3. Spring JDBC

- Spring의 DAO 프레임워크에서는 JDBC의 Connection 객체를 DataSource를 통해서 취득하며 Connection 객체를 프로그래머가 직접 다루지 않는다.
- 스프링은 데이터 접근 프로세스에 있어서 고정된 부분과 가변적인 부분을 명확히 분류하는데 고정적인 부분은 템플릿(template)이며, 가변적인 부분은 콜백(callback)이 된다.
- 템플릿은 프로세스의 고정적인 부분(트랜잭션 관리, 자원관리, 예외처리)을 관리하며 콜백(질의문 생성, 파라미터 바인딩, 결과집합 매핑)은 구체적인 구현을 넣어야 하는 장소이다. 프로그래머는 콜백부분만 구현하면 된다.
- Spring 프레임워크는 JDBC를 추상화한 API를 새롭게 제공하고 있는데 이것이 Spring JDBC이다. Spring JDBC는 기존의 JDBC를 추상화하고 있는 만큼 지금까지 개발자들이 직접 구현해 왔던 Connection 생성, PreparedStatement 생성, SQLException 처리와 같은 반복적인 작업들을 프레임워크가 담당하게 된다.
- Spring의 DAO 프레임워크가 던지는 모든 예외는 DataAccessException 이다. SQLException이나 HibernateException등과 같은 특정 기술에 의존적인 예외를 던지지 않는다. 이것을 `예외전환 서비스` 라고 부른다.
- 데이터 접근 인터페이스가 구현에 의존적인 예외가 아닌 스프링의 일반적인 예외를 던짐으로써 특정한 퍼시스턴스 구현에 결합되는 일이 없다.
- DataAccessException은 반드시 직접 처리할 필요는 없다. RuntimeException이기 때문에 Uncheck Exception에 속한다. Checked Exception이 과도한 catch나 throws 절을 야기시켜 코드가 난잡하게 되는 현상을 막기 위해서 Checked Exception을 Uncheck Exception으로 변경한다.
- Unchecked Exception이 발생하는 경우는 대부분 복구가 불가능한 것이므로 직접 처리할 필요는 없다.

비지니스 로직을 수행하는 중 발생하는 비즈니스 로직 오류는 Checked Exception으로 처리하고 그렇지 않으면 Unchecked Exception으로 처리하면 된다.

- SQLErrorCodeSQLExceptionTranslator 상속하여 커스텀 예외전환 클래스를 사용할 수 있다. JdbcTemplate.setExceptionTranslator() 메소드를 통해 설정하면 된다.

DataSource 설정

자바소스

```
@Configuration
@ComponentScan("com.example.demo")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/testdb");
        dataSource.setUsername("user_id");
        dataSource.setPassword("user_password");

        return dataSource;
    }
}
```

설정파일

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="url" value="jdbc:oracle:thin:@192.168.0.27:1521:topcredu"/>
  <property name="driverClassName" value="oracle.jdbc.OracleDriver"/>
  <property name="username" value="scott"/>
  <property name="password" value="tiger"/>
</bean>
```

DB와 대화하는 방법

- JdbcTemplate : 전형적인 Spring JDBC 접근법으로 SQL을 실행하며 자주 사용된다.
    - 모든 형태의 SQL 구문을 실행해서 원하는 결과 타입을 반환한다.
    - 스프링의 모든 데이터 접근 프레임워크는 템플릿 클래스를 포함하는데 이 경우 템플릿 클래스는 JdbcTemplate 클래스이다.
    - JdbcTemplate 클래스가 작업하기 위해 필요한 것은 DataSource 이며 스프링의 모든 DAO 템플릿 클래스는 스레드에 안전하기 때문에 애플리케이션 내의 각각의 DataSource에 대해서 하나의 JdbcTemplate 인스턴스만을 필요로 한다.

- NamedParameterJdbcTemplate : 전형적인 JDBC의 ?(순서 기반 위치보유자) 대신에 :name(이름 기반 위치보유자) 바인딩 파라미터를 제공하기 위한 것이다.
- SimpleJdbcTemplate : JdbcTemplate + NamedParameterJdbcTemplate 개념으로 도입되었으나 스프링 3.1부터 deprecated 되었다. 대신 JdbcTemplate과 NamedParameterJdbcTemplate이 SimpleJdbcTemplate의 모든 기능을 제공한다.
- SimpleJdbcInsert : 최소한의 구성으로 Insert가 가능하도록 지원하며 DataSource를 가진 클래스를 인스턴스한 직후 withTableName 메소드를 불러 테이블 이름을 지정해야 한다. 디비에서 제너레이트하는 키 값을 리턴하는 메소드를 지원한다.
- SimpleJdbcCall : 최소한의 설정으로 데이터베이스 내에 저장된 저장 프로시저, 함수 등을 호출하도록 지원한다.
- SqlUpdate : 재사용 가능한 SQL DML(insert, update, delete) 구문을 생성한다. 다수의 쿼리를 수행해야 하는 경우 유리하다.
- StoredProcedure : RDB의 저장 프로시저에 대한 추상 수퍼 클래스, 다양한 execute 메소드를 제공한다. SimpleJdbcCall 보다 설정이 복잡하다.

영속화(Persistence) 프로젝트의 개발순서

데이터를 파일 또는 디비에 저장하여 나중에 다시 데이터를 복원할 수 있도록 조치하는 작업을 영속화라고 부른다. 간단히 디비와 연동하는 프로그램을 만드는 것이라 볼 수 있다. 이러한 프로젝트를 개발하는 일반적인 개발순서를 살펴보고 그 순서를 따라서 실습을 해 보자.

1. 디비 스키마 확인
디비접속 정보, 테이블 및 칼럼 정보를 확인한다. 이클립스의 Data Source Explorer 뷰 또는 토드 같은 프로그램을 사용하여 접속가능여부를 테스트한다. 실무에서는 토드를 많이 사용한다. 또는 디비마다 최적화된 GUI 프로그램을 사용하기도 한다.

2. 디비 연동 기술 선택
스프링프레임워크 기반 프로젝트에서는 다음 기술 중 하나를 주로 사용한다.

- Spring JDBC
- MyBatis
- JPA

3. 프로젝트 생성, 디펜던시 설정
연동하는 디비에 따라서 디비 연결 드라이버, 임베디드 디비 등의 디펜던시를 설정한다.
스프링이 제공하는 기술말고 써드파티의 기술을 사용하고자 하는 경우, 디비와 연결을 담당할 DataSource 구현 기술의 디펜던시를 추가로 설정한다.
메이븐 `pom.xml` : 디비 연결 드라이버 디펜던시 설정

4. 빈 컨테이너 설정
스프링 `root-context.xml` : DataSource, JdbcTemplate

5. Domain 클래스

테이블 한 행의 데이터를 자바 객체로 취급하는 클래스인 도메인 클래스를 만든다.
`Emp.java`

6. DAO
`EmpDao.java` : 디비 연동로직은 관습적으로 인터페이스를 둔다.
`EmpDaoImpl.java` : 인터페이스를 구현하는 디비 연동로직을 제공하는 클래스

7. TEST
자바 프로그램이 다른 외부 프로그램과 연동할 때 테스트가 필요하다. `JUnit` 테스트 프레임워크를 사용하여 EmpDaoImpl 클래스의 로직을 테스트한다.

8. Controller
컨트롤러는 브라우저의 요청을 접수 받고 DAO에게 부탁하여 데이터를 구한 후 적절한 포맷으로 데이터를 가공하여 브라우저에게 응답하는 역할을 수행한다.
`EmpController.java` : 브라우저와 대화하는 URL Handler 클래스

9. JSP
서버사이드에서 동적으로 HTML을 만들기 위한 기술 중에 하나를 선택한다. 국내는 대부분의 프로젝트에서 JSP 기술을 주로 사용하고 있다. 스프링 부트는 기본 뷰 처리 기술로써 Thymeleaf를 선택했으므로 JSP를 사용하기 위해서는 별도의 설정이 필요하게 되었다.
`emp-list.jsp` : 동적으로 HTML을 생성하는 화면처리 기술

10. TEST
자바 프로그램이 다른 외부 프로그램과 연동할 때 테스트가 필요하다. `cURL`, `Postman` 같은 도구를 사용하면 편리하게 테스트를 수행할 수 있다. `JUnit` 테스트 프레임워크를 사용하여 컨트롤러 로직의 테스트를 자동화하면 더욱 좋다.

앞에서 살펴본 개발순서를 프로젝트의 규모에 따라서 여러번 반복한다. 이제 실제로 개발순서에 따라서 예제 프로젝트를 만들어 보자.

첫 번째 Spring JDBC Project

앞에서 살펴 본 일반적인 개발순서에 따라 Spring JDBC 기술을 사용하여 디비에서 데이터를 구하는 프로젝트를 만들어 보자.

1. 디비 스키마 확인

Oracle11g 데이터베이스, scott 테스트 계정을 사용한다.

2. 디비 연동 기술 선택

Spring JDBC 기술을 사용한다.

3. 프로젝트 생성, 디펜던시 설정

스프링 레가시 방식으로 프로젝트를 구성한다.

