

9 장 상품과 파일업로드

9.1 상품과 상품이미지 테이블

게시물과 다르게 상품은 이미지가 중심이 됩니다. 하나의 상품에는 여러 개의 이미지를 가지는 것이 일반적이기 때문에 일반적인 경우라면 상품과 상품 이미지는 별도의 테이블로 관리됩니다.

상품과 이미지가 하나의 테이블로 관리하는 경우가 가끔은 있는데 이런 경우는 '상품당 고정된 개수의 이미지만 올리게 하는 경우'이거나 조인 처리를 피해서 성능을 높이려고 하는 경우입니다. 이번 예제에서는 순수하게 상품과 해당 상품의 이미지를 별도의 테이블로 분리해서 처리하도록 합니다.

데이터베이스에서는 tbl_product 테이블을 설계해서 사용합니다. tbl_product 테이블에는 상품의 이름, 가격, 설명, 판매 여부 등을 추가합니다.

```
CREATE TABLE tbl_product (
  pno INT AUTO_INCREMENT PRIMARY KEY, -- 상품 번호(고유식별자)
  pname VARCHAR(200) NOT NULL,        -- 상품 이름
  pdesc VARCHAR(1000) NOT NULL,       -- 상품 설명
  price INT NOT NULL,                 -- 상품 가격
  sale BOOLEAN DEFAULT FALSE,         -- 판매 여부 (false)
  writer VARCHAR(100) NOT NULL,       -- 상품 등록자
  regdate DATETIME DEFAULT CURRENT_TIMESTAMP,
  moddate DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

상품 테이블을 참조하는 상품 이미지 테이블은 다음과 같이 설계합니다.

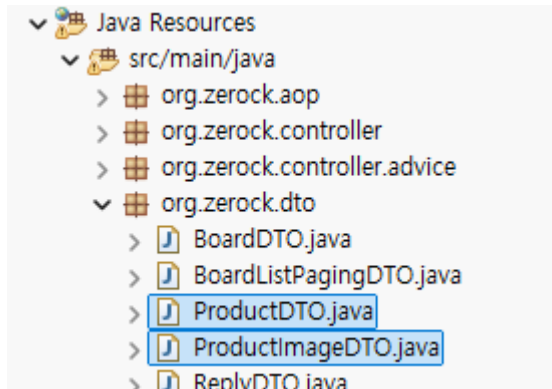
```
CREATE TABLE tbl_product_image (
  ino INT AUTO_INCREMENT PRIMARY KEY, -- 이미지 번호 (고유 식별자)
  pno INT NOT NULL,                  -- 상품 번호 (외래 키)
  filename VARCHAR(300) NOT NULL,   -- 실제 파일명 또는 저장된 경로
  uuid CHAR(36) NOT NULL,           -- 파일명 중복 방지를 위한 UUID
  ord INT DEFAULT 0,                -- 이미지 정렬 순서
  regdate DATETIME DEFAULT CURRENT_TIMESTAMP, -- 등록일
  FOREIGN KEY (pno) REFERENCES tbl_product(pno) ON DELETE CASCADE
);
```

상품 이미지는 주로 상품 번호로 인해서 검색되어서 사용되므로 미리 테이블 생성 단계에서 인덱스를 구성합니다.

```
CREATE INDEX idx_product_image_pno ON tbl_product_image(pno, ord);
```

9.2 DTO와 Mapper 작성

작성된 테이블에 데이터를 넣기 위해서 dto 패키지에 ProductDTO 와 ProductImageDTO 클래스를 구성합니다.



우선 각 DTO 는 테이블의 설계를 기준으로 작성합니다. tbl_product 를 기준으로 ProductDTO 를 작성합니다.

```
package org.zerock.dto;

import java.time.LocalDateTime;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductDTO {

    private Integer pno;

    private String pname;

    private String pdesc;

    private int price;

    private boolean sale;

    private String writer;

    private LocalDateTime regDate;

    private LocalDateTime modDate;

}
```

상품 이미지인 tbl_product_image 테이블을 기준으로 ProductImageDTO 를 작성합니다.

```
package org.zerock.dto;

import java.time.LocalDateTime;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductImageDTO {

    private Integer ino;

    private Integer pno;

    private String fileName;

    private String uuid;

    private int ord;

    private LocalDateTime regDate;
}
```

연관 관계 처리

관계형 데이터베이스에서는 '상품'과 '상품 이미지'는 '일대다' 혹은 '다대일'의 관계로 해석되고 오직 PK(주키)와 FK(외래키)의 관계를 통해서만 규정됩니다.

반면에 객체지향에서는 ProductDTO 에서 여러 개의 ProductImageDTO 객체들을 참조하도록 설계할 수도 있고 반대로 설계할 수도 있기 때문에 처음 설계할 때 주의해서 설계해야 합니다.

'상품'과 '상품 이미지'의 경우 실제 업무에서는 '상품'이 가장 중요한 단위가 됩니다. 엄밀하게 따지면 '상품'을 등록한다는 의미는 '상품 이미지'에 대한 처리가 포함됩니다.

데이터베이스 상에서 '상품 이미지'와 '상품'의 관계는 '댓글'과 '게시글'과 동일하게 보이지만 실제로는 중요한 차이가 있습니다.

- 주체: 게시글과 댓글의 경우 데이터를 생성하는 주체가 동일하지 않습니다. 하지만 상품의 경우에는 상품을 등록하는 작성자가 상품의 이미지도 같이 작성합니다.

- 시간: 게시글과 댓글은 독립적인 생성 시간, 수정 시간을 가지게 됩니다. 반면에 상품의 경우는 등록/수정/삭제 시에 상품과 함께 상품 이미지들 역시 같이 처리됩니다.

이러한 이유로 인해 '상품'과 '상품 이미지'의 경우 ProductDTO 에서 ProductImageDTO 를 관리하는 구조로 만들어지는 것이 적합합니다.

ProductDTO 에 여러 개의 ProductImageDTO 를 처리할 수 있도록 속성을 추가하고 상품의 이미지를 파일명만으로 처리할 수 있도록 addImage()와 같은 메서드를 추가합니다. ProductDTO 객체가 ProductImageDTO 를 컨트롤하게 되면 나중에 데이터를 다룰 때 조금 편리합니다.

```
package org.zerock.dto;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductDTO {

    private Integer pno;

    private String pname;

    private String pdesc;

    private int price;

    private boolean sale;

    private String writer;

    private LocalDateTime regDate;

    private LocalDateTime modDate;

    //상품 이미지들
    private List<ProductImageDTO> imageUrl;

    public void addImage(String uuid, String fileName) {

        if(imageList == null) {
            imageList = new ArrayList<>();
        }

        ProductImageDTO imageDTO = ProductImageDTO.builder()
```

```

        .uuid(uuid)
        .fileName(fileName)
        .pno(this.pno)
        .ord(this.imageList.size())
        .build();

    imageList.add(imageDTO);
}

}

```

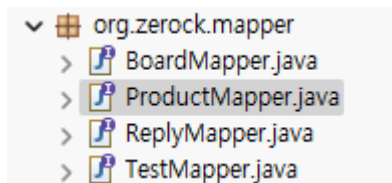
ProductMapper 와 상품 등록

연관 관계에서 살펴보았듯이 실제 업무는 '상품'을 다루는 것으로(이런 업무의 단위를 도메인(domain)이라고 합니다.) Mapper 를 작성할 때에도 ProductMapper 와 ProductImageMapper 와 같이 별도로 설계할 것인지 하나의 Mapper 를 설계할 것인지 결정해야 합니다.

예제에서는 상품을 처리하는 ProductMapper 를 설계해서 내부적으로 ProductImageDTO 를 처리하는 방식으로 구성하겠습니다.

Mapper 선언

mapper 패키지에 ProductMapper 인터페이스를 선언하고 상품과 상품 이미지를 추가하는 기능을 선언합니다. 특히 상품 이미지는 한 번에 여러 개를 추가하므로 ProductDTO 를 파라미터로 활용하고 MyBatis 의 <foreach>를 이용해서 imageList 를 처리합니다.



```

package org.zerock.mapper;

import org.zerock.dto.ProductDTO;

public interface ProductMapper {

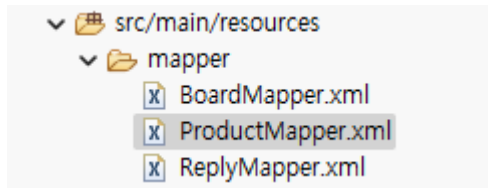
    int insert(ProductDTO productDTO);

    int insertImages( ProductDTO productDTO);

}

```

ProductMapper 인터페이스에 실제 SQL 을 작성하기 위한 ProductMapper.xml 을 추가합니다.



```
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "https://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.zerock.mapper.ProductMapper">

    <insert id="insert">

        <selectKey order="AFTER" keyProperty="pno" resultType="int">
            SELECT LAST_INSERT_ID()
        </selectKey>

        insert into tbl_product ( pname, pdesc, price, sale, writer)
        values ( #{pname}, #{pdesc}, #{price}, true, #{writer} )

    </insert>

    <insert id="insertImages">

        insert into tbl_product_image (pno, fileName, uuid, ord)
        values

        <foreach collection="imageList" item="image" separator=",">

            ( #{pno}, #{image.fileName}, #{image.uuid}, #{image.ord} )

        </foreach>

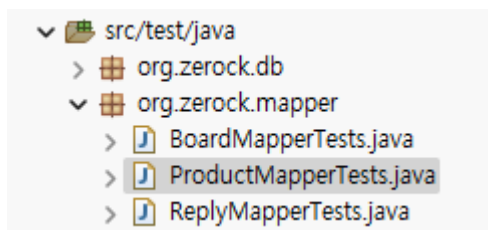
    </insert>

</mapper>
```

ProductDTO 를 추가하는 과정에서는 새롭게 만들어진 상품의 번호를 이용해서 상품 이미지를 추가해야 하기 때문에 LAST_INSERT_ID()를 이용해서 insert 된 상품의 번호를 알 수 있도록 구성합니다.

상품 등록 테스트

테스트 코드는 test 폴더에 ProductMapperTests 를 추가해서 구성합니다.



상품 등록 작업은 두 개의 테이블에 insert 가 여러 번 실행될 수 있으므로 @Transactional 과 테스트 후에 insert 결과를 저장하도록 @Commit 을 추가합니다.

```
package org.zerock.mapper;

import java.util.List;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.annotation.Commit;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.transaction.annotation.Transactional;
import org.zerock.dto.ProductDTO;
import org.zerock.dto.ProductImageDTO;

import lombok.extern.log4j.Log4j2;

@ExtendWith(SpringExtension.class)
@ContextConfiguration("file:src/main/webapp/WEB-INF/spring/root-context.xml")
@Log4j2
public class ProductMapperTests {

    @Autowired
    private ProductMapper productMapper;

    @Transactional
    @Commit
    @Test
    public void testInsert() {

        ProductDTO productDTO = ProductDTO.builder()
            .pname("Product")
            .pdesc("Product Desc")
            .writer("user1")
            .price(4000)
            .build();

        //insert into tbl_product
        productMapper.insert(productDTO);

        productDTO.addImage(UUID.randomUUID().toString(), i+"_test_1.jpg");
        productDTO.addImage(UUID.randomUUID().toString(), i+"_test__2.jpg");

        //insert into tbl_product_image
        productMapper.insertImages(productDTO.getImageList());
    }
}
```

테스트 코드의 경우 tbl_product 테이블에 1 번, tbl_product_image 테이블에 2 번의 insert 가 일어나고 select last_insert_id()가 일어나므로 총 4 번의 SQL 이 실행됩니다.


