

스프링(Spring) 개발

간단한 게시판을 하나 만들어 보면서 스프링의 MVC 구조와 DB, 트랜잭션, 파일 업로드, jQuery등을 살펴봅니다.

(10) 게시판 만들기 - 게시판 목록 조회 출력

대소문자의 구분이 굉장히 중요합니다. 대소문자의 구분을 잘 확인하세요.

10.1. DB

데이터베이스는 Oracle 11g XE를 기준으로 설명하려고 한다.

10.1.1). 테이블 생성

다음의 쿼리를 실행시키자.

```
1      CREATE TABLE TB_BOARD
2      (
3          IDX NUMBER PRIMARY KEY,
4          PARENT_IDX NUMBER,
5          TITLE VARCHAR2(100) NOT NULL,
6          CONTENTS VARCHAR2(4000) NOT NULL,
7          HIT_CNT NUMBER NOT NULL,
8          DEL_GB VARCHAR2(1) DEFAULT 'N' NOT NULL,
9          CREA_DTM DATE DEFAULT SYSDATE NOT NULL,
10         CREA_ID VARCHAR2(30) NOT NULL
11     );
12
13     COMMENT ON TABLE TB_BOARD IS '게시판';
14     COMMENT ON COLUMN TB_BOARD.IDX IS '인덱스';
15     COMMENT ON COLUMN TB_BOARD.PARENT_IDX IS '부모글 인덱스';
16     COMMENT ON COLUMN TB_BOARD.TITLE IS '제목';
17     COMMENT ON COLUMN TB_BOARD.CONTENTS IS '내용';
18     COMMENT ON COLUMN TB_BOARD.HIT_CNT IS '조회수';
19     COMMENT ON COLUMN TB_BOARD.DEL_GB IS '삭제구분';
20     COMMENT ON COLUMN TB_BOARD.CREA_DTM IS '생성일자';
21     COMMENT ON COLUMN TB_BOARD.CREA_ID IS '생성자 ID';
```

몇가지 컬럼만 살펴보자.

PARENT_IDX 는 추후 계층형 게시판으로 변형할 때, 사용하려고 미리 컬럼을 만들었다.
DEL_GB는 글을 삭제할 경우, 실제로 DELETE 쿼리를 이용하여 삭제하는것이 아닌, 삭제구분값만 바꾸려고 한다.

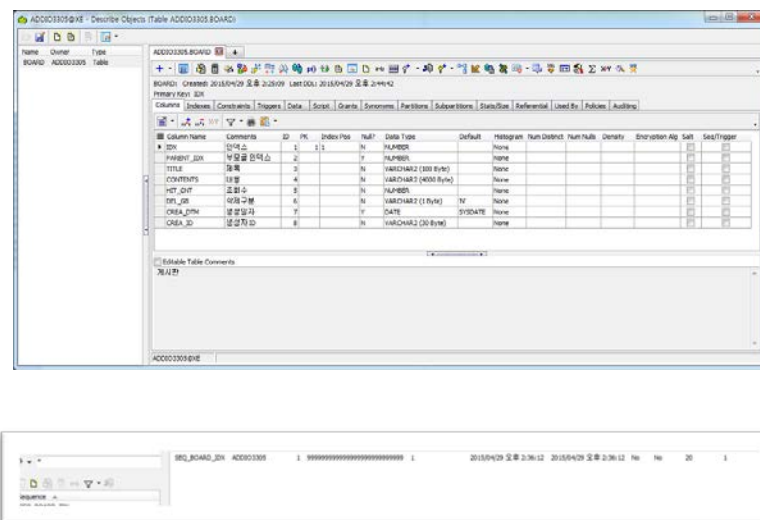
10.1.2). 시퀀스 생성

게시판의 인덱스를 자동 증가시키기 위하여 시퀀스를 하나 생성한다.
오라클은 MySql이나 MSSql에서 사용되는 Auto Increment가 없다.

다음의 쿼리를 실행시키자.

```
1 CREATE SEQUENCE SEQ_TB_BOARD_IDX
2 START WITH 1
3 INCREMENT BY 1
4 NOMAXVALUE
5 NOCACHE;
```

모두 완료하면 다음과 같은 테이블과 시퀀스가 만들어져 있다.



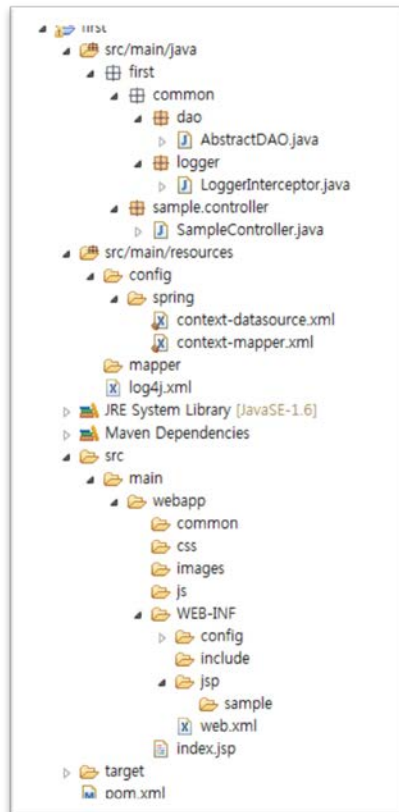
이렇게 DB는 준비가 완료되었다.

10.2. 데이터 조회

이제 스프링에서 해당 데이터를 어떻게 조회하는지 알아보자.
이제 본격적으로 비즈니스로직과 DB와의 연동을 시작한다.

10.2.1). 폴더 및 패키지 구성하기

현재 폴더 구조는 다음과 같다.



여기에서 각각 필요한 패키지 및 폴더를 만들어 보려고 한다.

1) service, dao 패키지 생성

src/main/java 밑의 first.sample 패키지 밑에 service, dao 패키지를 생성한다.

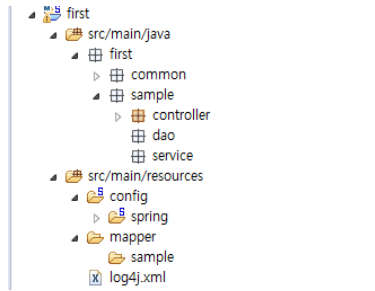
- service 패키지는 다시 Service 인터페이스와 그 인터페이스를 구현한 ServiceImpl 클래스가 위치하게 된다. ServiceImpl 클래스는 각 요청에 필요한 비즈니스 로직의 수행을 담당하게 된다.
- dao 패키지는 DAO클래스가 위치하게 되는곳으로, DAO 클래스를 통하여 데이터베이스에 접근하는 역할을 수행한다.

2) sample 폴더 생성

src/main/resources/ 밑의 mapper 폴더 밑에 sample 폴더를 생성한다.

이 폴더에는 게시판에 관련된 쿼리가 담긴 XML 파일이 위치하게 된다.

위의 패키지 및 폴더를 작성하면 다음과 같다.



10.2.2). 실제 기능 구현

이제 실제로 기능을 구현할 차례다. 기능 구현은 다음과 같은 순서로 작성한다.

- **Controller > Service > ServiceImpl > DAO > SQL(XML) > JSP**

1) Controller 구현

일단 SampleController에 다음의 내용을 작성한다.

```
1 package first.sample.controller;
2
3 import java.util.List;
4 import java.util.Map;
5
6 import javax.annotation.Resource;
7
8 import org.apache.log4j.Logger;
9 import org.springframework.stereotype.Controller;
10import org.springframework.web.bind.annotation.RequestMapping;
11import org.springframework.web.servlet.ModelAndView;
12
13@Controller
14public class SampleController {
15    Logger log = Logger.getLogger(this.getClass());
16
17    @Resource(name="sampleService")
18    private SampleService sampleService;
19
20    @RequestMapping(value="/sample/openSampleBoardList.do")
21    public ModelAndView openSampleBoardList(Map<string,object> commandMap)
22throws Exception{
23        ModelAndView mv = new ModelAndView("/sample/boardList");
24
25        List<Map<String,Object>> list = sampleService.selectBoardList(commandMap);
```

```

26     mv.addObject("list", list);
27
28     return mv;
29 }

```

일단 이 코드를 작성하면 18, 25번째줄에서 에러가 발생한다. SampleService 클래스가 없어서 에러가 나는것은 당연한 일이니 그대로 진행한다.

간단히 살펴보면,

컨트롤러(Controller)는 웹 클라이언트에서 들어온 요청을 해당 비즈니스 로직을 호출하고, 수행결과와 함께 응답을 해주는 Dispatcher 역할을 한다. 클래스의 선언부에 @Controller 어노테이션(Annotation)을 이용하여, Controller 객체임을 선언한다.

17,18번째 줄은 Service 영역의 접근을 위한 선언인데, 이는 추후 Service를 설명할때, 다시 설명한다.

@RequestMapping은 요청 URL을 의미한다. 우리가 /sample/openSampleBoardList.do 라는 주소를 호출하게 되면, 이 주소는 @RequestMapping 어노테이션과 매핑되어, 해당 메서드가 실행된다.

ModelAndView mv = new ModelAndView("/sample/sampleBoardList"); 는 우리가 화면에 보여줄 jsp파일을 의미한다. 아직 sampleBoardList.jsp 파일을 만들지는 않았지만, 추후 저 위치에 저 이름으로 생성할 거라서, 미리 작성하였다.

그 다음 중요한 부분이다.

List<Map<String,Object>> list = sampleService.selectBoardList(commandMap); 부분이다.

일단 왼쪽의 List<Map<String,Object>> list는 제너릭스 기능이 적용된 레퍼런스 변수의 선언이다.

하나씩 살펴보면 다음과 같다.

게시판 목록을 보여주기 때문에, 목록을 저장할 수 있는 List를 선언하였다. 그리고 그 List의 형식은 Map<String, Object>인데, 하나의 게시글 목록에도 여러가지 정보가 존재한다. 글번호, 글제목, 작성일 등의 내용을 Map에 저장하려는 것이다. Map은 다시 키(key)와 값(value)로 구분되어 있는데, 각각의 컬럼인 글번호, 글제목, 작성일 등의 키와 실제 값이 저장된다. 이는 잠시후, 데이터를 읽어와서 화면에 보여줄때, 다시 설명을 하도록 하겠다.

그 다음은 sampleService.selectBoardList(commandMap); 부분이다.

Controller는 단순히 어떤 주소와 화면을 연결하고, 비즈니스 로직을 호출하는 역할을 한다. 실제로 여러가지 비즈니스 로직은 Service에서 작성한다. 여기서는 단순히 게시글을 조회하는 역할을 하지만, 나중에 게시글 상세조회와 경우, 조회수 증가와 게시글 상세내용을 조회하는 두가지 기능이 필요한데, 이를 Service에서 처리해주게 된다.

마지막으로 `mv.addObject("list", list);`는 서비스로직의 결과를 `ModelAndView` 객체에 담아서 클라이언트, 즉 `jsp`에서 그 결과를 사용할 수 있도록 한다. `mv`에 값을 저장하는것은 `map`과 똑같이 키(key)와 값(value)로 구성이 되는데, `sampleService.selectBoardList` 메서드를 통해 얻어온 결과 `list`를 "list"라는 이름으로 저장하고 있다.

나중에, `jsp` 화면을 설명할 때, 이 값이 어떻게 사용되는지 좀 더 자세히 볼 예정이다.

2) Service 구현

Service 영역은 두 개의 파일로 구성된다.

Service 인터페이스와 이 인터페이스를 실제로 구현한 **ServiceImpl 클래스**로 구성이 되어있다.

