기말 프로젝트 최종 개발 보고서 [컴퓨터과학과 2018301034 서 용 주]

[사전 준비]

1) pom.xml 파일에 프로젝트에 필요한 라이브러리 dependency 추가

- DB 관련 처리를 위해 **MySql, Spring JDBC, MyBatis등 DB 관리 라이브러리를 추가**했다.
- WEB 사용을 위해 **Spring Web 라이브러리를 추가**했다.
- JSP 사용을 위해 **jstl 등 필요한 라이브러리를 추가**했다.
- 유효성 검사를 위해 **javax, valitator 관련 라이브러리를 추가**했다.

```
<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jdbc</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-jdbc</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>
		<dependency>
			<groupId>org.mybatis.spring.boot</groupId>
			<artifactId>mybatis-spring-boot-starter</artifactId>
			<version>2.2.2</version>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-devtools</artifactId>
			<scope>runtime</scope>
			<optional>true</optional>
		</dependency>
		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
		<!--Jsp위해-->
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-tomcat</artifactId>
			<scope>provided</scope>
		</dependency>

		<!-- JSP -->
		<dependency>
			<groupId>org.apache.tomcat.embed</groupId>
			<artifactId>tomcat-embed-jasper</artifactId>
		</dependency>

		<dependency>
			<groupId>javax.servlet</groupId>
			<artifactId>jstl</artifactId>
		</dependency>

		<!--javax  -->
		<dependency>
			<groupId>javax.validation</groupId>
			<artifactId>validation-api</artifactId>
```

```
                <scope>compile</scope>
                <optional>true</optional>
        </dependency>

        <!--javax Validator 유효성 검사 위해 추가함 -->
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <!-- DataSouce 설정  -->
        <dependency>
                <groupId>org.apache.commons</groupId>
                <artifactId>commons-dbcp2</artifactId>
                <version>2.7.0</version>
        </dependency>

</dependencies>
```

2) application.properties 파일에 설정 추가

- WEB 연결 시, jsp 파일의 기본 위치를 찾도록 기본 설정으로 추가했다.

[기본 구조 설정]

▶ JavaConfig : 자바 설정 클래스 작성

- **DataSource 클래스**로 MySql 데이터베이스에 연결 정보를 작성한 뒤, 빈으로 등록했다.

```
@Bean
public DataSource source() {
        BasicDataSource source = new BasicDataSource();
        source.setDriverClassName("com.mysql.cj.jdbc.Driver");
source.setUrl("jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8&s
erverTimezone=Asia/Seoul");
        source.setUsername("root");
        source.setPassword("1234");

        return source;
}
```

- **JdbcTemplate 클래스에 DataSource 객체를 주입**하여 sql 쿼리문 전달 객체를 실질적으로 빈으로 등록했다.

```
@Bean
public JdbcTemplate db(BasicDataSource source) {
        JdbcTemplate db = new JdbcTemplate(source); //DB 연동
        return db;
}
```

- UserBean : '회원' 사용자 데이터 객체를 빈으로 등록했다.
- ContentBean : 물품 재고 정보 데이터 객체를 빈으로 등록했다.
- SellingBean: 판매 정보 데이터 객체를 빈으로 등록했다.
- BoardInfoBean: 웹의 Top Menu 에 출력될 메뉴 이름을 따로 데이터 객체로 사용하고 빈으로 등록했다.
- BoardDao: 게시판에 입출력할 쿼리문 작성을 모두 여기에서 처리했으므로 이를 Bean으로 등록했다.
- SellingMapper: 판매정보 Select 할 경우 사용할 mapper를 빈으로 등록했다.

▶HomeController 홈 컨트롤러 작성

- 리다이렉트 처리 : Home 기본 경로를 /main으로 redirect 처리했다. (기본 경로)

▶부트스트랩 CDN 템플릿 활용

- 전체적인 웹 MVC를 감싸는 디자인은 부트스트랩 템플릿을 활용하여 구현했다.

▶JSP 파일 중복되는 부분 묶기

- JSP 파일 속 상단/하단부가 계속 중복되므로 include 폴더에 따로 분리해주었다.
- JSTL 태그 라이브러리 추가. <c:import> 태그 사용하여 원하는 위치 상에 주입.

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

▶메인 로고 버튼 클릭 시 Root 경로 ${root} 로 주기

-JSTL 라이브러리 활용.

```
<c:set var='root' value="${pageContext.request.contextPath }/"/>
```

▶메인 화면에 출력시킬 image 폴더 추가

- 이미지를 webapp의 하위 폴더에 resources/image 폴더를 만들어 png파일을 담아두었고, 이 값을 main.JSP파일에서 img src 경로를 주어 메인 뷰에 출력시켜주었다. \