```

DEBUG ==> Preparing: insert into tbl_product ( pname, pdesc, price, sale, writer) values ( ?, ?, ?, true, ? )
DEBUG ==> Parameters: Product(String), Product Desc(String), 4000(Integer), user1(String)
DEBUG <== Updates: 1
.java:135) DEBUG ==> Preparing: SELECT LAST_INSERT_ID()
.java:135) DEBUG ==> Parameters:
.java:135) DEBUG <== Total: 1
INFO -----
INFO [ProductImageDTO(ino=null, pno=48, fileName=test1.jpg, uuid=f46a0b93-6a9f-49f2-9ef5-e8fa2319582d, ord=0, regDate=null), Product
.java:135) DEBUG ==> Preparing: insert into tbl_product_image (pno, fileName, uuid, ord) values ( ?, ?, ?, ? ), ( ?, ?, ?, ? )
.java:135) DEBUG ==> Parameters: 48(Integer), test1.jpg(String), f46a0b93-6a9f-49f2-9ef5-e8fa2319582d(String), 0(Integer), 48(Integer),
.java:135) DEBUG <== Updates: 2

```

tbl_product 테이블에 결과를 확인합니다.

tbl_product (1r × 8c)									
#	pno	pname	pdesc	price	sale	writer	regdate	moddate	
1	1	Product	Product Desc	4,000	1	user1	2025-05-16 16:39:26	2025-05-16 16:39:26	

tbl_product_image 테이블에서도 결과를 확인합니다.

tbl_product_image (2r × 6c)							
#	ino	pno	filename	uuid	ord	regdate	
1	1	1	test1.jpg	e427d067-3228-11f0-a6e6-745d2200...	0	2025-05-16 16:39:26	
2	2	1	test2.jpg	e427ee5b-3228-11f0-a6e6-745d2200...	1	2025-05-16 16:39:26	

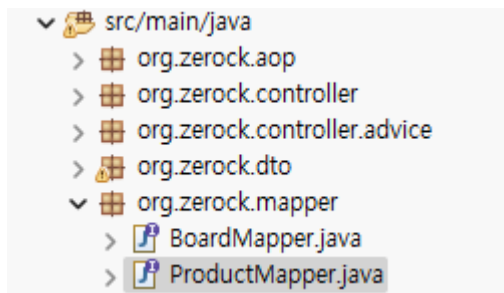
상품 조회와 <resultMap>

상품의 데이터가 tbl_product 와 tbl_product_image 로 나누어져 있으므로 상품 데이터의 조회시에는 두 테이블을 한번에 처리할 수 있는지 따로 처리해야 하는지 고민해야 합니다.

- 한 번에 처리하는 방법: tbl_product 와 tbl_product_image 테이블을 조인처리해서 한 번에 모든 이미지들까지 같이 SQL 로 처리하는 방식입니다. 데이터베이스를 한 번만 호출하기 때문에 성능면에서 유리합니다.
- 나누어 호출하는 방법: 처음에는 tbl_product 테이블만을 쿼리를 통해서 처리하고 나중에 화면에서 Ajax 등을 이용해서 tbl_product_image 테이블의 내용을 처리하는 방식입니다. 쿼리는 쉽게 작성할 수 있지만 전체적인 코드의 양은 많아집니다.

위의 두 가지 방식 중에서 예제에서 사용할 방식은 조인을 이용해서 두 테이블을 조인하고 이를 ProductDTO 로 만들어서 반환하는 방식입니다. 이를 구현할 때 MyBatis 의 <resultMap>을 이용하면 한번에 ProductDTO 와 객체 내부에 있는 ProductImageDTO 들을 쉽게 구성할 수 있습니다.

ProductMapper 에는 ProductDTO 를 반환하는 메서드를 선언합니다.



```
package org.zerock.mapper;

import org.apache.ibatis.annotations.Param;
import org.zerock.dto.ProductDTO;

public interface ProductMapper {

    int insert(ProductDTO productDTO);

    int insertImages( ProductDTO productDTO);

    ProductDTO selectOne( @Param("pno") Integer pno);

}
```

ProductMapper 를 개발하기 전에 특정 상품과 상품 이미지들을 가져오는 쿼리를 작성합니다.

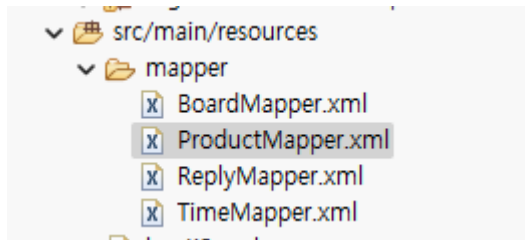
```
SELECT p.pno, pname, pdesc, price, sale, writer, p.regdate, ino, uuid, filename, ord
FROM
    tbl_product p LEFT OUTER JOIN tbl_product_image pimg ON pimg.pno = p.pno
WHERE p.pno = 1
```

쿼리의 결과로 상품과 모든 상품 이미지들이 출력되어야 합니다.

pno	pname	pdesc	price	sale	writer	regdate	ino	uuid	filename	ord
1	Product	Product Desc	4,000	1	user1	2025-05-16 16:39:26	1	e427d067-3228-11f0-a6e6-745d2200...	test1.jpg	0
1	Product	Product Desc	4,000	1	user1	2025-05-16 16:39:26	2	e427ee5b-3228-11f0-a6e6-745d2200...	test2.jpg	1

MyBatis 의 <select>는 한 행(row)의 결과를 특정한 타입으로 매핑해 주기 위해서 <resultMap>이라는 것을 사용합니다. 하지만 개발자가 직접 쿼리의 결과를 처리하려면 <resultMap>을 이용해서 처리합니다.

<resultMap> 내부에는 <collection>을 지정할 수 있는데 이를 이용하면 한 개의 ProductDTO 와 여러 개의 ProductImageDTO 를 구성할 수 있습니다.



```
<resultMap type="ProductDTO" id="selectMap">

  <id property="pno" column="pno"/>
  <result property="pname" column="pname"/>
  <result property="pdesc" column="pdesc"/>
  <result property="price" column="price"/>
  <result property="sale" column="sale"/>
  <result property="writer" column="writer"/>

  <collection property="imageList" ofType="ProductImageDTO">
    <id property="ino" column="ino"/>
    <result property="uuid" column="uuid"/>
    <result property="fileName" column="fileName"/>
    <result property="ord" column="ord"/>
  </collection>
</resultMap>
```

```
<select id="selectOne" resultMap="selectMap">

SELECT p.pno, pname, pdesc, price, sale, writer, p.regdate, ino, uuid, filename, ord
FROM
  tbl_product p LEFT OUTER JOIN tbl_product_image pimg ON pimg.pno = p.pno
WHERE p.pno = #{pno}
order by pimg.ord
</select>
```

작성된 XML 을 보면 <select>에 resultMap 이 지정되어 있고 해당 id 값을 가지는 <resultMap>에는 ProductDTO 를 구성한다는 것을 알 수 있습니다. <resultMap>의 내부에는 <collection>을 이용해서 List<ProductImageDTO>를 구성합니다. 각 객체를 구성하는 기준은 내부에서 사용하고 있는 <id>입니다.

테스트 코드를 이용해서 쿼리의 동작을 확인합니다.

```
@Test
public void testSelectOne() {

  Integer pno = 1;

  ProductDTO productDTO = productMapper.selectOne(pno);

  Log.info(productDTO);
}
```

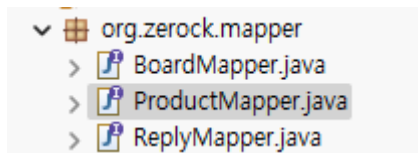
```
productDTO.getImageList().forEach(Log::info);
```

```
}
```

```
MapperTests.java:62) INFO ProductDTO(pno=1, pname=Product, pdesc=Product Desc, price=4000, sale=true, writer=user1, regDate=null,
INFO ProductImageDTO(ino=1, pno=null, fileName=test1.jpg, uuid=e427d067-3228-11f0-a6e6-745d2200eca3, ord=0, regDate=null)
INFO ProductImageDTO(ino=2, pno=null, fileName=test2.jpg, uuid=e427ee5b-3228-11f0-a6e6-745d2200eca3, ord=1, regDate=null)
```

상품 삭제

상품의 삭제는 sale 컬럼의 값을 false 로 처리해주는 것으로 처리합니다.



```
package org.zerock.mapper;

import org.apache.ibatis.annotations.Param;
import org.zerock.dto.ProductDTO;

public interface ProductMapper {

    int insert(ProductDTO productDTO);

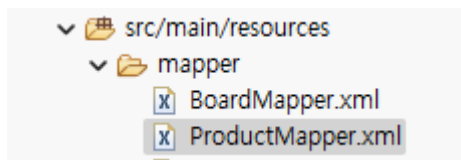
    int insertImages( ProductDTO productDTO);

    ProductDTO selectOne( @Param("pno") Integer pno);

    int deleteOne(@Param("pno") Integer pno);

}
```

ProductMapper.xml 의 내용은 다음과 같습니다.



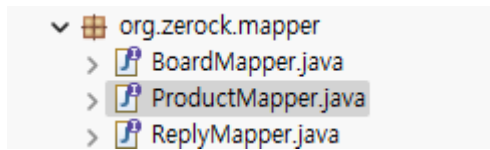
```
<update id="deleteOne">
UPDATE tbl_product SET sale = false WHERE pno = #{pno}
</update>
```

상품 수정

상품의 수정은 크게 상품 자체의 수정과 상품 이미지들의 수정으로 구분할 수 있습니다. 특히 상품 이미지들의 수정은 조금 생각해야 하는 점들이 있기 때문에 이에 대해서 먼저 다뤄보도록 합니다.

엄밀하게 상품 이미지 자체는 수정이라는 개념이 존재하지 않고 삭제된 후에 새로운 이미지 파일 정보가 추가되는 방식이면 충분합니다. 따라서 해당 상품의 모든 상품 이미지들을 삭제한 후에 다시 한번에 모든 상품 이미지들을 추가하는 방식이 더 간단합니다.

상품 이미지들을 한 번에 추가하는 기능은 이미 상품 등록 과정에서 추가되어 있으므로 상품 이미지들을 삭제하는 기능을 추가해 둡니다.



```
package org.zerock.mapper;

import org.apache.ibatis.annotations.Param;
import org.zerock.dto.ProductDTO;

public interface ProductMapper {

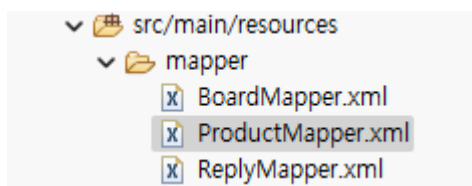
    int insertImages( ProductDTO productDTO);

    ...생략

    int deleteImages(@Param("pno") Integer pno);

}
```

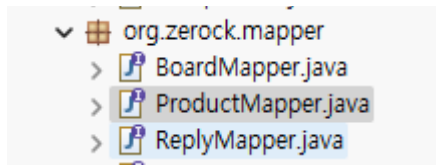
ProductMapper.xml 에 deleteImages()를 구현합니다.



```
<delete id="deleteImages">
DELETE FROM tbl_product_image WHERE pno = #{pno}
</delete>
```

상품의 이미지들을 한번에 추가하는 기능은 이미 상품 등록 과정에서 만든 insertImages()를 이용할 수 있습니다.

상품 자체의 수정은 상품이 이름(pname), 설명(pdsc), 가격(price)을 수정하는 것입니다. 이를 ProductMapper 를 이용해서 선언합니다.



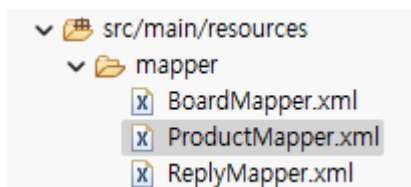
```
package org.zerock.mapper;

import org.apache.ibatis.annotations.Param;
import org.zerock.dto.ProductDTO;

public interface ProductMapper {

    ...생략

    int updateOne(ProductDTO productDTO);
}
```



```
<update id="updateOne">

UPDATE tbl_product

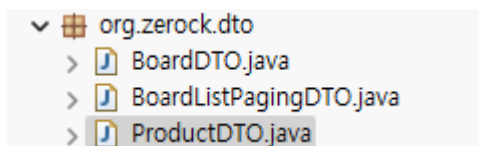
SET  pname = #{pname},
     pdesc = #{pdesc},
     price = #{price}

WHERE pno = #{pno}

</update>
```

ProductDTO 의 수정과 테스트

ProductDTO 에는 상품 이미지를 추가하는 기능이 존재하므로 상품 이미지들을 삭제하는 기능도 추가합니다. 삭제는 imageUrl 를 null 로 만들거나 clear()를 이용해서 내용물을 삭제하도록 구현합니다.



```
package org.zerock.dto;

...생략
```

```

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductDTO {

```

...생략

```

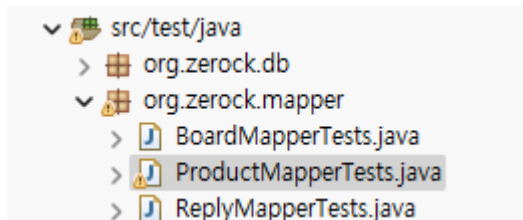
    public void clearImages() {

        imageUrl.clear();
    }

}

```

만들어진 모든 코드를 이용하는 테스트 코드를 작성해 봅니다.



상품의 수정은 1)상품의 기존 이미지를 삭제하고, 2)상품 정보를 수정, 3)수정된 상품 이미지 갱신의 순서로 처리할 수 있습니다. 테스트 코드는 아래와 같이 작성할 수 있습니다.