```
STS >> New >> Spring Legacy Project

Project Name : spring-jdbc-1
Templates : Spring MVC Project
Package : com.example.demo
```

스프링 레가시 방식으로 프로젝트를 만들면 3.1 버전을 사용하도록 설정되어 있다. 이를 스프링 4로 업그레이드해서 사용하기로 한다. pom.xml을 살펴보고 있는 부분은 수정하고 없는 부분은 추가한다.

pom.xml

```
<properties>
  <java-version>1.8</java-version>
  <org.springframework-version>4.3.20.RELEASE</org.springframework-version>
</properties>

<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>com.oracle.jdbc</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.1.0.6.0</version>
    <scope>runtime</scope>
  </dependency>

  <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.16.22</version>
      <scope>compile</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${org.springframework-version}</version>
  </dependency>
```

```
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${org.springframework-version}</version>
  </dependency>

  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>${org.aspectj-version}</version>
  </dependency>
</dependencies>

<!--
  메이븐 JAR 추적 순서 :
  1. 로컬 리파지토리 > 2. 센트럴 리파지토리 > 3. 추가지정 리파지토리
 -->
<repositories>
  <repository>
    <id>ojdbc6</id>
    <url>http://repo.boundlessgeo.com/main/</url>
  </repository>
</repositories>
```

springframework 디펜던시의 버전을 올린다면 그에 따라서 servlet, junit 디펜던시의 버전도 동반상승해야 한다. 자세한 버전 정보는 스프링 부트로 만든 프로젝트의 세부 디펜던시 버전정보를 참고하자.

오라클 디비 연결 드라이버 디펜던시는 불행하게도 메이븐 센트럴 리파지토리에 존재하지 않는다. 이는 오라클의 회사정책에 따른 결과다. 따라서, 오라클 디비 연결 드라이버를 다운 받을 수 있는 별도의 리파지토리를 메이븐에게 알려주어야 한다.

4. 빈 컨테이너 설정

디비연결 정보는 별도의 프로퍼티 파일을 두어 관리한다. XML 설정에서 context:property-placeholder 태그를 사용하여 프로퍼티 파일을 스프링에게 알려준다.

src\main\resources\jdbc.properties

```
jdbc.url = jdbc:oracle:thin:@192.168.0.27:1521:topcredu
jdbc.driverClassName = oracle.jdbc.OracleDriver
jdbc.username = scott
jdbc.password = tiger
```

src\main\webapp\WEB-INF\spring\root-context.xml

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

```
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">

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

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

      <context:component-scan base-package="com.example.demo"/>

      <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
          <constructor-arg ref="dataSource"/>
      </bean>

  </beans>
```

DAO 구현 객체에 JdbcTemplate 객체를 주입하기 위해서 빈으로 미리 등록해 놓는다.

5. Domain 클래스

롬복을 사용하여 장황하고 뻔한 코드를 컴파일 타임 시 자동으로 제너레이트 되어 추가되는 방식을 이용하면 도메인 클래스를 깔끔하게 관리할 수 있다.

Emp.java

```
package com.example.demo.model;

import lombok.Data;

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

도메인 클래스는 데이터베이스의 테이블과 1:1 대응되는 관계입니다. 테이블 내 한 줄의 데이터들을 객체에서 취급하기 위한 일종의 Value Object입니다. 때때로 취급하는 데이터의 개수가 다른 경우 별도의 DTO 클래스를 추가로 사용하기도 합니다.

자바빈 규약에 따라 클래스의 필드변수는 private 접근제어자를 두고 getter, setter 메소드를 제공하는 형태로

작성합니다. 이러한 클래스를 계속 만들다 보면 단순하고 반복적인 작업을 계속하고 있다고 생각하게 됩니다. 이 부분에서 Lombok이 제공하는 기술을 사용하면 개발자는 필드변수만 선언하고 생성자, getter, setter 메소드를 만드는 작업은 롬복이 대신 제너레이트하게 할 수 있습니다. 자세한 사용법은 부록을 참고하세요.

6. DAO

데이터베이스와 대화하는 로직을 취급하는 클래스를 작성합니다. 우선 인터페이스를 만들고 간단하게 CRUD(Create Read Update Delete) 메소드를 설계합니다.

EmpDao.java

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

import java.util.List;

import com.example.demo.model.Emp;

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);

    public List<Emp> findAll();
    public int count();
    public Emp findOne(int empno);
}
```

경험으로 데이터베이스 처리 로직은 자주 변경된다는 것을 알고 있기에 일반적으로 DAO 구현 클래스를 만들기 전에 인터페이스를 작성합니다. 이는 행안부가 지정한 전자정부표준 프레임워크의 권고사항이기도 합니다.

EmpDaoImpl.java

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

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Emp;

@Repository
public class EmpDaoImpl implements EmpDao {
    @Autowired
    public JdbcTemplate jdbcTemplate;
```

```
//    Row(테이블 행) 정보를 <Emp> 객체에 Mapper(매핑) 하기 위한 로직
//    어느 칼럼의 데이터를 어느 멤버변수에 넣어야 하는지 알려주는 로직
    private RowMapper<Emp> rowMapper = new RowMapper<Emp>() {
        @Override
        public Emp mapRow(ResultSet rs, int rowNum) throws SQLException {
            Emp e = new Emp();
            e.setEmpno(rs.getInt("empno"));
            e.setEname(rs.getString("ename"));
            e.setJob(rs.getString("job"));
            e.setSal(rs.getDouble("sal"));

            return e;
        }
    };

//    빈 컨테이너에 JdbcTemplate이 없을 때 대신 DataSource를 받아서 처리하는 방식
//    @Autowired
//    public void setDataSource(DataSource dataSource) {
//        this.jdbcTemplate = new JdbcTemplate(dataSource);
//    }

//    메소드마다 반복적으로 사용되는 부분을 밖으로 빼내는 방법을 고려해 보자.
    @Override
    public int insert(Emp emp) {
        String sql = "insert into EMP9(empno, ename, job, sal) values(?, ?, ?, ?)";
        int affected = jdbcTemplate.update(sql,
                emp.getEmpno(), emp.getEname(), emp.getJob(), emp.getSal());

        return affected;
    }

    @Override
    public int update(Emp emp) {
        String sql = "update EMP9 set ename=?, job=?, sal=? where empno=?";
        return jdbcTemplate.update(sql,
                emp.getEname(), emp.getJob(), emp.getSal(), emp.getEmpno());
    }

    @Override
    public int delete(int empno) {
        String sql = "delete EMP9 where empno=?";
        return jdbcTemplate.update(sql, empno);
    }

    @Override
    public List<Emp> findAll() {
        String sql = "select empno, ename, job, sal from EMP9 order by empno asc";
        return jdbcTemplate.query(sql, rowMapper);
    }

    @Override
    public int count() {
        String sql = "select count(*) from EMP9";
        return jdbcTemplate.queryForObject(sql, Integer.class);
    }

    @Override
    public Emp findOne(int empno) {
        String sql = "select empno, ename, job, sal from EMP9 where empno=?";
        return jdbcTemplate.queryForObject(sql, rowMapper, empno);
    }

}
```

각 메소드의 파라미터 정보는 이클립스의 도움말을 참고하자.

