6. MyBatis

마이바티스는 데이터베이스와 대화할 때 사용할 수 있는 기술 중에 하나입니다. SQL쿼리를 DAO클래스로부터 분리함으로써 SQL관리성을 증대시켜주는 데이터베이스 처리기술 입니다.

앞서 살펴 본 Spring JDBC 기술을 사용하는 경우 개발자는 2가지 종류의 로직만 기술하면 되었습니다. 첫 번째는 SQL쿼리작성이고 두 번째는 결과처리 로직이었습니다.

그런데 변하는 것과 변하지 않는 것이 같이 있다면 분리하는 것이 좋습니다. SQL 쿼리는 시간이 지남에 따라 변경될 가능성이 높습니다. 변하는 부분만 따로 분리하여 관리하면 그 만큼 관리성이 증대됩니다. 이 개념을 멋지게 적용한 기술이 MyBatis 라고 할 수 있습니다.

MyBatis는 학습곡선도 높지 않아서 국내외에서 널리 사용되고 있습니다. 프로젝트를 개발단계에만 초점을 맞추지 않고 운영 및 업그레이드 단계에도 초점을 맞추어 보면 Spring JDBC 기술보다 마이바티스를 선호하게 됩니다.

또한 쿼리 작성자와 이를 이용해서 DAO 로직을 만드는 개발자를 분리할 수 있다는 점에서 분업화에 유리하기 때문에 국내 SI프로젝트에서 많이 사용되고 있습니다. 마이바티스는 자바 표준 데이터베이스 처리기술인 JDBC를 바탕으로 동작한다는 점에서 Spring JDBC 기술과 비슷합니다. 마치 뿌리는 하나인데 가지가 두개인 모습입니다.

- 코드와 SQL 문자열을 분리한다. Spring JDBC는 SQL 문자열을 코드와 같이 작성한다. 마이바티스에서는 별도의 매퍼 파일에 SQL 구문을 작성하고 DAO에서 필요할 때 가져와서 사용한다.
- 개발자가 작성 할 코드가 줄어들어 생산성이 향상된다.
- SQL 문자열을 변경할 때 매퍼 파일만 변경하면 되므로 유지보수성이 향상된다. DAO 파일은 변경할 필요가 없다.

마이바티스 프로젝트

새 프로젝트 만들기

```
STS >> New >> Spring Starter Project

Project Name : spring-mybatis-1
Type : Maven
Packaging : Jar
Java : 8
Package : com.example.demo

Project Dependencies : Web, Lombok, MyBatis, H2
```

마이바티스 개발자들의 노력으로 스프링 부트 디펜던시 목록에 마이바티스가 추가되었습니다. 간단하게 디펜던시를 선택하는 것 만으로 설정작업이 자동으로 처리됩니다. 이전에는 마이바티스를 위한 환경설정을 개발자가 직접 처리해야 스프링 부트 프로젝트에서 사용할 수 있었습니다.

보다 자세한 내용은 다음 사이트를 참고하세요.
http://www.mybatis.org/spring-boot-starter/mybatis-spring-boot-autoconfigure/

환경설정

application.properties

```
# DataSource(JDBC Connection) Configuration
spring.datasource.url=jdbc:h2:mem:TESTDB;DB_CLOSE_DELAY=-1;
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

#spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# Use schema.sql, data.sql
spring.datasource.initialize=true

# SQL Holder File
mybatis.mapper-locations=/mybatis/mapper/*-mapper.xml
```

schema.sql

```
drop table if exists emp;

create table emp (
    empno bigint identity not null primary key,
    ename varchar(100),
    job varchar(100),
```

```
    sal bigint
);
```

data.sql

```
insert into emp(ename, job, sal) values('홍길동', '도적', 800);
insert into emp(ename, job, sal) values('일지매', '의적', 700);
insert into emp(ename, job, sal) values('임꺽정', '산적', 900);
```

도메인 클래스

Emp.java

```
package com.example.demo.domain;

import lombok.Data;

@Data
public class Emp {
    private int empno;
    private String ename;
    private String job;
    private double sal;
}
```

DAO 클래스

`@Mapper` 어노테이션을 인터페이스 클래스에 붙이면 마이바티스가 인터페이스 구현체를 만들고 그 객체를 빈 컨테이너에 등록합니다. 인터페이스를 작성한 것 만으로 개발자가 해야 할 작업은 끝났습니다. 나머지는 마이바티스와 스프링이 협력해서 개발자 대신 처리해 줄 것입니다. 마이바티스가 만드는 인터페이스 구현체의 로직은 정적이지 않고 동적입니다. 이 말에 뜻은 메소드가 사용하는 쿼리는 실제로 메소드가 호출되는 시점에 구해서 사용된다는 얘기입니다. 따라서, 객체를 빈 컨테이너에 등록하는 시점에 메소드가 사용하는 쿼리문이 없어도 예외는 발생하지 않습니다.