```

@Transactional
@Commit
@Test
public void testUpdateOne() {

    ProductDTO productDTO = ProductDTO.builder()
        .pno(1)
        .pname("Updated Product")
        .pdesc("update")
        .price(6000)
        .build();

    productDTO.addImage(UUID.randomUUID().toString(), "test3.jpg");
    productDTO.addImage(UUID.randomUUID().toString(), "test4.jpg");
    productDTO.addImage(UUID.randomUUID().toString(), "test5.jpg");

    //기존 이미지 삭제
    productMapper.deleteImages(productDTO.getPno());

    //상품 정보 수정
    productMapper.updateOne(productDTO);

    //상품 이미지 갱신
    productMapper.insertImages(productDTO);

}

```

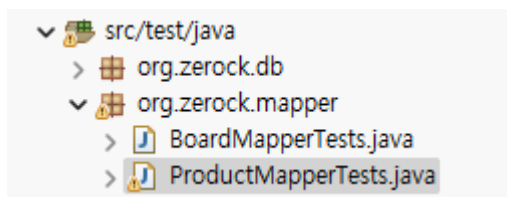
테스트 코드를 실행해 보면 아래와 같이 'DELETE..., UPDATE..., UPDATE...'와 같이 3 번에 걸쳐 실행되는 것을 확인할 수 있습니다.

```
-----
adding stats (total=2/10, idle=2/2, active=0, waiting=0)
==> Preparing: DELETE FROM tbl_product_image WHERE pno = ?
==> Parameters: 1(Integer)
<== Updates: 3
Preparing: UPDATE tbl_product SET pname = ?, pdesc = ?, price = ? WHERE pno = ?
Parameters: Updated Product(String), update(String), 6000(Integer), 1(Integer)
Updates: 1
==> Preparing: insert into tbl_product_image (pno, fileName, uuid, ord) values ( ?, ?, ?, ? ), ( ?, ?, ?, ? ), ( ?, ?, ?, ? )
==> Parameters: 1(Integer), test3.jpg(String), 5261804f-9b40-4d1a-ab90-87c44f3597ac(String), 0(Integer), 1(Integer), test4.jpg(String),
<== Updates: 3
```

상품 목록

상품 목록은 이전의 게시물과 유사하지만 화면에 상품의 이미지가 하나는 출력될 필요가 있습니다. 이 때문에 tbl_product_image 테이블에 ord 라는 컬럼을 부여한 것으로 ord 가 0 인 상품 이미지를 사용해서 하나의 상품과 하나의 상품 이미지가 출력될 수 있도록 하는 것입니다.

본격적인 목록 개발전에 테스트 코드를 이용해서 여러 개의 상품 데이터를 만들어 두도록 합니다.



```
@Transactional
@Commit
@Test
public void testInsertDummies() {

    for(int i = 0; i < 45; i++) {

        ProductDTO productDTO = ProductDTO.builder()
            .pname("Product " + i)
            .pdesc("Product Desc" + i)
            .writer("user" + (i % 10))
            .price(4000)
            .build();

        //insert into tbl_product
        productMapper.insert(productDTO);

        productDTO.addImage(i+"_test_1.jpg");
        productDTO.addImage(i+"_test__2.jpg");

        Log.info("-----");
        Log.info(productDTO.getImageList());

        //insert into tbl_product_image
        productMapper.insertImages(productDTO);
    }
}
```



```

    } //end for
}

```

testInsertDummies()는 기존 tbl_product 테이블에는 'Product0' 부터 'Product44'까지의 새로운 데이터를 추가하고 각 상품마다 2 개의 이미지가 추가됩니다. 아래 화면은 tbl_product_image 테이블에 상품 1 개당 2 개의 이미지가 추가된 결과입니다.

87	42	40_test_2.jpg	06e76f51-32f2-11f0-a6e6-745d2200e...	1	2025-05-17 16:39:10
88	43	41_test_1.jpg	06e79abc-32f2-11f0-a6e6-745d2200e...	0	2025-05-17 16:39:10
89	43	41_test_2.jpg	06e79b38-32f2-11f0-a6e6-745d2200e...	1	2025-05-17 16:39:10
90	44	42_test_1.jpg	06e7cb31-32f2-11f0-a6e6-745d2200e...	0	2025-05-17 16:39:10
91	44	42_test_2.jpg	06e7cbb9-32f2-11f0-a6e6-745d2200e...	1	2025-05-17 16:39:10
92	45	43_test_1.jpg	06e7ff58-32f2-11f0-a6e6-745d2200e...	0	2025-05-17 16:39:10
93	45	43_test_2.jpg	06e7ffe8-32f2-11f0-a6e6-745d2200e...	1	2025-05-17 16:39:10
94	46	44_test_1.jpg	06e82843-32f2-11f0-a6e6-745d2200e...	0	2025-05-17 16:39:10
95	46	44_test_2.jpg	06e828c6-32f2-11f0-a6e6-745d2200e...	1	2025-05-17 16:39:10

상품 목록에 대한 쿼리는 상품 하나당 하나의 이미지를 사용하기 위해서 ord 컬럼값을 0 으로 지정하고 LIMIT, OFFSET 을 이용해서 아래와 같이 작성할 수 있습니다.

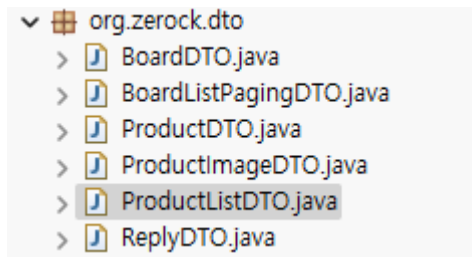
```

SELECT p.pno, pname, pdesc, price, sale, writer, p.regdate, ino, uuid, filename, ord
FROM
    tbl_product p LEFT OUTER JOIN tbl_product_image pimg ON pimg.pno = p.pno
WHERE
    pimg.ord = 0
ORDER BY p.pno desc
LIMIT 10 OFFSET 0

```

pno	pname	pdesc	price	sale	writer	regdate	ino	uuid	filename	ord
46	Product 44	Product Desc44	4,000		1 user4	2025-05-17 16:39:10	94	06e82843-32f2-11f0-a6e6-745d2200e...	44_test_1.jpg	0
45	Product 43	Product Desc43	4,000		1 user3	2025-05-17 16:39:10	92	06e7ff58-32f2-11f0-a6e6-745d2200e...	43_test_1.jpg	0
44	Product 42	Product Desc42	4,000		1 user2	2025-05-17 16:39:10	90	06e7cb31-32f2-11f0-a6e6-745d2200e...	42_test_1.jpg	0
43	Product 41	Product Desc41	4,000		1 user1	2025-05-17 16:39:10	88	06e79abc-32f2-11f0-a6e6-745d2200e...	41_test_1.jpg	0
42	Product 40	Product Desc40	4,000		1 user0	2025-05-17 16:39:10	86	06e76ec8-32f2-11f0-a6e6-745d2200e...	40_test_1.jpg	0
41	Product 39	Product Desc39	4,000		1 user9	2025-05-17 16:39:10	84	06e73ee4-32f2-11f0-a6e6-745d2200e...	39_test_1.jpg	0
40	Product 38	Product Desc38	4,000		1 user8	2025-05-17 16:39:10	82	06e708ee-32f2-11f0-a6e6-745d2200e...	38_test_1.jpg	0
39	Product 37	Product Desc37	4,000		1 user7	2025-05-17 16:39:10	80	06e6d568-32f2-11f0-a6e6-745d2200e...	37_test_1.jpg	0
38	Product 36	Product Desc36	4,000		1 user6	2025-05-17 16:39:10	78	06e6a894-32f2-11f0-a6e6-745d2200e...	36_test_1.jpg	0
37	Product 35	Product Desc35	4,000		1 user5	2025-05-17 16:39:10	76	06e67cb7-32f2-11f0-a6e6-745d2200e...	35_test_1.jpg	0

쿼리 결과는 상품 이미지가 하나 추가되어 있어서 기존의 ProductDTO 를 그대로 사용하기 보다는 별도의 DTO 를 구성하도록 합니다. dto 패키지에 ProductListDTO 클래스를 추가합니다.



```
package org.zerock.dto;

import lombok.Data;

@Data
public class ProductListDTO {

    private Integer pno;

    private String pname;

    private String pdesc;

    private int price;

    private boolean sale;

    private String writer;

    private String uuid;

    private String fileName;

}
```

상품 목록을 반환하는 list()기능과 상품 개수를 위한 listCount()을 ProductMapper 에 선언합니다.

```
public interface ProductMapper {

    int insert(ProductDTO productDTO);

    int insertImages( ProductDTO productDTO);

    ProductDTO selectOne( @Param("pno") Integer pno);

    int deleteOne(@Param("pno") Integer pno);

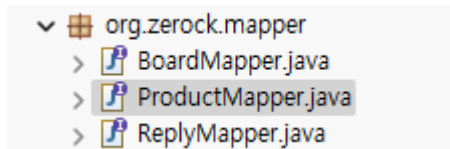
    int deleteImages(@Param("pno") Integer pno);

    int updateOne(ProductDTO productDTO);

    List<ProductListDTO> list(
        @Param("skip") int skip,
        @Param("count") int count);

    int listCount();

}
```



ProductMapper.xml 을 작성합니다.

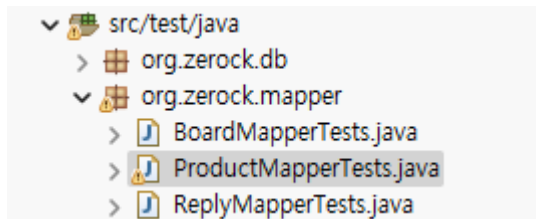
```
<select id="List">

SELECT
  p.pno, pname, pdesc, price, sale, writer, p.regdate, ino, uuid, filename, ord
FROM
  tbl_product p LEFT OUTER JOIN tbl_product_image pimg ON pimg.pno = p.pno
WHERE
  pimg.ord = 0
ORDER BY
  p.pno desc
LIMIT #{count} OFFSET #{skip}

</select>

<select id="ListCount">
SELECT
  count(*)
FROM
  tbl_product p
</select>
```

작성한 list()와 listCount()에 대해서 테스트 코드를 작성해서 마무리합니다.



```
@Test
public void testList() {

  productMapper.list(0, 10).forEach(Log::info);

  Log.info(productMapper.listCount());

}
```

```

INFO ProductListDTO(pno=46, pname=Product 44, pdesc=Product Desc44, price=4000, sale=true,
INFO ProductListDTO(pno=45, pname=Product 43, pdesc=Product Desc43, price=4000, sale=true,
INFO ProductListDTO(pno=44, pname=Product 42, pdesc=Product Desc42, price=4000, sale=true,
INFO ProductListDTO(pno=43, pname=Product 41, pdesc=Product Desc41, price=4000, sale=true,
INFO ProductListDTO(pno=42, pname=Product 40, pdesc=Product Desc40, price=4000, sale=true,
INFO ProductListDTO(pno=41, pname=Product 39, pdesc=Product Desc39, price=4000, sale=true,
INFO ProductListDTO(pno=40, pname=Product 38, pdesc=Product Desc38, price=4000, sale=true,
INFO ProductListDTO(pno=39, pname=Product 37, pdesc=Product Desc37, price=4000, sale=true,
INFO ProductListDTO(pno=38, pname=Product 36, pdesc=Product Desc36, price=4000, sale=true,
INFO ProductListDTO(pno=37, pname=Product 35, pdesc=Product Desc35, price=4000, sale=true,
seJdbcLogger.java:135) DEBUG ==> Preparing: SELECT count(*) FROM tbl_product p
seJdbcLogger.java:135) DEBUG ==> Parameters:
seJdbcLogger.java:135) DEBUG <==          Total: 1

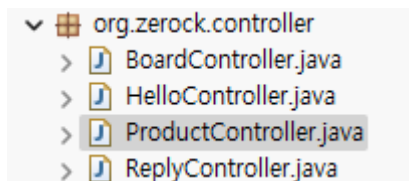
```

Win

9.3 컨트롤러의 파일 업로드

컨트롤러에서 가장 신경 써야 하는 점은 파일 업로드 기능을 구현하는 것입니다. 컨트롤러에서 브라우저가 전송하는 파일 데이터는 MultipartFile 타입으로 처리가 가능합니다.

상품 처리를 위해 controller 패키지에 ProductController 를 선언하고 GET 방식으로 동작하는 등록 화면을 보는 기능과 POST 방식으로 동작하고 ProductDTO 와 MultipartFile[]을 이용하는 등록 기능을 작성해 봅니다.