이는 Spring의 IoC/DI (Inversion of Control / Dependency Injection) 기능을 이용한 Bean 관리 기능을 사용하기 위함이다.

일단 작성을 하고, 추후 알아가도록 하자.

먼저 `service` 패키지에 `SampleService` 인터페이스와 `SampleServiceImpl` 클래스를 만들자.

SampleService.java

```
1 package first.sample.service;
2
3 public interface SampleService {
4
5 }
```

SampleServiceImpl.java

```
1 package first.sample.service;
2
3 import org.springframework.stereotype.Service;
4
5 @Service("sampleService")
6 public class SampleServiceImpl implements SampleService{
7
8 }
```

일단 지금은 클래스만 생성했으며, 이제 하나씩 채워나가보자.

먼저 `Service` 인터페이스는 비즈니스 로직의 수행을 위한 메서드를 정의한다.

`ServiceImpl` 클래스는 `Service` 인터페이스를 통해 정의된 메서드를 실제로 구현하는 클래스다.

`@Service` 어노테이션을 이용하여 `Service` 객체임을 선언하였고, 이 객체의 이름은 "sampleService"라고 선언하였다.

이렇게 Service 영역의 파일을 생성한 후, 다시 Controller로 돌아가서 Ctrl + Shift + O를 눌러보자. (이클립스 기준, 자동 import를 하는 단축키이다. 다른 에디터를 사용할 경우, 해당 단축키 또는 기능을 사용한다.)

그러면 import first.sample.service.SampleService; 가 추가되면서 기존 18번째 줄에서 나는 에러는 사라질 것이다.

아까 앞에서

@Resource(name="sampleService")

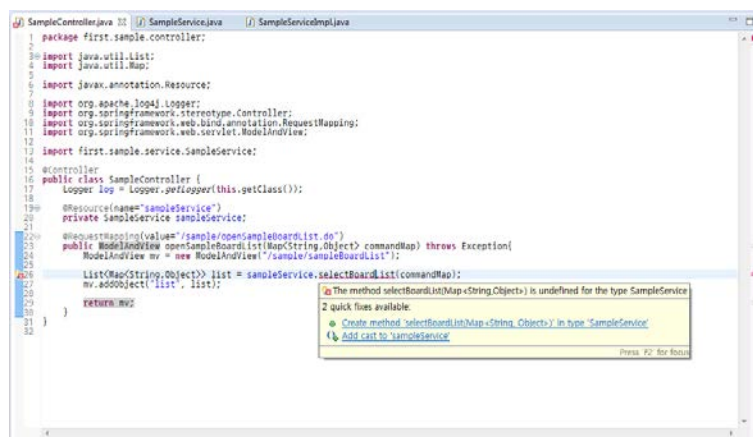
private SampleService sampleService;

이 부분은 Service 영역의 접근을 위한 선언이라고 이야기를 하였다. 우리가 SampleServiceImpl을 생성하고 그 Service를 "sampleService"라는 이름으로 선언을 하였는데, 이를 사용하기 위한 선언이다.

여기서 @Resource 어노테이션을 통해서 필요한 빈(bean)을 수동으로 등록하는것이다. 그리고 수동으로 등록할 빈의 이름이 "sampleService"이고, 이는 @Service("sampleService")라고 선언했을 때의 그 이름인 것을 확인한다.

다음, sampleService.selectBoardList(commandMap); 에서 selectBoardList에 아직 빨간 밑줄이 그려져 있을것이다. 해당 라인 위에 마우스를 올려보자.

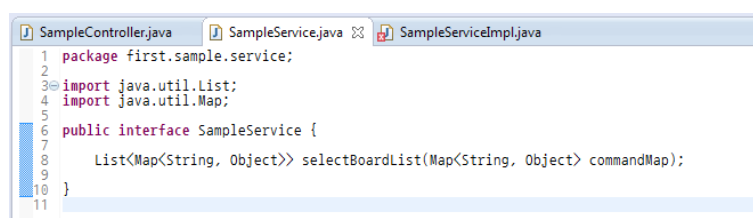
그러면 다음과 같은 화면이 나올것이다.



이는 sampleService에 selectBoardList라는 메서드가 정의되지 않았음을 의미한다.

Create method 'selectBoardList(Map<String, Object>)' in type 'SampleService' 를 클릭하자.

그러면 자동적으로 SampleService에 해당 메서드가 선언이 된다.



이 후, SampleService를 다음과 같이 import 구문을 추가하고 수정하자.

```
1 package first.sample.service;
2
3 import java.util.List;
4 import java.util.Map;
5
6 public interface SampleService {
7     List<Map<String, Object>> selectBoardList(Map<String, Object> map) throws
8     Exception;
9
10 }
```

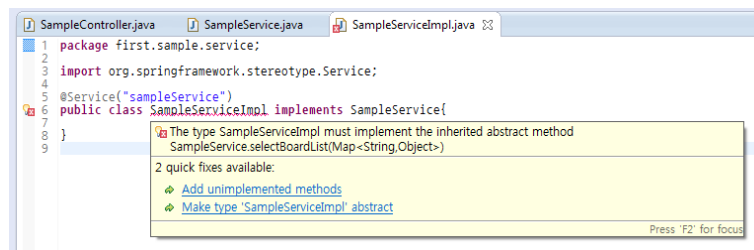
변수 이름을 map으로 바꾸었고, 뒤에 throws Exception을 붙였다.

이는 추후 에러처리를 위한 것으로, 원래 자바에서는 에러처리를 위하여 try, catch 문을 이용하여 적절한 에러처리를 해야 한다. 그렇지만 모든곳에서 동일한 에러처리를 하는것은 현실적으로 힘들고, 예상하지 못한 에러도 발생할 수 있다.

그래서 모든 메서드에서는 에러가 발생하면 Exception을 날리고, 공통으로 이 Exception을 처리하는 로직을 추후 추가하려고 한다.

위와 같이 수정한 후에는 SampleServiceImpl에서 에러가 발생할 것이다.

SampleServiceImpl.java 파일에서는 클래스 명에서 에러가 발생한다. 마우스를 올려보자.



아까 Controller에서 봤던 에러와는 약간 다르다.

Controller에서는 selectBoardList라는 메서드가 선언이 되지 않았다는 에러 메시지였는데, 여기서는 selectBoardList를 무조건 구현해야 한다는 메시지다.

Add unimplemented methods를 선택한다.



```
1 package first.sample.service;
2
3 import java.util.List;
4 import java.util.Map;
5
6 import org.springframework.stereotype.Service;
7
8 @Service("sampleService")
9 public class SampleServiceImpl implements SampleService{
10
11     @Override
12     public List<Map<String, Object>> selectBoardList(Map<String, Object> map)
13         throws Exception {
14         // TODO Auto-generated method stub
15         return null;
16     }
17
18 }
19 }
```

이렇게 인터페이스에 정의된 메서드를 실제 구현할 수 있는 코드가 작성된다.
이제 이 ServiceImpl 클래스를 다음과 같이 수정하자.

```
1 package first.sample.service;
2
3 import java.util.List;
4 import java.util.Map;
5
6 import javax.annotation.Resource;
7
8 import org.apache.log4j.Logger;
9 import org.springframework.stereotype.Service;
10
11 @Service("sampleService")
12 public class SampleServiceImpl implements SampleService{
13     Logger log = Logger.getLogger(this.getClass());
14
15     @Resource(name="sampleDAO")
16     private SampleDAO sampleDAO;
17
18     @Override
19     public List<Map<String, Object>> selectBoardList(Map<String, Object> map) throws
20     Exception {
21         return sampleDAO.selectBoardList(map);
22     }
23 }
```

먼저 여기서도 @Resource 어노테이션을 볼 수 있다.
Controller에서 Service 접근을 위한 선언을 한 것과 마찬가지로, Service에서는 데이터 접근을 위
한 DAO(Data Access Object) 객체를 선언하였다.

아직 SampleDAO 클래스가 작성되지 않았으니, 에러는 발생할 것이다.

그 다음으로 서비스의 selectBoardList의 결과값으로 sampleDAO 클래스의 selectBoardList 메서드를 호출하고, 그 결과값을 바로 반환(return) 하였다.

현재로는 Service 영역에서 수행해야 할 비즈니스 로직이 목록조회 밖에 없기 때문에, 바로 return을 했지만, 게시판 상세조회 등에서는 2개 이상의 로직이 필요하기 때문에, 비즈니스 로직에서 여러가지 일을 수행하도록 수정해야 할 것이다.

3) DAO 구현

이제 실제로 데이터베이스에 접근하는 DAO를 생성해 보자.

dao 패키지에 SampleDAO 클래스를 생성하자.

```
1      package first.sample.dao;
2
3      import org.springframework.stereotype.Repository;
4
5      import first.common.dao.AbstractDAO;
6
7      @Repository("sampleDAO")
8      public class SampleDAO extends AbstractDAO{
9
10     }
```

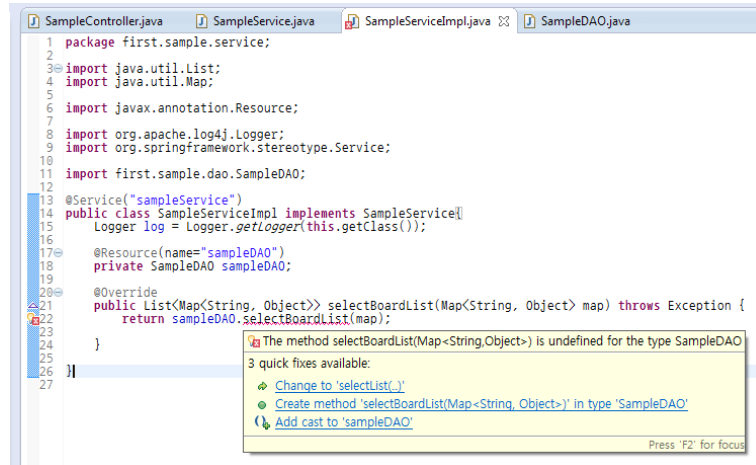
@Repository라는 어노테이션을 통해서 이 클래스가 DAO임을 선언하고 이름을 "sampleDAO"로 작성하였다.

SampleServiceImpl에서 @Resource(name="sampleDAO")로 bean을 수동으로 등록하였고, 거기서 사용된 빈이 방금 작성한 SampleDAO 클래스다.

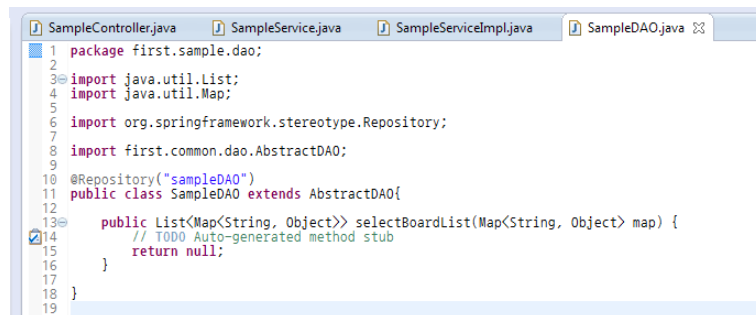
그리고 AbstractDAO 클래스를 상속받았다.

AbstractDAO 클래스는 순전히 본인의 필요에 의해서 만든 것으로, 필요없다고 생각된다면, AbstractDAO에서 사용중인 기능 등을 바로 SampleDAO 등 클래스에서 구현하면 된다.

이렇게 DAO 클래스를 만들었으면 다시 SampleServiceImpl.java 파일로 가서 아까와 마찬가지로 selectBoardList에 마우스를 올려놓자.