EmpDao.java

```
package com.example.demo.dao;

import java.util.List;

import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

import com.example.demo.domain.Emp;

// 인터페이스의 구현체 역할을 수행할 수 있는 객체(일종의 프록시)를
// 마이바티스가 빈 컨테이너에 등록합니다.
@Mapper
public interface EmpDao {
    // 리턴자료형 int: 작업결과로 영향받은 row의 개수를 의미합니다.
```

```
    public int insert(Emp emp);
    public int update(Emp emp);
    public int delete(int empno);

    // 마이바티스가 제공하는 애노테이션으로 사용할 SQL을 설정할 수 있다.
    @Select("select * from emp order by empno asc")
    public List<Emp> findAll();

    public int count();
    public Emp findOne(int empno);
}
```

findAll() 메소드가 사용해야 할 SQL 구문을 `@Select`으로 설정한다. `@Mapper` 애노테이션을 설정했기 때문에 빈 컨테이너에 EmpDao 자료형의 객체가 존재하고 이를 주입받아서 사용할 수 있다. findAll() 메소드는 제대로 작동하지만 다른 메소드들은 사용해야 할 SQL 구문을 설정하지 않았기 때문에 제대로 작동하지 않을 것이다.

일반적으로 SQL 쿼리는 예제처럼 짧지 않기 때문에 별도의 Mapper XML에 쿼리를 작성해 놓고 해당 인터페이스와 연계되어 사용되도록 설정한 후 사용하는 것을 선호합니다.

TEST

EmpDaoTest.java

```
package com.example.demo.dao;

import static org.junit.Assert.fail;

import java.util.List;

import javax.annotation.Resource;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.example.demo.domain.Emp;

@RunWith(SpringRunner.class)
@SpringBootTest
public class EmpDaoTest {
//    @Autowired
    @Resource(name="empDao")
    private EmpDao dao;

    @Test
    public void testFindAll() {
        List<Emp> emps = dao.findAll();
        System.out.println(emps.size());

        for (Emp emp : emps) {
            System.out.println(emp);
        }
    }
```

```
    @Test
    public void testCount() {
        int count = dao.count();
        System.out.println(count);
    }
}
```

testFindAll() 메소드는 성공할 것이다. 하지만, testCount() 메소드는 실패할 것이다. 왜냐하면 해당 메소드 처리 시 사용해야 할 SQL 구문을 설정하지 않았기 때문이다.

EmpDao 인터페이스의 메소드들이 사용해야 할 SQL 구문을 애노테이션으로 설정해도 되지만 별도의 XML파일에 설정할 수도 있다. 이어서 매퍼 파일을 작성해 보자.

emp-mapper.xml

`src\main\resources\mybatis\mapper\emp-mapper.xml` 위치에 만든다.

```
<?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.example.demo.dao.EmpDao">

<!--
    EmpDao 인터페이스에 정의된 추상메소드 정보를 바탕으로 매퍼설정을 진행한다.
    public int insert(Emp emp);
    public int update(Emp emp);
    public int delete(int empno);

    인터페이스의 풀패스 정보를 namespace 속성의 값으로 설정한다.
    메소드명을 id 속성의 값으로 설정한다.
    메소드가 받는 파라미터의 자료형을 parameterType 속성의 값으로 설정한다.
 -->

    <!--
        insert/update/delete 수행결과인 영향받은 로우의 개수 값은 자동으로 리턴된다.
     -->
    <insert id="insert" parameterType="com.example.demo.domain.Emp">
        insert into emp(ename, job, sal) values(#{ename}, #{job}, #{sal})
        <!--
            insert 작업을 수행하는 본 쿼리가 수행된 후에
            디비가 제너레이트 한 키 값이 무엇인지 알고 싶다.
            이 때, selectKey 태그로 제너레이트 된 키 값을 구하고
            그 값을 파라미터로 받은 객체의 멤버변수에 저장한다.
            파라미터로 받은 객체는 호출 측에서 넘겨준 객체이기 때문에
            호출 측에서 키 값을 꺼내어 사용할 수 있게 된다.
         -->
        <selectKey resultType="int" order="AFTER" keyProperty="empno">
            select LAST_INSERT_ID() as empno
        </selectKey>
    </insert>

    <update id="update" parameterType="com.example.demo.domain.Emp">
        update emp set ename=#{ename}, job=#{job}, sal=#{sal} where empno=#{empno}
    </update>

    <delete id="delete" parameterType="int">
        delete from emp where empno=#{empno}
    </delete>

<!--
```

```
    public int count();
    public Emp findOne(int empno);
    메소드의 리턴타입을 resultType 속성에 설정한다.
 -->

    <!-- 조회 쿼리는 결과를 어떻게 처리해서 리턴할 지 설정해야 한다. -->
    <select id="count" resultType="int">
        select count(*) from emp
    </select>

    <!--
        하나의 값이 아니라 복수의 값을 묶어서 리턴하기 위해서
        도메인 클래스를 설정한다. 칼럼명과 멤버변수명이 같은 경우
        resultType 속성에 Emp 도메인 클래스를 설정하면 된다.
     -->
    <select id="findOne" parameterType="int" resultType="com.example.demo.domain.Emp">
        select * from emp where empno=#{empno}
    </select>

</mapper>
```

