바이브코딩을 활용한 스프링부트 프로그래밍

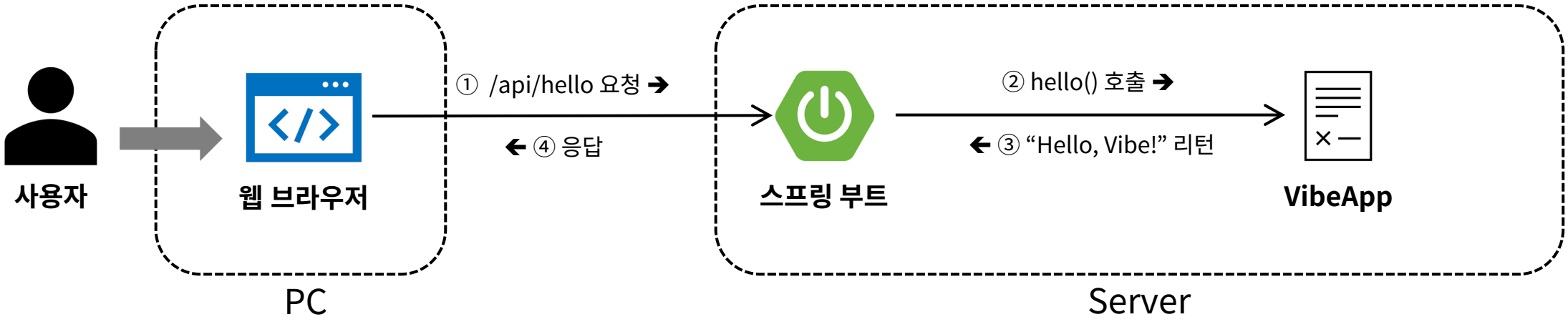
엄진영

1. 스프링부트 프로젝트 만들기

학습 목표

- 스프링부트 프로젝트의 “정체성”을 이해한다.
- 스프링부트 프로젝트 구조를 설명할 수 있다.
- 최소 기능 웹 애플리케이션을 실행할 수 있다.
- 바이브코딩 기반 개발 흐름을 경험한다.

1. 스프링부트 프로젝트 만들기 - 아키텍처



1. 스프링부트 프로젝트 만들기 – 디렉토리 구조

```
vibeapp/
├── .gradle/              ← Gradle 캐시 및 작업 디렉토리
├── .idea/                ← IntelliJ IDEA 설정 (IDE)
├── gradle/               ← Gradle Wrapper 파일
├── src/                  ← 소스 코드
├── build/                ← 빌드 결과물
├── out/                  ← IntelliJ IDEA가 생성하는 빌드 출력
├── build.gradle          ← Gradle 빌드 스크립트 (프로젝트 설정, 의존성 관리)
├── settings.gradle       ← Gradle 프로젝트 설정 (프로젝트 이름, 멀티 모듈 설정)
├── gradlew               ← Gradle Wrapper 실행 스크립트 (Unix/Mac)
├── gradlew.bat           ← Gradle Wrapper 실행 스크립트 (Windows)
├── .gitignore            ← Git 제외 파일 목록
└── PROJECT_SPEC.md       ← Gemini 에이전트용 프로젝트 명세서
```

1. 스프링부트 프로젝트 만들기 – 디렉토리 구조

```
.gradle/
├── 8.11.1/                    # Gradle 버전별 디렉토리
│   ├── executionHistory/      # 빌드 실행 히스토리
│   ├── fileChanges/           # 파일 변경 추적
│   ├── fileHashes/            # 파일 해시값
│   └── vcsMetadata/           # VCS 메타데이터
├── buildOutputCleanup/        # 빌드 정리 정보
└── vcs-1/                     # 버전 관리 정보
```

- **용도**
  - Gradle이 작업 중 생성하는 캐시 및 임시 파일
- **특징:**
  - Git에 커밋하지 않음 (.gitignore에 포함)
  - 삭제해도 다음 빌드 시 재생성됨
  - 빌드 속도 향상을 위한 캐시

1. 스프링부트 프로젝트 만들기 – 디렉토리 구조

```
gradle/
└── wrapper/
    ├── gradle-wrapper.jar          # Wrapper 실행 파일
    └── gradle-wrapper.properties   # Wrapper 설정
```

- **용도**
  - Gradle Wrapper 관련 파일
- **특징:**
  - `./gradlew` 실행 시 자동으로 Gradle 다운로드
  - Gradle 설치 없이 프로젝트 빌드 가능
  - Git에 커밋함 (팀원 간 Gradle 버전 통일)

1. 스프링부트 프로젝트 만들기 – 디렉토리 구조

```
src/main/
├── java/                                    # Java 소스 코드
│   └── com/example/vibeapp/
│       └── VibeappApplication.java          # 메인 클래스
└── resources/                               # 리소스 파일
    ├── static/                              # 정적 리소스(HTML, CSS, JavaScript, Images 등)
    ├── templates/                           # 뷰 템플릿
    ├── application.properties               # 메인 설정 파일 (또는 application.yml)
    ├── application-dev.properties           # 개발 환경 설정 (또는 application-dev.yml)
    ├── application-prod.properties          # 운영 환경 설정 (또는 application-prod.yml)
    ├── messages/                            # 다국어 메시지 (선택)
    │   ├── messages.properties              # 기본 (한글)
    │   ├── messages_en.properties           # 영어
    │   └── messages_ko.properties           # 한글 명시
    ├── schema.sql                           # DB 스키마 정의 (선택)
    └── data.sql                             # 초기 데이터 (선택)
```

1. 스프링부트 프로젝트 만들기 – 디렉토리 구조

```
src/test/
├── java/                                    # 단위 테스트 Java 소스 코드
│   └── com/example/vibeapp/
│       └── VibeappApplicationTests.java
└── resources/                               # 테스트 리소스
    ├── application-test.properties          # 테스트 설정 파일 (또는 application-test.yml)
    └── data.sql                             # 초기 데이터 (선택)
```

1. 스프링부트 프로젝트 만들기 – 디렉토리 구조

```
build/
├── classes/                    # 컴파일된 .class 파일
│   ├── java/
│   │   ├── main/               # 메인 클래스 파일
│   │   └── test/               # 테스트 클래스 파일
│   └── resources/
│       ├── main/               # 메인 리소스
│       └── test/               # 테스트 리소스
│
├── generated/                  # 자동 생성된 소스
│   └── sources/
│       └── annotationProcessor/
│           └── java/main/
│               └── com/example/vibeapp/domain/
│                   └── QPost.java # QueryDSL Q클래스
│
├── libs/                       # JAR/WAR 파일
│   └── vibeapp-0.0.1-SNAPSHOT.jar
│
├── reports/                    # 테스트 리포트
│   └── tests/
│       └── test/
│           └── index.html # 테스트 결과 HTML
│
├── resources/                  # 처리된 리소스
│   └── main/
│       ├── static/
│       ├── templates/
│       └── application.properties
│
├── test-results/               # 테스트 결과 XML
│   └── test/
│
└── tmp/                        # 임시 파일
    ├── compileJava/
    └── bootJar/
```

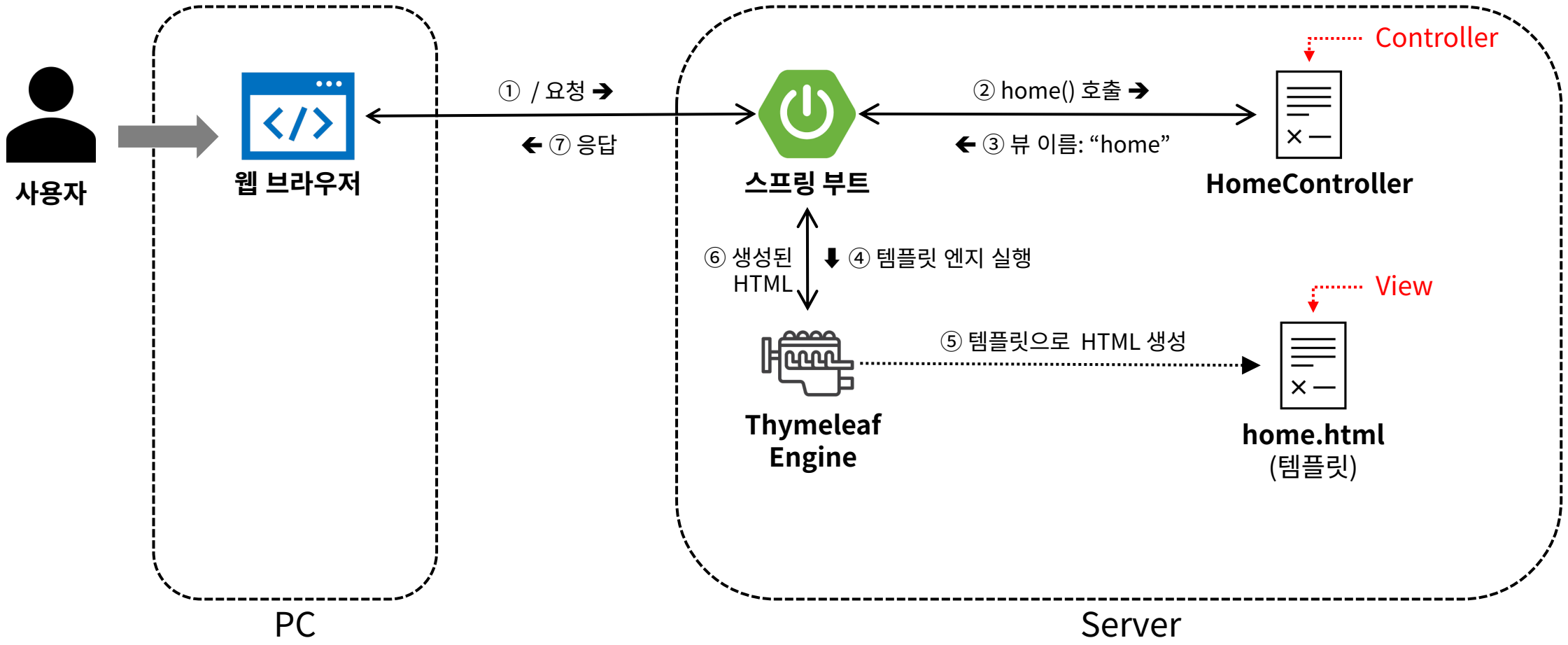
- **용도**
  - Gradle 빌드 결과물 및 임시 파일
- **특징:**
  - 빌드할 때마다 재생성됨
  - `./gradlew clean` 으로 폴더 삭제 가능
  - Git에 커밋하지 않음 (.gitignore 에 포함)

2. 뷰 템플릿 도입하기

학습 목표

- 웹 애플리케이션에서 템플릿 엔진의 필요성을 설명할 수 있다.
- Thymeleaf 템플릿 엔진을 설정하고 기본적인 사용법을 적용할 수 있다.
- MVC 패턴에서 Controller와 View를 연결할 수 있다.
- 바이브코딩을 활용하여 뷰 계층을 효율적으로 구축하는 개발 흐름을 경험한다.