```

package org.zerock.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.zerock.dto.ProductDTO;

import lombok.extern.log4j.Log4j2;

@Controller
@RequestMapping("/product")
@Log4j2

public class ProductController {

    @GetMapping("register")
    public void registerGET() {

        Log.info("product register");
    }
}

```

```
}
```

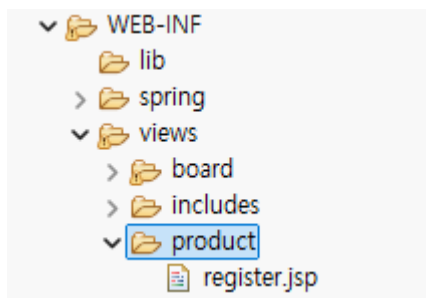
```
@PostMapping("register")
public String register(
    ProductDTO productDTO,
    @RequestParam("files") MultipartFile[] files,
    RedirectAttributes reAttr ) {

    Log.info("-----");

    Log.info(productDTO);
    Log.info(files);

    return "redirect:/product/list";
}
}
```

화면 구성을 위해서 views 폴더에 product 폴더를 추가하고 register.jsp 파일을 작성합니다.



register.jsp 에는 pname, pdesc, price, files 등을 <input>태그를 이용해서 작성합니다. 특히 주의해야 하는 점은 <form>태그의 enctype 속성을 지정하고 'multipart/form-data' 값을 지정하는 것입니다.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<%@include file="/WEB-INF/views/includes/header.jsp" %>

<div class="row justify-content-center">
<div class="col-lg-12">
    <div class="card shadow mb-4">
        <div class="card-header py-3">
            <h6 class="m-0 font-weight-bold text-primary">Product Register</h6>
        </div>

        <div class="card-body">

            <form action="/product/register" method="post" class="p-3" enctype="multipart/form-data">
                <div class="mb-3">
                    <label class="form-label">Product Name</label> <input type="text"
                        name="pname" class="form-control">
                </div>

                <div class="mb-3">
                    <label class="form-label">Product Desc</label>
                    <textarea class="form-control" name="pdesc" rows="3"></textarea>
                </div>

            </form>

        </div>
    </div>
</div>
</div>
```

```

<div class="mb-3">
  <label class="form-label">Price</label> <input type="number"
    name="price" class="form-control">
</div>

<div class="mb-3">
  <label class="form-label">Image Files</label> <input type="file"
    name="files" class="form-control" multiple="multiple">
</div>

<div class="mb-3">
  <label class="form-label">Writer</label> <input type="text"
    name="writer" class="form-control">
</div>

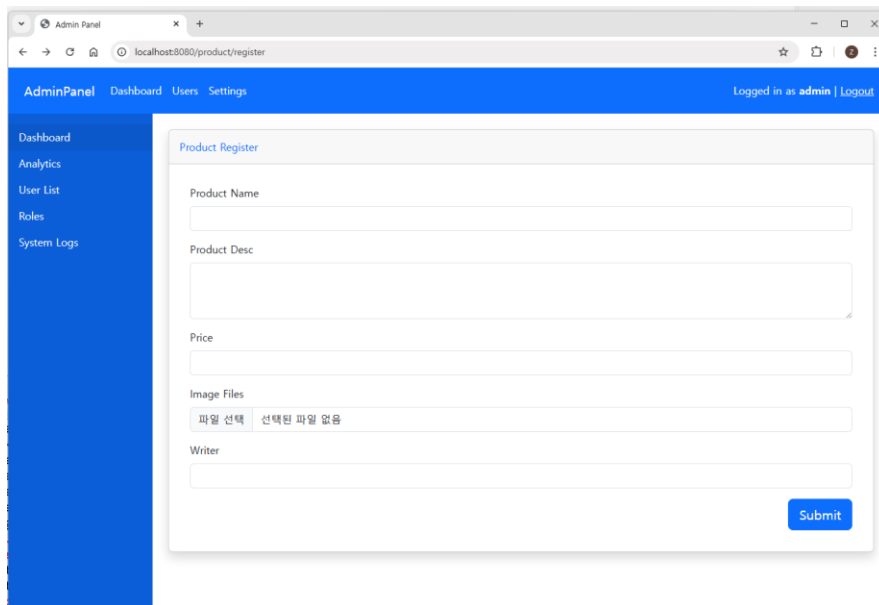
<div class="d-flex justify-content-end">
  <button type="submit" class="btn btn-primary btn-lg">Submit</button>
</div>
</form>

</div>
</div>
</div>
</div>

<%@include file="/WEB-INF/views/includes/footer.jsp" %>

```

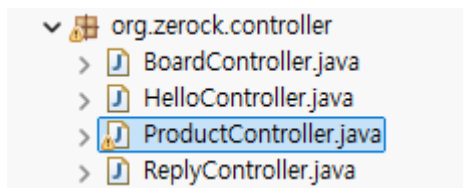
브라우저에서 테스트에 필요한 항목들을 입력하고 전송해서 POST 방식의 동작을 확인합니다.



```
INFO -----
INFO ProductDTO(pno=null, pname=Test, pdesc=aaaa, price=6000, sale=false, writer
INFO [Long.springframework.web.multipart.MultipartFile;@273484e8
cherServlet.java:1301) WARN No mapping for GET /product/list
```

서버내 파일 업로드 처리

브라우저에 전송한 파일 데이터는 MultipartFile 의 배열로 처리됩니다. MultipartFile 에서 실제 파일 업로드를 처리하는 작업은 uploadFiles()라는 이름의 메소드를 작성해서 처리합니다.



uploadFiles()는 C 드라이브내 upload 폴더에 파일들을 업로드 합니다. 만일 업로드 과정에서 문제가 발생하면 알 수 있도록 예외를 발생립니다.

```
private List<String> uploadFiles(MultipartFile[] files) throws RuntimeException {
    List<String> uploadNames = new ArrayList<>();

    if(files == null || files.length == 0) {
        return uploadNames;
    }

    String uploadPath = "C:\\upload";

    Log.info("-----uploadPath-----");
    Log.info(uploadPath);

    for (MultipartFile file : files) {
        if(file.isEmpty()) { continue; }

        String fileName = file.getOriginalFilename();

        File targetFile = null;
        targetFile = new File(uploadPath, fileName);

        try (
            InputStream fin = file.getInputStream();
            OutputStream fos = new FileOutputStream(targetFile);
        ){
            Log.info(targetFile.getAbsolutePath());

            FileCopyUtils.copy(fin, fos);
        }
    }
}
```

```

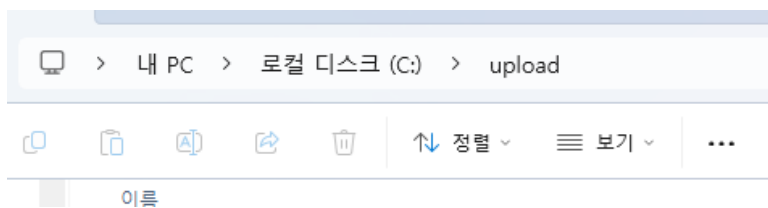
        uploadNames.add(fileName);
    }catch(Exception e) {
        Log.error(e.getMessage());
        throw new RuntimeException(e.getMessage());
    }
}

}

return uploadNames;
}

```

uploadFiles()는 C 드라이브내 upload 폴더로 파일을 업로드 합니다. 업로드가 진행되는 폴더는 미리 폴더를 생성해 둡니다.



register()에서는 우선적으로 파일의 업로드를 확인해 봅니다.

```

@PostMapping("register")
public String register(
    ProductDTO productDTO,
    @RequestParam("files") MultipartFile[] files,
    RedirectAttributes reAttr ) {

    Log.info("-----");

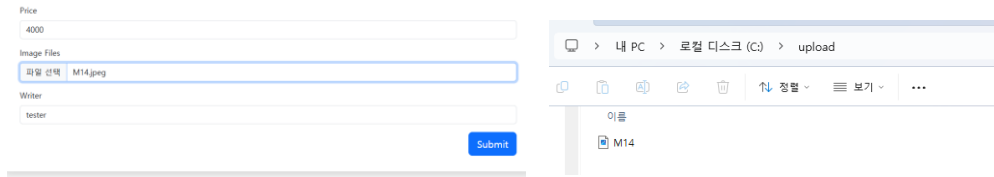
    Log.info(productDTO);
    Log.info(files);

    //upload file
    uploadFiles(files);

    return "redirect:/product/list";
}

```

브라우저에서 특정한 파일을 선택하고 전송하면 서버 내에 파일이 업로드 된 것을 확인할 수 있어야 합니다.



중복된 파일이름과 UUID

서버 내 파일 업로드에서 주의해야 하는 점 중에 하나는 동일한 이름의 파일이 업로드 되는 경우 기존 파일을 덮어쓰는 문제입니다. 사용자의 입장에서 다른 사람이 업로드한 파일로 변경될 수 있기 때문에 업로드하는 파일이 중복되지 않도록 임의의 문자열을 발생해서 처리하는데 이 때 UUID 를 이용합니다.

uploadFiles()의 내부의 for 문에서 파일 업로드시에 UUID 값 36 자리와 원본 파일의 이름을 '_'를 이용해서 연결하도록 수정합니다.

```
for (MultipartFile file : files) {
    if(file.isEmpty()) { continue; }

    String fileName = file.getOriginalFilename();

    String uploadName = UUID.randomUUID().toString() + "_" + fileName;

    File targetFile = new File(uploadPath, uploadName);

    try (
        InputStream fin = file.getInputStream();
        OutputStream fos = new FileOutputStream(targetFile);
    ){

        Log.info(targetFile.getAbsolutePath());

        FileCopyUtils.copy(fin, fos);

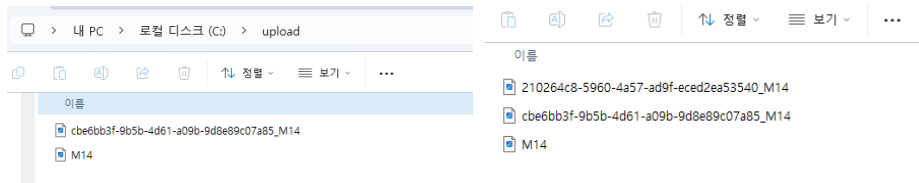
        uploadNames.add(uploadName);

    }catch(Exception e) {
        Log.error(e.getMessage());
        throw new RuntimeException(e.getMessage());
    }

}

} //end for
```

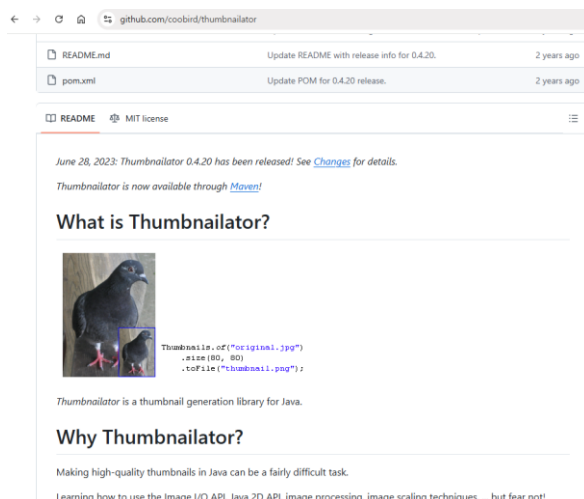
변경된 코드를 이용해서 다시 파일을 전송하면 이전과 달리 UUID 값이 추가된 파일이 만들어지는 것을 확인할 수 있습니다. 동일한 파일을 여러 번 추가해도 다른 UUID 값이 생성되므로 새로운 파일이 만들어집니다.



썸네일 이미지의 생성

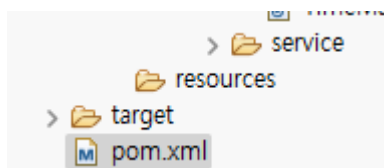
이미지 파일이 업로드 되는 과정에서 또 한가지 염두에 두어야 하는 사항은 썸네일이미지의 생성입니다. 일반적으로 원본 파일의 경우 용량이 큰 파일로 만들어지기 때문에 상품 목록처럼 여러 개의 상품에 대한 이미지들을 출력되려면 많은 양의 데이터를 받아야만 합니다.

이런 문제를 해결하기 위해서 파일이 업로드 되면 작은 용량의 썸네일 파일을 생성해서 목록 화면 등에서 활용하는 것이 일반적입니다. 썸네일 이미지의 생성에는 다양한 방법이 있지만 가장 흔한 방법은 Thumbnailator 라는 라이브러리를 활용하는 것입니다.



