AdminController.java(컨트롤러)
PostService.java(서비스)
PostDAO.java(다오)
PostMapper.xml(매퍼)
Board.html(화면)
의 관계

xml 와 dao 관계:

---

1 전체 구조 한눈에 보기

MyBatis에서는 보통 이렇게 나뉩니다.

```scss
[Controller / Service]
        ↓
     DAO (인터페이스)
        ↓   (연결)
   Mapper XML
        ↓
     Database
```

- **DAO**: 자바 코드에서 DB 작업을 호출하는 창구
- **XML Mapper**: 실제 SQL이 작성되는 곳
- **MyBatis**: DAO와 XML을 자동으로 연결해주는 중간 다리

---

2 XML 파일의 역할

📌 XML은 "SQL 저장소" 역할

```xml
<mapper namespace="kr.hi.boot.dao.PostDAO">
```

👉 이 부분이 **가장 중요**합니다.

- `namespace`는 **DAO 인터페이스의 전체 경로와 반드시 같아야 함**
- 이걸로 MyBatis가
  👉 "아, 이 XML은 `PostDAO`랑 연결되는구나" 라고 인식함

📌 select 태그 분석

```xml
<select id="getBoardList" resultType="Board">
    select * from board
</select>
```

요소	의미
`id`	DAO 메서드 이름과 **동일해야 함**
`resultType`	SQL 결과를 어떤 객체로 담을지
SQL	실제 실행될 쿼리

즉,

```java
getBoardList()
```

를 호출하면

👉 이 SQL이 실행됨

---

3 DAO 인터페이스의 역할

```java
public interface PostDAO {

    ArrayList<Board> getBoardList();

}
```

📌 DAO의 특징

- **구현 클래스가 없다**
- SQL도 없다
- 메서드 선언만 있음

👉 그럼 누가 구현하냐?

✔ **MyBatis가 런타임에 자동으로 구현**

4️⃣ XML ↔ DAO 연결 규칙 (시험·면접 단골)

✅ 반드시 지켜야 할 3가지 규칙

① namespace = DAO 전체 경로

xml Copy code

```xml
<mapper namespace="kr.hi.boot.dao.PostDAO">
```

java Copy code

```java
package kr.hi.boot.dao;
public interface PostDAO { ... }
```

✔ 패키지 + 인터페이스 이름 완전히 일치해야 함

---

② XML의 id = DAO 메서드명

xml Copy code

```xml
<select id="getBoardList">
```

java Copy code

```java
ArrayList<Board> getBoardList();
```

✔ 이름이 다르면 실행 안 됨 (에러 발생)

---

③ resultType = 반환 타입

xml Copy code

```xml
resultType="Board"
```

java Copy code

```java
ArrayList<Board>
```

- SQL 결과 한 행 → `Board`
- 여러 행 → `ArrayList<Board>`

---

5 실제 실행 흐름 (아주 중요)

java Copy code

```java
@Autowired
PostDAO postDAO;

List<Board> list = postDAO.getBoardList();
```

이 코드가 실행되면 내부적으로:

1 `getBoardList()` 호출
2 MyBatis가 `PostDAO`와 연결된 XML 찾음
3 `id="getBoardList"` 인 SQL 찾음
4 `select * from board` 실행
5 결과를 `Board` 객체로 변환
6 `ArrayList<Board>`로 반환

👉 **개발자는 SQL 호출 코드를 직접 작성하지 않음**

---

6 왜 이런 구조를 쓰는가?

✔ 장점

✅ SQL과 자바 코드 분리
✅ 유지보수 쉬움
✅ 인터페이스 기반 → 결합도 낮음
✅ XML만 수정해도 SQL 변경 가능

---

7 흔한 실수 TOP 3

❌ namespace 오타
❌ id 이름 불일치
❌ resultType 패키지 경로 누락

예:

xml Copy code

```xml
resultType="kr.hi.boot.model.vo.Board"
```

(설정에 따라 풀패키지 필요할 수도 있음)

---

8 한 문장으로 정리

> DAO는 메서드만 선언하고,
> XML은 SQL을 작성하며,
> MyBatis가 namespace + id를 기준으로 둘을 자동 연결한다.