2. 뷰 템플릿 도입하기 - 아키텍처

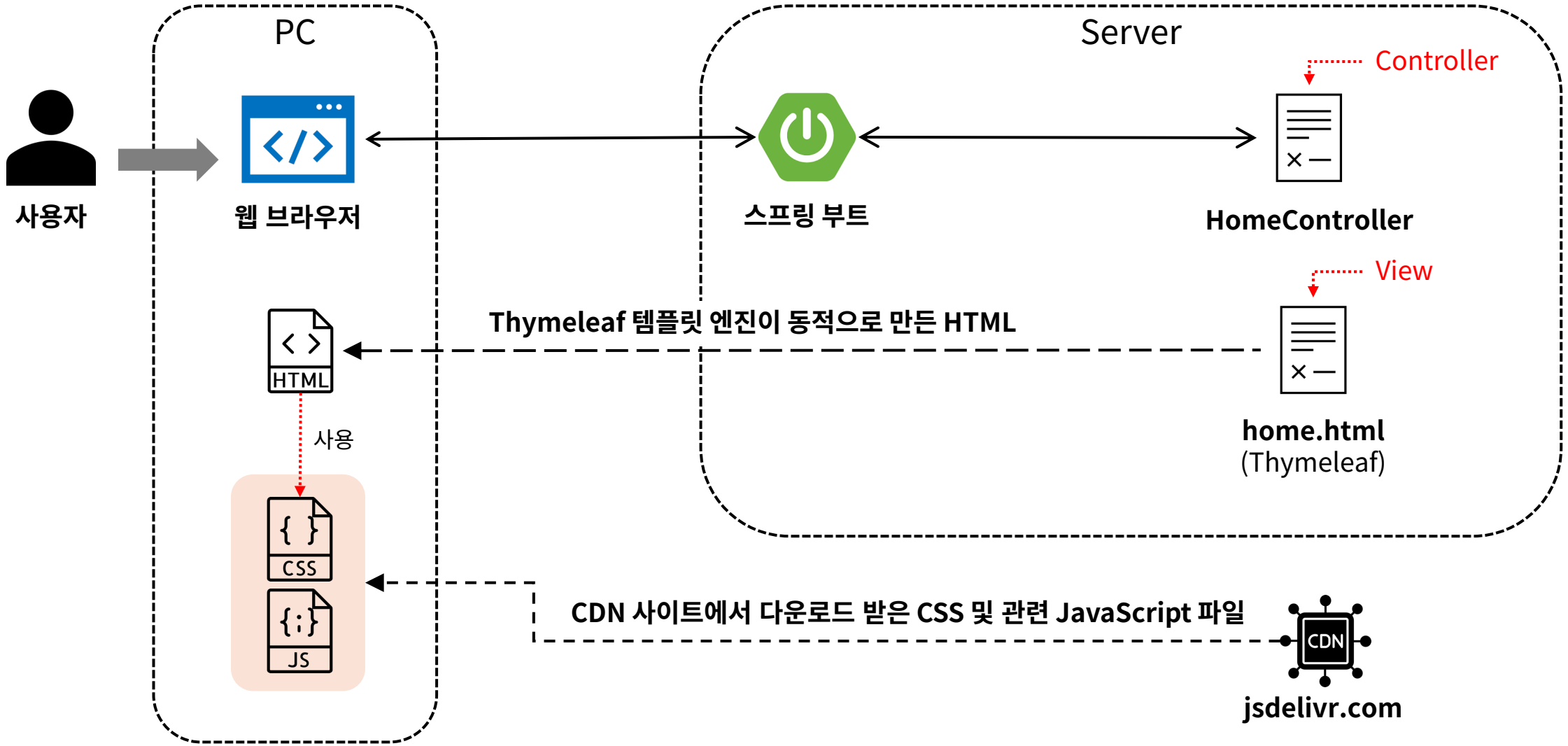


3. CSS 프레임워크 도입하기

학습 목표

- CSS 프레임워크의 필요성을 설명하고, 순수 CSS와의 차이를 구분할 수 있다.
- Bootstrap 5를 CDN 또는 로컬 방식으로 프로젝트에 적용할 수 있다.
- Bootstrap 컴포넌트와 Thymeleaf를 결합하여 동적 UI를 구현할 수 있다.
- 바이브코딩을 활용하여 Bootstrap 기반 UI를 생성하고 커스터마이징 할 수 있다.

3. CSS 프레임워크 도입하기 - 아키텍처



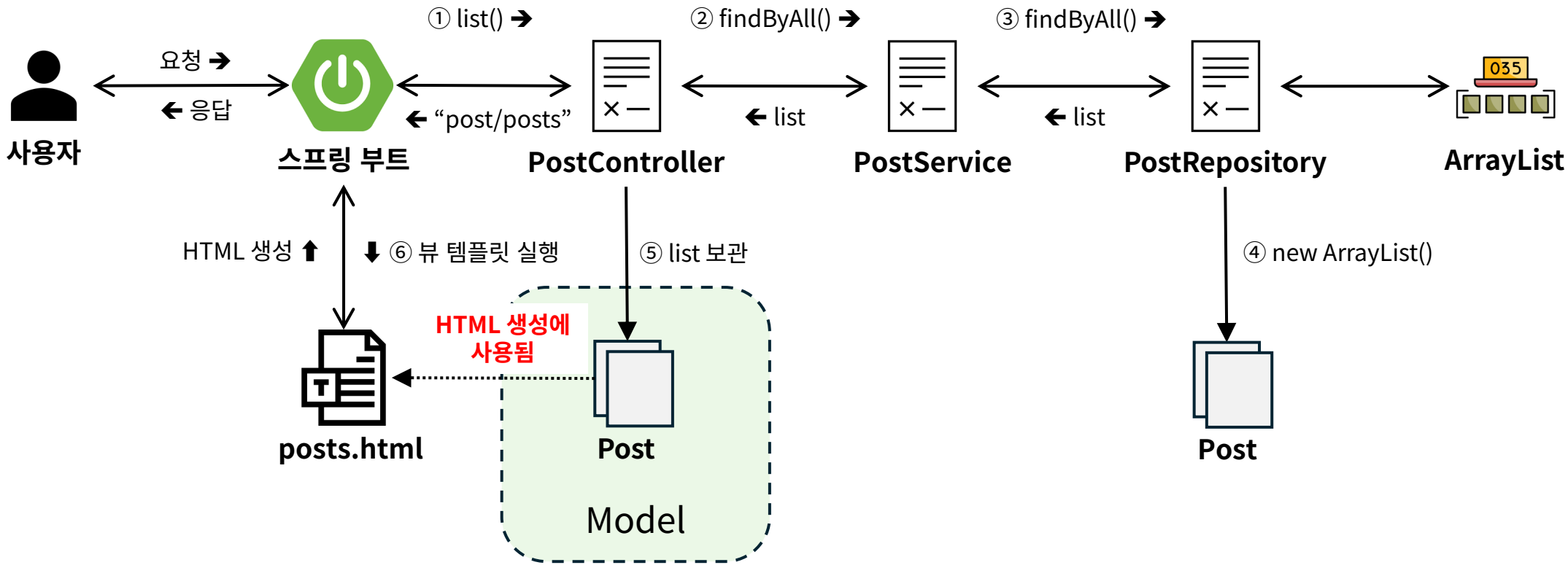
4. 게시글 CRUD 구현하기 (without DBMS)

학습 목표

- 웹 애플리케이션에서 **CRUD(Create, Read, Update, Delete)**의 개념과 역할을 설명할 수 있다.
- **Java Collection API**를 사용하여 게시글 데이터를 관리할 수 있다.
- 게시글 등록 / 조회 / 수정 / 삭제 기능을 **서버 렌더링** 기반으로 구현할 수 있다.
- **Controller**, **Service**, **Repository** 계층의 역할 분리를 이해하고 적용할 수 있다.
- Thymeleaf와 Bootstrap을 활용하여 CRUD 화면을 구성할 수 있다.
- 바이브코딩을 활용해 CRUD 기능을 구현하고, 생성된 코드를 분석·개선할 수 있다.