Thumbnailator 는 Thumbnails.of()와 같이 간단한 방법으로 지정된 크기의 이미지 파일을 생성할 수 있습니다. 코드내에서는 파일 업로드가 정상적으로 이루어진 후에 썸네일을 생성하도록 코드를 작성할 수 있습니다.

pom.xml 에 Thumbnailator 를 추가합니다.



```

<dependency>
  <groupId>net.coobird</groupId>
  <artifactId>thumbnailator</artifactId>
  <version>0.4.20</version>
</dependency>

```

uploadFiles()에서 업로드 중인 파일이 이미지인 경우라면 썸네일을 생성하도록 코드를 수정합니다.

```

private List<String> uploadFiles(MultipartFile[] files) throws RuntimeException {
    List<String> uploadNames = new ArrayList<>();

    if(files == null || files.length == 0) {
        return uploadNames;
    }

    String uploadPath = "C:\\\\upload";

    Log.info("-----uploadPath-----");
    Log.info(uploadPath);

    for (MultipartFile file : files) {
        if(file.isEmpty()) { continue; }

        String fileName = file.getOriginalFilename();

        String uploadName = UUID.randomUUID().toString() + "_" + fileName;

        File targetFile = new File(uploadPath, uploadName);

        try (
            InputStream fin = file.getInputStream();
            OutputStream fos = new FileOutputStream(targetFile);
        ){
            Log.info(targetFile.getAbsolutePath());

            FileCopyUtils.copy(fin, fos);

            uploadNames.add(uploadName);
        } catch (Exception e) {
            Log.error(e.getMessage());
            throw new RuntimeException(e.getMessage());
        }

        if(file.getContentType().startsWith("image")) {

            try {
                Thumbnails.of(targetFile)
                    .size(200, 200)
                    .toFile(new File(uploadPath, "s_" + uploadName));
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

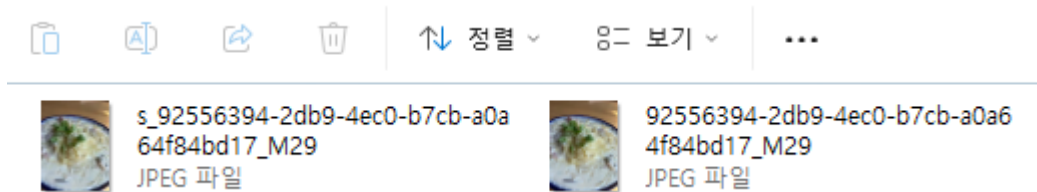
    //end for

    return uploadNames;
}

```

```
}
```

브라우저를 이용해서 이미지가 업로드 되면 원본 파일과 파일명이 's_'로 시작하는 썸네일 파일이 만들어진 것을 확인할 수 있습니다.



반환된 파일이름의 처리

ProductController 에서는 ProductDTO 에 업로드된 파일의 이름을 지정해 주어야 합니다. uploadFiles()에서 반환되는 파일의 이름은 'UUID 로 만들어진 36 자리 + _ + 원래 파일 이름'의 형태로 구성되므로 이를 이용해서 ProductDTO 의 addFile()을 호출합니다.

```
@PostMapping("register")
public String register(
    ProductDTO productDTO,
    @RequestParam("files") MultipartFile[] files,
    RedirectAttributes reAttr ) {

    Log.info("-----");

    Log.info(productDTO);
    Log.info(files);

    //upload file
    List<String> uploadNames = uploadFiles(files);

    uploadNames.forEach(name -> {

        String uuid = name.substring(0,36);
        String fileName = name.substring(37);

        Log.info(uuid);
        Log.info(fileName);

        productDTO.addImage(uuid, fileName);

    });

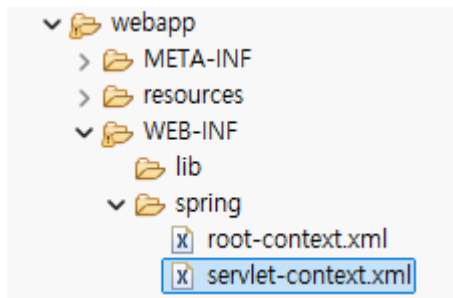
    return "redirect:/product/list";
}
```

위의 코드 실행 결과는 다음과 같이 UUID 값과 파일 이름을 구분하게 됩니다.

```
(ProductController.java:65) INFO 0466b788-2898-432c-b5cc-87280a419c08
(ProductController.java:66) INFO M28.jpeg
(ProductController.java:65) INFO 4762c854-ad79-4f69-9591-bc272df0d41a
(ProductController.java:66) INFO M29.jpeg
```

파일의 조회

사용자가 업로드한 결과를 브라우저에서 조회할 수 있도록 하기 위해서는 직접 스프링으로 해당 경로에 대한 리소스 접근이 가능하도록 설정하거나 메서드를 이용해서 구현할 수 있습니다. 예제에서는 servlet-context.xml의 설정을 이용하도록 합니다.

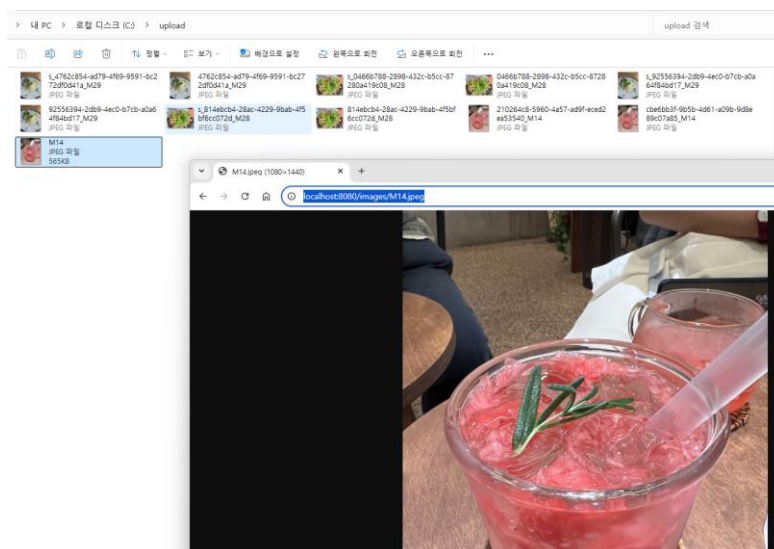


servlet-context.xml에는 호출 경로가 '/images/'로 시작하는 모든 호출은 C 드라이브 upload 폴더의 내용물을 서비스 하도록 수정합니다.

```
<mvc:resources mapping="/resources/**" location="/resources/" />

<mvc:resources mapping="/images/**"
                location="file:C:/upload/" />
```

브라우저를 통해서 upload 폴더에 있는 파일명으로 '/images/파일명'을 호출하면 아래 화면과 같이 조회가 가능합니다.



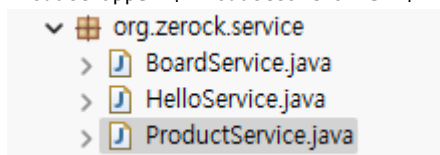
파일의 삭제

첨부 파일의 경우 수정이라는 개념 대신에 기존 파일을 삭제하고 새로운 파일들이 업로드 되는 방식이므로 경우에 따라서는 파일을 삭제하는 기능을 사용할 수도 있습니다. 이를 위해서 아래와 같이 파일 삭제 기능을 추가해 둡니다. 파일의 삭제는 예외가 발생하더라도 굳이 예외를 전달할 필요는 없습니다.

```
private void deleteFiles(List<String> fileNames) {  
    try {  
        File uploadPath = new File("C:\\upload");  
  
        for (String fileName : fileNames) {  
            File targetFile = new File(uploadPath, fileName);  
  
            targetFile.delete();  
  
            //thumbnailFile  
            File targetThumb = new File(uploadPath, "s_" + fileName);  
  
            targetThumb.delete();  
        }  
    }  
    catch(Exception e) {  
    }  
}
```

9.3 ProductService와 화면 구현

ProductMapper와 ProductController의 기본 구성이 완료되었다면 ProductService를 구현합니다.



등록 처리

ProductService 에서 가장 중요한 부분은 트랜잭션 처리입니다. 우선 등록 기능을 구현합니다.

```
package org.zerock.service;  
  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
import org.zerock.dto.ProductDTO;  
import org.zerock.mapper.ProductMapper;
```

```

import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;

@Service
@RequiredArgsConstructor
@Log4j2
@Transactional
public class ProductService {

    private final ProductMapper productMapper;

    public Integer register(ProductDTO productDTO) {

        productMapper.insert(productDTO);

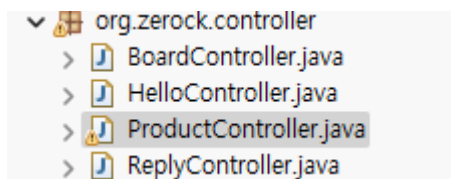
        Integer pno = productDTO.getPno();

        productMapper.insertImages(productDTO);

        return pno;
    }
}

```

작성된 기능은 ProductController 에서 연동해서 호출하도록 수정합니다.



```

package org.zerock.controller;

...생략

@Controller
@RequestMapping("/product")
@Log4j2
@RequiredArgsConstructor
public class ProductController {

    private final ProductService service;

    @GetMapping("register")
    public void registerGET() {

        log.info("product register");

    }

    @PostMapping("register")
    public String register(
        ProductDTO productDTO,
        @RequestParam("files") MultipartFile[] files,
        RedirectAttributes reAtr ) {

        log.info("-----");
    }
}

```

```

log.info(productDTO);
log.info(files);

//upload file
List<String> uploadNames = uploadFiles(files);

uploadNames.forEach(name -> {

    String uuid = name.substring(0,36);
    String fileName = name.substring(37);

    log.info(uuid);
    log.info(fileName);

    productDTO.addImage(uuid, fileName);

});

Integer pno = service.register(productDTO);

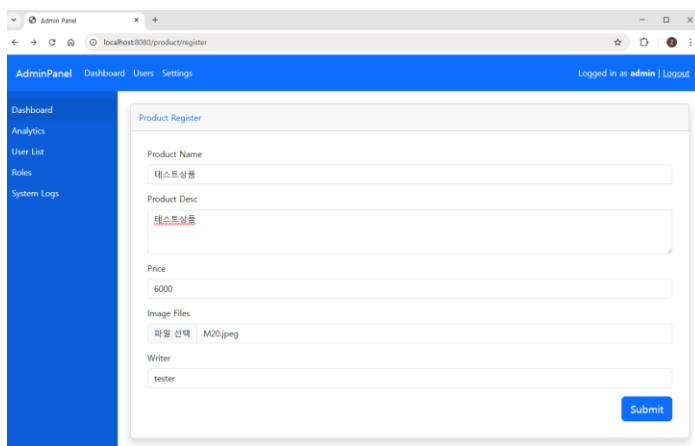
reAtr.addFlashAttribute("product", pno);

return "redirect:/product/list";
}

...생략
}

```

ProductController 의 등록 작업은 나중에 목록 화면으로 리다이렉트 처리되므로 실행 결과는 지금 당장 확인할 수는 없지만 업로드 폴더의 변화나, 테이블의 변화를 통해서 실행 결과를 확인할 수 있습니다.



The screenshot shows a web browser window with the URL 'localhost:8080/product/register'. The page has a blue header with 'AdminPanel' and navigation links: 'Dashboard', 'Users', 'Settings'. The user is logged in as 'admin'. A sidebar on the left lists 'Dashboard', 'Analytics', 'User List', 'Roles', and 'System Logs'. The main content area is titled 'Product Register' and contains a form with the following fields: 'Product Name' (text input with '테스트상품'), 'Product Desc' (text area with '테스트상품'), 'Price' (text input with '6000'), 'Image Files' (file selection with '파일 선택 M00.jpeg'), and 'Writer' (text input with 'tester'). A blue 'Submit' button is located at the bottom right of the form.

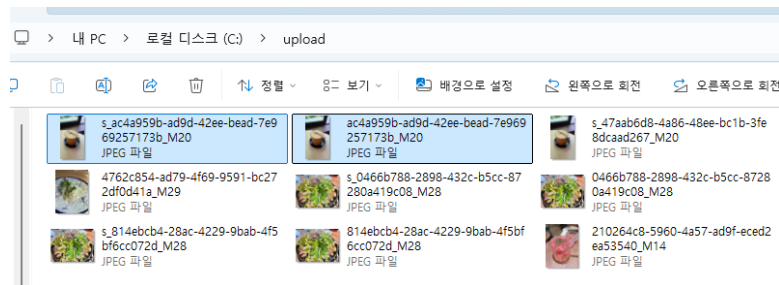
브라우저에서 호출하면 tbl_product 테이블에 insert 와 tbl_product_image 테이블에 대한 insert 가 실행됩니다.


```

DEBUG ==> Preparing: insert into tbl_product ( pname, pdesc, price, sale, writer) values ( ?, ?, ?, true, ? )
DEBUG ==> Parameters: 테스트상품(String), 테스트상품(String), 6000(Integer), tester(String)
DEBUG <== Updates: 1
java:135) DEBUG ==> Preparing: SELECT LAST_INSERT_ID()
java:135) DEBUG ==> Parameters:
java:135) DEBUG <== Total: 1
!135) DEBUG ==> Preparing: insert into tbl_product_image (pno, fileName, uuid, ord) values ( ?, ?, ?, ? )
!135) DEBUG ==> Parameters: 50(Integer), M20.jpeg(String), ac4a959b-ad9d-42ee-bead-7e969257173b(String), 0(Integer)
!135) DEBUG <== Updates: 1

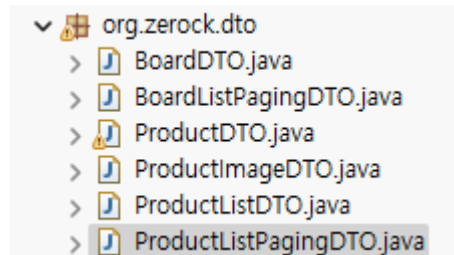
```

첨부파일 폴더에는 첨부된 이미지 파일이 만들어집니다.