`<mapper namespace="com.example.demo.dao.EmpDao">`

매퍼 XML을 마이바티스가 인식하도록 알려줄 필요가 있습니다. 네임 스페이스에 인터페이스를 풀패스로 지정하면 마이바티스는 이 파일 안에 담긴 쿼리들을 어떤 인터페이스의 추상메소드들이 사용해야 하는지 알게 됩니다.

`<select id="count" resultType="int">`

id로 추상 메소드명을 그대로 사용해서 설정하면 해당 메소드가 사용해야 하는 쿼리가 무엇인지 파악하게 됩니다. 더불어서 select 태그와 resultType의 자료형으로 마이바티스는 쿼리의 종류를 파악하고 쿼리 결과를 어떻게 처리해야 하는지 판단할 수 있습니다.

`<select id="findOne" parameterType="int" resultType="Emp">`

resultType의 자료형으로 모델클래스를 사용할 수 있습니다. 클래스의 풀패스를 써야 합니다. 풀패스를 클래스명만 짧게 써도 인식되게 만들려면 마이바티스에게 미리 알려주어야 합니다. 패키지명까지 써야 하는 모델클래스 이름을 클래스명만으로 간단하게 줄여 쓰기 위한 앨리어스 설정을 스프링 부트 설정파일 application.properties에 추가합니다.

```
mybatis.type-aliases-package=com.example.demo.domain
```

type-aliases-package 설정으로 해당 패키지에 있는 모든 클래스들은 매퍼 XML에서 클래스명만으로 설정해도 마이바티스는 풀패스가 무엇인지 파악할 수 있게 됩니다. 어노테이션 방식으로 SQL 쿼리를 알려주고 사용할 때는 mybatis.type-aliases-package 설정은 필요하지 않습니다.

마이바티스 표현식

매퍼 XML에서 파라미터를 쿼리문에 매핑할 때 마이바티스 표현식을 사용한다. 마이바티스 표현식에는 2가지 종류가 있으면 쿼리문 캐싱에서 약간의 차이점이 있다.

1. ${} : EAGER 매핑

```
select * from emp where empno=${empno}
// select * from emp where empno=1 형태로 쿼리문이 캐싱된다.
```

2. #{} : LAZY 매핑

```
select * from emp where empno=#{empno}
// select * from emp where empno=? 형태로 쿼리문이 캐싱된다.
```

마이바티스 표현식은 다음처럼 값을 매핑할 수 있다.

1. #{단일 값인 파라미터의 이름}
2. #{객체형인 파라미터의 멤버변수명}
3. #{@Param 애노테이션으로 설정한 이름}

앞서서 작성한 매퍼파일을 마이바티스가 인식할 수 있도록 다음 설정을 추가한다.

application.properties

```
# SQL Holder File
mybatis.mapper-locations=/mybatis/mapper/*-mapper.xml
```

- mapper-locations 설정으로 매퍼 XML 파일들의 위치를 마이바티스에게 알려준다.
- `/mybatis/mapper/*-mapper.xml` 패스 앞에 존재하는 `spring-mybatis/src/main/resources` 패스는 생략한다.
- `*` 기호를 사용하여 다수의 파일을 인식하도록 할 수 있다.
- `**` 기호를 사용하여 다수의 폴더를 인식하도록 할 수 있다.

EmpDao 인터페이스 안에 정의된 모든 메소드가 사용해야 할 SQL 구문을 설정했으니 잘 작동하는지 테스트를 수행할 차례다.