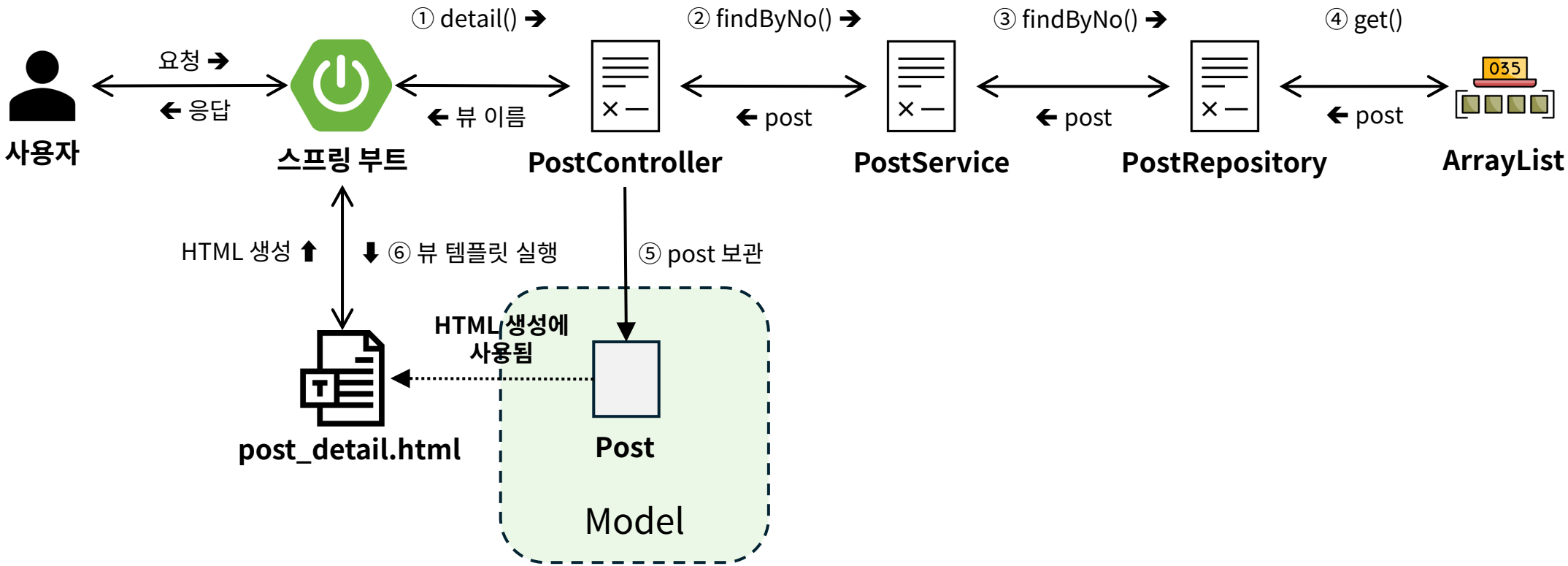
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 목록 조회)

URL: / posts



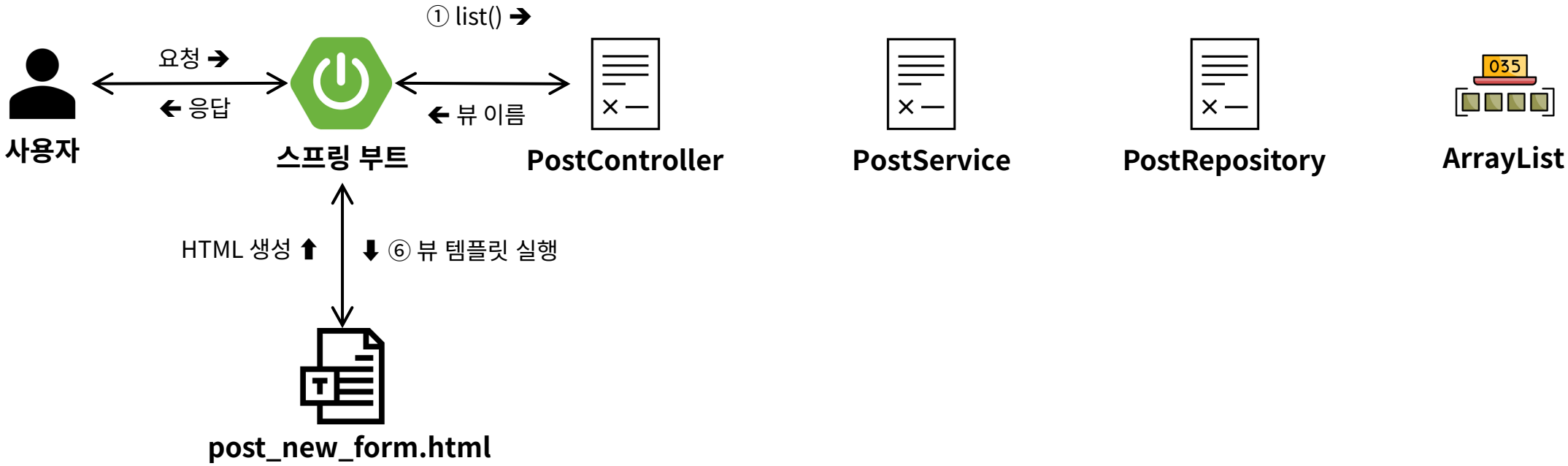
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 상세 조회)

URL: / posts/{no}



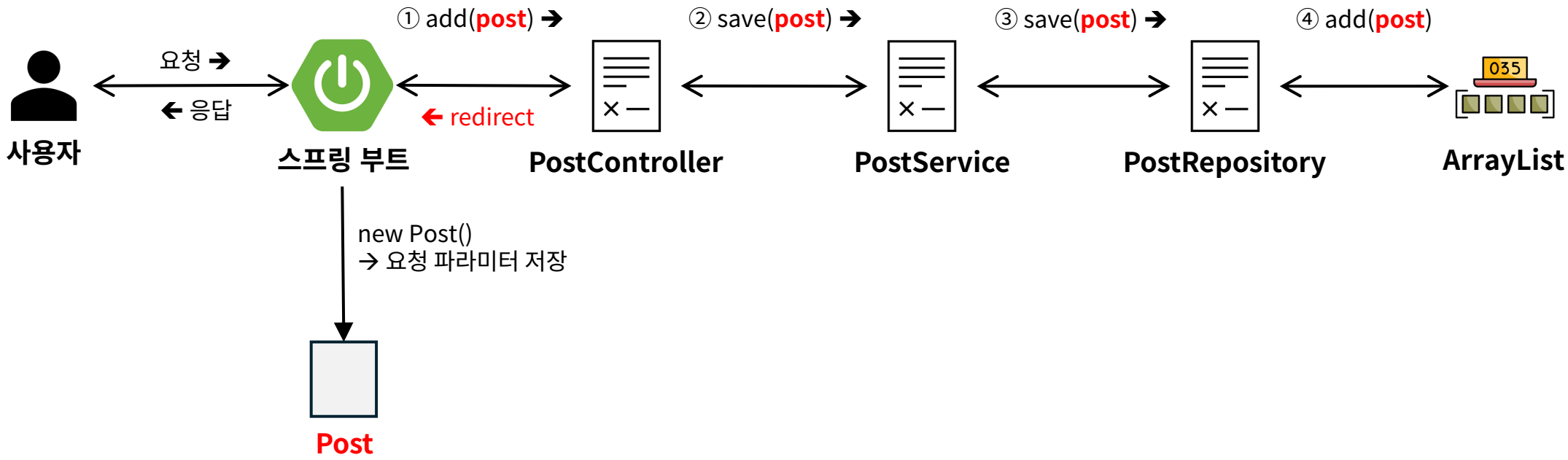
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 작성폼)

URL: / posts/new



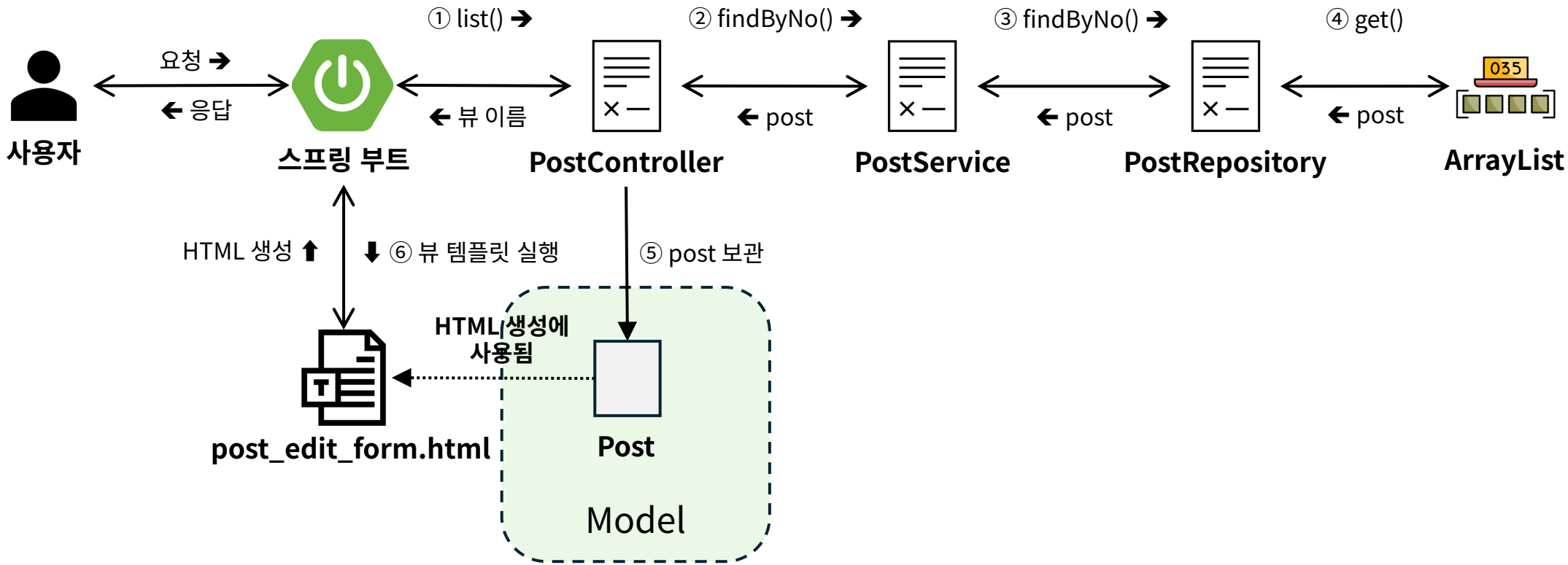
4. 게시글 CRUD 구현하기 – 아키텍처(새 게시글 등록)