새 로우의 추가, 수정, 삭제 시 jdbcTemplate.update() 메소드를 사용한다.

프로젝트 자바 버전이 1.8이상이면 테이블 한 행의 데이터를 모델 객체에 옮겨 담는 바인딩로직 RowMapper의 로직을 람다표현식을 사용하여 짧게 줄여서 작성할 수 있다.

데이터의 조회 시 사용해야 하는 메소드는 원하는 데이터의 구조의 따라서 메소드를 선택하면 된다. 달리 말하자면 메소드의 리턴자료형을 보고 사용해야 하는 메소드를 고르면 된다는 뜻이다.

원시타입(래퍼타입)을 리턴하는 경우

- 리턴 int : jdbcTemplate.queryForObject(sql, Integer.class)
- 리턴 long : jdbcTemplate.queryForObject(sql, Long.class)
- 리턴 double : jdbcTemplate.queryForObject(sql, Double.class)

참조타입을 리턴하는 경우

- 리턴 Emp : jdbcTemplate.queryForObject(sql, rowMapper, empno)
- 리턴 List : jdbcTemplate.query(sql, rowMapper)

7. TEST

디비와의 연동로직은 다양한 이유로 예외가 발생할 수 있다. 필수적으로 테스트를 수행해야만 DAO 로직이 제대로 작동함을 확신할 수 있게 된다. 자바 클래스 사이에 연동로직이 아닌 자바 클래스와 타 프로그램과의 연동로직인 경우 반드시 테스트 코드가 필요함을 상기하자.

EmpDaoImplTest.java

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

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.fail;

import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.example.demo.model.Emp;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations= {
  "file:src/main/webapp/WEB-INF/spring/root-context.xml"})
public class EmpDaoImplTest {
    @Autowired
    private EmpDao dao;
```

```
    @Test
    public void testDi() {
        System.out.println(dao);
        System.out.println(((EmpDaoImpl)dao).jdbcTemplate);
    }

    @Test
    public void testInsert() {
        Emp emp = new Emp();
        emp.setEmpno(3201);
        emp.setEname("홍길동");
        emp.setJob("도둑");
        emp.setSal(999);

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

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

    @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);
        }
    }

}
```

미 구현 테스트 메소드의 로직은 실습 시 직접 작성해 보자. 테스트 방법이 친숙해 졌다면 println() 메소드를 사용하여 개발자가 직접 눈으로 테스트 결과를 확인하는 대신 JUnit이 제공하는 단정메소드와 매처 라이브러리가 제공하는 매처메소드를 사용하여 테스트의 자동화를 꾀해 보자.

8. Controller

사용자(주로 브라우저)의 요청을 접수 받아서 적절한 데이터를 응답해 주는 컨트롤러를 만든다. 컨트롤러에 대한 자세한 설명은 Spring MVC 파트에서 설명한다.

EmpController.java

```
package com.example.demo.web;

import java.util.List;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.example.demo.dao.EmpDao;
import com.example.demo.model.Emp;

@Controller
public class EmpController {
    @Autowired
    private EmpDao empDao;

//    http://localhost:8080/demo/emps
    @RequestMapping(value="/emps", method=RequestMethod.GET)
    public String getAll(Model model) {
        // DAO에게 데이터를 구해달라고 요청한다.
        List<Emp> emps = empDao.findAll();
        // Model 객체에 데이터를 추가하면
        // 스프링이 HttpServletRequest 객체로 옮긴다.
        // HttpServletRequest 객체에 존재하는 데이터는
        // JSP에서 접근하여 사용할 수 있다.
        model.addAttribute("emps", emps);
        // 다음으로 연동할 JSP 뷰 파일명을 리턴한다.
        return "emp-list";
    }

}
```

9. JSP

브라우저에게 데이터를 HTML로 포장해서 전달하고자 하는 경우 JSP를 사용한다.

src\main\webapp\WEB-INF\views\emp-list.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title></title>
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</head>
<body>
    <div class="container">
        <h3>직원정보</h3>
        <p>직원들의 상세한 정보입니다.</p>

        <table class="table table-striped table-hover">
            <thead>
                <tr>
```

```
                    <th>직원번호</th>
                    <th>성명</th>
                    <th>직업</th>
                    <th>월급</th>
                </tr>
            </thead>
            <tbody>
                <c:forEach var="e" items="${emps }">
                <tr>
                    <td>${e.empno }</td>
                    <td>${e.ename }</td>
                    <td>${e.job }</td>
                    <td>${e.sal }</td>
                </tr>
                </c:forEach>
            </tbody>
        </table>
    </div>
</body>
</html>
```

10. TEST

크롬 브라우저를 사용하여 `http://localhost:8080/demo/emps` 주소로 접근한다.

JdbcTemplate 메소드 살펴보기

execute

- 대개 SQL 문자열로 부터 PreparedStatement를 생성하고 파라미터를 바인딩 하는 작업을 많이 하므로 JdbcTemplate은 execute 메소드를 제공한다.
- 주로 DDL 처리용으로 아래와 같은 경우 백그라운드에서 JdbcTemplate이 PreparedStatement와 PreparedStatementSetter를 생성 시킨다.

```
public int insertPerson(Person person) {
  String sql = "insert into person (id, firstname, lastname) values (?,?,?)";

  Object[] params = new Object[] {
    person.getId(),
    person.getFirstName(),
    person.getLastName()
  };

  return jdbcTemplate.execute(sql, params);
}
```

update

- 로우의 변화를 주고자 할 때 사용하는 메소드이다. 입력, 수정, 삭제 쿼리가 이에 해당한다.
- execute에 비해 NULL을 setting할때 이점이 있다.(TYPE을 인자로 받아들임)
- 주로 DML 처리용(insert, update, delete)으로 파라미터는 가변인자나 객체배열 형태로 제공하면 된다.
- 실행시 백그라운드에서 JdbcTemplate이 PreparedStatement와 PreparedStatementSetter를 생성 시킨다.

```
public int insert(Emp emp) {
  String sql = "insert into EMP9(empno, ename, job, sal) values(?, ?, ?, ?)";
  int affected = jdbcTemplate.update(sql,
      emp.getEmpno(), emp.getEname(), emp.getJob(), emp.getSal());
  return affected;
}

public int update(Emp emp) {
  String sql = "update EMP9 set ename=?, job=?, sal=? where empno=?";
  return jdbcTemplate.update(sql,
      emp.getEname(), emp.getJob(), emp.getSal(), emp.getEmpno());
}

public int delete(int empno) {
  String sql = "delete EMP9 where empno=?";
  return jdbcTemplate.update(sql, empno);
}
```

추가적으로 자료형을 지정할 수 있다. 많이 사용되는 Oracle, MySQL 디비와의 연동 작업 시에는 자료형 지정이 필요하지 않다.

```
public int insertPerson(Person person) {
  String sql = "insert into person (id, firstname, lastname) values (?,?,?)";

  Object[] params = new Object[] {
    person.getId(),
    person.getFirstName(),
    person.getLastName()
  };
  int[] types = new int[] { TYPES.INTEGER, TYPES.VARCHAR, TYPES.VARCHAR };

  return jdbcTemplate.update(sql, params, types);
}
```