Create method 'selectBoardList(Map<String, Object>)' in type 'SampleDAO' 를 선택하자.
그러면 SampleDAO에 다음과 같은 메서드가 생긴다.



이렇게 생성된 메서드를 실제 동작하도록 바꾸자.
다음과 같이 작성한다.

```

1 package first.sample.dao;
2
3 import java.util.List;
4 import java.util.Map;
5
6 import org.springframework.stereotype.Repository;
7
8 import first.common.dao.AbstractDAO;
9
10 @Repository("sampleDAO")
11 public class SampleDAO extends AbstractDAO{
12
13     @SuppressWarnings("unchecked")
14     public List<Map<String, Object>> selectBoardList(Map<String, Object> map) throws

```

```

15Exception{
16    return (List<Map<String, Object>>)selectList("sample.selectBoardList", map);
17 }
18
    }

```

DAO는 데이터베이스에 접근하여 데이터를 조작하는 (가져오거나 입력하는 등) 역할만 수행한다.

우리는 MyBatis 프레임워크를 사용하기 때문에 JDBC 연결 및 쿼리 수행을 편하게 할 수 있다. MyBatis는 프로그램의 소스코드에서 SQL을 분리하여 별도의 XML 파일에 저장하고, 이 둘을 연결하는 방식으로 작동한다. 여기서는 우리가 실행할 쿼리 이름과 해당 쿼리에 필요한 변수들을 매개변수로 MyBatis의 리스트 조회 메서드를 호출하였다.

여기서 사용한 selectList는 MyBatis의 기본 기능으로, 리스트를 조회할 때 사용한다. AbstractDAO를 생성할 때, MyBatis의 기본 기능과 동일한 이름으로 만들어 놔는데, 필요에 따라서 이름을 변경하는 등, 자신에 맞게끔 변경하여 사용하면 된다.

selectList 메서드의 인자는 두 가지이다.

첫번째는 쿼리 이름, 두번째는 쿼리가 실행되는데 필요한 변수들이다.

"sample.selectBoardList"가 쿼리 이름이고, Controller에서부터 계속 넘어온 Map<String, Object> map이 쿼리 실행시 필요한 변수들이다. 게시판 목록을 조회할때 지금 당장 필요한 변수들이 없기 때문에, 이 인자는 무의미하게 느껴질 수 있지만, 추후 다른 기능등에서 사용하게 될 것이다.

그리고 그 결과값은 List<Map<String,Object>> 형식으로 반환할 수 있도록 형변환을 하였다.

4) SQL 작성

이제 게시판 목록을 조회하는 쿼리를 작성하자.

src/main/resources 폴더 밑에 mapper/sample 폴더를 작성했었다. 여기에 **Sample_SQL.xml** 파일을 만들고, 다음의 내용을 작성하자.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4
5 <mapper namespace="sample">
6
7 </mapper>

```

우리는 MyBatis를 사용하고, 이 프레임워크는 SQL을 분리하여 별도의 XML 파일에 저장한다고 하였다. 지금 만든 Sample_SQL.xml 파일에 우리가 호출할 쿼리가 저장된다.

프로젝트에서는 기본적으로 여러 개의 <mapper>를 가지기 때문에 중복되는 이름을 가진 SQL이 존재할 수 있다. 따라서 각 <mapper>마다 namespace 속성을 이용하여 <mapper>간 유일성을 보장해야 한다. 여기서는 sample이라는 이름의 namespace를 사용하였다.

여기서 잘 봐야할 것은 아까 DAO에서 쿼리 아이디를 잘 생각해보자.

"sample.selectBoardList"라는 이름인 것을 확인할 수 있다. 여기서 앞에 붙은 sample이라는 부분이 바로 XML에서 설정한 namespace의 이름이다.

다시 말해, 모든 쿼리는 "NAMESPACE . SQL ID"의 구조로 구성된다. (namespace를 사용하지 않을 경우, 바로 SQL ID만 호출하여도 되지만 위에서 이야기한 것처럼 중복된 아이디가 있을 수 있다. 따라서 가능한 namespace를 사용하자.)

이제 쿼리를 작성해보자.

먼저 아까 작성한 테이블에서 목록을 조회하는 쿼리부터 작성해 보자.

```
1      SELECT  IDX,  TITLE,  HIT_CNT,  CREA_DTM
2      FROM    TB_BOARD
3      ORDER BY IDX DESC
```

게시판 목록에 보여줄 글번호, 제목, 조회수, 작성일만 조회를 하였다.

이 쿼리를 그대로 XML에 작성해야 한다. 먼저 XML에 다음의 내용을 작성하자.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE    mapper    PUBLIC    "-//mybatis.org//DTD    Mapper    3.0//EN"
3  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4
5  <mapper namespace="sample">
6
7      <select id="selectBoardList" parameterType="hashmap" resultType="hashmap">
8          <![CDATA[
9              SELECT  IDX,  TITLE,  HIT_CNT,  CREA_DTM
10             FROM    TB_BOARD
11             WHERE   DEL_GB = 'N'
12             ORDER BY IDX DESC
13         ]]>
14      </select>
15
16</mapper>
```

이게 MyBatis에서 사용하는 XML의 모습이다.

하나씩 살펴보자.

- 첫번째로, <select> 태그를 이용하여 이 쿼리가 select문이라는 것을 명시하였다.
- id="selectBoardList" 부분은 이 쿼리의 id는 selectBoardList 라고 정의하였다.
- parameterType="hashmap" 부분은 이 쿼리가 실행될때 필요한 변수는 HashMap 형태를 의미한다.
- resultType="hashmap" 부분은 이 쿼리의 결과값은 HashMap에 담겨서 반환된다는 것을 의미한다.

parameterType과 resultType으로 모두 HashMap을 사용할 것인데, 다른 클래스를 사용할 경우 해당 클래스의 이름을 적어주면 된다.

원래는 HashMap을 사용하려고 하면 parameterType="java.util.HashMap"이라고 패키지명까지 정확히 명시를 해야 하지만, MyBatis에서 우리가 많이 사용하는 변수형은 hashmap 과 같이 간단히 사용할 수 있도록 지원하고 있다.

이에 대한 상세한 내용은 <https://mybatis.github.io/mybatis-3/ko/configuration.html#typeAliases> 부분을 참조하자.

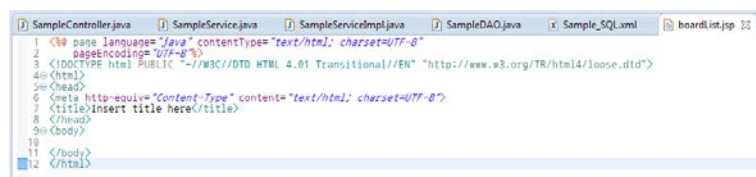
부가적으로, 모든 결과값은 Map을 사용하여 전달하고 전달받는데, 프로젝트에 따라서는 DTO를 사용하는 경우도 있다.

DTO는 Data Transfer Object의 약자로 TO, VO 등의 이름으로도 사용된다. 각 프로젝트별로 DTO, TO, VO를 혼용하여 사용하는데 TO와 VO는 약간 다른 개념이긴 하지만, 우리나라에서는 보통 혼용하여 사용된다.

5) jsp 구현

이제 마지막으로 사용자에게 보여줄 jsp를 구현하는 일만 남았다.

src/main/webapp/WEB-INF/jsp/sample 폴더 밑에 boardList.jsp 파일을 생성하자.



```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7   <title>Insert title here</title>
8 </head>
9 <body>
10
11 </body>
12 </html>
```

위와같이 생성된 jsp 파일에 다음과 같이 작성한다.

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <title>first</title>
```

```
7 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
8 <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
9 </head>
10<body>
11<h2>게시판 목록</h2>
12<table style="border:1px solid #ccc">
13    <colgroup>
14        <col width="10%"/>
15        <col width="*" />
16        <col width="15%"/>
17        <col width="20%"/>
18    </colgroup>
19    <thead>
20        <tr>
21            <th scope="col">글번호</th>
22            <th scope="col">제목</th>
23            <th scope="col">조회수</th>
24            <th scope="col">작성일</th>
25        </tr>
26    </thead>
27    <tbody>
28        <c:choose>
29            <c:when test="{fn:length(list) > 0}">
30                <c:forEach items="{list}" var="row">
31                    <tr>
32                        <td>${row.IDX }</td>
33                        <td>${row.TITLE }</td>
34                        <td>${row.HIT_CNT }</td>
35                        <td>${row.CREA_DTM }</td>
36                    </tr>
37                </c:forEach>
38            </c:when>
39            <c:otherwise>
40                <tr>
41                    <td colspan="4">조회된 결과가 없습니다.</td>
42                </tr>
43            </c:otherwise>
44        </c:choose>
45
```

```
46 </tbody>
47</table>
48</body>
</html>
```

전체 조회결과가 있으면 그 결과를 보여주고, 없으면 조회된 결과가 없다는 메시지를 보여주는 화면이다.

여기서 <c:forEach item="\${list}" var="row">라는 부분을 보자.

Controller에서 DB에서 조회된 결과 list를 mv.addObject를 이용하여 "list"라는 이름으로 mv에 값을 저장했었던 것을 떠올려 보자. (mv.addObject("list",list); 라고 작성했었다.)

JSTL의 forEach문을 통해서 테이블의 목록을 만드려고 하는데, 여기서 사용할 아이템은 \${list}라고 되어있다. 이게 바로 우리가 mv에서 추가를 해줬던 그 list를 의미한다.

그 다음으로 var="row"라고 되어있다. 우리는 List<Map<String, Object>> 라는 형식으로 list를 선언하고, DB에서 조회된 결과를 받았다.

이 list에서 하나의 데이터쌍을 가져오면 (Java에서 list.get(index)를 의미한다.) 그 데이터는 Map<String,Object> 형식의 데이터 한줄이 나오게 되고, 그 데이터는 row라는 이름의 변수에 저장된다.

그 후, row에서 원하는 데이터를 뽑아서 사용하면 된다.

row.IDX, TITLE, HIT_CNT, CREA_DTM의 이름에서 볼 수 있듯이, 이는 우리가 select 쿼리를 실행했을 때 뽑아온 데이터의 이름이다. **(대소문자를 구분한다.)**

쿼리의 select 문에서 뽑아온 이름이 Map의 키(key)로 사용되고, 화면에서는 그 키를 이용하여 데이터에 접근할 수 있다.

그럼 이제 실행시켜 보자.

톰캣 서버를 실행시키고, 주소표시줄에 아까 Controller에서 설정한 주소와 이름을 입력한다.

여기까지 모두 정상적으로 되었으면 다음과 같은 화면이 보인다.



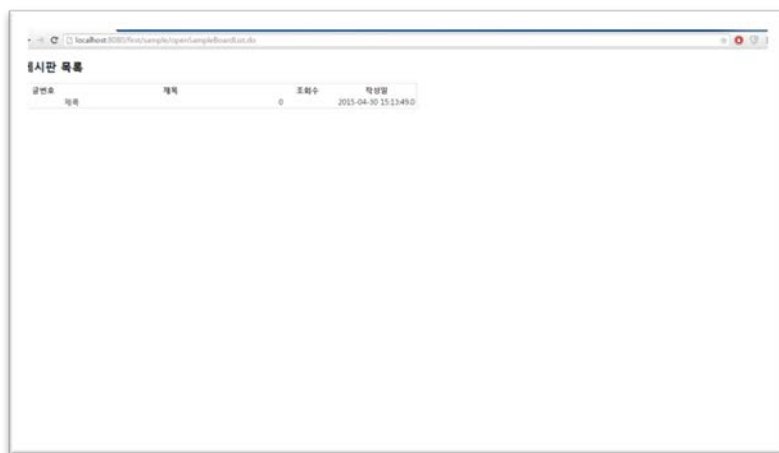
우리는 아까 테이블만 만들고 실제 데이터가 하나도 없기 때문에, 위와 같이 조회된 결과가 없다고 보여지고 있다.

그럼 DB에 임의의 데이터를 하나 입력하고 결과를 살펴보자.