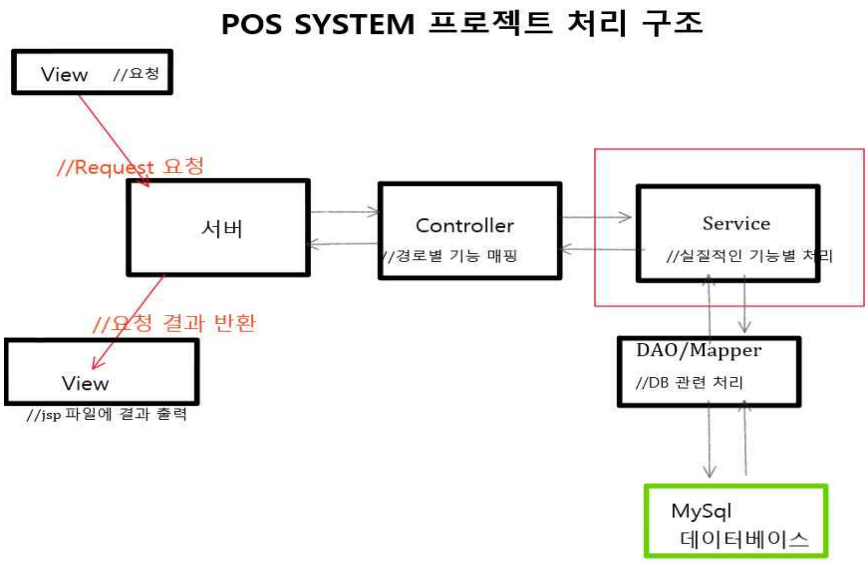
▶JSP 파일에서 사용자 입력값 등을 편하게 처리하기 위해 spring 제공 커스텀 태그 라이브러리 추가했다.

```
<%@ page contentType="text/html; charset=utf-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="tf" tagdir="/WEB-INF/tags"%>
```

[프로젝트 기본 처리 구조]

: 요청 -> controller -> service -> DAO(Mapper) -> DB

: view 출력 <- controller <- service <- DAO(Mapper) <- DB



[Spring JDBC 활용한 데이터 입출력 처리 구조]

[DB에 '데이터 입력']

- 뷰에서 ModelAttribute 로 전달받은 객체를 컨트롤러가 @ModelAttribute 파라미터로 받아서 서비스 객체에게 보낸다. 서비스 객체는 다시 내부적으로 Dao 에게 입력 객체를 보내주면 Dao는 다시 **JdbcTemplate의 update() 메소드를 활용**하여 DB 상에 최종 입력/수정 처리를 하는 구조로 데이터 입력을 처리했다.

▶ **Dao 객체**

```
// 판매 정보 저장하기
public void addSellingInfo(SellingBean sellingInfo) {
    String sql = "INSERT INTO selling_table(selling_name, selling_price) VALUES (?, ?)";
    jdbcTemplate.update(sql, sellingInfo.getSelling_name(), sellingInfo.getSelling_price());
}
```

[DB의 '데이터 출력']

1) **'한 행' 데이터 출력** : jdbcTemplate의 **queryForObject()를 사용**하여 Select 쿼리.

▶ **Dao 객체**

```
// '상품명 판매 계산 기능' 사용자가 판매 상품명 입력 -> 가격 가져오기
public int getSellPrice(String content_name) {
    String sql = "SELECT content_price FROM content_table WHERE content_name = ?";
    int price = jdbcTemplate.queryForObject(sql, Integer.class, content_name);
    return price;
}
```

2) **'여러 행' 데이터 출력** : jdbcTemplate의 **query()를 활용**하되,
코드 복잡도를 줄이기 위해서 RowMapper를 implements 한 Mapper객체를 생성하여 내부에 mapRow() 메소드를 재정의하는 방식으로 데이터 타입을 매칭시켜주었다.

▶ **Mapper 객체 : RowMapper< > 구현한 객체**

```
public class BoardMapper implements RowMapper<ContentBean> {

    @Override
    public ContentBean mapRow(ResultSet rs, int rowNum) throws SQLException {
        // TODO Auto-generated method stub

        //DB 속 데이터 타입과 코드의 데이터 객체 필드 타입 매칭시킴
        ContentBean bean = new ContentBean();
        bean.setContent_idx(rs.getInt("content_idx"));
        bean.setContent_date(rs.getString("content_date"));
        bean.setContent_name(rs.getString("content_name"));
        bean.setContent_price(rs.getInt("content_price"));;
        bean.setContent_count(rs.getInt("content_count"));

        return bean;
    }
}
```

▶ **Dao 객체 : jdbcTemplate.query() 사용**

```
 // '최근 한달' 통계 정보 가져오기
public List<SellingBean> getOneMonthSellingInfo() {
    String sql = "SELECT * FROM selling_table WHERE selling_date BETWEEN DATE_ADD(NOW(),
INTERVAL -1 MONTH ) AND NOW() ORDER BY selling_date ASC";

    List<SellingBean> list = jdbcTemplate.query(sql, sellingMapper);
    return list;
}
```

>> 계정 관리 기능 (회원가입, 로그인, 로그아웃, 정보수정)

▷[User 데이터 구조]

-idx는 저장 시 자동 증가 처리되도록 auto_increment 로 선언했다.