RowMapper

- RowMapper 인터페이스
  여러 건의 레코드(여러 person 객체)를 얻을 수 있는 메소드가 필요하다면 RowMapper를 구현해서 데이터를 추출할 수 있는데 RowMapper는 ResultSet의 SELECT된 레코드와 객체를 매핑 시키는 역할을 한다.

```
public class PersonRowMapper implements RowMapper {
  public Object mapRow(ResultSet rs, int index) throws SQLException {
    Person person = new Person();
    person.setId(new Integer(rs.getInt("id")));
```

```
    person.setFirstName(rs.getString("firstname"));
    person.setLastName(rs.getString("lastname"));

    return person;
  }
}
```

queryForObject

한 개의 레코드 처리용, 객체타입으로 결과를 리턴해 준다.

```
public String getLastNameById(Integer id) {
  String sql = "SELECT lastname FROM person WHERE id = ?";
  Object[] args = new Object[] { id };  // ?에 대입되는 매개변수

  String lastname = (String)jdbcTemplate.queryForObject(
    sql, args, String.class);
  return lastname;
}
```

```
// 사용 예
int rowCount = jdbcTemplate.queryForObject(
  "select count(*) from t_actor", Integer.class);
// 사용 예
int count = jdbcTemplate.queryForObject(
  "select count(*) from emp where ename = ?", Integer.class, "SMITH");
// 사용 예
String lastName = this.jdbcTemplate.queryForObject(
  "select ename from emp where empno = ?", new Object[]{7369L}, String.class);
// 사용 예
String sql = "SELECT dateCol FROM Table";
Date d  = (Date) jdbcTemplate.queryForObject(sql, Date.class);
```

query

여러 행의 정보를 구할 때 사용한다. 여러 칼럼의 정보를 도메인 클래스에 담기 위한 로직인 RowMapper가 필요하다.

```
private RowMapper<Emp> rowMapper = new RowMapper<Emp>() {
  @Override
  public Emp mapRow(ResultSet rs, int rowNum) throws SQLException {
    Emp e = new Emp();
    e.setEmpno(rs.getInt("empno"));
    e.setEname(rs.getString("ename"));
    e.setJob(rs.getString("job"));
    e.setSal(rs.getDouble("sal"));

    return e;
  }
};

public List<Emp> findAll() {
  String sql = "select empno, ename, job, sal from EMP order by empno asc";
```

```
  // RowMapper를 멤버변수나 별도의 클래스로 만들어서 사용하면 재 사용 시 편리하다.
  return jdbcTemplate.query(sql, rowMapper);
}

public List<Emp> findAll2() {
  String sql = "select * from EMP order by empno asc";
  // 칼럼명과 멤버변수명이 일치할 때 BeanPropertyRowMapper를 사용하면 편리하다.
  return jdbcTemplate.query(sql, new BeanPropertyRowMapper<Emp>(Emp.class));
}
```

queryForList

도메인 클래스를 따로 만들지 않고 사용할 때 유리하다. 도메인 클래스로 매핑해야 한다면 그냥 query() 메소드를 사용하자.

```
public List<Emp> findAll3(){
  String sql = "select * from EMP order by empno asc";
  List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);

  List<Emp> emps = new ArrayList<Emp>();
  for (Map<String, Object> row : rows) {
    Emp e = new Emp();
    e.setEmpno((int)row.get("empno"));
    e.setEname((String)row.get("ename"));
    e.setJob((String)row.get("job"));
    e.setSal((double)row.get("sal"));

    emps.add(e);
  }
  return emps;
}
```

두 번째 Spring JDBC Project

이번에는 스프링 부트 프로젝트에서 Spring JDBC를 사용해 보자. 작업순서를 상기하면서 진행하도록 한다.

1. 디비 스키마 확인

H2 디비를 프로젝트 임베디드 방식으로 사용한다. 프로젝트 기동 시 디비와 테이블을 생성해서 사용한다. 일반적으로 개발자의 로컬 컴퓨터에서 개발한 다음 단위 테스트를 거친 후 개발서버로 통합되고 통합 테스트가 끝난 다음 일정에 맞추어 운영 서버에 적용된다. 개발자의 로컬 컴퓨터에서의 빠른 개발을 원할 때 임베디드 방식으로 디비를 사용하는 것이 도움이 된다.

테스트 테이블 및 테스트 데이터를 입력해 놓기 위해서 application.properties 파일이 위치해 있는 src/main/resources 폴더에 다음 파일들을 생성합니다.

schema.sql

```
drop table if exists emp;

create table emp (
    empno int identity not null primary key,
    ename varchar(100),
    job varchar(100),
    sal double
);
```

조회 쿼리 테스트를 위해서 약간의 데이터를 입력해 놓는다.

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);
```

스프링 부트는 기동 시 위 파일들을 발견하면 schema.sql 파일의 DDL쿼리를 실행하고 다음으로 data.sql의 DML쿼리를 수행합니다. 개발 시 매우 편리한 기능입니다.

2. 디비 연동 기술 선택

Spring JDBC 기술을 사용한다.

3. 프로젝트 생성, 디펜던시 설정

스프링 부트는 프로젝트 생성 시 디펜던시를 미리 선택한다. 물론, 프로젝트 생성 후에도 pom.xml을 수정하는 것으로 추가 및 변경이 가능하다.

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

Project Name : spring-jdbc-2
Type : Maven
Packaging : Jar
Java : 8
Package : com.example.demo

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

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
```

```
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!--
      디비를 별도로 설치해서 사용하지 않고 프로젝트 안에서 추가된 기능으로
      사용하고 싶을 때, 임베디드 방식으로 h2 디비의 디펜던시를 추가해서
      사용할 수 있습니다.
     -->
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

4. 빈 컨테이너 설정

DataSource 설정을 위한 디비 연결정보를 등록한다. url 문자열 jdbc:h2:mem:TESTDB 중에서 mem 문자열은 메모리 디비로 H2 디비를 사용한다는 것을 나타낸다.
웹 브라우저를 이용한 H2 디비 접속 콘솔을 사용하기 위해서 spring.h2.console.path 설정을 추가한다.

src\main\resources\application.properties

```
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

# Using for schema.sql, data.sql
spring.datasource.initialize=true
```

spring.datasource.initialize 속성은 기본 값이 true이기 때문에 명시적으로 설정하지 않아도 schema.sql, data.sql 파일이 처리된다.

`spring.datasource.platform=mariadb`
이 설정은 필수는 아니지만 스프링 부트에서 SQL의 DDL과 DML을 자동생성 하기위한 schema-mariadb.sql,

data-mariadb.sql 파일들을 사용할 수 있도록 해주기 때문에 여러 데이터베이스를 사용하는 개발 환경인 경우 설정해 놓으면 편리하다. 기본적으로 부트가 프로젝트 기동 시 처리해 주는 schema.sql, data.sql 파일도 같이 사용할 수 있다.

pom.xml에 spring-boot-starter-jdbc 디펜던시 설정이 있는데 application.properties 파일에 datasource 설정이 없다면 에러가 발생한다. 스프링 부트는 자동환경설정을 진행하려 하는데 datasource 설정에서 꼭 필요한 디비연결 정보가 없기 때문이다. 디비를 당장 사용하지 않는 경우, 스프링부트 기동클래스에 Annotation을 추가함으로써 스프링부트 시작 시 실행되는 Auto Configuration 작업 중 DatatSource 설정부분을 제외시킬 수 있다.

```
@EnableAutoConfiguration(exclude={ DataSourceAutoConfiguration.class })
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

5. Domain 클래스

도메인 클래스의 설정은 간단히 롬복으로 해결한다.