상품 목록 처리

상품의 목록 데이터 처리를 위해서는 ProductListDTO 객체들의 리스트와 상품 데이터의 개수를 처리해야 합니다. 이를 위해서 dto 패키지에 ProductListPagingDTO 클래스를 추가합니다.



ProductListPagingDTO 는 게시물관리에서 작성했던 BoardListPagingDTO 와 거의 동일한 구조이고 데이터의 타입만 ProductListDTO 를 다루는 것이 차이점입니다.

```

package org.zerock.dto;

import java.util.List;
import java.util.stream.IntStream;

import lombok.Data;

@Data
public class ProductListPagingDTO {

    private List<ProductListDTO> productDTOList;

    private int totalCount;

    private int page, size;
}

```

```

private int start, end;

private boolean prev, next;

private List<Integer> pageNums;

public ProductListPagingDTO(List<ProductListDTO> productDTOList, int totalCount, int page, int size )
{
    this.productDTOList = productDTOList;
    this.totalCount = totalCount;
    this.page = page;
    this.size = size;

    //start계산을 위한 end 페이지
    int tempEnd = (int)(Math.ceil(page/10.0)) * 10;

    this.start = tempEnd - 9;

    this.prev = start != 1; //start값이 1이 아니라면 이전 페이지로 이동 필요

    //임시 end 값 * size가 totalCount 보다 크다면 totalCount로 다시 계산 필요
    if( (tempEnd * size) > totalCount ) {
        this.end = (int) ( Math.ceil(totalCount / (double)size) );
    }else {
        this.end = tempEnd;
    }

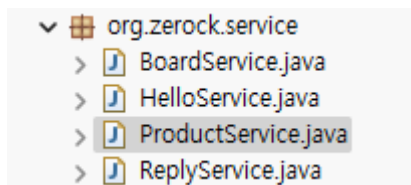
    //end 값 * size 보다 totalCount가 크다면 next로 이동 가능
    this.next = totalCount > (this.end * size);

    //화면에 출력한 번호들 계산

    this.pageNums = IntStream.rangeClosed(start, end).boxed().toList();
}
}

```

ProductService 에서 이를 이용하여 list()기능을 완성합니다.



```

public ProductListPagingDTO getList(int page, int size) {
    //페이지 번호가 0보다 작으면 무조건 1페이지
    page = page <= 0? 1 : page;
    //사이즈가 10보다 작거나 100보다 크면 10

```

```

size = (size <= 10 || page > 100) ? 10: size;

int skip = (page -1 ) * size;

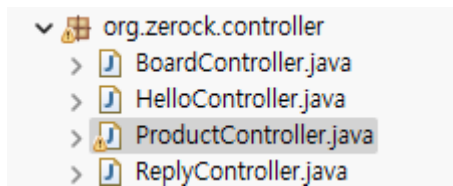
List<ProductListDTO> list = productMapper.list(skip, size);

int total = productMapper.listCount();

return new ProductListPagingDTO(list, total, page, size);
}

```

ProductController 에서는 ProductService 의 반환 결과를 Model 을 이용해서 list.jsp 로 전달합니다.



```

@GetMapping("list")
public void list(
    @RequestParam( name = "page", defaultValue = "1" ) int page,
    @RequestParam( name = "size", defaultValue = "10" ) int size,
    Model model) {

    ProductListPagingDTO dto = service.getList(page, size);

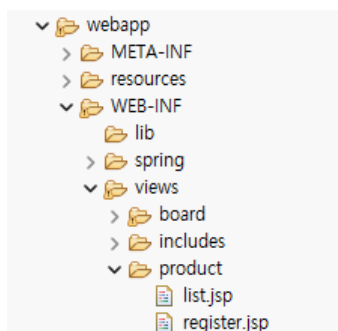
    model.addAttribute("dto", dto);

}

```

목록 화면 구성

WEB-INF/views/product/list.jsp 파일을 추가합니다.



list.jsp 와 거의 유사하게 상품의 정보를 출력합니다. 상품의 경우 uuid 와 fileName 을 이용해서 상품의 이미지를 출력하도록 구성합니다.

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@include file="/WEB-INF/views/includes/header.jsp"%>
<div class="row justify-content-center">
    <div class="col-lg-12">
        <div class="card shadow mb-4">
            <div class="card-header py-3">
                <h6 class="m-0 font-weight-bold text-primary">Product List</h6>
            </div>

            <div class="card-body">

                <table class="table table-bordered" id="dataTable">
                    <thead>
                        <tr>
                            <th>No</th>
                            <th>Product Name</th>
                            <th>Price</th>
                            <th>Writer</th>
                        </tr>
                    </thead>
                    <tbody class="tbody">

                        <c:forEach var="product" items="${dto.productDTOList}">

                            <tr data-bno="${product.pno}" >
                                <td>
                                    <a href='/product/read/${product.pno}''>
                                        <c:out value="${product.pno}"/>
                                    </a>
                                </td>

                                <td>

                                    
                                    <c:out value="${product.pname}"/>
                                </td>
                                <td><c:out value="${product.price}"/></td>
                                <td><c:out value="${product.writer}"/></td>
                            </tr>

                        </c:forEach>

                    </tbody>
                </table>

                <div class="d-flex justify-content-center">

                    <ul class="pagination">

                        <c:if test="${dto.prev}">
                            <li class="page-item">
                                <a class="page-link" href="" tabindex="-1">Previous</a>
                            </li>
                        </c:if>

                        <c:forEach var="num" items="${dto.pageNums}">
                            <li class="page-item ${dto.page == num ? 'active':'' } ">
                                <a class="page-link" href="${num}"> ${num} </a>
                            </li>
                        </c:forEach>

                        <c:if test="${dto.next}">
                            <li class="page-item">
                                <a class="page-link" href="">Next</a>

```

```

        </li>
      </c:if>
    </ul>

  </div>

</div>

</div>
</div>
</div>

<div class="modal fade" id="myModal" tabindex="-1" aria-labelledby="exampleModalLabel" aria-
hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="exampleModalLabel">Modal title</h5>
        <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>
      </div>
      <div class="modal-body">
        New Product Added
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-primary">Save changes</button>
        <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>
      </div>
    </div>
  </div>
</div>
</div>

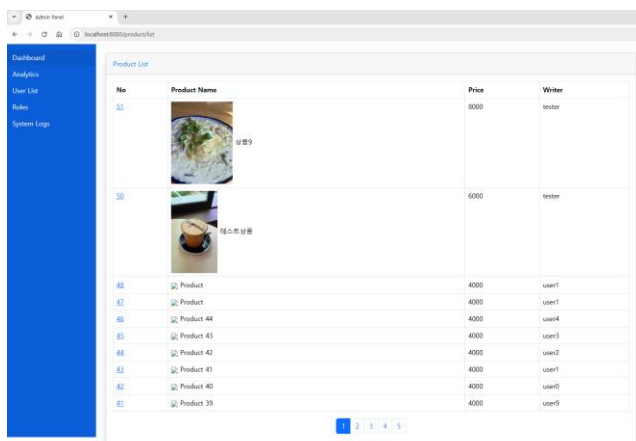
<script type="text/javascript" defer="defer">

</script>

<%@include file="/WEB-INF/views/includes/footer.jsp" %>

```

기존에 테스트 과정에서 추가된 상품들의 경우 상품의 이미지가 정상적이지 않기 때문에 이미지가 제대로 출력되지 않습니다.



list.jsp 의 <script>에는 페이징 처리와 등록 결과에 따라 모달 창을 보여주는 코드를 작성합니다.

```
<script type="text/javascript" defer="defer">

const pno = '${product}'

const myModal = new bootstrap.Modal(document.getElementById('myModal'))

if(pno){
    myModal.show()
}

const pagingDiv = document.querySelector(".pagination")
pagingDiv.addEventListener("click", (e) => {

    e.preventDefault()
    e.stopPropagation()

    const target = e.target

    //console.log(target)

    const targetPage = target.getAttribute("href")

    const size = ${dto.size} || 10 // BoardListPagingDT의 size

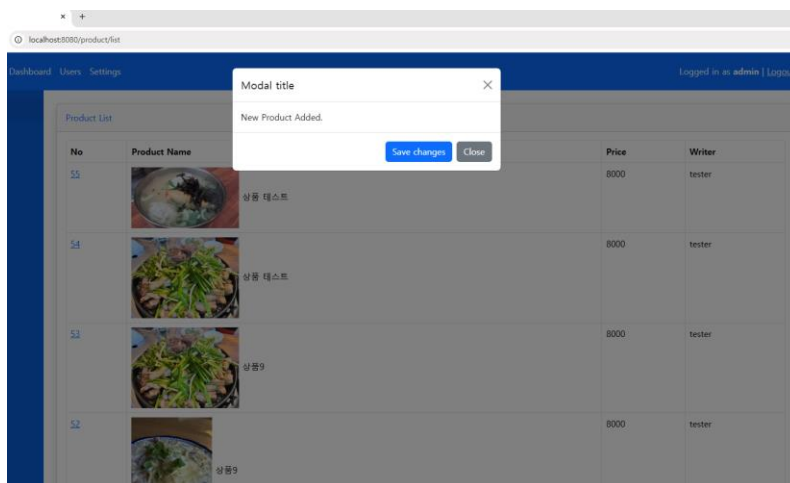
    const params = new URLSearchParams({
        page: targetPage,
        size: size
    });

    console.log(params.toString())

    self.location = `/product/list?${params.toString()}` //JavaScript 백틱, 템플릿
}, false)

</script>
```

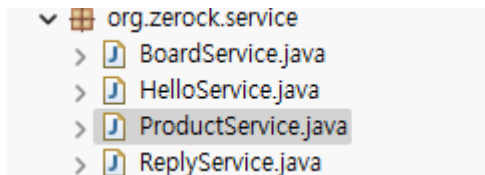
페이지 이동과 등록 후 모달 창을 확인합니다.



상품 조회

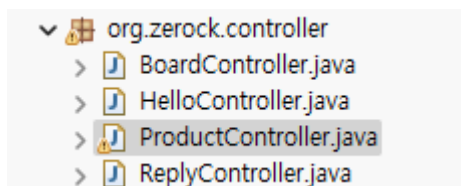
상품 조회는 ProductService 에 read()를 이용해서 ProductDTO 를 반환하도록 구성합니다.

ProductDTO 내부에는 tbl_product_image 와 조인 처리된 결과물로 List<ProductImageDTO>를 가지고 있으므로 이를 이용해서 상품의 이미지까지 한 번에 출력해 줄 수 있습니다.



```
public ProductDTO read(Integer pno) {  
  
    return productMapper.selectOne(pno);  
}
```

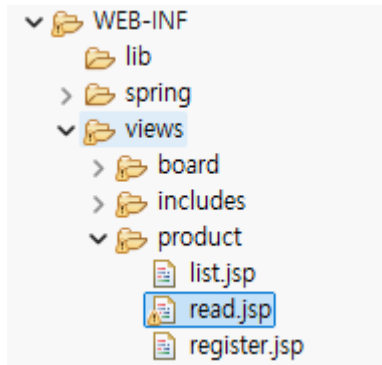
ProductController 에는 GET 방식으로 동작하는 read()기능을 구현합니다.



read()에서는 'product'라는 이름으로 ProductDTO 를 Model 에 추가합니다.

```
@GetMapping("read/{pno}")  
public String read(  
    @PathVariable("pno")Integer pno, Model model) {  
  
    Log.info("pno: "+ pno);  
  
    model.addAttribute("product", service.read(pno));  
  
    return "/product/read";  
}
```

화면은 read.jsp 를 구성해서 처리합니다. 조회 화면에서는 썸네일이 아닌 원본 파일을 볼 수 있도록 출력합니다.