사용자 User 정보 테이블

User_idx	User_name	User_id	User_pw

//user_idx : auto-increment

1) 회원가입 기능

- 사용자가 폼에 사용자의 name/id/pw 등을 입력하면,
입력값의 유효성 검사(아이디 중복 여부 등)을 거친 뒤 DB 상에 회원 정보를 담는 기능.

▶ 회원가입 폼의 '사용자 입력값' 유효성 검사

- [UserBean 데이터 객체] 의 필드에 **@Size, @Pattern 으로 유효성 제한**을 걸어준다.

- [UserController 컨트롤러]

: 사용자가 join.jsp 폼에 입력한 값은 @ModelAttribute 파라미터로 컨트롤러에 들어오게 되는데, 이때 @Valid를 붙여서 폼에 입력된 회원가입 User 객체는 파라미터로 들어온 동시에 유효성 검사를 하도록 설정한다.

- [UserValidator 객체]

: 이 validator 객체는 validator를 구현한 객체로, 내부에 validate() 메소드를 재정의한다.
: @Valid로 파라미터값에 대한 유효성 검사를 할 때, 동시에 validate() 메소드 내부가 실행되어 별도의 유효성 검사를 처리하는데, 여기서 target 객체로 받은 User 객체의 실질적인 유효성 검사를 실시한다.

▶ 회원 가입할 아이디 '중복' 확인 : [DB 상에 이미 존재하는 회원인지 여부 확인]

-[UserDao]

: 중복되는 id 데이터를 가져올 수 있도록 'select 쿼리문' 작성한다.
: 사용자가 입력한 id 값을 읽어들이면 해당 id가 존재하는 user_name을 가져와야 한다.

```
@Select("SELECT user_name FROM user_table WHERE user_id = #{user_id}")
String checkUserIdExist(String user_id);
```

-[UserService] 서비스 : DB 상에 중복 id 있는지 checkUserIdExist()로 확인

```
	//아이디 중복 확인 체크
	public boolean checkUserIdExist(String user_id) {
		//들어온 id 값과 일치하는 데이터 존재여부를 매퍼로 접근해서 가져옴
		String user_name = userDao.checkUserIdExist(user_id);

		//DB 상에 해당 데이터 없다면 = 중복X
		if(user_name == null) {
			return true; //T 반환
		}else { //DB 상에 데이터 이미 존재하면 중복 O
			return false;
		}
	}
```

-[UserController] 컨트롤러 : 폼에 입력한 가입정보를 얻어옴

▶ 사용자가 폼에 입력한 'User 정보'를 DB 에 저장

-[UserDao]

: 사용자가 form에 입력한 회원가입 User 객체를 DB 상에 Insert 처리할 쿼리문 작성

```
@Insert("INSERT INTO user_table (user_name, user_id, user_pw)
   VALUES ( #{user_name},#{user_id},#{user_pw} )")
void addUserInfo(UserBean joinUserBean);
```

-[UserService 서비스]

: UserDao를 매개로 User객체 담을 서비스 작성

```
//가입한 회원정보 DB에 저장
public void addUserInfo(UserBean joinUserBean) {
        userDao.addUserInfo(joinUserBean);
}
```

-[UserController 컨트롤러]

: User객체 입력받는 폼 주소 매핑 시, 입력값을 파라미터로 받아서 서비스 객체로 주입함

```
@PostMapping("/join_pro") //폼에 입력한 값 객체로 가져와서 유효성 검사하여 받음
public String join_pro(@Valid @ModelAttribute("joinUserBean") UserBean joinUserBean, BindingResult
result)
{
        if(result.hasErrors()) {
                return "user/join";
        }
        userService.addUserInfo(joinUserBean); //서비스로 보냄

        return "user/join_success";
}
```

2) 로그인 기능

- 사용자가 폼에 입력한 id/pw 값과 일치하는 User 데이터를 DB 상에서 select 하여 로그인 User가 기존 회원인지 확인 후, 로그인 성공/실패 처리를 하는 기능이다.
- 로그인 User에 한해서만 접근 가능한 페이지를 작성하여 보안 기능을 강화했다.

- [UserDao] 쿼리문 작성

: 사용자가 로그인한 id/pw값과 일치하는 데이터를 DB 상에서 select 처리할 쿼리문

```
//로그인 기능 - 사용자가 입력한 id,pw 정보기준으로 name select하여 UserBean 객체로 반환하는 쿼리문
@Select("SELECT user_name FROM user_table WHERE user_id = #{user_id} AND user_pw = #{user_pw}")
UserBean getLoginUserInfo(UserBean tempLoginUserBean);
```

▶사용자가 입력한 ID/PA의 유효성 검사

- [UserController] 컨트롤러

: 사용자가 (.jsp) 속 로그인 form 에 입력한 id/pw 값은 @ModelAttribute를 통해 UserBean 객체 단위의 파라미터로 컨트롤러에 들어올 것이다. 해당 파라미터 객체의 값에 대해 @Valid 유효성 검사 후 이상 없으면 입력받은 로그인 User객 체의 정보와 일치하는 데이터 존재 확인을 위해 UserService를 내부적으로 호출한다.