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;
}
```

6. DAO

EmpDao.java

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

import java.util.List;

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

```
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);

    public List<Emp> findAll();
    public int count();
    public Emp findOne(int empno);
}
```

첫 번째 프로젝트에서 사용한 코드와 어떻게 다른지 비교해 보십시오.

EmpDaoImpl.java

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

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.annotation.PostConstruct;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;

import com.example.demo.domain.Emp;

@Repository
public class EmpDaoImpl implements EmpDao {
    @Autowired
    public JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert jdbcInsert;
    @Autowired
    private NamedParameterJdbcTemplate nTemplate;

//    Row(테이블 행) 정보를 <Emp> 객체에 Mapper(매핑) 하기 위한 로직
//    어는 칼럼의 데이터를 어는 멤버변수에 넣어야 하는지 알려주는 로직
    private RowMapper<Emp> rowMapper = new RowMapper<Emp>() {
        @Override
        public Emp mapRow(ResultSet rs, int rowNum) throws SQLException {
            Emp e = new Emp();
            e.setEmpno(rs.getInt("empno"));
            e.setEname(rs.getString("ename"));
            e.setJob(rs.getString("job"));
            e.setSal(rs.getDouble("sal"));

            return e;
        }
    };
```

```
    @PostConstruct
    public void init() {
        jdbcInsert = new SimpleJdbcInsert(jdbcTemplate)
                .withTableName("EMP").usingGeneratedKeyColumns("empno");
        // SimpleJdbcInsert를 사용하면 디비가 자동으로 증가시키는 키 값을 얻을 수 있다.
    }

//    @Autowired
//    public void setDataSource(DataSource dataSource) {
//        this.jdbcTemplate = new JdbcTemplate(dataSource);
//    }

    @Override
    public int insert(Emp emp) {

        if (emp.getEmpno() == 0) {
            // 도메인 클래스의 멤버변수 이름과 테이블의 칼럼명이 일치하는 경우
            // 멤버변수를 칼럼명으로 사용하여 자동으로 SQL insert 쿼리문을 작성합니다.
            // jdbcInsert 객체를 사용할 때 따로 SQL 쿼리문을 알려줄 필요가 없습니다.
            SqlParameterSource param = new BeanPropertySqlParameterSource(emp);
            // 자바에서 숫자를 취급하는 Number객체 안에 디비가 제너레이트한 키 값이
            // 존재합니다.
            Number number = jdbcInsert.executeAndReturnKey(param);
            // 디비가 제너레이트한 키 값을 insert() 메소드가 받은 파라미터 객체에
            // 추가하면 객체는 참조이기 때문에 이 메소드를 호출한 측에서 바로 객체에서
            // 키 값을 꺼내서 사용할 수 있습니다.
            emp.setEmpno(number.intValue());

            return 1;
        } else {
            // PK인 empno 값을 개발자가 직접 지정해서 저장하는 방식입니다.
            // 디비가 PK 값을 제너레이트 하는 경우라면 이 방식을 사용하지 않습니다.
            String sql = "insert into EMP(empno, ename, job, sal) values(?, ?, ?, ?)";
            return jdbcTemplate.update(sql,
                    emp.getEmpno(), emp.getEname(), emp.getJob(), emp.getSal());
        }

    }

    @Override
    public int update(Emp emp) {
//        String sql = "update EMP set ename=?, job=?, sal=? where empno=?";
//        return jdbcTemplate.update(sql,
//                emp.getEname(), emp.getJob(), emp.getSal(), emp.getEmpno());

        // 순서 기반 위치보유자 ? 대신,
        // 이름 기반 위치보유자 : 를 사용하면
        // 파라미터로 받은 값을 SQL 문자열과 결합할 때,
        // 순서가 헷갈려서 벌어지는 실수를 예방할 수 있습니다.
        String sql = "update EMP set ename=:ename, job=:job, sal=:sal "
                + "where empno=:key";

        Map<String, Object> map = new HashMap<>();
        map.put("key", emp.getEmpno());
        map.put("ename", emp.getEname());
        map.put("job", emp.getJob());
        map.put("sal", emp.getSal());

        return nTemplate.update(sql, map);
    }

    @Override
    public int delete(int empno) {
        String sql = "delete EMP where empno=?";
```

```
        return jdbcTemplate.update(sql, empno);
    }

    @Override
    public List<Emp> findAll() {
        String sql = "select empno, ename, job, sal from EMP order by empno asc";
        // RowMapper를 멤버변수나 별도의 클래스로 만들어서 사용하면 재 사용 시 편리하다.
        return jdbcTemplate.query(sql, rowMapper);
    }

    public List<Emp> findAll2() {
        String sql = "select * from EMP order by empno asc";
        // 칼럼명과 멤버변수명이 일치할 때 BeanPropertyRowMapper를 사용하면 편리하다.
        return jdbcTemplate.query(sql, new BeanPropertyRowMapper<Emp>(Emp.class));
    }

    public List<Emp> findAll3(){
        String sql = "select * from EMP order by empno asc";
        // DTO를 따로 만들지 않고 사용할 때 유리하다.
        List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);

        List<Emp> emps = new ArrayList<Emp>();
        for (Map<String, Object> row : rows) {
            Emp e = new Emp();
            e.setEmpno((int)row.get("empno"));
            e.setEname((String)row.get("ename"));
            e.setJob((String)row.get("job"));
            e.setSal((double)row.get("sal"));

            emps.add(e);
        }
        return emps;
    }

    @Override
    public int count() {
        String sql = "select count(*) from EMP";
        return jdbcTemplate.queryForObject(sql, Integer.class);
    }

    @Override
    public Emp findOne(int empno) {
        String sql = "select empno, ename, job, sal from EMP where empno=?";
        return jdbcTemplate.queryForObject(sql, rowMapper, empno);
    }

}
```

insert/update 메소드를 하나로 합친 save 메소드를 사용할 수도 있다. 다음을 참고하자.

```
// save 메소드는 insert/update를 모두 지원하는 upsert 메소드입니다.
public int save(Emp emp) {
  // 키 값이 없다면(0 값인 상태) 새 로우를 추가하는 insert 쿼리를 수행한다.
  if (emp.getEmpno() == 0) {
    Map<String, Object> paramMap = new HashMap<>();
    paramMap.put("ename", emp.getEname());
    paramMap.put("job", emp.getJob());
    paramMap.put("sal", emp.getSal());

    Number number = jdbcInsert.executeAndReturnKey(paramMap);
    emp.setEmpno(number.intValue());
```

```
      return 1;
    } else {
      // 키 값이 있다면 기존 로우를 수정하는 update 쿼리를 수행한다.
      String sql = "update EMP set ename=:ename, job=:job, sal=:sal where empno=:key";

      // 멤버변수와 칼럼명이 다르다면 BeanPropertySqlParameterSource를
      // 사용할 수 없다. 이 때, MapSqlParameterSource를 사용한다.
      SqlParameterSource paramSource = new MapSqlParameterSource()
          .addValue("ename", emp.getEname())
          .addValue("job", emp.getJob())
          .addValue("sal", emp.getSal())
          .addValue("key", emp.getEmpno());

      return nTemplate.update(sql, paramSource);
    }
  }
```

테스트 코드를 작성하여 테스트를 수행해 보자. 이해를 돕기위해서 로그에 출력하도록 작성한다. 저장 전, 저장 후/수정 전, 수정 후에 객체 상태를 살펴보자. 저장 전에는 키 값이 0인 상태이지만 저장 후/수정 전 상태에서는 키 값이 존재함을 확인하자.

```
@Test
public void testSave() {
  Emp emp = new Emp();
  emp.setEname("Tom");
  emp.setJob("Actor");
  emp.setSal(999);

  System.out.println("저장 전: " + emp);

  // Insert
  int affected = dao.save(emp);
  System.out.println("affected = " + affected);

  System.out.println("저장 후/수정 전: " + emp);

  emp.setJob("None");
  emp.setSal(0);

  // Update
  affected = dao.save(emp);
  System.out.println("affected = " + affected);

  // Select
  Emp e = dao.findOne(emp.getEmpno());
  System.out.println("수정 후: " + e);
}
```

7. TEST

언제나 테스트는 중요하다. 개발자는 코드로써 말하고 코드로써 증명하는 직업이다.