```
1      INSERT INTO TB_BOARD
2      ( IDX, TITLE, CONTENTS, HIT_CNT, DEL_GB, CREA_DTM, CREA_ID)
3      VALUES( SEQ_TB_BOARD_IDX.NEXTVAL, '제목', '내용', 0, 'N', SYSDATE,
4      'Admin');
```

IDX에 아까 만든 시퀀스를 이용하여 값을 넣어주었고, 작성자를 의미하는 CREA_ID에는 임시로 값을 넣었다.

이렇게 쿼리를 실행시킨 후, 다시 주소를 호출하여보자.



위와 같이 정상적으로 데이터가 나오는것을 볼 수 있다.

지금은 단순히 하나의 데이터만 넣었는데, 여러 개의 데이터를 넣어보고 테스트를 해보기를 바란다.

 first2.zip

(11) HandlerMethodArgumentResolver 적용

HandlerMethodArgumentResolver 가 없어도 개발은 할 수 있지만, 개발을 더욱 편하게 할 수 있습니다.

1. HandlerMethodArgumentResolver 란?

HandlerMethodArgumentResolver 는 스프링 3.1에서 추가된 인터페이스이다.

스프링 3.1 이전에는 WebArgumentResolver 라는 인터페이스였는데, 스프링 3.1 이후부터 HandlerMethodArgumentResolver 라는 이름으로 바뀌었다.

이 인터페이스의 역할은 다음과 같다.

스프링 사용시, 컨트롤러(Controller)에 들어오는 파라미터(Parameter)를 수정하거나 공통적으로 추가를 해주어야 하는 경우가 있다.

예를 들어, 로그인을 한 사용자의 사용자 아이디나 닉네임등을 추가하는것을 생각해보자.

보통 그런 정보는 세션(Session)에 담아놓고 사용하는데, DB에 그러한 정보를 입력할 때에는 결국 세션에서 값을 꺼내와서 파라미터로 추가를 해야한다.

그런 경우가 한두번 있다면 몰라도, 여러번 사용되는 값을 그렇게 일일이 세션에서 가져오는건 상당히 번거로운 일이다.

HandlerMethodArgumentResolver 는 사용자 요청이 Controller에 도달하기 전에 그 요청의 파라미터들을 수정할 수 있도록 해준다.

자세한건 소스를 보면서 하나씩 살펴보자.

1). CommandMap 클래스 생성

request에 담겨있는 파라미터를 Map에 담아주는 역할을 하는 클래스다.

앞서 작성된 컨트롤러를 살펴보면,

```
public ModelAndView openSampleBoardList(Map<String,Object> commandMap) throws Exception{ 라고 선언을 했었다.
```

여기서 Map<String,Object> commandMap에 사용자가 넘겨준 파라미터가 저장된다는 것이다.

그런데 여기서 문제는 HandlerMethodArgumentResolver는 컨트롤러의 파라미터가 Map 형식이면 동작하지 않는다.

스프링 3.1에서 HandlerMethodArgumentResolver를 이용하여 그러한 기능을 만들더라도, 컨트롤러의 파라미터가 Map 형식이면 우리가 설정한 클래스가 아닌, 스프링에서 기본적으로 설정된 ArgumentResolver를 거치게 된다.

항상 그렇게 동작하는것은 아니고, 스프링의 <mvc:annotation-driven/>을 선언하게 되면 위에서 이야기한것처럼 동작하게 된다. 따라서 <mvc:annotation-driven/>을 선언하려면 Map을 그대로 사용할 수 없고, 선언하지 않으면 Map을 그대로 사용할 수 있다. 그렇지만 앞으로 작성할 내용중에는 <mvc:annotation-driven/>을 선언해야 하는 경우가 있기때문에, 여기서는 Map을 대신할 CommandMap을 작성한다.

first 프로젝트의 common 패키지 밑에 common 패키지를 만들고, 다음을 작성하자.

CommandMap.java

```
1  package first.common.common;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Map.Entry;
6  import java.util.Set;
7
8  public class CommandMap {
9      Map<String,Object> map = new HashMap<String,Object>();
10
11     public Object get(String key){
12         return map.get(key);
13     }
14
15     public void put(String key, Object value){
16         map.put(key, value);
17     }
18
19     public Object remove(String key){
20         return map.remove(key);
21     }
22
23     public boolean containsKey(String key){
24         return map.containsKey(key);
25     }
26
27     public boolean containsValue(Object value){
28         return map.containsValue(value);
29     }
30
31     public void clear(){
32         map.clear();
33     }
34
35     public Set<Entry<String, Object> > entrySet(){
36         return map.entrySet();
37     }
38 }
```

```

37     }
38
39     public Set<String> keySet(){
40         return map.keySet();
41     }
42
43     public boolean isEmpty(){
44         return map.isEmpty();
45     }
46
47     public void putAll(Map<? extends String, ?extends Object> m){
48         map.putAll(m);
49     }
50
51     public Map<String,Object> getMap(){
52         return map;
53     }
54 }

```

클래스는 별다른 부분은 없다. 내부적으로 Map을 하나 생성하고, 그 맵에 모든 데이터를 담는 역할을 한다.

여기서 중요한 점은 절대로 Map을 상속받으면 안된다.

Map을 상속받게 되면, 우리가 작성할 ArgumentResolver를 거치지 않게 되니 주의하자.

여러가지 메서드들이 보이는데, 거의 대부분은 map의 기본 기능을 호출하기 위한것이다.

그리고 다른곳에서 이 CommandMap을 map과 똑같이 사용할 수 있도록 getMap 메서드를 추가했다.

2). HandlerMethodArgumentResolver 작성

first > common 패키지 밑에 resolver 패키지를 작성 후 다음을 작성하자.

CustomMapArgumentResolver.java

```

1 package first.common.resolver;
2
3 import java.util.Enumeration;
4
5 import javax.servlet.http.HttpServletRequest;
6
7 import org.springframework.core.MethodParameter;
8 import org.springframework.web.bind.support.WebDataBinderFactory;

```

```

9 import org.springframework.web.context.request.NativeWebRequest;
10import org.springframework.web.method.support.HandlerMethodArgumentResolver;
11import org.springframework.web.method.support.ModelAndViewContainer;
12
13import first.common.common.CommandMap;
14
15public class CustomMapArgumentResolver implements HandlerMethodArgumentResolver{
16    @Override
17    public boolean supportsParameter(MethodParameter parameter) {
18        return CommandMap.class.isAssignableFrom(parameter.getParameterType());
19    }
20
21    @Override
22    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer
23mavContainer, NativeWebRequest webRequest, WebDataBinderFactory binderFactory)
24throws Exception {
25        CommandMap commandMap = new CommandMap();
26
27        HttpServletRequest request = (HttpServletRequest) webRequest.getNativeRequest();
28        Enumeration<?> enumeration = request.getParameterNames();
29
30        String key = null;
31        String[] values = null;
32        while(enumeration.hasMoreElements()){
33            key = (String) enumeration.nextElement();
34            values = request.getParameterValues(key);
35            if(values != null){
36                commandMap.put(key, (values.length > 1) ? values:values[0] );
37            }
38        }
39        return commandMap;
40    }
41}

```

이제 하나씩 살펴보자.

HandlerMethodArgumentResolver 인터페이스를 상속하여 두가지 메서드 supportsParameter 메서드와 resolveArgument 메서드를 반드시 구현한다.

이름에서 알수 있듯이 supportsParameter 메서드는 Resolver가 적용 가능한지 검사하는 역할을 하고, resolveArgument 메서드는 받은 파라미터와 기타 정보를 꺼내서 객체에 담아서 객체를 반

환한다.

supportparameter 메서드는 컨트롤러의 파라미터가 CommandMap 클래스인지 검사하도록 하였다.

이를 위해서 추후 Controller의 Map<String, Object> 형식을 CommandMap이라고 변경할 것이다.

그 다음 중요한것이 resolverArgument 메서드다.

먼저, 25번 줄에서 CommandMap 객체를 생성하였다.

그 다음으로, 36번 줄에서 request에 담겨있는 모든 키(key)와 값(value)을 commandMap에 저장하였다.

30번 줄부터 34번 줄까지는 request에 있는 값을 iterator를 이용하여 하나씩 가져오는 로직이다.

마지막으로 39번 줄에서 모든 파라미터가 담겨있는 commandMap을 반환하였다.

3). CustomMapArgumentResolver 등록

이제 CustomMapArgumentResolver를 등록하자.

CustomMapArgumentResolver는 root context 영역에 등록이 되어야 한다. 따라서 action-servlet.xml에 등록해야 한다.

action-servlet.xml에 다음과 같이 등록하자.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns:context="http://www.springframework.org/schema/context"
3     xmlns:p="http://www.springframework.org/schema/p"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xmlns="http://www.springframework.org/schema/beans"
6     xmlns:mvc="http://www.springframework.org/schema/mvc"
7     xsi:schemaLocation="http://www.springframework.org/schema/beans
8     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
9     http://www.springframework.org/schema/context
10    http://www.springframework.org/schema/context/spring-context-3.0.xsd
11    http://www.springframework.org/schema/mvc
12    http://www.springframework.org/schema/mvc/spring-mvc.xsd">
13
14     <context:component-scan base-package="first"> </context:component-scan>
15
16     <mvc:annotation-driven>
17         <mvc:argument-resolvers>
18             <bean
19 class="first.common.resolver.CustomMapArgumentResolver"> </bean>
20         </mvc:argument-resolvers>
```

```

21 </mvc:annotation-driven>
22
23 <mvc:interceptors>
24     <mvc:interceptor>
25         <mvc:mapping path="/**"/>
26         <bean                                id="loggerInterceptor"
27class="first.common.logger.LoggerInterceptor"> </bean>
28     </mvc:interceptor>
29 </mvc:interceptors>
30
31 <bean
32class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>
33
34 <bean class="org.springframework.web.servlet.view.BeanNameViewResolver" p:order="0"
35/>
36 <bean                                id="jsonView"
    class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
    <bean
        class="org.springframework.web.servlet.view.UrlBasedViewResolver" p:order="1"
        p:viewClass="org.springframework.web.servlet.view.JstlView"
        p:prefix="/WEB-INF/jsp/" p:suffix=".jsp">
    </bean>
</beans>

```

위의 action-servlet.xml의 기존 글에 CustomMapArgumentResolver를 등록한 내용이다. 실제로 등록을 한 부분은 16~21번째 줄이다.

<mvc:"argument-resolvers"> 태그를 이용하여 우리가 만든 CustomMapArgumentResolver의 빈(bean)을 수동으로 등록했다.

4). Controller의 수정 및 테스트

이제 위에서 작성한 것들이 정확히 동작하는지 확인해 보자.

Controller에 다음을 추가한다.

```

1 @RequestMapping(value="/sample/testMapArgumentResolver.do")
2 public ModelAndView testMapArgumentResolver(CommandMap commandMap) throws
3 Exception{
4     ModelAndView mv = new ModelAndView("");
5

```

```

6    if(commandMap.isEmpty() == false){
7        Iterator<Entry<String,Object> > iterator =
8                                commandMap.getMap().entrySet().iterator();
9        Entry<String,Object> entry = null;
10       while(iterator.hasNext()){
11           entry = iterator.next();
12           log.debug("key : "+entry.getKey()+" , value : "+entry.getValue());
13       }
14   }
       return mv;
   }

```

좀 복잡해보일수도 있는데, 간단한 내용이다.