```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ include file="/WEB-INF/views/includes/header.jsp" %>

<div class="row justify-content-center">
  <div class="col-lg-12">
    <div class="card shadow mb-4">
      <div class="card-header py-3">
        <h6 class="m-0 fw-bold text-primary">Product Read</h6>
      </div>
      <div class="card-body">

        <div class="mb-3 input-group input-group-lg">
          <span class="input-group-text">No</span>
          <input type="text" class="form-control" value="<c:out value='${product.pno}'/>" readonly>
        </div>

        <div class="mb-3 input-group input-group-lg">
          <span class="input-group-text">Product Name</span>
          <input type="text" name="title" class="form-control" value="<c:out
value='${product.pname}'/>" readonly>
        </div>

        <div class="mb-3 input-group input-group-lg">
          <span class="input-group-text">Desc</span>
          <textarea class="form-control" name="content" rows="3" readonly><c:out
value='${product.pdesc}'/></textarea>
        </div>

        <div class="mb-3 input-group input-group-lg">
          <span class="input-group-text">Writer</span>
          <input type="text" name="writer" class="form-control" value="<c:out
value='${product.writer}'/>" readonly>
        </div>

        <div class="mb-3 input-group input-group-lg">
          <span class="input-group-text">Price</span>
          <input type="text" name="price" class="form-control" value="<c:out
value='${product.price}'/>" readonly>
        </div>

        <div class="float-end">

          <a href="/product/List" class="btn">
            <button type="button" class="btn btn-info btnList"> LIST </button>
          </a>

          <a href="/product/modify/${product.pno}" class="btn">
            <button type="button" class="btn btn-warning btnModify">MODIFY</button>
          </a>


```



```

        </div>

    </div>
</div>
</div>
</div>

<div class="mb-3">
    <label class="form-label fw-bold">Product Images</label>
    <div class="row">
        <c:forEach var="image" items="${product.imageList}">
            <div class="col-md-3 mb-3">
                <div class="card">
                    <a href="/images/${image.uuid}_${image.fileName}" target="_blank">
                        
                    </a>
                </div>
            </div>
        </c:forEach>
    </div>
</div>

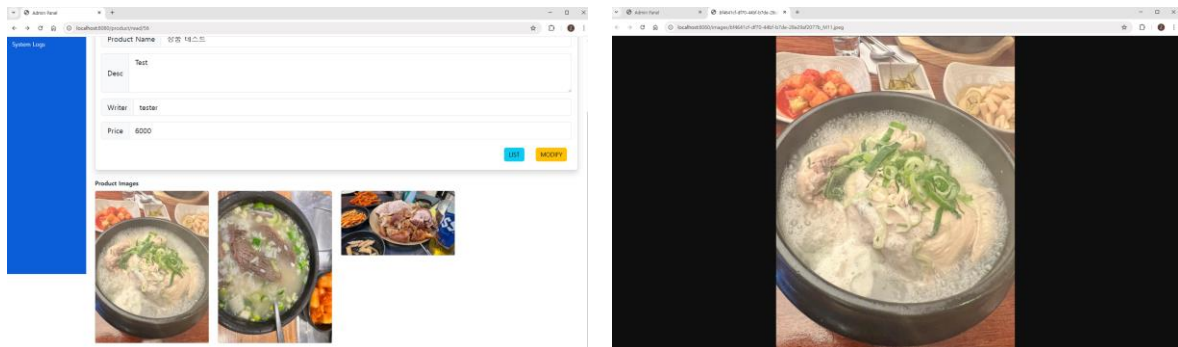
<script>

</script>

<%@ include file="/WEB-INF/views/includes/footer.jsp" %>

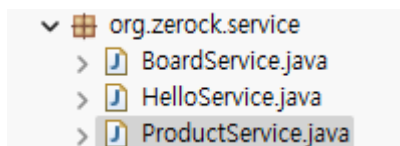
```

read.jsp 에서 상품의 이미지는 클릭하면 새로운 창으로 조회가 가능합니다.



상품 수정/삭제

상품의 수정/삭제는 ProductService 에 기능을 먼저 구현합니다. 특히 상품 수정의 경우 기존 상품 이미지들을 삭제하고 새로운 상품 이미지들을 추가하는 방식이므로 주의가 필요합니다.



```

public void remove(Integer pno) {
    productMapper.deleteOne(pno);
}

public void modify(ProductDTO productDTO) {

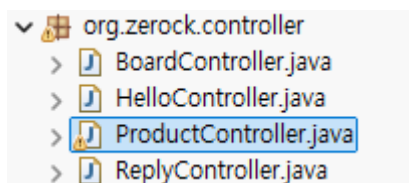
    //기존 이미지 삭제
    productMapper.deleteImages(productDTO.getPno());

    //상품 정보 수정
    productMapper.updateOne(productDTO);

    //상품 이미지 갱신
    productMapper.insertImages(productDTO);
}

```

상품의 수정과 삭제는 모두 ProductController 에서 '/product/modify/상품번호' 경로를 통해서 이루어지도록 구성합니다.



```

@GetMapping("modify/{pno}")
public String modifyGET(
    @PathVariable("pno") Integer pno, Model model) {

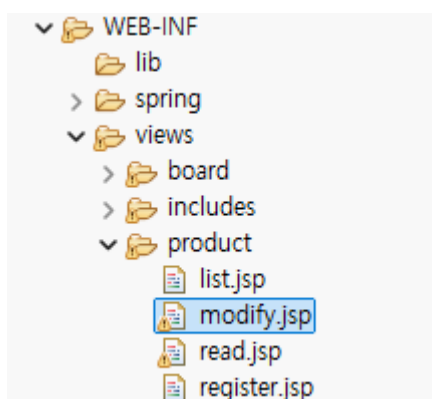
    Log.info("pno: "+ pno);

    model.addAttribute("product", service.read(pno));

    return "/product/modify";
}

```

화면은 modify.jsp 를 구성해서 처리합니다. modify.jsp 는 기본적으로 read.jsp 와 거의 동일하지만 <form>태그를 이용해서 감싸고 pname, pdesc, price 는 수정이 가능하도록 처리합니다.



```

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ include file="/WEB-INF/views/includes/header.jsp" %>

<div class="row justify-content-center">
  <div class="col-lg-12">
    <div class="card shadow mb-4">
      <div class="card-header py-3">
        <h6 class="m-0 fw-bold text-primary">Product Modify</h6>
      </div>
      <div class="card-body">
        <form id="form1" action="/product/modify/${product.pno}" method="post" enctype="multipart/form-
data">
          <div class="mb-3 input-group input-group-lg">
            <span class="input-group-text">No</span>
            <input type="text" name="pno" class="form-control" value="<c:out value='${product.pno}'/>"
readonly>
          </div>

          <div class="mb-3 input-group input-group-lg">
            <span class="input-group-text">Product Name</span>
            <input type="text" name="pname" class="form-control" value="<c:out
value='${product.pname}'/>" >
          </div>

          <div class="mb-3 input-group input-group-lg">
            <span class="input-group-text">Desc</span>
            <textarea class="form-control" name="pdesc" rows="3"><c:out
value='${product.pdesc}'/></textarea>
          </div>

          <div class="mb-3 input-group input-group-lg">
            <span class="input-group-text">Writer</span>
            <input type="text" name="writer" class="form-control" value="<c:out
value='${product.writer}'/>" readonly>
          </div>

          <div class="mb-3 input-group input-group-lg">
            <span class="input-group-text">Price</span>
            <input type="number" name="price" class="form-control" value="<c:out
value='${product.price}'/>">
          </div>

          <div class="float-end">
            <button type="button" class="btn btn-info btnList">LIST</button>
            <button type="button" class="btn btn-warning btnModify">MODIFY</button>
            <button type="button" class="btn btn-danger btnRemove">REMOVE</button>
          </div>
        </form>

      </div>
    </div>
  </div>
</div>

<div class="mb-3">
  <label class="form-label fw-bold">Product Images</label>
  <div class="row">
    <c:forEach var="image" items="${product.imageList}">
      <div class="col-md-3 mb-3">
        <div class="card">
          <a href="/images/${image.uuid}_${image.fileName}" target="_blank">
            
          </a>
        </div>
      </div>
    </c:forEach>
  </div>

```

```

</div>
</div>

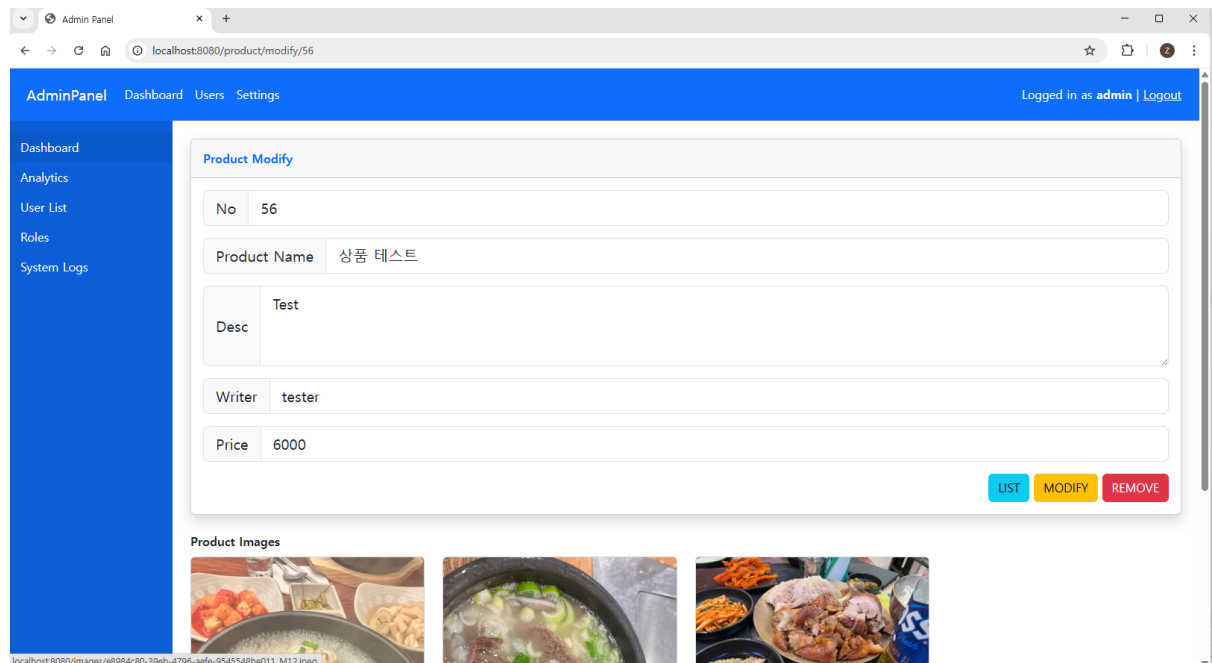
<script>

</script>

<%@ include file="/WEB-INF/views/includes/footer.jsp" %>

```

브라우저를 통해서 버튼들과 화면을 확인합니다.



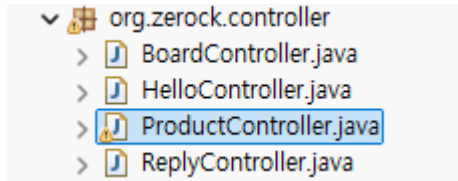
상품의 삭제

상품 삭제의 처리는 상품의 sale 값을 false 로만 변경합니다. 한 가지 고민을 상품 이미지의 삭제 여부인데 이에 대한 판단 기준은 '나중에 다시 상품 판매가 가능한가? 혹은 나중에 삭제된 상품을 조회할 수 있는가?' 입니다.

예를 들어 지금 잠시 상품을 판매하지 않기로 했다면 상품 이미지들을 그대로 유지하는 것이 안전합니다. 상품의 경우 등록자가 상품 이미지를 가지고 있기 때문에 나중에 다시 상태를 변경하는 것이 수월하지 않기 때문입니다.

만일 상품이 삭제된 후에 그냥 새로운 상품을 등록하는 방식으로 결정되었다면 그냥 상품의 이미지들을 역시 삭제할 수 있습니다. 그럼에도 데이터베이스상에서 상품 데이터는 나중에 통계나 구매 데이터와 연결되기 때문에 삭제하지 않는 것이 좋습니다.

ProductController에서는 POST 방식으로 삭제 부분을 구현합니다. 삭제 후에는 다시 상품 목록 화면으로 이동하도록 구현합니다.



```
@PostMapping("remove")
public String remove( @RequestParam("pno")Integer pno , RedirectAttributes rttr) {

    service.remove(pno);

    rttr.addFlashAttribute("result", "deleted");

    return "redirect:/product/list";

}
```

modify.jsp에서는 버튼에 대한 처리를 위해 이벤트 처리를 추가합니다.

```
<script>

const form = document.querySelector("#form1")

document.querySelector(".btnRemove").addEventListener("click", e => {

    e.preventDefault()
    e.stopPropagation()

    form1.action = "/product/remove"
    form1.submit()

},false)

</script>
```

브라우저에서 'REMOVE'버튼을 클릭해서 서버를 호출하는 것을 확인합니다.