EmpDaoImplTest.java

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

// CoreMatchers 클래스에 is 메소드는 static 메소드입니다.
// 원래 사용방법은 CoreMatchers.is() 방식인데
// 클래스를 명시하지 않고 is() 형태로 사용하고 싶어서
// import 구문에 static 설정을 추가합니다.
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.fail;

import java.util.List;

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
    private EmpDao dao;

    @Test
    public void testInsert() {
        int oldCount = dao.count();

        Emp emp = new Emp();
        emp.setEname("Tom");
        emp.setJob("Actor");
        emp.setSal(999);

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

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

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

        int nowCount = dao.count();

        assertThat("한 행이 추가되었으므로 nowCount는 oldCount 보다 1이 커야 합니다.", nowCount, is(oldCc
    }

    @Test
    public void testUpdate() {
        Emp emp = new Emp();
        emp.setEname("Tom");
        emp.setJob("Actor");
        emp.setSal(999);

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

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

        emp.setJob("None");
        emp.setSal(0);

        affected = dao.update(emp);
```

```
        System.out.println("affected = " + affected);

        Emp e = dao.findOne(emp.getEmpno());

        System.out.println("후: " + e);
    }

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

    @Test
    public void testFindAll() {
        List<Emp> emps = dao.findAll();
        int count = dao.count();

        assertEquals("리스트의 사이즈는 카운트 값과 일치해야 한다.",  emps.size(), count);
        System.out.println("count = " + count);

        emps.forEach(System.out::println);
    }

    @Test
    public void testFindAll2() {
        List<Emp> emps = ((EmpDaoImpl)dao).findAll2();
        int count = dao.count();

        assertEquals("리스트의 사이즈는 카운트 값과 일치해야 한다.",  emps.size(), count);
        System.out.println("count = " + count);

        emps.forEach(System.out::println);
    }

    @Test
    public void testFindAll3() {
        List<Emp> emps = ((EmpDaoImpl)dao).findAll3();
        int count = dao.count();

        assertEquals("리스트의 사이즈는 카운트 값과 일치해야 한다.",  emps.size(), count);
        System.out.println("count = " + count);

        emps.forEach(System.out::println);
    }

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

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

}
```

구현되지 않은 테스트 메소드는 직접 테스트 코드를 작성해 보자.

8. Service

서비스는 비즈니스 로직을 배치하는 곳이다. 컨트롤러는 브라우저와 대화하는 역할을 수행한다. DAO는 디비와 대화하는 역할을 수행한다. 추가적으로 비즈니스 로직을 배치해야 할 때, 서비스 클래스를 만든다.

EmpService.java

EmpDao 인터페이스와 동일한 상태로 만들었지만 실무에서는 대부분의 경우 비즈니스 업무에 따라서 달라지게 된다.

```
package com.example.demo.service;

import java.util.List;

import com.example.demo.domain.Emp;

public interface EmpService {
    public int insert(Emp emp);
    public int update(Emp emp);
    public int delete(int empno);

    public List<Emp> findAll();
    public int count();
    public Emp findOne(int empno);
}
```

이어서 서비스 인터페이스 구현 클래스를 만든다.

EmpServiceImpl.java

```
package com.example.demo.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import com.example.demo.dao.EmpDao;
import com.example.demo.domain.Emp;

@Service
public class EmpServiceImpl implements EmpService {
    @Autowired
    private EmpDao empDao;

    @Override
    public int insert(Emp emp) {
        // 로직을 추가해야 할 때 서비스 클래스의 메소드 공간이 필요합니다.
        return empDao.insert(emp);
    }

    @Override
    public int update(Emp emp) {
```

```
        return empDao.update(emp);
    }

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

    @Override
    public List<Emp> findAll() {
        return empDao.findAll();
    }

    @Override
    public int count() {
        return empDao.count();
    }

    @Override
    public Emp findOne(int empno) {
        return empDao.findOne(empno);
    }

}
```

서비스 레이어를 두는 것은 현명한 선택입니다. 서비스 레이어의 존재 이유를 현재까지 작성한 코드만을 보고 생각해 내기 어렵지만 실제로 서비스를 구축하게 되면 자연스럽게 여러 비즈니스 로직이 필요하게 되고 그에 따라 서비스 레이어에 여러 로직이 추가됩니다.

전자정부 표준프레임워크에서도 퍼시스턴스 레이어와 서비스 레이어에는 항상 인터페이스를 두고 개발하기를 권고하고 있습니다. 이는 많은 개발자들이 수많은 프로젝트를 수행하면서 인터페이스를 두고 개발하는 것이 좋다는 결론에 동의한 결과라고 볼 수 있습니다.

예를 들어 신혼부부가 결혼하면 태어날 아기를 염두에 두고 방 2개 이상의 집을 구하듯이 컨트롤러와 리파지토리가 연동하면 추가될 비즈니스 로직을 염두에 두고 서비스를 만들어 두는 것입니다.

9. Controller

앞서서 만든 프로젝트를 참고해서 직접 작성해 보자.

EmpController.java

```
// 직접 작성해 보세요.
```

클라이언트와 JSON 포맷의 문자열로 대화하는 컨트롤러를 작성해 보자.

EmpRestController.java

```
package com.example.demo.controller;

import java.util.List;
```

```java
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.domain.Emp;
import com.example.demo.service.EmpService;

@RestController
public class EmpRestController {
    @Autowired
    private EmpService empService;

    /*
     * JavaScript ~~HTTP~~ Controller
     *
     * HTTP 요청방식
     * 1. GET ==> Select 쿼리
     * 2. POST ==> Insert 쿼리
     * 3. PUT(PATCH) ==> Update 쿼리
     * 로우 전체를 변경하면 PUT, 로우의 일부분을 변경하면 PATCH
     * 4. DELETE ==> Delete 쿼리
     * 5. OPTIONS : 어떤 요청방식을 제공하는 가 결정, 환경설정으로 처리
     */

    // 1. GET ==> Select 쿼리
    @RequestMapping(value="/emps", method=RequestMethod.GET)
    public Object getAll() {
        List<Emp> emps = empService.findAll();
        return emps;
    }

    // {id} : URI 문자열 부분은 가변적이어도 연동이 된다.
    // @PathVariable("id") : {id} 부분에 값을 변수에 할당한다.
    @RequestMapping(value="/emps/{id}", method=RequestMethod.GET)
    public Object getOne(@PathVariable("id") int empno) {
        Emp emp = empService.findOne(empno);
        return emp;
    }

    // 2. POST ==> Insert 쿼리
    // @RequestBody : 클라이언트가 HTML Form 방식(x-www-form-urlencoded)이
    // 아닌 JSON 포맷의 문자열로 데이터를 보낼 때 이를 스프링이 처리하도록
    // 요청한다. 클라이언트는 추가적으로 Header 부분에 다음 설정을 해야 한다.
    // Content-Type: application/json
    @RequestMapping(value="/emps", method=RequestMethod.POST)
    public Object postOne(@RequestBody Map<String, Object> map) {
        // 클라이언트가 보내는 파라미터들이 스프링에 의해서 map 객체에 추가된다.
        Emp emp = new Emp();
        emp.setEname((String)map.get("ename"));
        emp.setJob((String)map.get("job"));
        emp.setSal((Integer)map.get("sal"));

        int affected = empService.insert(emp);
        return emp;
    }

    // 3. PUT ==> Update 쿼리
```

```
    @RequestMapping(value="/emps/{id}", method=RequestMethod.PUT)
    public Object putOne(
            @PathVariable("id") int empno,
            @RequestBody Map<String, Object> map) {
        // PUT 방식은 해당 로우를 전체적으로 모두 수정하는 것이므로
        // 클라이언트가 수정하지 않는 기존 데이터도 보내주어야 한다.
        Emp emp = new Emp();
        emp.setEmpno(empno);
        emp.setEname((String)map.get("ename"));
        emp.setJob((String)map.get("job"));
        emp.setSal((Integer)map.get("sal"));

        int affected = empService.update(emp);
        return emp;
    }

    // 4. DELETE ==> Delete 쿼리
    @RequestMapping(value="/emps/{id}", method=RequestMethod.DELETE)
    public Object deleteOne(@PathVariable("id") int empno) {
        int affected = empService.delete(empno);

