerSearch) {  
    return new SQL() {{  
        SELECT("* ");  
        FROM("CUSTOMER");  
        if(customerSearch.getName() != null) {  
            WHERE("NAME LIKE #{name}");  
        }  
        if(customerSearch.getAddress() != null) {  
            WHERE("ADDRESS LIKE #{address}");  
        }  
        if(customerSearch.getEmail() != null) {  
            WHERE("EMAIL LIKE #{email}");  
        }  
    }}.toString();  
}
```

```
@SelectProvider(type=CustomerSqlProvider.class, method="find")  
@ResultMap("com.ensoa.order.entity.mapper.CustomerMapper.CustomerResult")  
List<CustomerEntity> find(CustomerSearch customerSearch);
```

## ▪ 저장 프로시저 호출

```
@Select(value= "{CALL GET_CUSTOMERS(#{name, mode=IN, jdbcType=VARCHAR})}")  
@ResultMap("com.ensoa.order.entity.mapper.CustomerMapper.CustomerResult")  
@Options(statementType = StatementType.CALLABLE)  
List<CustomerEntity> findBySp(String name);
```