먼저 확인해야 할 것은 메소드의 해더인

public ModelAndView testMapArgumentResolver(CommandMap commandMap) throws Exception{ 부분이다.

앞서 Controller에 public ModelAndView openBoardList(Map<String,Object> commandMap) throws Exception{ 메소드를 작성했었다.

여기서 Map<String, Object>가 방금 만든 CommandMap으로 바뀌었다.

그 후, commandMap에 있는 모든 파라미터를 iterator를 이용하여 출력하였다.

이제 서버를 실행시키고 테스트를 해보자.

주소창에 localhost:8080/**first/sample/testMapArgumentResolver.do?aaa=temp** 를 입력해보자.

방금 위에서 만든 컨트롤러에 get방식을 이용하여 aaa라는 키로 temp라는 값을 추가하였다.

이클립스의 콘솔창을 확인하면 다음과 같은 결과를 볼 수 있다.



key : aaa, value : temp라는 로그를 보자.

위에서 우리가 get으로 전송한 aaa라는 키와 temp라는 값이다.

위와 같은 결과가 나오면 정상적으로 CustomMapArgumentResolver가 등록된 것이다.

이번에는 두개의 키와 값을 전송해보자.

localhost:8080/**first/sample/testMapArgumentResolver.do?aaa=value1&bbb=value2** 라고 입력해보자.




정상적으로 aaa와 bbb에 해당하는 값 value1과 value2가 commandMap에 담겨져서 출력됨을 알 수 있다.

사용자가 게시글을 작성하면, 그 데이터는 request에 담겨서 서버로 전송이 되고, 서버에서는 그 데이터를 DB에 입력 및 수정을 하게 됩니다.

만약 MapArgumentResolver를 등록하지 않았다면, 컨트롤러에서 request.getParameter 메서드 등을 이용하여 하나씩 파라미터 값을 꺼내야 하기 때문에 상당히 번거롭고 코드가 길어지게 됩니다.

사실 Annotation을 이용하여 그러한 과정을 생략할 수도 있지만, 값을 수정하거나 추가하기 위해서 MapArgumentResolver를 등록했습니다.

 first3.zip

(12) 게시판 상세조회 및 글 등록 기능 만들기

1. 스타일 및 공통기능 작성

1). 스타일

간단히 스타일을 추가해 봅니다.

src/main/webapp 폴더 밑의 css 폴더밑에 **ui.css** 파일을 생성하고 다음의 내용을 작성한다.

```
@CHARSET "UTF-8";

a:link, a:visited {text-decoration: none; color: #656565;}
1
2
3 .board_list {width:100%;border-top:2px solid #252525;border-bottom:1px solid #ccc}
4 .board_list thead th:first-child{background-image:none}
5 .board_list thead th {border-bottom:1px solid #ccc;padding:12px 0 13px
6 0;color:#3b3a3a;vertical-align:middle}
7 .board_list tbody td {border-top:1px solid #ccc;padding:10px 0;text-align:center;vertical-
8 align:middle}
9 .board_list tbody tr:first-child td {border:none}
10 .board_list tbody td.title {text-align:left; padding-left:20px}
11 .board_list tbody td a {display:inline-block}
12
13 .board_view {width:50%;border-top:2px solid #252525;border-bottom:1px solid #ccc}
14 .board_view tbody th {text-align:left;background:#f7f7f7;color:#3b3a3a}
15 .board_view tbody th.list_tit {font-size:13px;color:#000;letter-spacing:0.1px}
16 .board_view tbody .no_line_b th, .board_view tbody .no_line_b td {border-bottom:none}
17 .board_view tbody th, .board_view tbody td {padding:15px 0 16px 16px;border-bottom:1px
18 solid #ccc}
19 .board_view tbody td.view_text {border-top:1px solid #ccc; border-bottom:1px solid
20 #ccc;padding:45px 18px 45px 18px}
21 .board_view tbody th.th_file {padding:0 0 0 15px; vertical-align:middle}
22
23 .wdp_90 {width:90%}
24
25 .btn {border-radius:3px;padding:5px 11px;color:#fff !important; display:inline-block;
26 background-color:#6b9ab8; border:1px solid #56819d;vertical-align:middle}
```

2). 자바스크립트 공통함수

가장 자주 사용되는 두가지 기능을 만든다.

첫번째는 널(null) 체크하는 함수고, 두번째는 submit 기능을 하는 함수를 만든다.

보통 전송기능인 submit은 form과 <input type="submit"> 을 많이 사용한다.

그렇지만 이는 파라미터의 동적 추가나 공통적인 파라미터 추가, 아무것도 없을때의 화면이동이 불편한 경우가 많다. 따라서 숨겨둔 form을 하나 만들어놓고, 그 폼을 이용하여 페이지의 이동 및 입력값 전송을 하려고 한다.

src/main/webapp 밑의 **js 폴더에 common.js**를 만들고 다음의 내용을 작성하자.

```
1 function gfn_isNull(str) {
2     if (str == null) return true;
3     if (str == "NaN") return true;
4     if (new String(str).valueOf() == "undefined") return true;
5     var chkStr = new String(str);
6     if( chkStr.valueOf() == "undefined" ) return true;
7     if (chkStr == null) return true;
8     if (chkStr.toString().length == 0 ) return true;
9     return false;
10}
11
12function ComSubmit(opt_formId) {
13    this.formId = gfn_isNull(opt_formId) == true ? "commonForm" : opt_formId;
14    this.url = "";
15
16    if(this.formId == "commonForm"){
17        $("#commonForm")[0].reset();
18    }
19
20    this.setUrl = function setUrl(url){
21        this.url = url;
22    };
23
24    this.addParam = function addParam(key, value){
25        $("#"+this.formId).append("<input type='hidden' name='"+key+"' id='"+key+"'
26value='"+value+"' >");
27    };
28
29    this.submit = function submit(){
30        var frm = $("#"+this.formId)[0];
31        frm.action = this.url;
```

```

32     frm.method = "post";
33     frm.submit();
34 };
    }

```

위의 내용은 jQuery를 이용하여 작성하였다.

3). 공통으로 포함될 헤더(header) 및 바디(body) 파일 작성

위에서 작성된 스타일 시트 및 자바스크립트는 어떤 페이지에서도 공통적으로 사용된다.

이를 각 화면마다 똑같이 추가하는것은 비효율적이기 때문에, 하나의 파일에 모든 리소스를 선언하고 그 파일을 include 하게 되면, 공통적으로 쓰이는 모든 기능을 쉽게 관리할 수 있다.

3-1). include-header.jspf 파일 작성

먼저 화면 상단에 포함이 되어야 할 리소스를 선언할 파일이다.

여기에는 화면의 메타정보, 스타일 시트 및 화면 호출이 완료되기 전에 가져와야 하는 스크립트 등이 선언이 된다.

WEB-INF 폴더 밑의 include 폴더 밑에 include-header.jspf 파일을 만들고 다음의 내용을 작성한다.

```

1 <%@ page pageEncoding="utf-8"%>
2
3 <meta charset="utf-8">
4 <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
5 <title>first</title>
6
7 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
8 <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
9
10<link rel="stylesheet" type="text/css" href="<c:url value='/css/ui.css'/>" />
11
12<!-- jQuery -->
13<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
14<script src="<c:url value='/js/common.js'/>" charset="utf-8"></script>

```

jspf 파일은 web.xml을 이용하여 모든 페이지에 자동으로 포함되도록 할 수 있는데 여기서는 그 기능은 사용하지 않고, jsp 페이지에서 include를 하는 방식으로 진행할 것이다. 그렇기 때문에 꼭 jspf 로 만들 필요는 없고, jsp 파일로 만들어도 무관하다.

3-2). include-body.jspf 파일 작성

다음은 화면 하단에 포함될 리소스가 선언된 파일이다.

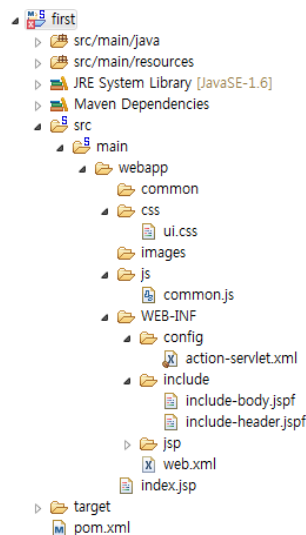
여기에는 화면의 하단에 공통적으로 사용할 태그나 조금 늦게 호출될 자바스크립트 등이 포함된다. 위에서 작성한 common.js에 있는 submit 기능을 위한 숨김 폼 하나만 넣어둔다.

만드는 방법은 include-header.jspf 와 동일하다.

```
1      <%@ page pageEncoding="utf-8"%>
2      <form id="commonForm" name="commonForm"> </form>
```

여기까지 작성을 완료했다면, 기본적인 준비는 되었다.

여기까지 작성이 된 파일구조는 다음과 같다.



2. 등록 페이지

이제 본격적인 게시판 등록 페이지를 만들어 보자.

1). 화면 구성

먼저 WEB-INF/jsp 폴더 아래에 **boardWrite.jsp** 페이지를 만들고, 다음과 같이 작성한다.

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2 pageEncoding="UTF-8"%>
3 <!DOCTYPE html>
4 <html lang="ko">
5 <head>
6 <%@ include file="/WEB-INF/include/include-header.jspf" %>
7 </head>
8 <body>
```

```

9    <form id="frm">
10    <table class="board_view">
11        <colgroup>
12            <col width="15%">
13            <col width="*"/>
14        </colgroup>
15        <caption>게시글 작성</caption>
16        <tbody>
17            <tr>
18                <th scope="row">제목</th>
19                <td><input          type="text"          id="TITLE"          name="TITLE"
20class="wdp_90"></input></td>
21            </tr>
22            <tr>
23                <td colspan="2" class="view_text">
24                    <textarea    rows="20"    cols="100"    title="내용"    id="CONTENTS"
25name="CONTENTS"></textarea>
26                </td>
27            </tr>
28        </tbody>
29    </table>
30
31    <a href="#this" class="btn" id="write" >작성하기</a>
32    <a href="#this" class="btn" id="list" >목록으로</a>
33 </form>
34
35 <%@ include file="/WEB-INF/include/include-body.jspf" %>
36 <script type="text/javascript">
37     $(document).ready(function(){
38
39     });
    </script>
</body>
</html>

```

웹표준과 접근성에 맞춘 HTML 태그는 아니고, 화면을 단순하게 보여주려는 목적임을 참고한다.

먼저 6번, 35번줄에 위에서 작성한 include-header.jspf, include-body.jspf 파일이 include가 되어있는 것을 볼 수 있다.

앞으로 어떠한 화면을 만들더라도, <body> 태그 안쪽의 내용만 바뀌고, 나머지는 항상 똑같이 작성될 것이다.

그 다음으로는 이 화면이 제대로 만들어졌는지 확인하자.

SampleController.java 에서 글쓰기 액션을 호출하는 매핑을 등록하자.