TEST

EmpDaoTest.java

```
package com.example.demo.dao;

import static org.junit.Assert.fail;

import java.util.List;

import javax.annotation.Resource;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.example.demo.domain.Emp;
```

```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class EmpDaoTest {
//    @Autowired
    @Resource(name="empDao")
    private EmpDao dao;

    @Test
    public void testInsert() {
        fail("Not yet implemented");
    }

    @Test
    public void testUpdate() {
        Emp emp = new Emp();
        emp.setEname("Tom");
        emp.setJob("Programmer");
        emp.setSal(700);

        System.out.println("전: " + emp);

        int affected = dao.insert(emp);
        System.out.println("affected = " + affected);

        // 수정을 위해서는 키 값인 empno 값이 필요합니다.
        // 별다른 작업을 하지 않는다면 여전히 0 값이 상태이고
        // 이러면 수정할 수 없습니다.

        System.out.println("후: " + emp);

        emp.setJob("Designer");
        emp.setSal(900);

        affected = dao.update(emp);
        System.out.println("affected = " + affected);

        Emp e = dao.findOne(emp.getEmpno());
        System.out.println(e);

        affected = dao.delete(emp.getEmpno());
        System.out.println("affected = " + affected);
    }

    @Test
    public void testDelete() {
        fail("Not yet implemented");
    }

    @Test
    public void testFindAll() {
        List<Emp> emps = dao.findAll();
        System.out.println(emps.size());

        for (Emp emp : emps) {
            System.out.println(emp);
        }
    }

    @Test
    public void testCount() {
        int count = dao.count();
        System.out.println(count);
    }
```

```
    @Test
    public void testFindOne() {
        int empno = 1;
        Emp emp = dao.findOne(empno);
        System.out.println(emp);
    }

}
```

`@Mapper` 애노테이션을 사용하지 않고 개발자가 직접 인터페이스 구현 클래스를 작성한 다음 마이바티스가 제공하는 SqlSessionTemplate을 사용해서 처리할 수도 있다. 과거에 만든 프로젝트에서 많이 사용하던 방식이다.

이를 확인하기 위해서 다음 코드를 작성하자.

EmpDaoImpl.java

```
package com.example.demo.dao;

import java.util.List;

import org.apache.ibatis.session.SqlSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.example.demo.domain.Emp;

@Repository
public class EmpDaoImpl implements EmpDao {
    @Autowired
    private SqlSession sqlSession;

    @Override
    public int insert(Emp emp) {
        return 0;
    }

    @Override
    public int update(Emp emp) {
        return 0;
    }

    @Override
    public int delete(int empno) {
        return 0;
    }

    @Override
    public List<Emp> findAll() {
        // "com.example.demo.dao.EmpDao.findAll" 문자열로
        // SQL 문자열을 가지고 있는 매퍼파일을 찾는다.
        // 리턴자료형을 보고 판단하여 사용해야 할 메소드를 선택한다.
        return sqlSession.selectList("com.example.demo.dao.EmpDao.findAll");
    }

    @Override
    public int count() {

        return 0;
    }
```

```
    @Override
    public Emp findOne(int empno) {
        return null;
    }

}
```

앞서서 만든 findAll() 메소드가 잘 작동하는지 테스트 해 보자.

EmpDaoImplTest.java

```
package com.example.demo.dao;

import static org.junit.Assert.*;

import java.util.List;

import javax.annotation.Resource;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.example.demo.domain.Emp;

@RunWith(SpringRunner.class)
@SpringBootTest
public class EmpDaoImplTest {
//    @Autowired
    @Resource(name="empDaoImpl")
    private EmpDao dao;

    @Test
    public void testFindAll() {
        List<Emp> emps = dao.findAll();
        System.out.println(emps.size());

        for (Emp emp : emps) {
            System.out.println(emp);
        }
    }

}
```

정리

마이바티스는 SQL 쿼리를 분리할 수 있는 유용한 기술입니다. 클래스 안에 쿼리를 기술하는 경우, 이를 수정하려면 코드를 이해하는 개발자가 필요하겠지만 별도의 파일로 분리해 놓았다면 굳이 개발자가 아니더라도 수정할 수 있게 됩니다.

마이바티스는 DAO 인터페이스의 추상메소드의 파라미터, 리턴자료형, SQL 쿼리의 종류를 보고서 처리해야 하는 로직을 판단할 수 있으므로 개발자 대신 인터페이스 구현체를 만들어 주는 서비스를 제공합니다.

따라서 마이바티스를 사용하면 개발자가 해야 하는 업무는 DAO 인터페이스 생성, 추상 메소드 작성, 추상 메소드가 사용하는 쿼리 지정 입니다. 개발자가 구현 클래스를 직접 만들지 않아도 되는 점에서 Spring JDBC 보다 편리합니다.