: 만약 DB 상에 로그인User 의 id/pw와 일치하는 데이터 존재할 경우 로그인 성공, 아닌 경우 로그인 실패 처리를 한다.

- [UserService] 서비스

: 컨트롤러에서 받은 로그인 User 객체 정보를 다시 UserDao에게 넘긴다. UserDao는 쿼리문으로 해당 객체와 일치하는 데이터 select 해온다.

▶로그인한 User에 한해서 접근 허용 처리 : 정보수정/로그아웃

-[loginUserBean] 빈을 RootAppContext에 @SessionScope 으로 등록한다.

: Bean을 @SessionScope로 정의했기 때문에 브라우저가 최초의 요청을 할 때 주입된다.

: 한 번 주입된 객체를 계속 사용하여 로그인 여부에 따라 유효 페이지 작성이 가능해짐 O

-> Session 영역에 로그인 정보 저장

-[CheckLoginInterceptor] 인터셉터

: HandlerInterceptor 인터페이스 구현한 객체

: 내부에 **preHandle() 메소드를 재정의**: 여기서 현재 경로 -> not_login 으로 경로 재설정

: 이 인터셉터를 ServletAppContext에 addInterceptor() 메소드로 등록함

```
//인터셉터 등록하기 - 재정의
@Override
public void addInterceptors(InterceptorRegistry registry) {
		// TODO Auto-generated method stub
		WebMvcConfigurer.super.addInterceptors(registry);
		CheckLoginInterceptor checkLoginInterceptor = new CheckLoginInterceptor(loginUserBean);
		InterceptorRegistration reg2 = registry.addInterceptor(checkLoginInterceptor);
		reg2.addPathPatterns("/user/modify", "/user/logout");
                                                    //이 경로에 거칠 경우 인터셉터 거치도록함
		reg2.excludePathPatterns("/board/main"); //이 경로는 제외하고 인터셉터 겨치도록함
		}
```

3) 회원정보 수정 기능 -> 비밀번호 변경 기능

- 로그인한 User에 한해서만 회원정보 수정 페이지에 접근이 가능하다.
- 현재 로그인한 (세션영역) User 객체의 id 값을 기준으로 DB 속 회원의 id, name 데이터를 가져와 회원 정보 폼에 일부 띄워주고, 회원은 password 값을 변경할 수 있다.
- 폼에서 변경한 값들은 다시 DB 로 '현재 로그인 USER'와 동일한 idx 행에 저장된다.

▶ 현재 로그인 사용자의 '일부 정보' 띄워주기 (name, id값)

-[UserDao] 쿼리문 작성

: jsp 파일 form의 수정 정보 객체 정보를 가져와서 나머지 select 처리할 쿼리문 작성

```
//정보수정 시  - 현재 로그인한 사용자의 id 값 기준으로 로그인 사용자 정보 일부를 가져옴
@Select("SELECT user_id, user_name FROM user_table WHERE user_id = #{user_id}")
UserBean getModifyUserInfo(String user_id);
```

-[UserService] 서비스 작성

: 수정할 객체에 현재 로그인한 사용자의 객체값을 기준으로 수정할 객체를 세팅해줌

: 수정할 객체와 현재 로그인한 객체는 동일한 DB에 저장되어야 하므로 현재 로그인 User객체의 idx, id, name값을 수정할 객체의 값에 일치시켜준다.

-[UserController] 컨트롤러 작성

: 수정 form에서 얻은 객체를 받아서 UserService의 getModifyUserInfo() 메소드에 보낸다.

: 그렇게 되면 수정 페이지의 수정 객체는 Service에서 현재의 로그인 사용자 객체의 정보를 기반으로 세팅되고 그 정보가 폼에 입력된다.

▶ 사용자가 변경한 비밀번호를 다시 DB 에 UPDATE 처리

-[UserDao] 쿼리문 작성

: jsp 파일에는 현재 로그인한 사용자의 name, id 값이 올라와있고, 사용자는 비밀번호만 변경 가능하다.

: 따라서 사용자 id 값과 일치하는 password 값을 SET 처리하여 UPDATE 하는 쿼리문

```
//정보수정 처리
@Update("UPDATE user_table SET user_pw = #{user_pw} WHERE user_id = #{user_id}")
void modifyUserInfo(UserBean modifyUserBean);
```

-[UserService] 서비스

: 폼에 입력된 수정객체는 현재 로그인 객체의 idx값과 일치하는 idx값을 갖도록 처리해주고 (그래야 DB 상 동일한 idx 값에 저장되므로) userDao를 토대로 해당 수정객체를 DB 에 업데이트 처리해준다.

-[UserController] 컨트롤러

: 폼에 입력한 수정 객체는 @ModelAttribute 파라미터로 받아서 UserService에 보낸다.

4) 로그아웃 기능

-컨트롤러에서 (세션 영역) Login 정보 객체의 UserLogin 정보를 false 처리해주면 해당 User는 로그아웃 처리가 완료된다.