Service 와 dao 관계

1️⃣ Service와 DAO의 역할 차이

✅ DAO (Data Access Object)

java

Copy code

```java
public interface PostDAO {
    ArrayList<Board> getBoardList();
}
```

📌 역할

- DB와 직접 소통하는 계층
- SQL 실행 책임
- **"데이터 접근만 담당"**

📌 특징

- DB 구조에 가장 가까움
- MyBatis XML과 1:1로 연결됨
- 비즈니스 로직 ❌

✅ Service

```java
@Service
public class PostService {

    @Autowired
    PostDAO postDAO;

    public ArrayList<Board> getBoardList() {
        return postDAO.getBoardList();
    }
}
```

📌 **역할**

- 비즈니스 로직 담당
- 여러 DAO를 조합할 수 있음
- Controller와 DAO 사이의 **중간 관리자**

📌 특징

- 트랜잭션 처리 위치
- 로직 변경이 가장 많이 일어나는 계층
- DB 구조를 직접 알 필요 없음

2️⃣ Service와 DAO의 관계 (한 문장)

> **Service는 "무엇을 할지"를 결정하고,**
> **DAO는 "어떻게 DB에서 가져올지"만 담당한다.**

3️⃣ @Autowired의 의미 (핵심)

```java
@Autowired
PostDAO postDAO;
```

이 한 줄에서 벌어지는 일

1️⃣ Spring이 실행되면서
2️⃣ `PostDAO` 타입의 Bean을 찾음
3️⃣ MyBatis가 만들어준 DAO 구현체를 발견
4️⃣ `postDAO` 변수에 자동 주입 (DI)

👉 **Service는 DAO 구현체를 전혀 몰라도 됨**

---

4 전체 실행 흐름 (실무에서 매우 중요)

csharp Copy code

```
[Controller]
    ↓
[Service]
    ↓
[DAO]
    ↓
[MyBatis Mapper XML]
    ↓
[DB]
```

실제 예시 흐름

java Copy code

```
postService.getBoardList();
```

1. Controller가 Service 호출
2. Service가 `postDAO.getBoardList()` 호출
3. MyBatis가 XML에서 SQL 실행
4. DB 결과 → Board 객체 변환
5. DAO → Service → Controller 반환

5️⃣ 지금 Service가 "얇은 이유"

java Copy code

```java
public ArrayList<Board> getBoardList() {
    return postDAO.getBoardList();
}
```

👉 현재는 **단순 전달 역할만 함**
👉 하지만 이게 잘못된 구조 ❌ 전혀 아님

나중에 이렇게 확장됨

java Copy code

```java
public ArrayList<Board> getBoardList() {

    // 1. 권한 체크
    // 2. 파라미터 검증
    // 3. 여러 DAO 호출
    // 4. 데이터 가공

    return postDAO.getBoardList();
}
```

✔ 그래서 **처음부터 Service를 반드시 둔다**

6️⃣ Service가 꼭 필요한 이유

❌ Controller → DAO 직접 호출하면?

- 비즈니스 로직이 Controller에 섞임
- 재사용 불가
- 테스트 어려움
- 코드 스파게티 🍝

✅ Controller → Service → DAO 구조

✔ 역할 분리 (SRP 원칙)
✔ 유지보수 쉬움
✔ 테스트 용이
✔ 트랜잭션 관리 가능

7️⃣ Service에서 자주 하는 일들

기능	설명
비즈니스 로직	계산, 조건 분기
트랜잭션	`@Transactional`
여러 DAO 조합	A + B + C
예외 처리	DB 예외 → 서비스 예외
데이터 가공	DTO 변환 등

8️⃣ DAO는 "부품", Service는 "조립자"

비유로 정리하면 👇

- **DAO** = 나사, 부품
- **Service** = 조립 설명서 + 조립자
- **Controller** = 버튼 누르는 사람

9️⃣ 한 줄 요약 (면접용)

> **Service는 비즈니스 로직 계층이며,
> DAO는 데이터 접근 계층이고,
> Service가 DAO를 호출하여 기능을 완성한다.**

Service 와 controller관계

1 Controller 와 Service의 기본 관계

✔ Controller

- **요청(Request)을 받는 역할**
- URL 매핑, 파라미터 처리
- Service를 호출
- 결과를 **View(JSP/Thymeleaf)** 로 전달

✔ Service

- **비즈니스 로직 담당**
- 데이터 가공, 검증, 예외 처리
- DAO(Repository)를 호출하여 DB 작업 수행

👉 즉,

markdown Copy code

```
사용자 요청
 → Controller
   → Service
     → DAO
       → DB
```

2️⃣ 현재 코드에서의 관계 분석

📌 Controller (AdminController)

java Copy code

```
@Autowired
PostService postService;
```

- `PostService`를 **의존성 주입(DI)** 받음
- Controller는 **Service 없이는 동작하지 않음**
- 하지만 **구현 내용은 모름** (역할 분리)

예시 ① 게시판 목록 조회

java Copy code

```
ArrayList<Board> list = postService.getBoardList();
```

- Controller는
  👉 *"게시판 목록 줘"* 라고만 요청
- 실제 DB 조회 방법은 **Service/DAO에 위임**

예시 ② 게시판 등록

java Copy code

```
postService.insertBoard(name);
```

- 입력값을 받아 Service로 전달
- Controller는 **검증, 중복 처리 안 함**

📌 Service (PostService)

java Copy code

```java
@Service
public class PostService {
```

- 비즈니스 로직 계층
- Controller와 DAO의 **중간 다리 역할**

예시 ① 데이터 검증

java Copy code

```java
name = name.trim();
if(name.length()==0) {
    return;
}
```