URL: / posts/add



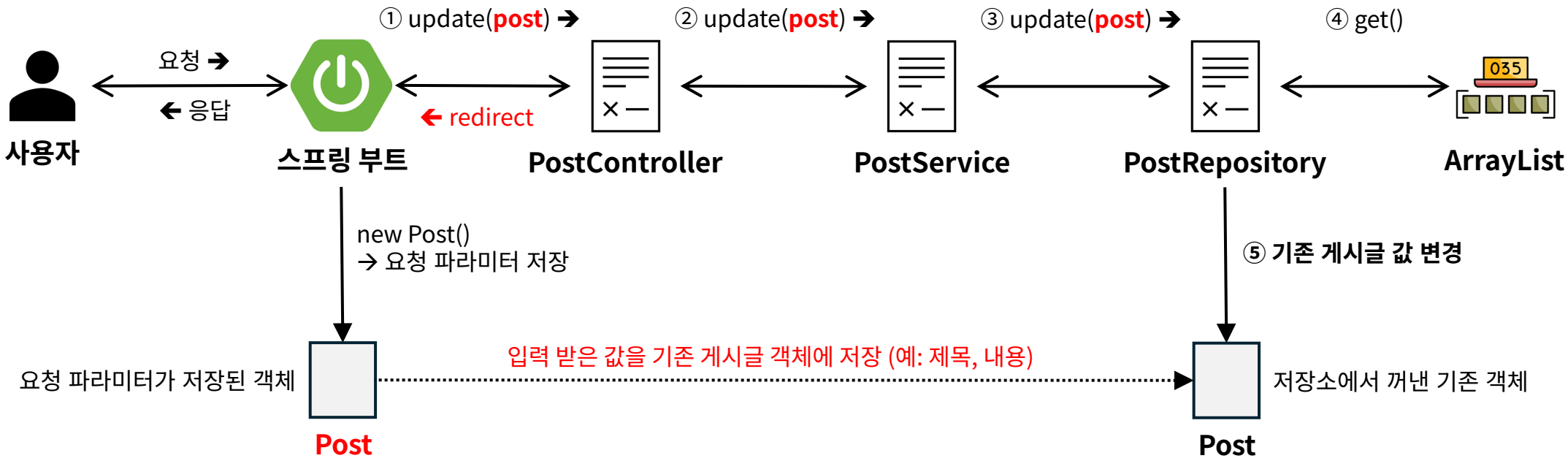
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 수정폼)

URL: / posts/{no}/edit



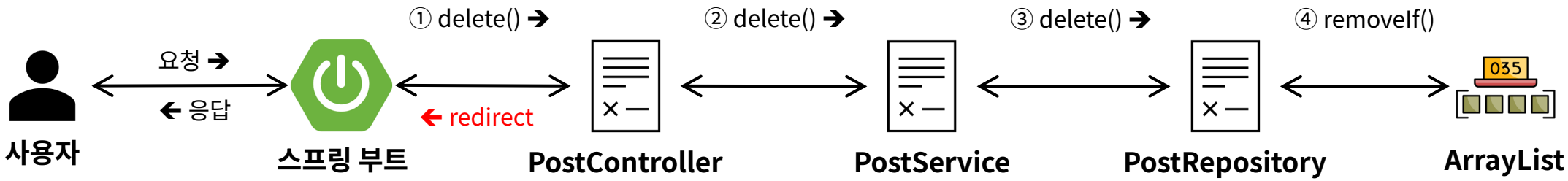
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 수정)

URL: / posts/{no}/save



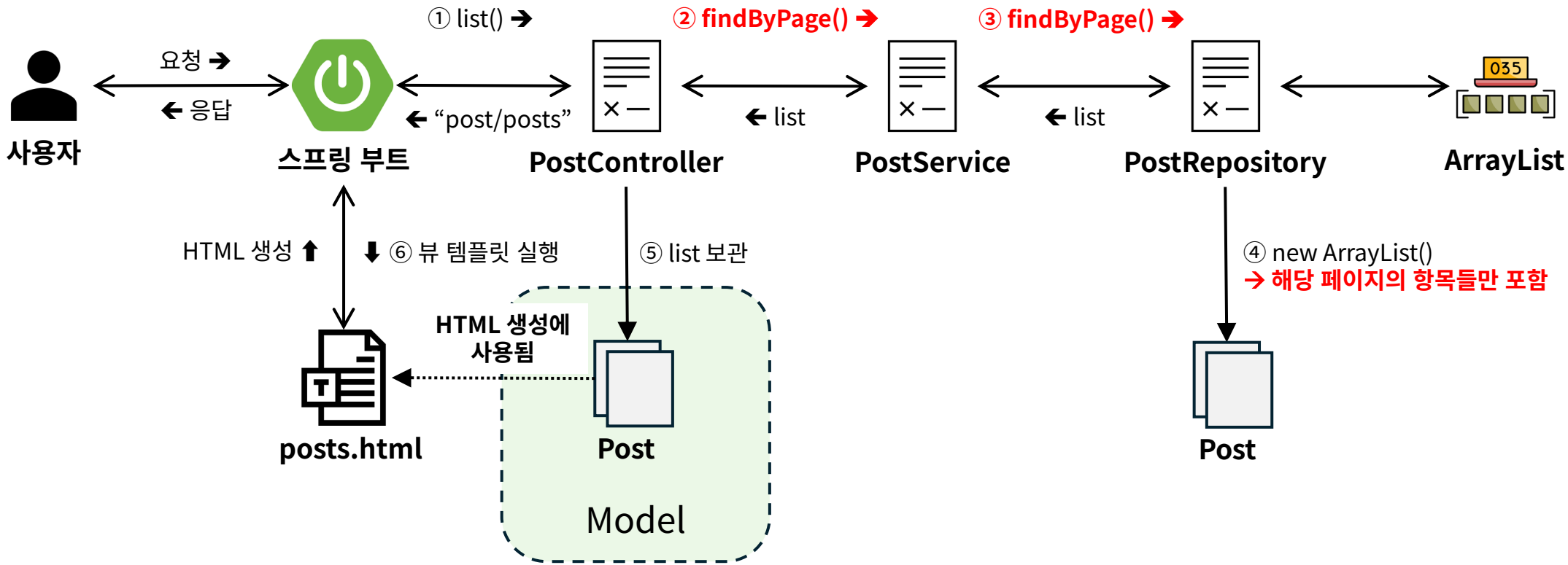
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 삭제)

URL: / posts/{no}/delete



4. 게시글 CRUD 구현하기 – 아키텍처(게시글 목록 페이징 처리)

URL: / posts



4. 게시글 CRUD 구현하기 – 자바 패키지 구조 변경

변경 전

```
com.example.vibeapp
├── VibeApp.java
├── HomeController.java
├── Post.java
├── PostController.java
├── PostRepository.java
└── PostService.java
```

기능형 구조

```
com.example.vibeapp
├── VibeApp.java
├── home
│   └── HomeController.java
└── post
    ├── Post.java
    ├── PostController.java
    ├── PostRepository.java
    └── PostService.java
```

비즈니스 도메인/기능 단위로 분류
DDD(Domain-Driven-Design) 접근법
장점:

- 응집도 높음 – 관련 코드가 한 곳에 있음
- 변경 영향 범위 파악 용이
- **대규모 프로젝트**에 적합
- **마이크로서비스 전환 쉬움**
- **팀 단위 개발에 유리** (각 팀이 도메인 담당)

단점:

- 초보자에게 직관적이지 않을 수 있음
- 공통 기능 처리 고민 필요
- 패키지 간 의존성 관리 필요

계층형 구조

```
com.example.vibeapp
├── VibeApp.java
├── controller
│   └── HomeController.java
│   ├── PostController.java
├── service
│   └── PostService.java
├── repository
│   └── PostRepository.java
└── domain
    └── Post.java
```

기술적 역할(계층)에 따라 분류
장점:

- 직관적, 학습 용이
- **소규모 프로젝트**에 적합

단점:

- 프로젝트가 커지면 패키지가 비대해짐
- 기능 단위 작업시 여러 패키지를 오가야 함
- **모듈화/마이크로서비스 전환 어려움**

4. 게시글 CRUD 구현하기 – 뷰 템플릿 파일 위치 변경

변경 전:

```
templates/
├── home.html
├── post_detail.html
├── post_edit_form.html
├── post_new_form.html
└── posts.html
```

변경 후:

```
Templates
├── home/
│   └── home.html
└── post/
    ├── post_detail.html
    ├── post_edit_form.html
    ├── post_new_form.html
    └── posts.html
```

5. DTO 패턴 적용하기

학습 목표

- **DTO의 역할**과 **Entity와의 분리 필요성**을 설명할 수 있다.
- 요청(Request)과 응답(Response) **DTO를 설계하고 유효성 검증을 적용**할 수 있다.
- **DTO ↔ Entity 간 변환 방법을 이해**하고 적절한 위치에서 구현할 수 있다.
- **Controller ↔ Service** 계층 사이에서 **DTO 기반 데이터 전달 구조를 구현**할 수 있다.
- 바이브코딩을 활용하여 **기존 코드를 DTO 패턴으로 리팩토링**할 수 있다.

"**DTO**는 **데이터를 옮기기 위한** 전용 객체이며,
Entity를 보호하고 **계층 간 결합을 낮추기 위한** 핵심 도구이다."

5. DTO 패턴 적용하기 – DTO 개요

```
@PostMapping("/posts")
public String create(Post post) {  // Entity 직접 사용
  ...
}
```

DTO 적용 전:

1. **내부 구조 노출**

   Entity 필드가 그대로 외부에 노출됨

2. **변경에 취약**

   Entity 필드 변경 → Controller / View / API 전부 영향

3. **보안 문제**

   비밀번호, 내부 상태 값 노출 위험

4. **의미 없는 데이터 전달**

   화면에 필요 없는 필드까지 함께 전달됨

5. DTO 패턴 적용하기 – DTO 개요

```
@PostMapping("/posts")
public String create(PostCreateDto dto) {
  ...
}
```

DTO 적용 후:

1. **필요한 데이터만 전달**

   클라이언트가 알 필요 없는 정보 숨김

2. **외부와 내부 모델을 분리**

   Entity 변경해도 API는 안정적

3. **계층 간 명확한 계약(contract) 형성**

   입력/출력 스펙이 명확해짐

4. **유지보수성 향상**

   각 계층의 책임이 명확해짐

DTO를 만드는 신호:

- ✓ Controller에서 Entity를 직접 받고 있다
- ✓ 화면/API 요구사항이 Entity와 다르다
- ✓ 보안상 숨겨야 할 필드가 있다
- ✓ 입력 검증이 필요한데 Entity에 넣기 애매하다
- ✓ 여러 Entity를 조합한 데이터를 반환해야 한다
- ✓ 같은 Entity지만 화면마다 보여줄 필드가 다르다
- ✓ 나중에 REST API로 확장할 가능성이 있다