```
DEBUG ==> Preparing: UPDATE tbl_product SET sale = false WHERE pno = ?
DEBUG ==> Parameters: 55(Integer)
DEBUG <== Updates: 1
```

목록 화면 처리

실제로 상품 데이터가 삭제된 후에 전달되는 result 값을 이용해서 모달 창을 띄울 수 있도록 구성합니다.

```
<script type="text/javascript" defer="defer">

const pno = '${product}'

const result = '${result}'

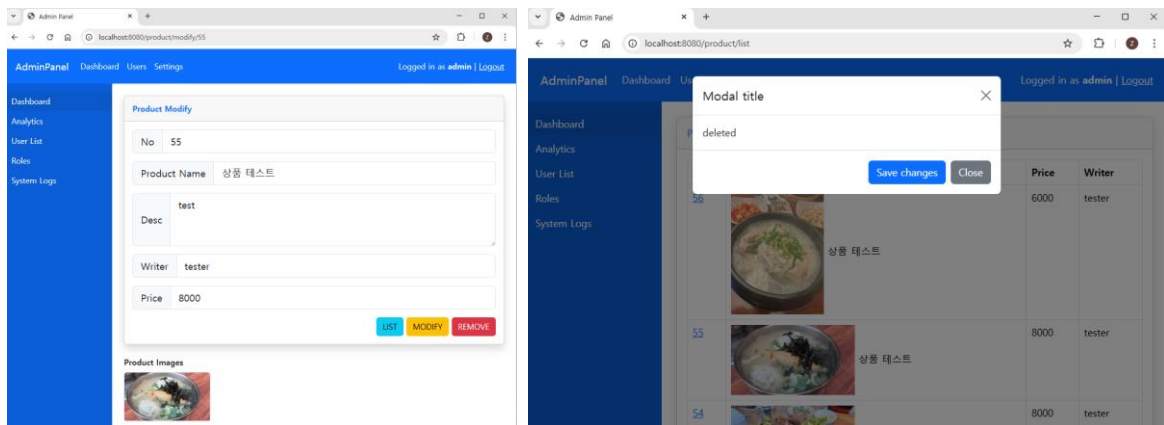
const myModal = new bootstrap.Modal(document.getElementById('myModal'))

if(result) {

    document.querySelector(".modal-body").innerHTML = result

}

if(pno || result){
    myModal.show()
}
....
```



화면상에서 삭제된 상품이 구분될 수 있도록 <style>을 적용합니다.

```
<style>
.deleted-row {
    background-color: #f0f0f0;
    color: #888;
    text-decoration: line-through;
    font-style: italic;
}
.deleted-row img {
    opacity: 0.4;
}
</style>
```

상품 목록 중에 판매되지 않는 상품에는 class 를 적용합니다.

```
<c:forEach var="product" items="${dto.productDTOList}">
```

```

<tr data-bno="${product.pno}" class="${not product.sale ? 'deleted-row' : ''}" >
  <td>
    <a href="/product/read/${product.pno}" >
      <c:out value="${product.pno}"/>
    </a>
  </td>




  <td>

    
    <c:out value="${product.pname}"/>
  </td>
  <td><c:out value="${product.price}"/></td>
  <td><c:out value="${product.writer}"/></td>
</tr>

</c:forEach>

```

브라우저에서는 sale 값에 의해서 판매중인 상품이 구분됩니다.

No	Product Name	Price	Writer
56	 상품 테스트	6000	tester
55	 상품 테스트	8000	tester
54	 상품 테스트	8000	tester

상품의 수정

상품 수정 과정에서 가장 중요한 부분은 상품 이미지들이 변경되는 것에 대한 처리입니다. 상품 이미지 처리는 크게 1) 기존 상품 이미지의 삭제와 2) 새로운 상품 이미지의 추가로 구분될 수 있습니다.

기존 상품 이미지의 삭제는 <form>태그의 전송 전에 사용자가 기존 상품 이미지들 중에서 삭제하고 남은 이미지들만 남겨서 같이 전송되어야 합니다. <input type='hidden;'> 으로 구성해서 상품 수정에도 원래 파일들 중에서 남겨진 파일들의 목록을 확인해야 합니다.

새로운 상품 이미지들은 <input type='file'>을 이용해서 상품 등록과 같이 파일들을 업로드 처리해야 합니다.

화면에서의 기존 상품 이미지 처리

등록했던 상품의 이미지 중에 사용자가 원하는 상품 이미지는 삭제가 가능하도록 각 상품의 이미지가 삭제가 가능하도록 버튼을 추가해야 합니다.

우선 modify.jsp 에서 상품 이미지들을 출력할 때 각 상품에 'Delete' 버튼을 추가하고 나중에 이벤트를 처리를 위해서 이미지들을 포함하고 있는 <div>에 'productImages'라는 class 속성값을 부여합니다.

```
<div class="mb-3 productImages">
  <label class="form-label fw-bold">Product Images</label>
  <div class="row">
    <c:forEach var="image" items="${product.imageList}">
      <div class="col-md-3 mb-3">
        <div class="card">
          <a href="/images/${image.uuid}_${image.fileName}" target="_blank">
            
          </a>
          <button type="button" class="btn btn-danger btn-sm position-absolute top-0 end-0 m-2 delete-
image-btn"
            data-file="${image.uuid}_${image.fileName}">
            Delete
          </button>
        </div>
      </div>
    </c:forEach>
  </div>
</div>
```

브라우저에서는 각 상품 이미지에 버튼이 추가된 것을 확인합니다.

Product Images



각 상품 이미지의 삭제에 대한 이벤트를 처리를 추가합니다.

```
document.querySelector(".productImages").addEventListener("click", e => {
  e.preventDefault()
  e.stopPropagation()

  const target = e.target
```



```

const fileName = target.getAttribute("data-file")

if(!fileName) {
  return
}

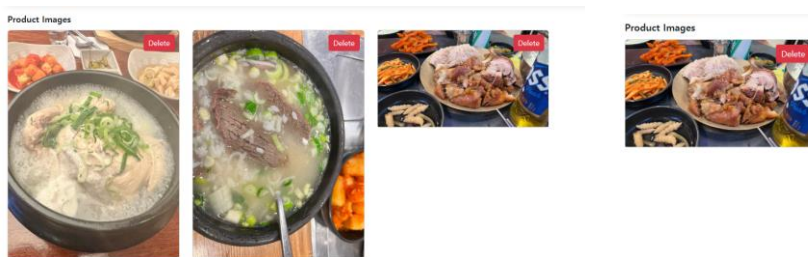
//해당 <div>를 찾아가야 함

const divObj = target.closest(".col-md-3")
divObj.remove()

}, false)

```

브라우저에서는 각 상품 이미지의 'Delete'버튼을 통해서 화면에서 삭제가 가능합니다.



새로운 상품 이미지 추가와 전송

새로운 상품 이미지를 추가하기 위해서 <form>태그 내부에 <input>을 추가합니다.

```

<div class="mb-3">
  <input type="file"
    name="files" class="form-control" multiple="multiple">
</div>

<div class="float-end">
  <button type="button" class="btn btn-info btnList">LIST</button>
  <button type="button" class="btn btn-warning btnModify">MODIFY</button>
  <button type="button" class="btn btn-danger btnRemove">REMOVE</button>
</div>
</form>

```

Price	6000
파일 선택	선택된 파일 없음

LIST
MODIFY
REMOVE

브라우저에서 'MODIFY' 버튼을 처리하는 이벤트를 작성합니다. 이벤트 처리 과정에서 화면상에 남아 있는 이미지들은 'oldImages'라는 이름으로 <form>태그내 추가합니다.

```

document.querySelector(".btnModify").addEventListener("click", (e) => {

    e.preventDefault()
    e.stopPropagation()

    form1.action = '/product/modify';
    form1.method = 'post';

    const imageArr = document.querySelectorAll(".productImages button")

    if(imageArr){

        let str = ""

        for(let image of imageArr){

            const imageFile = image.getAttribute("data-file")

            str += `<input type='hidden' name='oldImages' value='${imageFile}'>`

        }

        form1.querySelector("div:last-child").insertAdjacentHTML("beforeend", str)
    }
});

```

브라우저에서는 'Modify'버튼을 클릭하면 <form>태그가 끝나기 전에 <input type='hidden'..> 태그들이 생성되는 것을 확인할 수 있습니다.

```

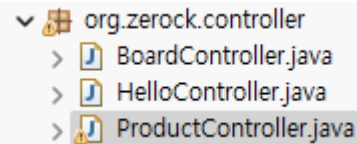
<form id="form1" action="/product/modify" method="post" enctype="multipart/form-data">
  <div class="mb-3 input-group input-group-lg">...</div>
  <div class="mb-3 input-group input-group-lg">...</div>
  <div class="mb-3 input-group input-group-lg">...</div>
  <div class="mb-3 input-group input-group-lg">...</div>
  <div class="mb-3 input-group input-group-lg">...</div>
  <div class="mb-3">
    <input type="file" name="files" class="form-control" multiple="multiple">
  </div>
  <div class="float-end">
    <button type="button" class="btn btn-info btnList">LIST</button>
    <button type="button" class="btn btn-warning btnModify">MODIFY</button>
    <button type="button" class="btn btn-danger btnRemove">REMOVE</button>
    <input type="hidden" name="oldImages" value="bf4641cf-df70-44bf-b7de-28e29af2077b_M11.jpeg">
    <input type="hidden" name="oldImages" value="e8984c80-39eb-4796-aefe-9545548be011_M12.jpeg">
    <input type="hidden" name="oldImages" value="fc1ba42a-f940-422e-aed9-7d122fd89efd_M13.jpeg">
  </div>
</form>

```

컨트롤러의 수정 처리

ProductController 는 다음과 같은 파라미터가 필요합니다.

- 상품 정보 자체의 수정을 위한 ProductDTO
- 기존 상품 이미지들(유지해야 하는 이미지들)
- 새로운 파일 데이터(새로운 이미지들)



```
@PostMapping("modify")
public String modifyPost(ProductDTO productDTO,
    @RequestParam("oldImages") String[] oldImages,
    @RequestParam("files") MultipartFile[] files ) {

    List<String> newFileNames = uploadFiles(files);

    //oldImages
    if(oldImages != null && oldImages.length > 0) {

        for (String oldImage : oldImages) {

            String uuid = oldImage.substring(0,36);
            String fileName = oldImage.substring(37);

            productDTO.addImage(uuid, fileName);
        }
    }

    if(newFileNames != null && newFileNames.size() > 0) {

        for (String newImage : newFileNames) {

            String uuid = newImage.substring(0,36);
            String fileName = newImage.substring(37);

            productDTO.addImage(uuid, fileName);
        }
    }

    service.modify(productDTO);

    return "redirect:/product/read/" + productDTO.getPno();
}
```

마지막으로 modify.jsp 에서 서버 전송을 호출합니다.

```
document.querySelector(".btnModify").addEventListener("click", (e) => {

    e.preventDefault()
    e.stopPropagation()

    form1.action = '/product/modify';
    form1.method = 'post';

    const imageArr = document.querySelectorAll(".productImages button")

    if(imageArr){

        let str = ""

        for(let image of imageArr){

            const imageFile = image.getAttribute("data-file")

            str += `<input type='hidden' name='oldImages' value='${imageFile}'>`
        }
    }
}
```

```

    } //end for

    form1.querySelector("div:last-child").insertAdjacentHTML("beforeend", str)
}

form1.submit() //서버로 전송
});

```

브라우저를 통해서 최종적인 동작을 확인하면 3 번의 SQL 이 실행되는 것을 확인할 수 있습니다.

```

==> Preparing: DELETE FROM tbl_product_image WHERE pno = ?
==> Parameters: 56(Integer)
<== Updates: 3
    Preparing: UPDATE tbl_product SET pname = ?, pdesc = ?, price = ? WHERE pno = ?
    Parameters: 상품 테스트 (String), Test(String), 6000(Integer), 56(Integer)
    Updates: 1
==> Preparing: insert into tbl_product_image (pno, fileName, uuid, ord) values ( ?, ?, ?, ? ), ( ?, ?, ?, ? ), ( ?
==> Parameters: 56(Integer), M11.jpeg(String), bf4641cf-df70-44bf-b7de-28e29af2077b(String), 0(Integer), 56(Integer),
<== Updates: 4

```

수정 작업이 끝나면 다시 조회 화면으로 이동하게 됩니다.

Product Modify

No56

Product Name상품 테스트

Test

Desc

Writertester

Price6000



파일 선택선택된 파일 없음

LIST

MODIFY

REMOVE

Product Images

Product Read

No56

Product Name상품 테스트

Test

Desc

Writertester

Price6000

LIST

MODIFY

Product Images