```
1 @RequestMapping(value="/sample/openBoardWrite.do")
2 public ModelAndView openBoardWrite(CommandMap commandMap) throws Exception{
3     ModelAndView mv = new ModelAndView("/sample/boardWrite");
4
5     return mv;
6 }
```

이제, 화면이 제대로 만들어졌는지 확인하기 위해 서버를 실행시키고 브라우저에서 위 액션 호출에 대한 url을 입력해 보자.

<http://localhost:8080/first/sample/openBoardWrite.do>



위와 같은 화면이 나오면 정상적으로 화면 구성은 된 것이다.

이제 글 작성 및 목록으로 가는 기능을 구현해보자.

2). 글 작성 및 목록으로 이동 기능

2.1). '목록으로' 버튼 기능

먼저 "목록으로" 버튼 기능을 먼저 작성하도록 하자.

다음의 코드를 작성하자.

```
1 <script type="text/javascript">
2     $(document).ready(function(){
```

```

3      $("#list").on("click", function(e){
4          e.preventDefault();
5          fn_openBoardList();
6      });
7  });
8
9  function fn_openBoardList(){
10     var comSubmit = new ComSubmit();
11     comSubmit.setUrl("<c:url value='/sample/openBoardList.do' />");
12     comSubmit.submit();
13 }
14</script>

```

위에서 작성한 boardWrite.jsp 파일의 36번 줄부터 작성하면 된다.

먼저, var comSubmit = new ComSubmit(); 부분은 자바스크립트 객체를 생성하는 부분이다.

위에서 common.js 에 submit 기능을 하는 ComSubmit 함수를 만들었는데, 그것이 ComSubmit 객체이다.

그 다음으로 ComSubmit 객체 내의 setUrl 함수를 이용하여 호출하고 싶은 주소를 입력하도록 하였다. 여기서 <c:url> 태그는 JSTL 태그로, 이 태그를 이용하여 ContextPath를 자동으로 붙이도록 하였다.

만약 JSTL을 이용하지 않는다면, comSubmit.setUrl("/first/sample/openBoardList.do"); 라고 작성하면 된다. 그렇지만 <c:url> 태그를 이용하기를 권장한다.

마지막으로 ComSubmit 객체의 submit 함수를 이용하여 전송(submit)을 처리하였다.

많은 사람들이 폼(form)을 이용하여 전송을 할 때는 다음과 같이 폼을 만든다.

```
<form id="frm" name="frm" method="post" action="/first/sample/openBoardList.do"></form>
```

그 후, <input type="submit" value="전송"/>와 같은 태그를 이용하여 전송(submit)을 한다.

그렇지만 이 경우, 폼을 만들 필요가 없는 부분에서도 폼을 만들거나, 동일한 내용을 반복해서 작성해야 하는 불편함이 있다. 그래서 위와 같이 submit 기능을 하는 객체를 만들어서 사용한다.

지금 여기서는 오히려 더 불편하게 생각될 수도 있는데, 프로젝트를 진행하다 보면 이렇게 하는 방식의 장점을 발견할 수 있을 것이다.

이렇게 작성을 하고 버튼을 눌렀을 때, 목록화면으로 이동을 하면 "목록으로" 기능은 완료된 것이다.

2.2). '작성하기' 버튼 기능

작성하기 기능은 목록으로 이동하는 것보다 할 일이 조금 더 많다.

자바 스크립트에서 서버로 데이터를 넘겨주고, 서버에서는 그 결과를 DB에 저장한 후, 다시 게시

글 목록으로 이동시켜야 한다.

먼저 스크립트부터 작성해보자.

```
1 $(document).ready(function(){
2     $("#list").on("click", function(e){ //목록으로 버튼
3         e.preventDefault();
4         fn_openBoardList();
5     });
6
7     $("#write").on("click", function(e){ //작성하기 버튼
8         e.preventDefault();
9         fn_insertBoard();
10    });
11 });
12
13 function fn_openBoardList(){
14     var comSubmit = new ComSubmit();
15     comSubmit.setUrl("<c:url value='/sample/openBoardList.do' />");
16     comSubmit.submit();
17 }
18
19 function fn_insertBoard(){
20     var comSubmit = new ComSubmit("frm");
21     comSubmit.setUrl("<c:url value='/sample/insertBoard.do' />");
22     comSubmit.submit();
23 }
```

"목록으로" 버튼과 비슷하다.

차이점은 ComSubmit 객체를 생성할 때 form의 아이디인 frm을 인자값으로 넘겨주었다.

ComSubmit 객체는 객체가 생성될 때, 폼의 아이디가 인자값으로 들어오면 그 폼의 파라미터들을 전송하고, 파라미터가 없으면 숨겨둔 폼을 이용하여 데이터를 전송하도록 구현하였다.

따라서 전송할 데이터가 있는 frm이라는 id를 가진 form을 이용하도록 id를 넘겨주었다.

다음으로는 서버의 Controller, Service, DAO, 쿼리를 작성할 차례다.

insert는 간단하기 때문에 각 파일에 해당하는 소스만 추가 작성한다.

SampleController.java

```
1 @RequestMapping(value="/sample/insertBoard.do")
```

```

2 public ModelAndView insertBoard(CommandMap commandMap) throws Exception{
3     ModelAndView mv = new ModelAndView("redirect:/sample/openBoardList.do");
4
5     sampleService.insertBoard(commandMap.getMap());
6
7     return mv;
8 }

```

SampleService.java

```

1 public interface SampleService {
2
3     List<Map<String, Object>> selectBoardList(Map<String, Object> map) throws Exception;
4
5     void insertBoard(Map<String, Object> map) throws Exception;
6
7 }

```

SampleServiceImpl.java

```

1 @Override
2 public void insertBoard(Map<String, Object> map) throws Exception {
3     sampleDAO.insertBoard(map);
4 }

```

SampleDAO.java

```

1 public void insertBoard(Map<String, Object> map) throws Exception{
2     insert("sample.insertBoard", map);
3 }

```

Sample_SQL.xml

```

1     <insert id="insertBoard" parameterType="hashmap">
2         <![CDATA[
3             INSERT INTO TB_BOARD
4             (IDX, TITLE, CONTENTS, HIT_CNT, DEL_GB, CREA_DTM, CREA_ID)
5             VALUES (SEQ_TB_BOARD_IDX.NEXTVAL, #{TITLE}, #{CONTENTS},
6                 0, 'N', SYSDATE, 'Admin')
7         ]]>
8     </insert>

```

위에 작성된 쿼리는 지난번 게시판 목록 글에서 사용된 쿼리와 거의 유사하지만, 제목과 내용이

들어갈 부분에 #{TITLE}, #{CONTENTS}라고 되어있는 것을 볼 수 있다.

MyBatis에서는 변수 대입을 #{변수명}으로 사용한다.

여기서 사용된 TITLE, CONTENTS라는 변수명은 위의 HTML 파일에서 제목과 내용을 입력받았던, `<input type="text" id="TITLE" name="TITLE" class="wdp_90"></input>` 와 `<textarea rows="20" cols="100" title="내용" id="CONTENTS" name="CONTENTS">` 태그의 name과 같다는 것을 기억하자.

JSP 내의 Tag에 입력된 값은 그 태그의 name을 키(key)로 데이터가 서버에 전송이 된다.

그럼 서버를 실행시켜서 제목과 내용을 입력하고 저장하기 버튼을 눌러서 게시글이 전송되는지 확인하자.

<http://localhost:8080/first/sample/openBoardWrite.do>



The screenshot shows a web browser window with the URL `localhost:8080/first/sample/openBoardWrite.do`. The page title is '게시글 작성' (Write Post). It contains a form with a '제목' (Title) input field, a '게시글 내용' (Post Content) text area, and two buttons at the bottom: '저장하기' (Save) and '목록으로' (Back to List).

먼저 게시글의 제목과 내용에 각각 입력하고, 저장하기를 누르면 다음과 같이 목록으로 이동하고, 방금 작성한 내용이 있는 것을 볼 수 있다.



The screenshot shows a web browser window with the URL `localhost:8080/first/sample/openBoardList.do`. The page title is '게시판 목록' (Board List). It displays a table with the following data:

글번호	제목	조회수	작성일
2	게시글 제목	0	2015-06-03 14:34:43.0
1	제목	0	2015-04-30 15:15:49.0

마지막으로 여기까지의 과정이 로그로 출력되었는지 콘솔창에서 확인하자.

```

2015-06-03 14:34:32.509 DEBUG [first.common.logger.LoggerInterceptor] ===== START =====
2015-06-03 14:34:32.510 DEBUG [first.common.logger.LoggerInterceptor] Request URI : /first/sample/openBoardWrite.do ===== END =====
2015-06-03 14:34:43.093 DEBUG [first.common.logger.LoggerInterceptor] ===== START =====
2015-06-03 14:34:43.094 DEBUG [first.common.logger.LoggerInterceptor] Request URI : /first/sample/insertBoard.do ===== END =====
2015-06-03 14:34:43.105 INFO [sample] : INSERT INTO TB_BOARD
(
  ID,
  TITLE,
  CONTENT,
  HIT_CNT,
  DEL_YN,
  CRE_A_DTM,
  CRE_ID
)
VALUES
(
  SEQ_TB_BOARD_IDX.NEXTVAL,
  '게시글 제목',
  '게시글 내용',
  0,
  'N',
  SYSDATE,
  ADMIN
)
2015-06-03 14:34:43.109 DEBUG [first.common.logger.LoggerInterceptor] ===== END =====
2015-06-03 14:34:43.115 DEBUG [first.common.logger.LoggerInterceptor] ===== START =====
2015-06-03 14:34:43.115 DEBUG [first.common.dao.abstractDAO] Request URI : /first/sample/openBoardList.do ===== END =====
2015-06-03 14:34:43.116 INFO [sample] : SELECT
ID,
TITLE,
CONTENT,
HIT_CNT,
CRE_A_DTM
FROM
TB_BOARD
ORDER BY ID DESC
2015-06-03 14:34:43.117 INFO [dbc.resultsettable] =====
2015-06-03 14:34:43.117 INFO [dbc.resultsettable] ID | TITLE | HIT_CNT | CRE_A_DTM |
2015-06-03 14:34:43.118 INFO [dbc.resultsettable] 1 | 게시글 제목 | 0 | 2015-06-03 14:34:43.8 |
2015-06-03 14:34:43.118 INFO [dbc.resultsettable] 2 | 게시글 제목 ID | 2015-06-03 14:34:43.8 |
2015-06-03 14:34:43.119 INFO [dbc.resultsettable] 1 | 제목 | 0 | 2015-06-03 14:34:43.8 |
2015-06-03 14:34:43.119 INFO [dbc.resultsettable] =====
2015-06-03 14:34:43.119 DEBUG [first.common.logger.LoggerInterceptor] ===== END =====

```

이클립스의 로그창에 다음과 같은 로그가 찍혀있을 것이다. 잘 살펴보면 프로그램의 흐름이 보인다. START, END로 구분되어진 것이 하나의 호출이다. 총 3번의 Controller 호출이 일어난 것을 볼 수 있다.

첫번째로는 게시글 작성 페이지를 호출한 openBoardWrite.do가 호출되었다.

그 다음으로 중요한 게시글 작성이다. 먼저 insertBoard.do가 호출되었고, sample.insertBoard라는 id를 가진 쿼리가 실행되었음을 알 수 있다. 또한, 실행된 쿼리는 어떤지도 보여지고 있다.

정상적으로 저장이 완료된 후, openBoardList.do가 호출되고, 게시글 목록을 조회해오는 것을 볼 수 있다.

3. 상세보기 페이지

1). JSP

이제 작성한 게시글을 볼 수 있는 상세보기 페이지를 만들자.

먼저 게시글 목록페이지를 약간 수정해서, 게시글 제목을 클릭할 경우 그에 해당하는 상세페이지를 호출하도록 변경하자.

boardList.jsp 파일을 다음과 같이 수정한다.