5. DTO 패턴 적용하기 – DTO vs Entity vs VO

구분	DTO	Entity	VO (Value Object)
목적	계층간 데이터 전달	도메인 핵심 객체	도메인의 "값" 표현
비교 기준	**비교 불필요**	동일성(**ID 기반**)	동등성(**값 기반**)
비즈니스 로직	△ 최소한의 변환/검증만	**O** (상태 변경, 규칙)	**O** (값 검증, 불변 규칙)
변경 빈도	화면/API에 따라 자주 변경	상대적으로 안정적	매우 낮음
변경 가능성	**Mutable (상황에 따라 자유)**	Mutable (상태 **변경 가능**)	**Immutable(불변,** 새 객체 생성)
사용 위치	경계 (Controller, API, View)	Service, Repository 에서 사용됨 도메인에 속함	Service 내부에서 사용 Entity에 포함됨 도메인 내부에서 사용
외부 노출	**O** 외부 계약이므로 노출	**X** 내부 구현 → 노출 금지	△ 도메인 내부에서만 사용 권장
생명주기	요청/응답 단위 (일회성)	영속성 생명 주기 (DB와 연동)	Entity에 포함되어 존재
써야 할 때	"Entity를 노출하면 위험한가?" "입력에서 검증하고 싶은가?" "도메인과 입출력 형태가 다른가?"	"이 객체는 시간이 지나도 같은 대상인가?" DB에 저장되고 조회되는 대상인가? 수정/삭제의 개념이 있는가?	"이 값 자체가 도메인 개념인가" 값 검증/규칙이 중요한다. ID 없이도 충분한가?
예시	PostRequestDto PostResponseDto	User, Post, Order	Email, Money, Period, Address, Title

5. DTO 패턴 적용하기 – DTO vs Entity vs VO (코드 예)

VO

```
public class Email {
  private final String value;

  public Email(String value) {
    // 도메인 규칙 포함
    if (!value.contains("@")) {
      throw new IllegalArgumentException(
        "Invalid email");
    }
    this.value = value;
  }

  public String getValue() {
    return value;
  }
}
```

DTO

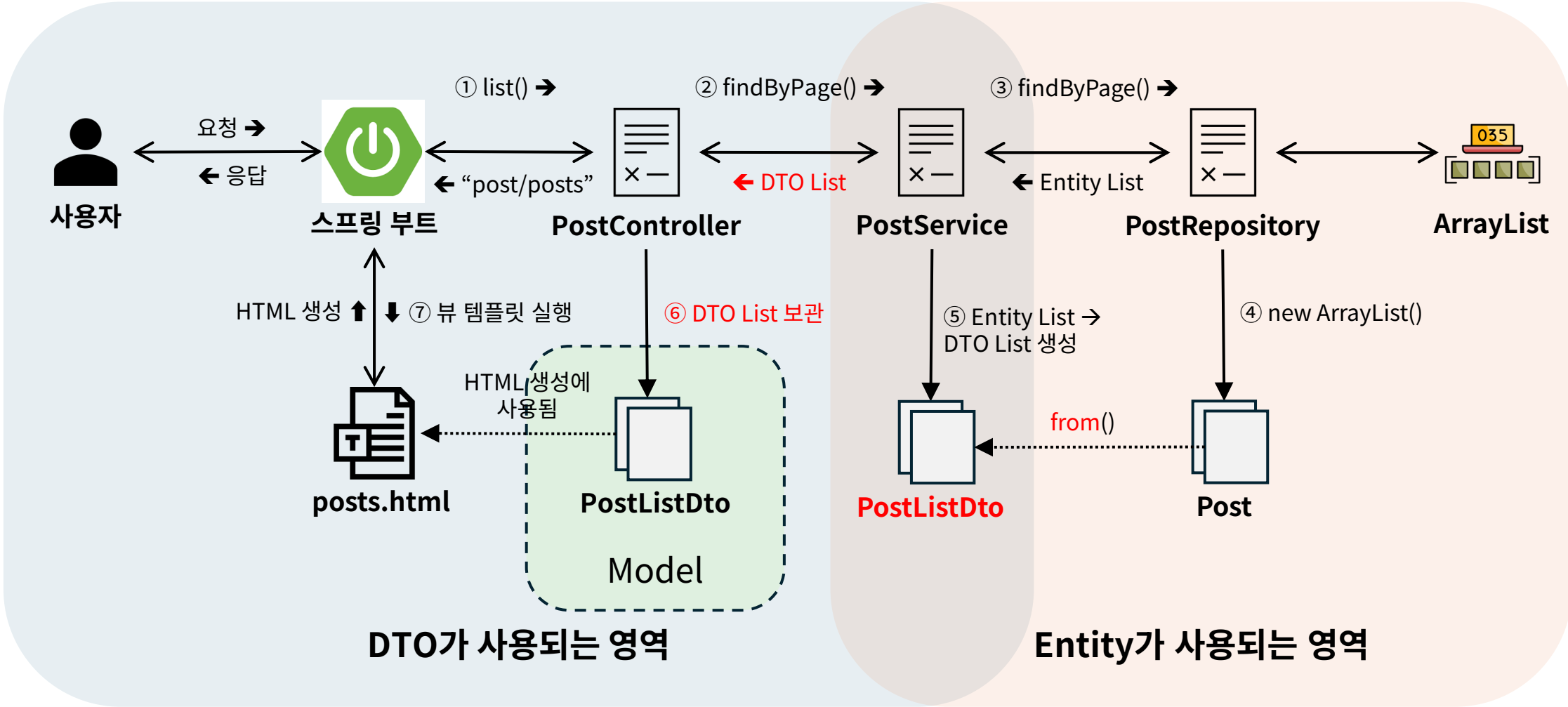
```
public class PostCreateDto {
    private String title;
    private String content;
}

public class PostUpdateDto {
    private Long id;
    private String title;
    private String content;
}
```

Entity

```
public class Post {
    private Long id; // ID로 동일성 판단
    private String title;
    private String content;
    private Email email;       // VO
}
```

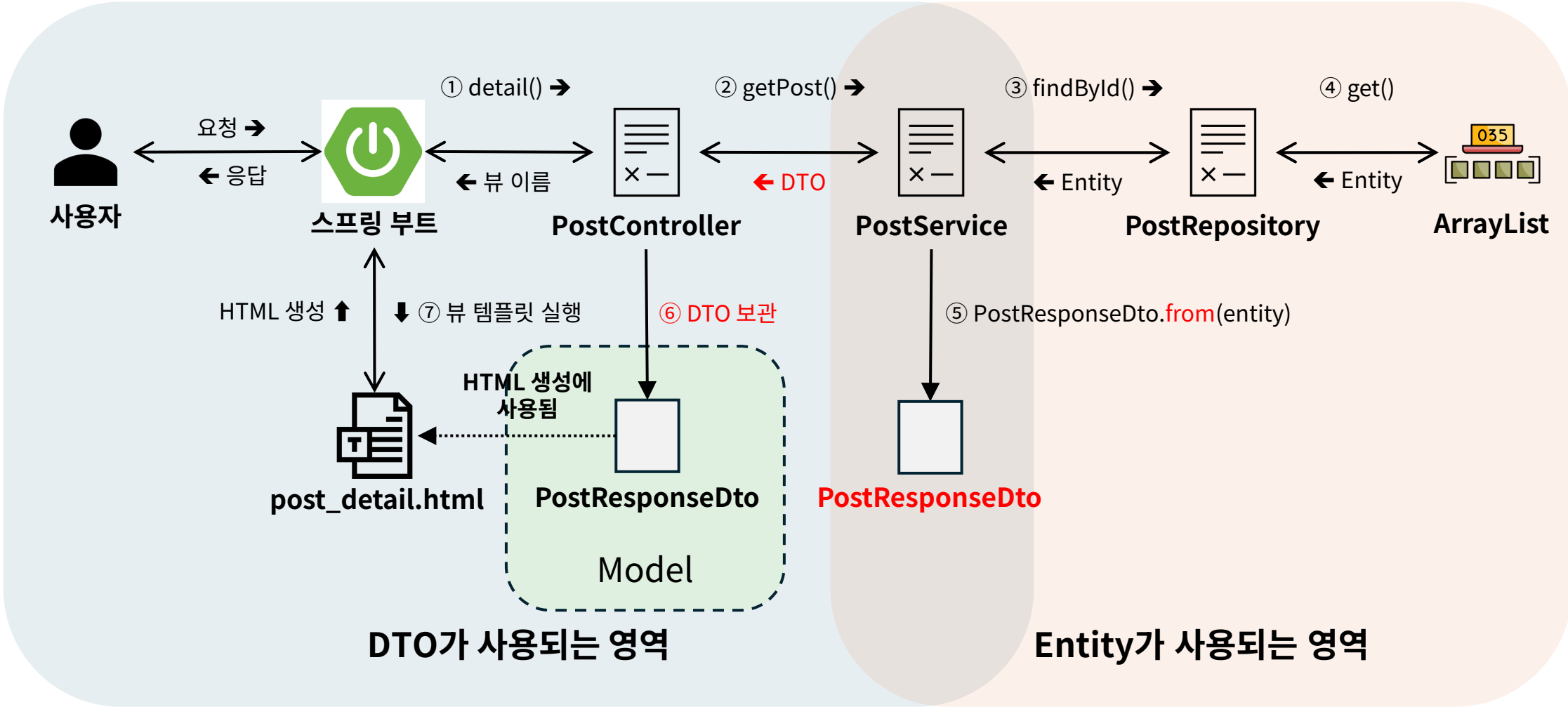
5. DTO 패턴 적용하기 – 아키텍처(게시글 목록 조회)