>> 물품 관리 기능 (등록, 상품명으로 검색, 목록출력)

▷**[물품 데이터 구조]**

- content_idx 는 저장 시 자동 증가되도록 **auto_increment 로 선언**하여 만들었다.
- content_date 는 상품입고(추가) 처리 시, **자동으로 '현재 시간'을 defualt 값**으로 담도록 생성했다.

Content_table 정보

Content_idx	Content_date	Content_name	Content_price	Content_count	Content_board_idx

1) 상품 등록 기능

-사용자가 뷰의 폼에 상품명, 상품코드, 상품가격, 상품수량을 입력하면, 해당 상품 정보 객체를 ModelAttribute로 받아서 DB 상에 값을 Insert 시켜주는 기능이다.

-[ContentBean] : 데이터 객체 생성

: DB의 상품 테이블과 동일한 필드값 작도록 생성했다.

-[BoardDao] : Insert 쿼리문 작성

: DB 상에 물품 정보를 삽입 처리할 쿼리문을 작성했다.

: idx, date 값은 자동 처리되기 때문에 상품명/상품코드/가격/수량 정보를 삽입한다.

: **JdbcTemplate.update() 사용하여 DB에 상품 정보 등록**한다.

```
// 사용자가 쓴 입고 물품 쓰기
public void addContentInfo(ContentBean writeContentBean) {
    String sql = "INSERT INTO content_table(content_name,content_price, content_count, content_board_idx) VALUES (?, ?, ?, ?)";
    jdbcTemplate.update(sql, writeContentBean.getContent_name(), writeContentBean.getContent_price(),
    writeContentBean.getContent_count(), writeContentBean.getContent_board_idx());
}
```

-[BoardService] : 서비스

: DB에 삽입할 상품 객체를 컨트롤러로부터 받아서 BoardDao에게 보내고, Dao의 반환값을 다시 컨트롤러에게 보낸다.

-[MenuController] : 컨트롤러

: 해당 경로를 매핑하는 컨트롤러에서 사용자가 뷰 폼에 입력해놓은 '입고 상품 정보 객체'를 @ModelAttribute 파라미터로 받고, 내부적으로 유효성 검사를 @Valid 로 거친 뒤, 문제없으면 서비스 객체를 호출하여 입고 상품 객체를 보낸다.

2) '상품명'으로 검색 기능

- 사용자가 폼에 '상품명'을 입력하면, DB 상에 해당 상품명을 갖는 데이터 묶음을 가져와 뷰에 다시 출력시켜주는 기능이다.

-[BoardController] 컨트롤러

: 사용자가 폼에 입력한 '상품명'은 해당 경로를 매핑하는 컨트롤러에서 @ModelAttribute (SearchBean객체) 파라미터로 받아서, 만약 해당 객체에 getName() 처리 시 null 이 아니라면, BoardService에서 해당 객체에 입력된 '상품명'을 넣고 List타입으로 해당 상품 데이터 묶음을 받는다. 이 데이터를 뷰에 전달해야 하므로 Model 객체에 add 처리해준다. 뷰에서는 이 정보들을 나열하여 사용자에게 보여준다.

-[BoardService] 서비스

: 컨트롤러에서 보낸 객체의 이름값을 받아서 다시 BoardDao에게 전달해주고, Dao의 쿼리 반환값을 다시 컨트롤러에게 전달해준다.

-[BoardDao] 쿼리문 작성

: 서비스에서 보낸 객체의 이름값과 동일한 데이터를 DB 상에서 탐색한 뒤, 상품명, 상품가격, 상품수량 등의 정보를 List 타입으로 받는다.

3) 모든 상품목록을 뷰에 출력하는 기능

-사용자가 '재고관리' 메뉴를 클릭한 경우, DB 상에 저장된 모든 상품목록을 Select해서 뷰에 모든 목록을 출력해주는 기능이다.

-[boardDao] 쿼리문 작성

: 모든 상품목록을 얻기 위해, 메뉴 인덱스 값을 기준으로 상품목록의 모든 데이터 값을 Select하여 List<ContentBean> 타입으로 받는다.

```
// 상품 재고 목록 가져오기
public List<ContentBean> getContentList(int board_info_idx) {
  String sql = "SELECT content_idx, content_date as content_date, content_name, content_price, content_count FROM content_table WHERE content_board_idx = ? ORDER BY content_idx DESC ";

  List<ContentBean> contentBean = jdbcTemplate.query(sql, boardMapper, board_info_idx);

  return contentBean;
}
```

-[boardService] 서비스

: 컨트롤러에게 받은 메뉴 인덱스 번호를 다시 Dao에게 보내주고, Dao가 반환한 List타입 데이터를 다시 컨트롤러에게 보내주는 역할을 한다.

```
public List<ContentBean> getContentList(int board_info_idx) {
        return boardMapper.getContentList(board_info_idx);
}
```

-[boardController] 컨트롤러

: 사용자가 '메뉴' 버튼에서 클릭한 idx 정보가 @RequestParam으로 컨트롤러에 넘어온다. 해당 값은 메뉴의 인덱스 정보로서, 이 정보를 기준으로 사용자에게 출력되는 내용물이 달라야 한다. 상품목록을 모두 보여주기 위해서 우선 해당 idx 정보를 서비스에 보내어 모든 상품

목록을 Select 처리하여 List 타입으로 받는다. 받은 값을 뷰에 출력해주어야 하므로 Model 타입 객체에 'contentList' 라는 이름으로 addAttribute() 처리해준다. 뷰에서는 Model에 전달된 값을 기준으로 전체 상품 목록을 출력시켜준다.

