# 1. @Transaction 어노테이션을 Controller에서 Service(Model 계층)로 이전

## 1-1. 적용 이유

: 트랜잭션의 속성 중 하나인 원자성(Atomicity)은 한 명령에서 여러 데이터의 변경사항이 있을때 하나라도 오류가 발생하면 이를 취소하고 다시 명령전으로 되돌린다. 이를 이용하여 Controller단에서 Transaction 어노테이션을 적용했으나 추후 이러한 특성을 고려했을 때 데이터의 흐름 방식을 가공하는 Service라는 Model 계층이 더 부합하다고 판단하여 변경.

## 1-2. 적용전 예시

```
/* AdminController에 있는 메소드 */

// 변경 전
@PostMapping("/mypage/regist/seller")
@Transactional(rollbackFor = Exception.class)
public ModelAndView registing_Seller(@ModelAttribute MbrShippingVO
mbrShippingVO, ModelAndView mav) throws Exception {
    log.info("registing seller page........");

    ResponseEntity<String> entity = null;
    mbrShippingVO.setMbr_nickname(mbrShippingVO.getMbr_name());
    log.info("rest_update..");

    try {
        adminService.addSeller(mbrShippingVO);
        log.info("update seller info");
        entity = new ResponseEntity<String>("SUCCESS", HttpStatus.OK);
        mav.setViewName("redirect:/admin/mypage/seller");
    } catch (Exception e) {
        e.printStackTrace();
        entity = new ResponseEntity<String>(e.getMessage(),
HttpStatus.BAD_REQUEST);
    }

    return mav;
}

// 변경 후
@PostMapping("/mypage/regist/seller")
public ModelAndView registing_Seller(@ModelAttribute MbrShippingVO
mbrShippingVO, ModelAndView mav) throws Exception {
    log.info("registing seller page........");

    ResponseEntity<String> entity = null;
    mbrShippingVO.setMbr_nickname(mbrShippingVO.getMbr_name());
    log.info("rest_update..");

    try {
        adminService.addSeller(mbrShippingVO);
        log.info("update seller info");
        entity = new ResponseEntity<String>("SUCCESS", HttpStatus.OK);
```

```
            mav.setViewName("redirect:/admin/mypage/seller");
        } catch (Exception e) {
            e.printStackTrace();
            entity = new ResponseEntity<String>(e.getMessage(),
HttpStatus.BAD_REQUEST);
        }

        return mav;
    }
```

### 1-3. 적용후 예시

```
/* AdminServiceImple에 있는 메소드 */

// 변경 전
@Override
public void addSeller(MbrShippingVO mbrShippingVO) {
    log.info("addSeller");

    String pw = passEncoder.encode(mbrShippingVO.getMbr_pw());
    mbrShippingVO.setMbr_pw(pw);

    adminMapper.addSellerInfo(mbrShippingVO);
    adminMapper.addSellerAddress(mbrShippingVO);
}

// 변경 후
@Override
@Transactional(rollbackFor = Exception.class)
public void addSeller(MbrShippingVO mbrShippingVO) {
    log.info("addSeller");

    String pw = passEncoder.encode(mbrShippingVO.getMbr_pw());
    mbrShippingVO.setMbr_pw(pw);

    adminMapper.addSellerInfo(mbrShippingVO);
    adminMapper.addSellerAddress(mbrShippingVO);
}
```

## 2. 불필요한 파라미터 제거

### 2-1. 적용 이유

: 소스코드 관리에 있어서 다른 인원과의 협업 가능성을 두었을 때 어떤 파라미터에 의해 데이터가 흐르는지 파악할 수 있는 여건이 필요하다. 그러기에 실제로 사용되지 않는 파라미터들을 제거하여 혼란 여지를 주지 않도록 고려한다.

### 2-1. 적용전 예시

```
// 공지사항 작성(관리자)
// 파라미터에 있는 ModelAndView 타입의 mav는 해당 메소드에서 결과적으로 사용되지 않아 삭제하여 코드를 리팩토링 해야 한다
@PostMapping("/admin/board/notice/write")
```

```
public ResponseEntity<String> noticeWrite(@RequestBody BoardVO boardVO,
ModelAndView mav) {
    ResponseEntity<String> entity = null;

    log.info("noticeWrite..");
    try {
        boardService.setNoticeWrite(boardVO);
        entity = new ResponseEntity<String>("SUCCESS", HttpStatus.OK);
    } catch (Exception e) {
        e.printStackTrace();
        entity = new ResponseEntity<String>(e.getMessage(),
HttpStatus.BAD_REQUEST);
    }

    return entity;
}
```

## 2-2. 적용후 예시

```
@PostMapping("/admin/board/notice/write")
public ResponseEntity<String> noticeWrite(@RequestBody BoardVO boardVO) {
    ResponseEntity<String> entity = null;

    log.info("noticeWrite..");
    try {
        boardService.setNoticeWrite(boardVO);
        entity = new ResponseEntity<String>("SUCCESS", HttpStatus.OK);
    } catch (Exception e) {
        e.printStackTrace();
        entity = new ResponseEntity<String>(e.getMessage(),
HttpStatus.BAD_REQUEST);
    }

    return entity;
}
```