URL: / posts



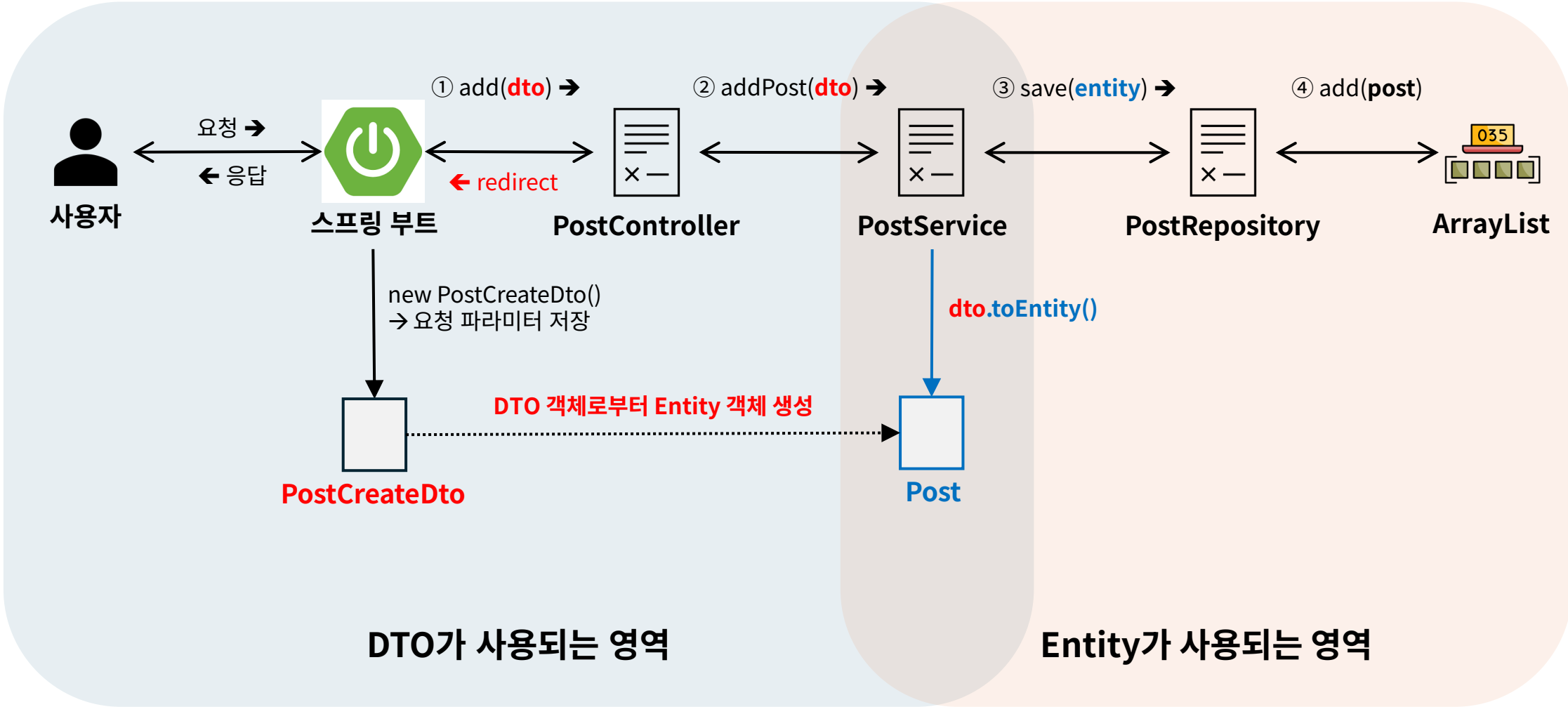
5. DTO 패턴 적용하기 – 아키텍처(게시글 상세 조회)

URL: / posts/{no}



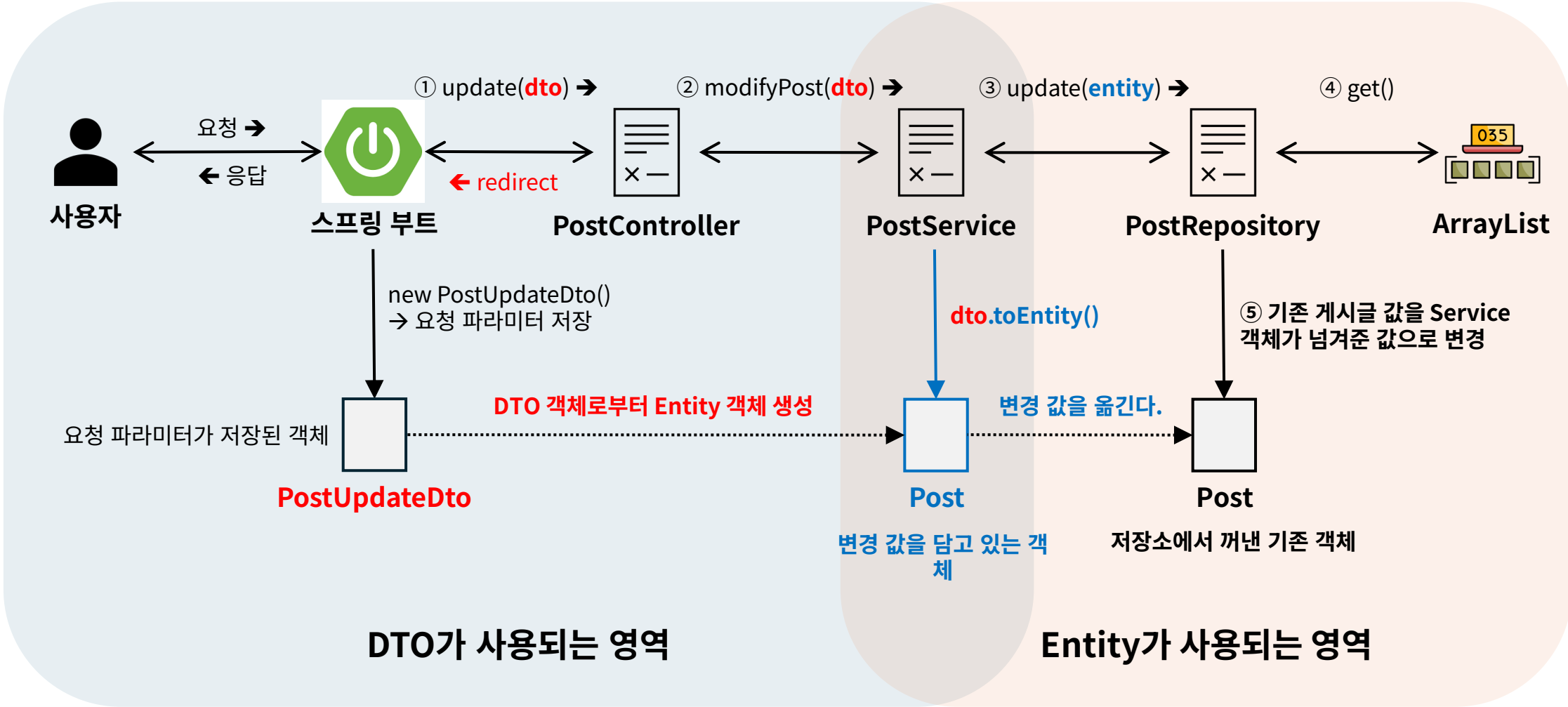
5. DTO 패턴 적용하기 – 아키텍처(새 게시글 등록)

URL: / posts/add



5. DTO 패턴 적용하기 – 아키텍처(게시글 수정)

URL: / posts/{no}/save



6. DTO를 record 문법으로 리팩토링 하기

학습 목표

- Java **record 문법의 목적과 특징을 설명**할 수 있다.
- class 기반 DTO와 record 기반 DTO의 **차이를 비교**할 수 있다.
- Response DTO를 record로 변환하고, **불변성의 장점을 설명**할 수 있다.
- record에서 **Bean Validation**과 **Compact Constructor**를 활용할 수 있다.
- record의 **제약사항을 이해**하고 DTO 종류별 적절한 선택을 할 수 있다.
- 바이브코딩을 활용하여 DTO를 record로 리팩토링하고 개선할 수 있다.

"**record**는 **DTO의 의도**를 **언어 차원에서 강제하는 도구**다."

6. DTO를 record 문법으로 리팩토링 하기

record DTO의 이점:

1. **DTO의 의도를 언어 차원에서 강제한다** – record는 본질적으로 데이터 전달용 타입. 필드는 final, setter 없음
2. **불변성(immutability)으로 안정성이 높아진다** – 생성 후 값 변경 불가. 스레드 안전. 계층 간 이동 중 데이터 변조 없음
3. **코드가 극적으로 줄어든다** – 생성자, getter, equals, hashCode, toString 자동 생성. 중복 코드 제거 → 가독성 상승
4. **equals/hashCode/toString 품질이 기본 보장된다** – 값 기반 비교 자동 구현. 테스트, 로깅, 디버깅 유리. 실수 감소
5. **Validation 구조가 더 명확해진다** – 생성 시점 검증. 잘못된 값은 객체 자체가 생성되지 않음
6. **JSON 직렬화/역직렬화와 궁합이 좋다** – Jackson 공식 지원. REST API 요청/응답에 최적
7. **DTO와 Entity의 경계가 더 선명해진다** – Entity는 class. DTO는 record. 역할 구분이 코드 구조로 드러남
8. **바이브코딩(AI 코드 생성) 품질이 좋아진다** – 불필요한 boilerplate 제거. 생성 코드 일관성 향상

6. DTO를 record 문법으로 리팩토링 하기 – class vs record

기능	class	record	호환성
Bean Validation	✅	✅	100%
JSON 역직렬화	✅	✅	100% (JDK 17+, Jackson 2.12.3+)
JSON 직렬화	✅	✅	100%
생성자	✅	✅	100%
Getter	✅	✅	100%(메서드명만 변경)
Setter	✅	❌	불변성 강제
toEntity()	✅	✅	100%
from()	✅	✅	100%
equals/hashCode	수동	✅	자동 생성
toString	수동	✅	자동 생성