```

1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2 pageEncoding="UTF-8"%>
3 <!DOCTYPE html>
4 <html lang="ko">
5 <head>
6 <%@ include file="/WEB-INF/include/include-header.jspf" %>
7 </head>
8 <body>
9 <h2>게시판 목록</h2>
10 <table class="board_list">
11 <colgroup>
12 <col width="10%"/>

```

```

13     <col width="*" />
14     <col width="15%" />
15     <col width="20%" />
16 </colgroup>
17 <thead>
18     <tr>
19         <th scope="col">글번호</th>
20         <th scope="col">제목</th>
21         <th scope="col">조회수</th>
22         <th scope="col">작성일</th>
23     </tr>
24 </thead>
25 <tbody>
26     <c:choose>
27         <c:when test="{fn:length(list) > 0}">
28             <c:forEach items="{list}" var="row">
29                 <tr>
30                     <td>${row.IDX }</td>
31                     <td class="title">
32                         <a href="#this" name="title">${row.TITLE }</a>
33                         <input type="hidden" id="IDX" value="${row.IDX }">
34                     </td>
35                     <td>${row.HIT_CNT }</td>
36                     <td>${row.CREA_DTM }</td>
37                 </tr>
38             </c:forEach>
39         </c:when>
40         <c:otherwise>
41             <tr>
42                 <td colspan="4">조회된 결과가 없습니다.</td>
43             </tr>
44         </c:otherwise>
45     </c:choose>
46 </tbody>
47 </table>
48 <br/>
49 <a href="#this" class="btn" id="write">글쓰기</a>
50
51 <%@ include file="/WEB-INF/include/include-body.jspf" %>

```

```

52 <script type="text/javascript">
53     $(document).ready(function(){
54         $("#write").on("click", function(e){ //글쓰기 버튼
55             e.preventDefault();
56             fn_openBoardWrite();
57         });
58
59         $("a[name='title']").on("click", function(e){ //제목
60             e.preventDefault();
61             fn_openBoardDetail($(this));
62         });
63     });
64
65
66     function fn_openBoardWrite(){
67         var comSubmit = new ComSubmit();
68         comSubmit.setUrl("<c:url value='/sample/openBoardWrite.do' />");
69         comSubmit.submit();
70     }
71
72     function fn_openBoardDetail(obj){
73         var comSubmit = new ComSubmit();
74         comSubmit.setUrl("<c:url value='/sample/openBoardDetail.do' />");
75         comSubmit.addParam("IDX", obj.parent().find("#IDX").val());
76         comSubmit.submit();
77     }
78 </script>
79</body>
</html>

```

일단 위에서 설정한 것처럼 include 파일을 포함시키고, 테이블의 제목에 <a> 태그를 이용하여 링크가 가능하도록 수정하였다. 또한 각 게시글을 조회할 때 필요한 정보인 해당 게시글의 번호를 알기 위해서 <a> 태그와 더불어 <input type="hidden"> 태그를 이용하여 각 글번호를 숨겨두었다.

목록의 제목을 클릭 하였을 때의 이벤트를 바인딩하는것은 59번째 줄부터이다.

각 제목을 클릭하면 fn_openBoardDetail 이라는 함수가 실행된다.

이 함수가 실행될 때 인자값으로 \$(this)가 넘겨지는 것을 볼 수 있다. 이는 jQuery 객체를 뜻하는데, 여기서는 게시글 제목인 <a> 태그를 의미한다.

fn_openBoardDetail 함수에서는 기존과 달라진 점은 addParam 이라는 함수를 사용한 점이다.
위에서 ComSubmit 객체를 만든 이유중에 하나가 폼에 동적으로 값을 추가하는 기능을 편하게
사용하기 위함이라고 했었는데, addParam이라는 함수가 그 역할을 해준다.
addParam은 서버로 전송될 키(key)와 값(value)를 인자값으로 받는다.

obj.parent().find("#IDX").val()은 jQuery를 이용하여 선택된 <a> 태그의 부모 노드 내에서 IDX라는
값을 가진 태그를 찾아서, 그 태그의 값을 가져오도록 한 것이다.

그 다음으로는 상세화면 페이지를 먼저 작성하자.

boardDetail.jsp를 생성하고 다음의 내용을 작성한다.

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2 pageEncoding="UTF-8"%>
3 <!DOCTYPE html>
4 <html lang="ko">
5 <head>
6 <%@ include file="/WEB-INF/include/include-header.jspf" %>
7 </head>
8 <body>
9 <table class="board_view">
10 <colgroup>
11 <col width="15%"/>
12 <col width="35%"/>
13 <col width="15%"/>
14 <col width="35%"/>
15 </colgroup>
16 <caption>게시글 상세</caption>
17 <tbody>
18 <tr>
19 <th scope="row">글 번호</th>
20 <td>${map.TITLE }</td>
21 <th scope="row">조회수</th>
22 <td>${map.HIT_CNT }</td>
23 </tr>
24 <tr>
25 <th scope="row">작성자</th>
26 <td>${map.CREA_ID }</td>
27 <th scope="row">작성시간</th>
```

```

28         <td>${map.CREA_DTM }</td>
29     </tr>
30     <tr>
31         <th scope="row">제목</th>
32         <td colspan="3">${map.TITLE }</td>
33     </tr>
34     <tr>
35         <td colspan="4">${map.CONTENT }</td>
36     </tr>
37 </tbody>
38 </table>
39
40 <a href="#this" class="btn" id="list">목록으로</a>
41 <a href="#this" class="btn" id="update">수정하기</a>
42
43 <%@ include file="/WEB-INF/include/include-body.jspf" %>
44 <script type="text/javascript">
45     $(document).ready(function(){
46         $("#list").on("click", function(e){ //목록으로 버튼
47             e.preventDefault();
48             fn_openBoardList();
49         });
50
51         $("#update").on("click", function(e){
52             e.preventDefault();
53             fn_openBoardUpdate();
54         });
55     });
56
57     function fn_openBoardList(){
58         var comSubmit = new ComSubmit();
59         comSubmit.setUrl("<c:url value='/sample/openBoardList.do' />");
60         comSubmit.submit();
61     }
62
63     function fn_openBoardUpdate(){
64         var idx = "${map.IDX}";
65         var comSubmit = new ComSubmit();
66         comSubmit.setUrl("<c:url value='/sample/openBoardUpdate.do' />");

```



```

67         comSubmit.addParam("IDX", idx);
68         comSubmit.submit();
69     }
70 </script>
71</body>
</html>

```

전체적으로 게시글 작성과 큰 차이가 없지만, 글 상세보기 페이지이기 때문에 글 번호, 조회수, 작성자, 작성시간을 추가로 보여주면서, 편집할 수 없도록 <input> 태그를 사용하지 않았다.

2). Java

SampleController.java 에 소스 추가한다.

```

1@RequestMapping(value="/sample/openBoardDetail.do")
2public ModelAndView openBoardDetail(CommandMap commandMap) throws Exception{
3    ModelAndView mv = new ModelAndView("/sample/boardDetail");
4
5    Map<String,Object> map = sampleService.selectBoardDetail(commandMap.getMap());
6    mv.addObject("map", map);
7
8    return mv;
9}

```

SampleService.java 에 추가한다.

```

1Map<String, Object> selectBoardDetail(Map<String, Object> map) throws Exception;

```

SampleServiceImpl.java 에 추가한다.

```

@Override
1public Map<String, Object> selectBoardDetail(Map<String, Object> map) throws Exception
2{
3    sampleDAO.updateHitCnt(map);
4    Map<String, Object> resultMap = sampleDAO.selectBoardDetail(map);
5    return resultMap;
6}

```

SampleDAO.java 에 추가한다.

```

1public void updateHitCnt(Map<String, Object> map) throws Exception{
2    update("sample.updateHitCnt", map);
3}

```

```

4
5@SuppressWarnings("unchecked")
6public Map<String, Object> selectBoardDetail(Map<String, Object> map) throws Exception{
7    return (Map<String, Object>) selectOne("sample.selectBoardDetail", map);
8}

```

먼저 컨트롤러에서 방금 작성한 boardDetail를 호출할 액션에 대한 매핑이름을 등록하였다.
그리고 게시판 상세 내용은 목록과는 다르게 단 하나의 행(record)만 조회하기 때문에, Map의 형태로 결과값을 받았다.

SampleServiceImpl 에서는 기존과 다르게 2개의 메소드를 호출한다는 것이 **굉장히 중요한점이다**.

게시글 상세를 조회하기 위해서는 다음의 두가지의 동작이 필요하다.

- 1) 해당 게시글의 조회수를 1 증가시킨다.
- 2) 게시글의 상세내용을 조회한다.

즉, 이 두가지 동작은 하나의 트랜잭션으로 처리가 되어야 하는 부분이다.

실제 주소가 등록되는 Controller에는 하나의 URL만 등록이 된다.

앞에서 ServiceImpl 에는 하나의 페이지를 호출할 때 필요한 비즈니스 로직을 묶어서 처리하는 곳이라고 하였다.

여기서는 게시글을 조회하기 위해서 위의 2가지 동작을 수행하는 역할을 하고있다.

DAO는 단순히 DB에 접속하여 데이터를 조회하는 역할만 수행하는 클래스다.

따라서 **DAO에서 2개 이상의 동작을 수행하면 안된다**.

마지막으로 DAO에서는 위에서 설명한 두가지 동작에 대한 쿼리를 각각 구현하고, ServiceImpl 에서 두개의 메소드를 호출하여, updateHitCnt 라는 쿼리와 selectBoardDetail 이라는 쿼리를 각각 수행하는 것을 볼 수 있다.

3). SQL

마지막으로 쿼리를 볼 차례이다.

```

1 <update id="updateHitCnt" parameterType="hashmap">
2   <![CDATA[
3     UPDATE TB_BOARD
4     SET HIT_CNT = NVL(HIT_CNT, 0) + 1
5     WHERE IDX = #{IDX}
6   ]]>
7 </update>
8
9 <select id="selectBoardDetail" parameterType="hashmap" resultType="hashmap">

```

```

10  <![CDATA[
11      SELECT IDX, HIT_CNT, CREA_ID, CREA_DTM, TITLE, CONTENTS
12      FROM TB_BOARD
13      WHERE IDX = #{IDX}
14  ]]>
15</select>

```

이렇게 작성을 완료하였으면 정확히 개발되었는지 확인해보자.
먼저 목록 화면을 호출하자.

글번호	제목	조회수	작성일
2	게시글 제목	2	2015-06-03 14:34:43.0
1	제목	0	2015-04-30 15:13:49.0

현재 게시글 제목에 마우스를 올려놓고, 마우스가 손 모양으로 바뀌면 하단에 해당 글의 링크 주소가 보여주고 있는것도 확인한 후에, 제목을 클릭하자.

글번호	제목	조회수	작성자	작성시간
2	게시글 제목	3	Admin	2015-06-03 14:34:43.0

위와 같이 게시글 상세 화면이 정상적으로 나오는것을 볼 수 있다.