- 이런 로직은 **Controller가 아니라 Service에서 처리**
- 이유:

  👉 다른 Controller에서도 재사용 가능

예시 ② DAO 호출

java Copy code

```java
postDAO.insertBoard(name);
```

- Service는 DB를 직접 만지지 않음
- DAO에게 **"저장해"** 라고 지시

↓

3 왜 이렇게 나누는가? (핵심 이유)

◆ 역할 분리 (Separation of Concerns)

계층	역할
Controller	요청/응답, 화면 연결
Service	비즈니스 로직
DAO	DB 접근

👉 한 곳에 다 넣으면

- 코드 복잡
- 수정 어려움
- 테스트 힘듦

---

4 한 줄로 정리하면

> **Controller는 Service를 호출하고,**
> **Service는 비즈니스 로직을 처리한 뒤 DAO를 호출한다.**

또는

> **Controller = 지휘자**
> **Service = 두뇌**
> **DAO = 손**

---

5 추가로 알면 좋은 포인트 💡

- `@Autowired`
  → Controller는 Service 객체를 **new 하지 않는다**
- Service를 거치지 않고 DAO를 직접 호출 ❌
- Controller에 로직이 많아지면 ❌

---

html 와 controller관계

1️⃣ HTML 과 Controller의 기본 관계

✔ Controller

- **어떤 HTML을 보여줄지 결정**
- HTML에서 사용할 **데이터(Model)** 를 만들어 전달
- HTML에서 보낸 **요청(form, link)** 을 받음

✔ HTML (Thymeleaf)

- Controller가 준 **데이터를 화면에 출력**
- 사용자 입력을 **Controller로 전송**

👉 즉,

```css
요청
HTML  ⇄  Controller
```

2️⃣ 현재 코드의 흐름 (중요 ⭐)

🔁 전체 흐름 요약

```bash
브라우저
 → /admin/board 요청
   → AdminController.board()
     → Model에 list 저장
     → admin/board.html 반환
 → Thymeleaf가 HTML 렌더링
```

3️⃣ Controller → HTML (데이터 전달)

📌 Controller 코드

java Copy code

```java
@GetMapping("board")
public String board(Model model) {
    ArrayList<Board> list = postService.getBoardList();
    model.addAttribute("list", list);
    return "admin/board";
}
```

✔️ 핵심 포인트

- `"admin/board"`
  👉 `resources/templates/admin/board.html`
- `model.addAttribute("list", list)`
  👉 HTML에서 `${list}` 사용 가능

📌 HTML에서 데이터 받기

html Copy code

```html
<form th:each="board : ${list}">
```

- `${list}` ← Controller가 준 데이터
- `board` ← 반복문 변수

html Copy code

```html
<input th:value="${board.bo_name}">
```

👉 Controller → Model → HTML 로 데이터 전달됨

4️⃣ HTML → Controller (요청 전달)

📌 게시판 등록 Form

```html
<form th:action="@{/admin/board/insert}" method="post">
    <input type="text" name="name">
    <button>등록</button>
</form>
```

✔ 작동 방식

1. 버튼 클릭
2. POST `/admin/board/insert`
3. Controller의 이 메서드 실행됨 👇

```java
@PostMapping("/board/insert")
public String boardInsert(
    @RequestParam("name") String name
)
```

- `name="name"` ↔ `@RequestParam("name")`
- HTML input 이름과 Controller 파라미터 이름이 **연결**

5 redirect 의 의미 (중요 ⭐)

```java
return "redirect:/admin/board";
```

✔ 이유

- 새로고침 시 중복 등록 방지
- URL을 다시 `/admin/board`로 변경
- 다시 GET 요청 발생

👉 **HTML을 직접 부르는 게 아니라 Controller를 다시 호출**

---

6 HTML은 Controller를 직접 모르고, Controller도 HTML을 직접 모르지 않는다

느슨한 연결 구조

구분	역할
Controller	데이터 준비 + 화면 이름 반환
HTML	데이터 사용 + 요청 전송

- Controller는 HTML 내부 구조 모름
- HTML은 Controller 내부 로직 모름
- **이름(경로, 변수명)으로만 연결**

↓

7 한 줄 요약 ⭐⭐⭐

> **Controller는 HTML에 보여줄 데이터를 주고,**
> **HTML은 Controller가 준 데이터를 사용하며 요청을 다시 Controller로 보낸다.**

또는

> **Controller = 화면 선택 + 데이터 공급**
> **HTML = 화면 표시 + 사용자 입력 전달**

---

8 추가로 아주 중요한 포인트 💡

- `${list}` 는 **Model에서 온 데이터**
- `th:action` 은 **Controller URL**
- `name="xxx"` 는 **Controller 파라미터 이름**
- HTML은 DB/Service/DAO를 절대 모른다 ❌