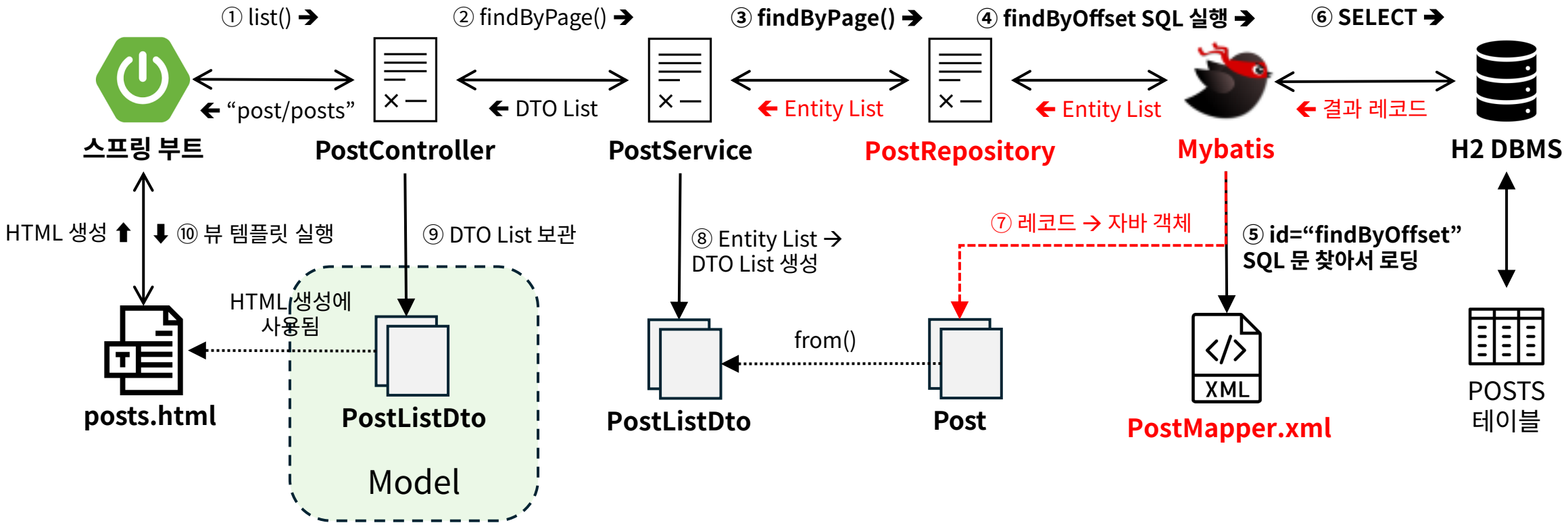
7. DBMS + Mybatis SQL Mapper 도입하기

학습 목표

- **H2 데이터베이스**와 **MyBatis**를 **스프링부트에 연동**하고 기본 설정을 구성할 수 있다.
- 게시글 Entity 기준으로 **테이블 DDL과 CRUD SQL을 직접 작성**할 수 있다.
- MyBatis XML Mapper를 사용하여 **SQL을 실행하고 결과를 매핑**할 수 있다.
- Repository 구현체를 Collection 기반에서 MyBatis 기반으로 교체할 수 있다.
- 데이터 저장소 변경 후에도 **Controller와 DTO 계층이 영향받지 않음을 확인**할 수 있다.
- 바이브코딩을 활용하여 SQL과 Mapper를 작성하고 실행 흐름을 분석할 수 있다.

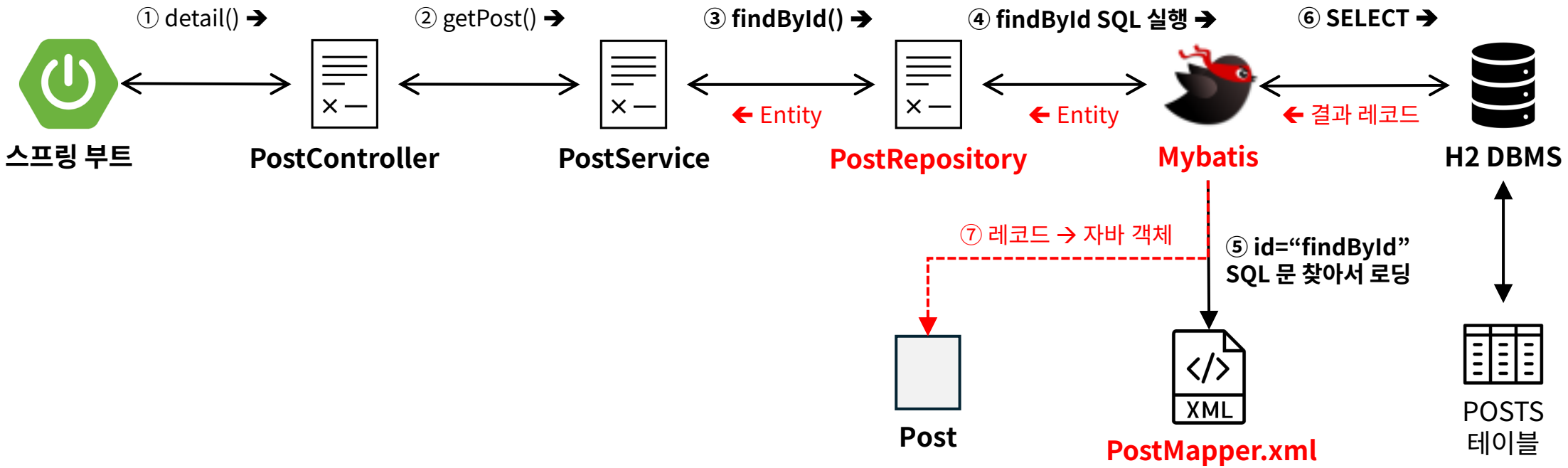
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 목록 조회)

URL: / posts



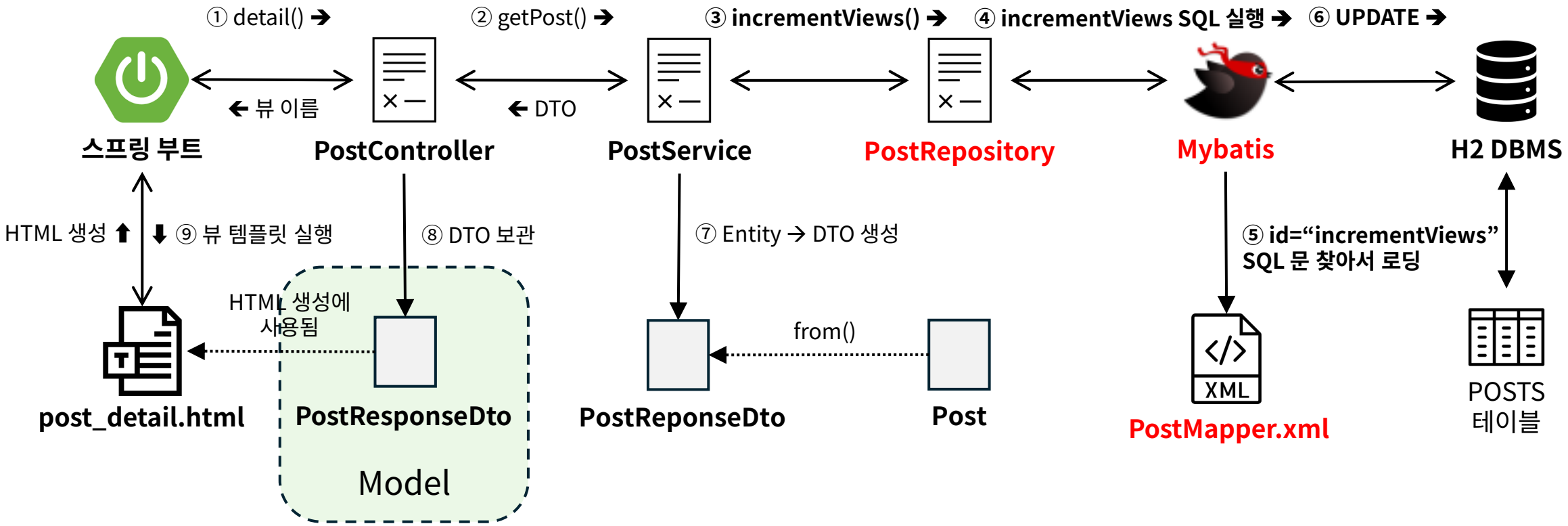
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 상세 조회)

URL: / posts/{no}



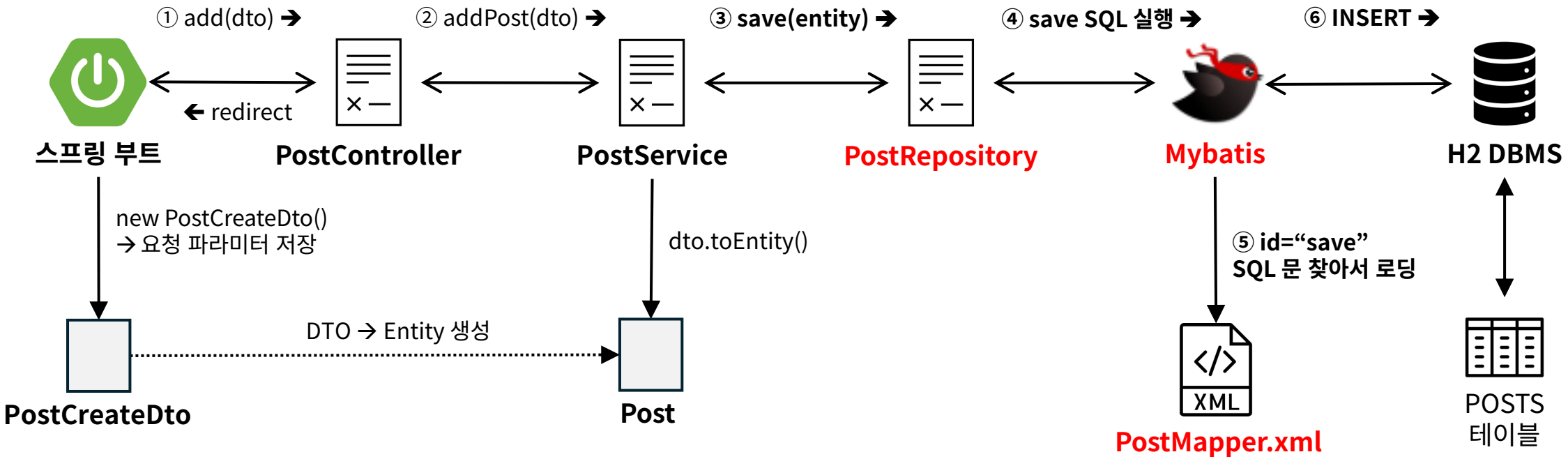
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 조회수 증가)