마지막으로 콘솔창의 로그를 확인해보자.

```

015-06-03 15:35:42.189 DEBUG [first.common.Logger.Logger$TraceLogger] Request URI : /first/sample/openBoardDetail.do
015-06-03 15:35:42.189 DEBUG [first.common.dao.AbstractDAO] QueryId : sample.updateHitCnt
015-06-03 15:35:42.190 INFO SQL : UPDATE TB_BOARD
SET
HIT_CNT = NVL(HIT_CNT, 0) + 1
WHERE
ID = '2'
015-06-03 15:35:42.199 DEBUG [first.common.dao.AbstractDAO] QueryId : sample.selectBoardDetail
015-06-03 15:35:42.199 INFO SQL :
SELECT
ID,
HIT_CNT,
CREA_ID,
CREA_DTM,
TITLE,
CONTENTS
FROM
TB_BOARD
WHERE
ID = '2'
015-06-03 15:35:42.201 INFO [jdbc:resultsettable] ID HIT_CNT CREA_ID CREA_DTM TITLE CONTENTS
015-06-03 15:35:42.202 INFO [jdbc:resultsettable] 2 1 Admin 2015-06-03 14:34:43.9 게시물 목록 (게시글 내용)
015-06-03 15:35:42.203 INFO [jdbc:resultsettable]
015-06-03 15:35:42.203 INFO [first.common.Logger.Logger$TraceLogger]

```

로그에서 우리가 호출했던 두가지 쿼리, updateHitCnt와 selectBoardDetail이 각각 실행되었음을 알 수 있다.

여기서 잘 보면 쿼리는 2개가 실행되었지만 단 하나의 START, END를 볼 수 있다.

이렇게 하나의 트랜잭션에서 2개 이상의 쿼리를 수행시켜서 데이터의 무결성을 유지할 수 있다.

4. 수정 페이지

1). JSP

수정 페이지는 앞의 상세보기 페이지와 다르게 거의 없다.

수정 페이지는 제목과 내용을 입력받을 수 있도록 변경하고, 삭제하기 버튼이 추가되었다.

```

1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2 pageEncoding="UTF-8"%>
3 <!DOCTYPE html>
4 <html lang="ko">
5 <head>
6 <%@ include file="/WEB-INF/include/include-header.jspf" %>
7 </head>
8 <body>
9 <form id="frm">
10 <table class="board_view">
11 <colgroup>
12 <col width="15%"/>
13 <col width="35%"/>
14 <col width="15%"/>
15 <col width="35%"/>
16 </colgroup>
17 <caption>게시글 상세</caption>
18 <tbody>
19 <tr>
20 <th scope="row">글 번호</th>
21 <td>

```

```

22         ${map.IDX }
23         <input type="hidden" id="IDX" name="IDX" value="${map.IDX }">
24     </td>
25     <th scope="row">조회수</th>
26     <td>${map.HIT_CNT }</td>
27 </tr>
28 <tr>
29     <th scope="row">작성자</th>
30     <td>${map.CREA_ID }</td>
31     <th scope="row">작성시간</th>
32     <td>${map.CREA_DTM }</td>
33 </tr>
34 <tr>
35     <th scope="row">제목</th>
36     <td colspan="3">
37         <input type="text" id="TITLE" name="TITLE" class="wdp_90"
38value="${map.TITLE }"/>
39     </td>
40 </tr>
41 <tr>
42     <td colspan="4" class="view_text">
43         <textarea rows="20" cols="100" title="내용" id="CONTENTS"
44name="CONTENTS">${map.CONTENTS }</textarea>
45     </td>
46 </tr>
47 </tbody>
48 </table>
49 </form>
50
51 <a href="#this" class="btn" id="list">목록으로</a>
52 <a href="#this" class="btn" id="update">저장하기</a>
53 <a href="#this" class="btn" id="delete">삭제하기</a>
54
55 <%@ include file="/WEB-INF/include/include-body.jspf" %>
56 <script type="text/javascript">
57     $(document).ready(function(){
58         $("#list").on("click", function(e){ //목록으로 버튼
59             e.preventDefault();
60             fn_openBoardList();

```

```

61     });
62
63     $("#update").on("click", function(e){ //저장하기 버튼
64         e.preventDefault();
65         fn_updateBoard();
66     });
67
68     $("#delete").on("click", function(e){ //삭제하기 버튼
69         e.preventDefault();
70         fn_deleteBoard();
71     });
72 });
73
74 function fn_openBoardList(){
75     var comSubmit = new ComSubmit();
76     comSubmit.setUrl("<c:url value='/sample/openBoardList.do' />");
77     comSubmit.submit();
78 }
79
80 function fn_updateBoard(){
81     var comSubmit = new ComSubmit("frm");
82     comSubmit.setUrl("<c:url value='/sample/updateBoard.do' />");
83     comSubmit.submit();
84 }
85
86 function fn_deleteBoard(){
87     var comSubmit = new ComSubmit();
88     comSubmit.setUrl("<c:url value='/sample/deleteBoard.do' />");
89     comSubmit.addParam("IDX", $("#IDX").val());
90     comSubmit.submit();
91
92 }
</script>
</body>
</html>

```

그동안 작성했던 내용과 다르지 않다. 앞에서 게시글 작성 및 상세 페이지에서 했던 내용의 반복이라고 볼 수 있다..

2). Java

SampleController.java

```
1 @RequestMapping(value="/sample/openBoardUpdate.do")
2 public ModelAndView openBoardUpdate(CommandMap commandMap) throws Exception{
3     ModelAndView mv = new ModelAndView("/sample/boardUpdate");
4
5     Map<String,Object> map = sampleService.selectBoardDetail(commandMap.getMap());
6     mv.addObject("map", map);
7
8     return mv;
9 }
10
11 @RequestMapping(value="/sample/updateBoard.do")
12 public ModelAndView updateBoard(CommandMap commandMap) throws Exception{
13     ModelAndView mv = new ModelAndView("redirect:/sample/openBoardDetail.do");
14
15     sampleService.updateBoard(commandMap.getMap());
16
17     mv.addObject("IDX", commandMap.get("IDX"));
18     return mv;
19 }
```

SampleService.java

```
1 void updateBoard(Map<String, Object> map) throws Exception;
```

SampleServiceImpl.java

```
1 @Override
2 public void updateBoard(Map<String, Object> map) throws Exception{
3     sampleDAO.updateBoard(map);
4 }
```

SampleDAO.java

```
1 public void updateBoard(Map<String, Object> map) throws Exception{
2     update("sample.updateBoard", map);
3 }
```

SampleController에서 게시글을 수정하고 나면, 해당 게시글의 상세화면으로 이동하도록 하였다. 따라서, 해당 게시글의 글 번호를 mv.addObject 메서드를 이용하여 다시 전송하도록 하였다.

3). SQL

쿼리 역시 별다른 부분은 없다.

```
1      <update id="updateBoard" parameterType="hashmap">
2          <![CDATA[
3              UPDATE TB_BOARD
4              SET TITLE = #{TITLE},
5                  CONTENTS = #{CONTENTS}
6              WHERE IDX = #{IDX}
7          ]]>
8      </update>
```

이제 수정기능을 확인해보자.

상세페이지에서 수정하기 버튼을 눌러보자.



이와 같이 수정할 수 있는 화면이 나오고, 해당 글의 내용도 정상적으로 조회됨을 알 수 있다. 게시글의 내용을 변경하고 저장하기 버튼을 클릭하자.



정상적으로 제목과 내용이 변경되고, 게시글 상세화면으로 이동한 것을 볼 수 있다.

5. 게시글 삭제

마지막으로 게시글을 삭제하는 기능이다.

게시글 수정하기에서 JSP의 내용은 모두 만들어놨기 때문에, 여기서는 Java와 SQL만 보도록 하겠다.

1). Java

SampleController.java

```
1 @RequestMapping(value="/sample/deleteBoard.do")
2 public ModelAndView deleteBoard(CommandMap commandMap) throws Exception{
3     ModelAndView mv = new ModelAndView("redirect:/sample/openBoardList.do");
4
5     sampleService.deleteBoard(commandMap.getMap());
6
7     return mv;
8 }
```

SampleService.java

```
1 void deleteBoard(Map<String, Object> map) throws Exception;
```

SampleServiceImpl.java

```
1 @Override
2 public void deleteBoard(Map<String, Object> map) throws Exception {
3     sampleDAO.deleteBoard(map);
4 }
```

SampleDAO.java

```
1 public void deleteBoard(Map<String, Object> map) throws Exception{
2     update("sample.deleteBoard", map);
3 }
```

여기서 살펴봐야 할것은 SampleDAO.java에서 update문을 호출하였다는 것이다.

다음의 쿼리에서 볼 수 있겠지만, delete문을 통해서 게시글을 삭제하기보다는 구분값을 변경하여 게시글의 삭제 여부를 변경하려고 한다.

2). SQL

```
1 <update id="deleteBoard" parameterType="hashmap">
```

```

2      <![CDATA[
3          UPDATE TB_BOARD
4          SET DEL_GB = 'Y'
5          WHERE IDX = #{IDX}
6      ]]>
7  </update>

```

쿼리의 id는 delete라고 붙였지만, 실제 실행되는 것은 update인 것을 유의하자.

앞서 게시판 목록글에서 TB_BOARD 테이블에 DEL_GB 컬럼을 만들었다.

이 컬럼은 해당 게시글의 삭제여부를 저장하는 컬럼으로, DEL_GB = 'Y'이면 삭제가 된 글로 처리한다. 만약 해당 게시글을 완전히 삭제시키려고 하면, update문 대신 delete문을 실행시키면 된다.

이제 삭제하기의 결과를 확인할 차례다.

위와 같은 게시글 수정화면에서 삭제하기 버튼을 클릭하자.

글번호	제목	조회수	작성일
1	제목	0	2015-04-30 15:13:49.0

그러면 위와 같이 해당 게시글이 삭제된것을 볼 수 있다.

마지막으로 이클립스의 로그를 확인해 보자.

```
2015-06-03 17:33:41.876 DEBUG [first.common.Logger.LoggerInterceptor] ===== START =====
2015-06-03 17:33:41.876 DEBUG [first.common.Logger.LoggerInterceptor] Request URI : /first/sample/deleteBoard.do
2015-06-03 17:33:41.877 DEBUG [first.common.Logger.LoggerInterceptor] QueryId : sample.deleteBoard
2015-06-03 17:33:41.877 INFO SQL
SET
DEL_ID = '1'
WHERE
ID = '1'
2015-06-03 17:33:41.882 DEBUG [first.common.Logger.LoggerInterceptor] ===== END =====
2015-06-03 17:33:41.887 DEBUG [first.common.Logger.LoggerInterceptor] ===== START =====
2015-06-03 17:33:41.887 DEBUG [first.common.Logger.LoggerInterceptor] Request URI : /first/sample/getBoardList.do
2015-06-03 17:33:41.888 DEBUG [first.common.Logger.LoggerInterceptor] QueryId : sample.getBoardList
2015-06-03 17:33:41.888 INFO SQL
SELECT
ID,
TITLE,
CONTENT,
CREATE_DT
FROM
TB_BOARD
WHERE
DEL_ID = '0'
ORDER BY ID DESC
2015-06-03 17:33:41.890 INFO [org.springframework] [=====]
2015-06-03 17:33:41.890 INFO [org.springframework] [ID TITLE CONTENT]
2015-06-03 17:33:41.891 INFO [org.springframework] [=====]
2015-06-03 17:33:41.891 INFO [org.springframework] [1 标题 内容]
2015-06-03 17:33:41.891 INFO [org.springframework] [=====]
2015-06-03 17:33:41.891 INFO [org.springframework] [=====]
2015-06-03 17:33:41.891 DEBUG [first.common.Logger.LoggerInterceptor] ===== END =====
```

위와 같이 작성한 deleteBoard라는 쿼리가 정상적으로 수행된 후, 게시글의 목록을 조회한 것을 확인할 수 있다.

스프링의 기본적인 내용은 끝났습니다.

실제 프로젝트에서는 당연히 더 복잡하고 내용도 많지만, 기본적인 아키텍처는 게시판의 연장선 일 뿐입니다.



first4.zip