>> 판매 기능 (계산, 저장, 목록출력)

▷[판매 정보 데이터 구조]

-selling_idx 값은 저장 시 자동 증가 처리되도록 auto_increment 로 선언했다.

-selling_date 값은 판매 정보 저장 시의 시간을 자동으로 담도록 default 값을 넣었다.

판매정보 selling_table 정보

Selling_idx	Selling_date	Selling_name	Selling_price	Selling_board_idx

1) 사용자가 [상품명, 수량] 입력 시 -> 계산 기능

- 사용자가 폼에서 구매할 상품명과 수량을 입력하면, 물품 목록 테이블에서 입력된 상품명과 일치하는 데이터의 'price' 정보를 Select 해온다. 컨트롤러에서는 사용자가 입력한 수량과 가져온 가격 정보를 곱한 뒤, 다시 뷰에 계산 결과를 보여준다.

-[boardDao] 쿼리문 작성

: 서비스에게 받은 '상품명' 이름을 기준으로 물품 테이블에서 **'가격' 정보만 가져와서 반환.**

: 한 행만 가져올 것이기 때문에 **queryForObject() 사용**

```
// '상품명 판매 계산 기능' 사용자가 판매 상품명 입력 -> 가격 가져오기
public int getSellPrice(String content_name) {
   String sql = "SELECT content_price FROM content_table WHERE content_name = ?";
   int price = jdbcTemplate.queryForObject(sql, Integer.class, content_name);
   return price;
}
```

-[boardService] 서비스

: 컨트롤러에게 받은 '상품명' 이름을 다시 Dao에게 보내서 가격 정보를 얻어온다.

```
public int getSellPrice(String content_name) {
        return boardDao.getSellPrice(content_name);
}
```

-[boardController] 컨트롤러

: 사용자가 폼에서 입력한 '상품명, 수량' 정보가 저장된 객체를 @ModelAttribute 파라미터로 받은 뒤, 해당 정보가 null이 아니라면 Service에게 '상품명'을 보내서 해당 물품의 '1개' 가격을 얻어온다. 얻어온 price 값에 사용자가 입력한 count 수량 정보를 곱한 뒤, 곱한 가격 정보를 Model 객체에게 add 처리하여 뷰에서 model에게 준 계산 결과값을 출력할 수 있도록 처리한다.

2) 판매 정보를 DB 에 저장하는 기능

- 이 기능은 사용자가 계산과 동시에 판매 정보를 selling_table DB 테이블에 저장하도록 하는 기능이다. 실제로 1)의 계산과 동시에 판매 정보를 저장하도록 구현해놓았다.

-[boardController] 컨트롤러

: 사용자가 '상품명', '수량'을 입력하여 총 판매 금액을 계산했던 경로와 동일한 경로의 컨트롤러 메소드에서 사용자가 입력한 '상품명' 정보와 계산을 마친 총 판매금액 정보를 곧장 판매 정보 객체인 SellingBean 타입에 set 처리해준다. 판매 정보 세팅이 완료된 판매 객체를 다시 서비스에게 보내서 DB의 selling_table 속에 저장을 시도한다.

-[boardService] 서비스

: 컨트롤러에서 판매 정보 세팅 후 보낸 판매 정보 객체를 다시 boardDao에게 보낸다.

-[boardDao] 쿼리문 작성

: 서비스에게 받은 판매 정보 객체의 모든 필드값을 DB상의 selling_table에 INSERT 처리하도록 쿼리문을 작성했다.

```
// 판매 정보 저장하기
public void addSellingInfo(SellingBean sellingInfo) {
   String sql = "INSERT INTO selling_table(selling_name, selling_price) VALUES (?, ?)";
   jdbcTemplate.update(sql, sellingInfo.getSelling_name(), sellingInfo.getSelling_price());
}
```

3) 저장된 판매 정보 목록을 뷰에 출력해주는 기능

- 이 기능은 사용자가 '판매 기능' 메뉴를 클릭 시, (2)에서 판매 처리 후 DB 상에 저장된 모든 판매 정보 목록을 다시 뷰에 출력시켜주는 기능이다.