        // 삭제된 로우는 존재하지 않으므로 삭제작업이 성공했다는 의미만 전달한다.
        return new ResponseEntity<>(HttpStatus.OK);
    }

    // 컨트롤러의 메소드가 정상작동 하는지 Postman을 사용해서 테스트 해 보자.
}
```

10. JSP

src\main\webapp\WEB-INF\views\emp-list.jsp

```
<!-- 직접 작성해 보세요. -->
```

11. TEST

크롬 브라우저를 사용하여 `http://localhost:8080/demo/emps` 주소로 접근한다.

세 번째 Spring JDBC Project

이번에는 Spring JDBC 기술로 디비의 프로시저와 연동하는 방법을 살펴 볼 것이다.

1. 디비 스키마 확인

MySQL 디비 계열의 마리아 디비를 사용해 보자. 설치되어 있지 않다면 `부록 C. MariaDB 설치방법`을 참고하여 디비를 설치한 다음 다음을 진행하자. 프로젝트 기동 시 디비와 테이블을 생성해서 사용한다.

schema.sql

```
DROP TABLE IF EXISTS emp;

CREATE TABLE emp (
    empno INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    ename VARCHAR(100),
    job VARCHAR(100),
    sal DOUBLE
) ENGINE=InnoDB;

DROP TABLE IF EXISTS member;

-- 이미 존재하는 테이블의 정보를 이용하여 새 테이블 member를 만든다.
-- 필터: 이미 존재하는 테이블의 데이터는 복사하지 않는다.
-- 주의: 이미 존재하는 테이블의 키 설정은 복사되지 않는다.
CREATE TABLE member SELECT * FROM emp WHERE 1=0;
```

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);

--추가 연습용이다.
--INSERT INTO emp(ename, job, sal) VALUES('구운몽', '백수', 999);
--INSERT INTO emp(ename, job, sal) VALUES('온달', '장군', 600);
```

pro_fn_tri.sql

마리아 디비에 HeidiSQL을 사용하여 접속한 후 다음 프로시저를 생성하자.

```
--================================================
-- # 프로시저
-- 결과를 resultset으로 처리하기 위해서 함수 대신 프로시저를 사용해야 한다.
-- (https://dev.mysql.com/doc/refman/8.0/en/create-function-udf.html)
-- 프로시저에서 사용한 질의문에 기반하여 implicit objects가 리턴된다.
-- 실행방법:
-- call pro_find_all;
--================================================

DELIMITER //

DROP PROCEDURE IF EXISTS pro_find_all//

CREATE PROCEDURE pro_find_all()
BEGIN
    SELECT * FROM emp;
END
//

DELIMITER ;
```

```
--=============================================
-- # 프로시저
-- 실행방법:
-- call pro_find_one(1, @ename, @job, @sal);
-- SELECT @ename, @job, @sal;
--=============================================

DELIMITER //

DROP PROCEDURE IF EXISTS pro_find_one//

CREATE PROCEDURE pro_find_one (
  IN in_empno INT,
  OUT out_ename VARCHAR(100),
  OUT out_job VARCHAR(100),
  OUT out_sal DOUBLE)
BEGIN
  SELECT ename, job, sal
  FROM emp WHERE empno = in_empno
  INTO out_ename, out_job, out_sal;
END
//

DELIMITER ;

--=============================================
-- # 프로시저
-- 실행방법:
-- call pro_emp_count(@count);
-- select @count;
--=============================================

DELIMITER //

DROP PROCEDURE IF EXISTS pro_emp_count//

CREATE PROCEDURE pro_emp_count(OUT out_count INT)
BEGIN
    SELECT COUNT(*) FROM emp INTO out_count;
END
//

DELIMITER ;

--=============================================
-- # 함수
-- 프로시저와 비슷해 보이지만 함수는 결과값을 리턴할 수 있다.
-- 작성문법:
-- CREATE FUNCTION 함수이름(인자 인자타입) RETURNS 반환 타입
-- BEGIN
--     sql문
--     RETURN 반환값 식
-- END
--
-- 실행방법:
-- SELECT fn_emp_count();
--=============================================

DELIMITER //
CREATE FUNCTION fn_emp_count() RETURNS INT
BEGIN
    DECLARE result INT;
    SELECT count(*) FROM emp INTO result;
```

```
    RETURN result;
END
//
DELIMITER ;
```

2. 디비 연동 기술 선택

Spring JDBC 기술을 사용한다.

3. 프로젝트 생성, 디펜던시 설정

스프링 부트로 프로젝트를 생성하여 환경설정 시간을 단축할 것이다.

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

Project Name : spring-jdbc-3
Type : Maven
Packaging : Jar
Java : 8
Package : com.example.demo

Project Dependencies : Web, Lombok, JDBC, MySQL
```

pom.xml

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.17.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
```

```
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

4. 빈 컨테이너 설정

src\main\resources\application.properties

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/test_spring_jdbc?createDatabaseIfNotExist=true&useUr
spring.datasource.username=root
spring.datasource.password=1111
```

MariaDB를 설치하면 자동으로 test라는 DB는 존재하지만 `test_spring_jdbc`라는 DB는 존재하지 않습니다.
URL 연결구문에서 "createDatabaseIfNotExist=true"라고 추가해 놓으면 연결시도 시 디비가 생성됩니다.
password는 디비를 설치할 때 설정한 것을 사용하자.

5. Domain 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;
}
```

6. DAO

간단히 프로시저와의 연동방법을 살펴보는 것이므로 DAO 인터페이스를 따로 만들지 않을 것이다.

EmpDaoWithProc.java

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

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import java.util.Map;

import javax.annotation.PostConstruct;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.stereotype.Repository;

import com.example.demo.domain.Emp;

@Repository
public class EmpDaoWithProc {
    @Autowired
    public JdbcTemplate jdbcTemplate;

    private SimpleJdbcCall procFindEmpAll;
    private SimpleJdbcCall procFindEmpByEmpno;
    private SimpleJdbcCall procEmpCount;
    private SimpleJdbcCall fnEmpCount;

    private RowMapper<Emp> rowMapper = new RowMapper<Emp>() {
        @Override
        public Emp mapRow(ResultSet rs, int rowNum) throws SQLException {
            Emp e = new Emp();
            e.setEmpno(rs.getInt("empno"));
            e.setEname(rs.getString("ename"));
            e.setJob(rs.getString("job"));
            e.setSal(rs.getDouble("sal"));

            return e;
        }
    };

    @PostConstruct
    public void init() {
        this.procFindEmpAll = new SimpleJdbcCall(jdbcTemplate)
                  .withProcedureName("pro_find_all")
                  .returningResultSet("returnedEmps", rowMapper);
        this.procFindEmpByEmpno = new SimpleJdbcCall(jdbcTemplate)
                  .withProcedureName("pro_find_one");
        this.procEmpCount = new SimpleJdbcCall(jdbcTemplate)
                  .withProcedureName("pro_emp_count");
        this.fnEmpCount = new SimpleJdbcCall(jdbcTemplate)
                  .withFunctionName("fn_emp_count");
    }
```

```
    @SuppressWarnings("unchecked")
    public List<Emp> findAll() {
        Map<String, Object> out = procFindEmpAll.execute();
        return (List<Emp>) out.get("returnedEmps");
    }

    public Emp findEmpByEmpno(int empno) {
        SqlParameterSource parameterSource = new MapSqlParameterSource()
                  .addValue("in_empno", empno);
        Map<String, Object> out = procFindEmpByEmpno.execute(parameterSource);

        Emp emp = new Emp();
        emp.setEmpno(empno);
        emp.setEname((String) out.get("out_ename"));
        emp.setJob((String) out.get("out_job"));
        emp.setSal((Double) out.get("out_sal"));

        return emp;
    }

    public int countUsingProc() {
        Map<String, Object> out = procEmpCount.execute();
        return (int) out.get("out_count");
    }

    public int countUsingFn() {
        return fnEmpCount.executeFunction(Integer.class);
    }

}
```