-[boardDao] 쿼리문 작성

: 우선 모든 판매 정보를 불러오도록 SELECT 문을 작성하여 전체 판매 데이터 목록을 List 타입으로 받는다.

```
//판매 정보 목록 모두 가져오기
public List<SellingBean> getSellingInfo() {
   String sql = "SELECT * FROM selling_table ORDER BY selling_idx DESC";
   List<SellingBean> list = jdbcTemplate.query(sql, sellingMapper);
   return list;
}
```

-[boardService] 서비스

: 컨트롤러로부터 요청받은 서비스는 다시 Dao에게 전체 판매 목록 데이터를 가져오도록 내부적으로 호출한다. 이후 Dao가 전달한 List 값은 다시 컨트롤러에게 보내준다.

```
//모든 판매 목록 가져오기
public List<SellingBean> getSellingInfo() {
        return boardDao.getSellingInfo();
}
```

-[boardController] 컨트롤러

: board/main 경로를 매핑하는 메소드에서 만약 메뉴 인덱스값이 2인 경우에 한해, Service에게 모든 판매 정보 목록 데이터를 List 타입으로 받고, 이를 뷰에 출력시켜주기 위해 Model 타입 객체에 addAttribute() 처리해준다.

>> 판매통계 기능 (최근 일주일, 최근 한달, 지정 기간별 총판매 통계 출력)

1) 최근 일주일 판매 통계 기능

: 이 기능은 사용자의 현재 시간값 기준으로 최근 일주일 기간의 판매 정보를 SELECT 해와서 판매 가격을 누적 합 시켜준 뒤, 총 판매액을 사용자에게 출력시켜주는 기능이다.

[boardDao] 쿼리문 작성

: 사용자가 통계 기능을 알고 싶어하는 '현재' 날짜 값을 Now() 값으로 얻어서 해당 날짜로부터 일주일 전 기간 사이에 판매된 판매 정보들을 불러와야 한다. 따라서 다음과 같이 쿼리문을 작성했다. 쿼리 결과는 List<SellingBean> 타입으로 반환받는다.

```
// '최근 일주일' 통계 정보 가져오기
public List<SellingBean> getOneWeekSellingInfo() {
  String sql = "SELECT * FROM selling_table WHERE selling_date BETWEEN
DATE_ADD(NOW(), INTERVAL -1 WEEK ) AND NOW() ORDER BY selling_date ASC";
  List<SellingBean> list = jdbcTemplate.query(sql, sellingMapper);

  return list;
}
```

[boardService] 서비스

: 컨트롤러에게 호출받은 서비스는 다시 Dao에게 동일한 요청을 내부적으로 보낸다. 또한, Dao가 쿼리문 실행 결과를 반환하면 서비스는 이를 받아서 다시 컨트롤러에게 돌려준다.

```
//최근 일주일 판매 정보 가져오기
public List<SellingBean> getOneWeekSellingInfo() {
        return boardDao.getOneWeekSellingInfo();
}
```

[boardController] 컨트롤러

: 최근 일주일 통계는 ("board/onWeekTotal") 경로에서 확인할 수 있도록 처리했다. 컨트롤러에서는 해당 경로에 대하여 @RequestMapping("/oneWeekTotal") 메소드로 처리하는데, Model 타입 객체를 파라미터로 받아서 뷰에 출력시켜줄 데이터를 세팅해준다.

: 우선 이 컨트롤러 안에서 최근 일주일 간의 판매목록에 대한 select 쿼리결과를 List 타입으로 받아둔다. for문을 돌면서 해당 List에 담겨진 각각의 판매 정보 객체를 얻고, price 값에 대해 누적합을 처리해준다. 그렇게 되면 해당 기간의 '총판매액'이 price값에 저장되므로 이 값을 model 객체에 addAttribute() 처리해주어 뷰에 출력시켜줄 수 있도록 처리했다.

: 또한 해당 기간의 판매 목록을 뷰에도 출력시켜주기 위해 아까 반환받았던 List 타입의 쿼리 결과를 또 다시 model 객체에 add 처리시켜주어 뷰에 출력시켜줄 수 있게 처리했다.

[oneWeekTotal.jsp] 뷰

: 이 jsp 파일에서 컨트롤러가 처리 결과로 보내준 데이터를 ${ }를 토대로 출력시켜준다.

2) 최근 한달 판매 통계 기능

: 이 기능은 사용자의 현재 시간값 기준으로 최근 한달 기간의 판매 정보를 SELECT 해와서 판매 가격을 누적 합 시켜준 뒤, 총 판매액을 사용자에게 출력시켜주는 기능이다.

[boardDao] 쿼리문 작성

: 사용자가 통계 기능을 알고 싶어하는 '현재' 날짜 값을 Now() 값으로 얻어서 해당 날짜로부터 '한달' 전 기간 사이에 판매된 판매 정보들을 불러와야 한다. 따라서 다음과 같이 쿼리문을 작성했다. 쿼리 결과는 List<SellingBean> 타입으로 반환받는다.

```
// '최근 한달' 통계 정보 가져오기
public List<SellingBean> getOneMonthSellingInfo() {
   String sql = "SELECT * FROM selling_table WHERE selling_date BETWEEN DATE_ADD(NOW(),
INTERVAL -1 MONTH ) AND NOW() ORDER BY selling_date ASC";
   List<SellingBean> list = jdbcTemplate.query(sql, sellingMapper);
   return list;
}
```

[boardService] 서비스

: 컨트롤러에게 호출받은 서비스는 다시 Dao에게 동일한 요청을 내부적으로 보낸다. 또한, Dao가 쿼리문 실행 결과를 반환하면 서비스는 이를 받아서 다시 컨트롤러에게 돌려준다.

```
//최근 한달 판매 정보 가져오기
public List<SellingBean> getOneMonthSellingInfo(){
          return boardDao.getOneMonthSellingInfo();
}
```

[boardController] 컨트롤러

: 최근 일주일 통계는 ("board/oneMonthTotal") 경로에서 확인할 수 있도록 처리했다. 컨트롤러에서는 해당 경로에 대하여 @RequestMapping("/oneMonthTotal") 메소드로 처리하는데, Model 타입 객체를 파라미터로 받아서 뷰에 출력시켜줄 데이터를 세팅해준다.

: 우선 이 컨트롤러 안에서 최근 한달 간의 판매목록에 대한 select 쿼리결과를 List 타입으로 받아둔다. for문을 돌면서 해당 List에 담겨진 각각의 판매 정보 객체를 얻고, price 값에 대해 누적 합을 처리해준다. 그렇게 되면 해당 기간의 '총판매액'이 price값에 저장되므로 이 값을 model 객체에 addAttribute() 처리해주어 뷰에 출력시켜줄 수 있도록 처리했다.

: 또한 해당 기간의 판매 목록을 뷰에도 출력시켜주기 위해 아까 반환받았던 List 타입의 쿼리 결과를 또 다시 model 객체에 add 처리시켜주어 뷰에 출력시켜줄 수 있게 처리했다.

[oneMonthTotal.jsp] 뷰

: 이 jsp 파일에서 컨트롤러가 처리 결과로 보내준 데이터를 ${ }를 토대로 출력시켜준다.

3) 사용자 지정 기간 판매 통계 기능

: 이 기능은 사용자가 폼에 입력한 start 기간과 end 기간을 받아서 해당 기간의 판매 정보를 SELECT 해온 뒤, 판매 가격을 누적 합 시켜 총 판매액을 사용자에게 출력시켜주는 기능이다. 이 기능에서는 폼에 사용자가 입력한 start 값과 end 값을 받아서 처리해야 하기 때문에 별도의 DayCommand라는 '커맨드 객체'를 새롭게 정의하여 사용자 입력값을 받았다.

[DayCommand] 커맨드 객체

: 사용자가 View의 폼에 입력한 지정 날짜값을 얻을 용도로 커맨드 객체를 하나 생성했다.
이 객체는 내부 필드로 start 날짜와 end 날짜를 갖는다.

[BoardDao] 쿼리문 작성

: 사용자가 폼에 입력한 지정 기간(start 날짜, end 날짜) 값 사이에 들어있는 모든 판매 목록을 Select 해서 LIst<SellingBean> 타입으로 반환시킬 쿼리문을 작성한다.

```
// 지정날짜 판매 정보 가져오기
public List<SellingBean> getTotalInfo(String start, String end) {
  String sql = "select * from selling_table where selling_date between date(?) and date(?) +1  order by selling_date ASC";
  List<SellingBean> list = jdbcTemplate.query(sql, sellingMapper, start, end);
  return list;
}
```

[BoardService] 서비스

: 컨트롤러가 요청한 쿼리 결과를 다시 Dao에게 요청하고, Dao가 쿼리 결과로 얻어온 List<SellingBean> 타입 결과를 서비스가 받아서 다시 컨트롤러에게 보내주는 역할을 한다.

```
//날짜 조회해서 가져오기
public List<SellingBean> getTotalInfo(String start, String end){
        return boardDao.getTotalInfo(start, end);
}
```

[BoardController] 컨트롤러

:컨트롤러에서 @ModelAttribute 로 사용자가 입력한 '지정 날짜' 값이 들어있는 커맨드 객체를 받아서 해당 값에 대한 쿼리 결과를 서비스 객체를 통해 얻어서 List<SellingBean> 타입으로 받는다. 받은 List를 for문 차례로 돌면서 각 판매 정보 객체의 price 값을 누적합 시킨다. 그렇게 얻은 사용자 지정 기간의 '총판매 금액'을 model 타입에게 보내주어 뷰에서 지정 기간에 대한 총 판매액을 확인할 수 있게 처리해준다.

4) 각 기간별 사용자 판매 목록 (판매날짜, 판매가격) 정보 목록을 바로 보여주는 기능

: 이 기능은 각각의 기간별 판매 통계를 사용자가 요청할 때, 해당 기간의 판매 정보 '목록'도 함께 출력시켜주는 기능이다.

[BoardController] 컨트롤러

: 1~3 기능에서 각각 기간별 통계 시 얻어줬던 기간별 판매 목록 List 타입을 각 컨트롤러 내부에서 Model 타입으로 뷰에 전달해준다.

[oneWeekTotal / oneMonthTotal / Total] 뷰 - JSP 파일

뷰에서는 해당 값을 <c:forEach> 태그로 for 문을 돌면서 차례로 뷰에 '판매 목록'을 출력시켜주도록 구현해주었다.