7. TEST

테스트는 언제나 중요하다. 테스트 코드를 잘 작성할 수록 확신을 갖게 된다. 테스트 코드를 제대로 작성했다면 주말에 전화가 오는 일은 발생하지 않을 것이다.

EmpDaoWithProcTest.java

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

import java.util.List;

import javax.annotation.Resource;

import org.junit.Test;
import org.junit.runner.RunWith;
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 EmpDaoWithProcTest {
    @Resource(name="empDaoWithProc")
    private EmpDaoWithProc dao;
```

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

    @Test
    public void testFindEmpByEmpno() {
        Emp emp = dao.findEmpByEmpno(1);
        System.out.println(emp);
    }

    @Test
    public void testCountUsingProc() {
        int count = dao.countUsingProc();
        System.out.println("count = " + count);
    }

    @Test
    public void testCountUsingFn() {
        int count = dao.countUsingFn();
        System.out.println("count = " + count);
    }
}
```

다음 개발 과정은 앞서서 살펴보았으므로 생략한다. 직접 작성해 보자.

```
8. Service
9. Controller
10. JSP
11. TEST : URI
```

배치작업

갱신(update)을 배치로 그룹핑해서 데이터베이스에 라운드트립하는 수를 줄인다. 같은 프리페어드 스테이트먼트를 여러번 호출하는 배치작업 시 향상된 성능을 제공한다.

EmpDaoBatch.java

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

import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ParameterizedPreparedStatementSetter;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSourceUtils;
import org.springframework.stereotype.Repository;
```

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

@Repository
public class EmpDaoBatch {
    @Autowired
    public JdbcTemplate jdbcTemplate;
    @Autowired
    public NamedParameterJdbcTemplate nTemplate;

    public int[] batchUpdateUsingJdbcTemplate(List<Emp> emps) {
        String sql = "INSERT INTO emp (ename, job, sal) VALUES (?, ?, ?)";

        return jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws SQLException {
                Emp emp = emps.get(i);
                ps.setString(1, emp.getEname());
                ps.setString(2, emp.getJob());
                ps.setDouble(3, emp.getSal());
            }

            @Override
            public int getBatchSize() {
                return emps.size();
            }
        });
    }

    public int[][] batchUpdateUsingJdbcTemplateBatchSize(List<Emp> emps) {
        String sql = "INSERT INTO emp (ename, job, sal) VALUES (?, ?, ?)";

        // 큰 배치작업일 때, 50개 단위로 잘라서 처리할 수 잇다.
        return jdbcTemplate.batchUpdate(sql, emps, 50,
                new ParameterizedPreparedStatementSetter<Emp>() {
            @Override
            public void setValues(PreparedStatement ps, Emp emp) throws SQLException {
                ps.setString(1, emp.getEname());
                ps.setString(2, emp.getJob());
                ps.setDouble(3, emp.getSal());
            }
        });
    }

    public int[] batchUpdateUsingNamedParameterJdbcTemplate(List<Emp> emps) {
        String sql = "INSERT INTO emp (ename, job, sal) VALUES (:ename, :job, :sal)";

        SqlParameterSource[] batch = SqlParameterSourceUtils
                  .createBatch(emps.toArray());
        int[] updateCounts = nTemplate.batchUpdate(sql, batch);
        return updateCounts;
    }
}
```

EmpDaoBatchTest.java

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

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```
import javax.annotation.Resource;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
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 EmpDaoBatchTest {
    @Resource(name = "empDaoBatch")
    private EmpDaoBatch dao;
    private List<Emp> emps = new ArrayList<>();

    @Before
    public void setUp() throws Exception {
        emps.add(new Emp().setEname("Tom").setJob("Actor").setSal(500));
        emps.add(new Emp().setEname("Chris").setJob("Teacher").setSal(500));
    }

    @Test
    public void testBatchUpdateUsingJdbcTemplate() {
        int[] affecteds = dao.batchUpdateUsingJdbcTemplate(emps);
        System.out.println(Arrays.toString(affecteds));
    }

    @Test
    public void testBatchUpdateUsingJdbcTemplateBatchSize() {
        int[][] affecteds = dao.batchUpdateUsingJdbcTemplateBatchSize(emps);
        for (int[] updates : affecteds) {
            System.out.println(Arrays.toString(updates));
        }
    }

    @Test
    public void testBatchUpdateUsingNamedParameterJdbcTemplate() {
        int[] affecteds = dao.batchUpdateUsingNamedParameterJdbcTemplate(emps);
        System.out.println(Arrays.toString(affecteds));
    }
}
```

정리

Spring JDBC 기술의 핵심은 반복적이고 뻔한 작업은 스프링이 대신 처리하도록 맡기고 개발자는 상황에 따라 변하는 부분인 SQL쿼리 작성과 질의결과의 처리 부분만 작업하면 된다는 것 입니다. 주요 파일의 역할을 다시 점검해서 숙지하시기 바랍니다.

- pom.xml
  프로젝트에서 사용하는 디펜던시를 설정한다.

- application.properties
  데이터베이스 연결정보를 설정한다.

- Emp.java
  테이블의 1 Row 를 자바 객체의 1 Object 로 취급하는 용도의 모델 클래스를 만든다.

- EmpDao.java, EmpDaoImpl.java
  데이터베이스 처리로직을 갖고 있는 DAO 클래스를 작성한다.

- EmpController.java
  사용자의 URL 요청을 받아서 결과를 돌려주는 기능을 수행하는 컨트롤러 클래스를 생성한다. 필요하다면 JSP 를 별도로 사용하여 HTML과 데이터를 결합한 후 전달한다.

여러 개의 파일을 역할에 따라 체계적으로 만든 후 연동해서 처리합니다. 사실 대부분의 작업은 미래를 위한 일종의 투자였습니다. 로직이 추가되거나 변화될 것을 대비해서 미리 예비작업을 수행했다고 볼 수 있습니다.

데이터베이스와 대화하는 로직을 갖고 있는 EmpDaoImpl 클래스의 메소드를 살펴보면 코드는 두 줄에 불과합니다. 이는 스프링이 반복적으로 사용되는 JDBC의 Connection. Statement, ResultSet 객체들을 대신 처리해 주기 때문에 가능한 일입니다.

더불어서 스프링은 체크드 예외를 언체크드 예외로 전환해서 던져주기 때문에 논리적으로 의미없는 try-catch 구문을 개발자가 매번 코딩할 필요도 없어졌습니다.

하지만 데이터베이스 로직만을 개발해서 다른 개발자에게 제공하는 경우에는 사용의 오용을 방지하기 위해서 try구문이 필요할 수도 있습니다. 이런 경우 여러 데이터베이스가 제 각각 다르게 던지는 예외를 일관된 예외로 변경해서 돌려주는 스프링의 예외전환 서비스의 혜택을 느껴 볼 수 있을 것 입니다.