URL: / posts/{no}



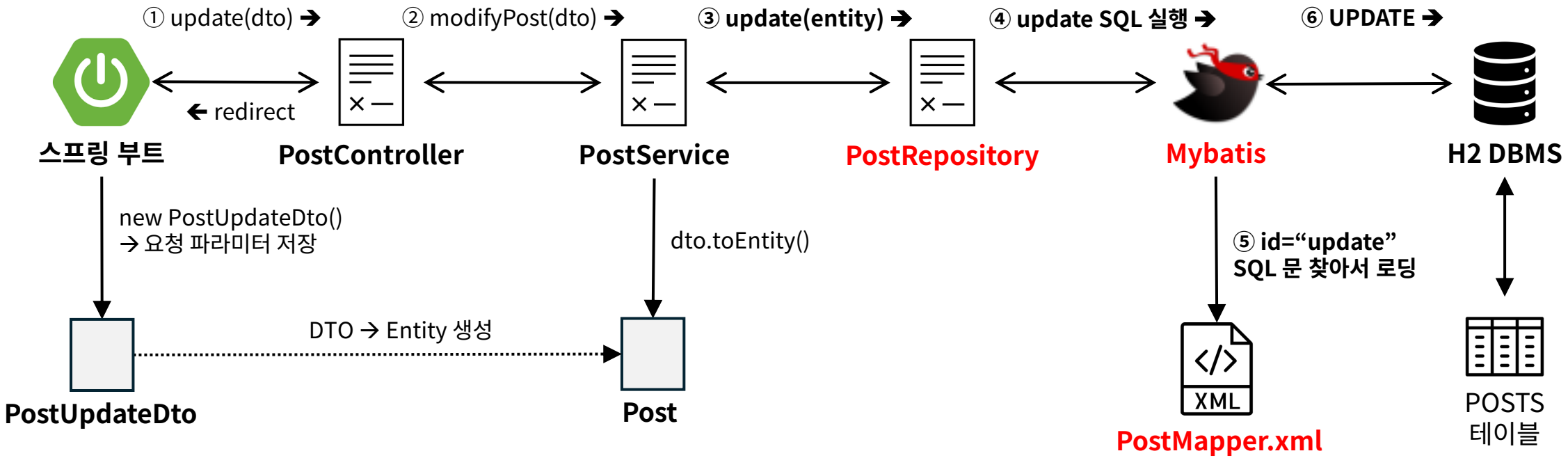
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(새 게시글 등록)

URL: / posts/add



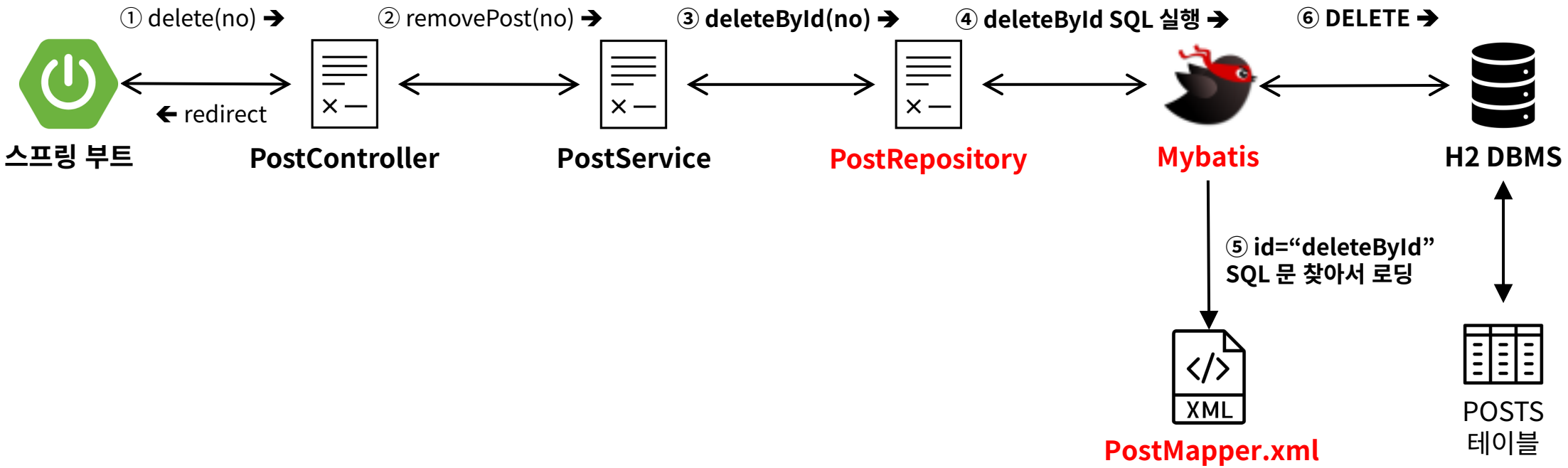
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 수정)

URL: / posts/{no}/save



7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 삭제)

URL: / posts/{no}/delete



8. 트랜잭션 적용하기

학습 목표

- 트랜잭션 없이 여러 SQL을 실행할 때 발생하는 **데이터 불일치 문제를 경험하고 설명**할 수 있다.
- 트랜잭션의 개념과 **ACID 특성을 이해하고 설명**할 수 있다.
- 스프링의 **선언적 트랜잭션 관리 방식**과 @Transactional 동작 원리를 설명할 수 있다.
- @Transactional을 적용하여 여러 SQL 작업을 하나의 **원자적 단위로 묶을 수** 있다.
- 예외 발생 시 **커밋/롤백 결정 규칙을 이해**하고 필요시 커스터마이징할 수 있다.
- readOnly 속성과 트랜잭션 전파 속성의 기본 개념을 이해한다.
- 바이브코딩을 활용하여 트랜잭션 적용 코드를 작성하고 실행 결과를 검증할 수 있다.

"**트랜잭션**은 여러 데이터 변경 SQL문들을

한 단위의 업무로 묶는(원자성을 보장하는) 장치이다."

전부 성공 또는 전부 실패

8. 트랜잭션 적용하기 – ACID 특성

Atomicity(원자성) 예시 코드:

```
@Transactional
public void transferPost(Long fromUserId, Long toUserId, Long postId) {
    // 3개 작업이 모두 성공하거나 모두 실패
    postRepository.removeFromUser(fromUserId, postId);
    postRepository.addToUser(toUserId, postId);
    auditRepository.logTransfer(fromUserId, toUserId, postId);
}
```

8. 트랜잭션 적용하기 – ACID 특성

Consistency(일관성) 예시 코드:

```
@Transactional
public void createPost(PostCreateRequest dto) {
    Post post = dto.toEntity();
    // 비즈니스 규칙 검증
    if (post.getTitle().length() > 100) {
        throw new IllegalArgumentException("제목 길이 초과");
    }
    postRepository.save(post);
    // 트랜잭션 커밋 시 DB 제약조건도 검증됨
}
```

8. 트랜잭션 적용하기 – ACID 특성

Isolation(격리성) 예시 코드:

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public PostResponse findById(Long id) {
    // 다른 트랜잭션의 미커밋 데이터는 안 보임
    return PostResponse.from(postRepository.findById(id));
}
```

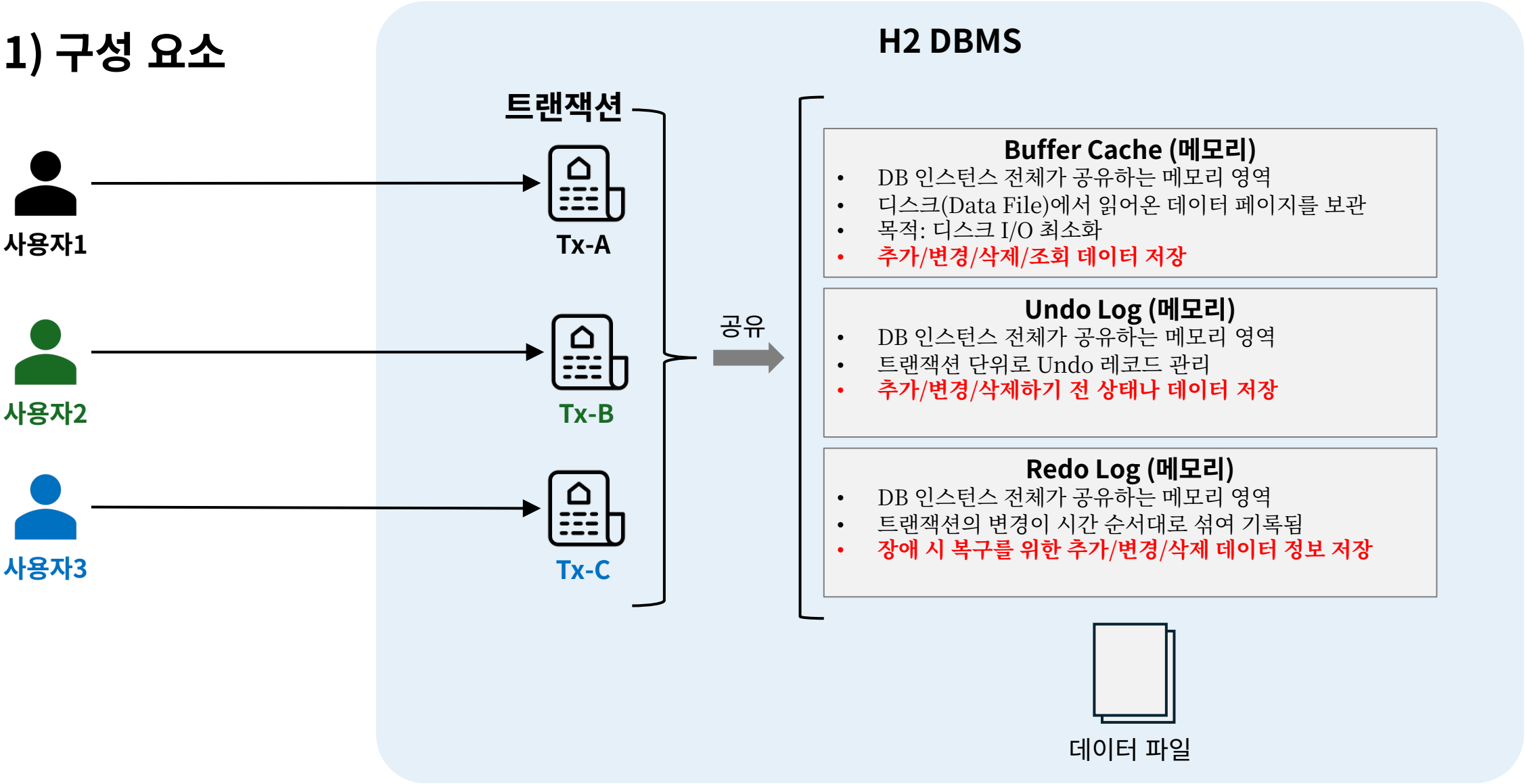
8. 트랜잭션 적용하기 – ACID 특성

Durability(지속성) 예시 코드:

```
@Transactional
public void createPost(PostCreateRequest dto) {
    postRepository.save(dto.toEntity());
    // 커밋되면 영구적으로 저장
    // 시스템 장애 발생해도 데이터 유지
}
```

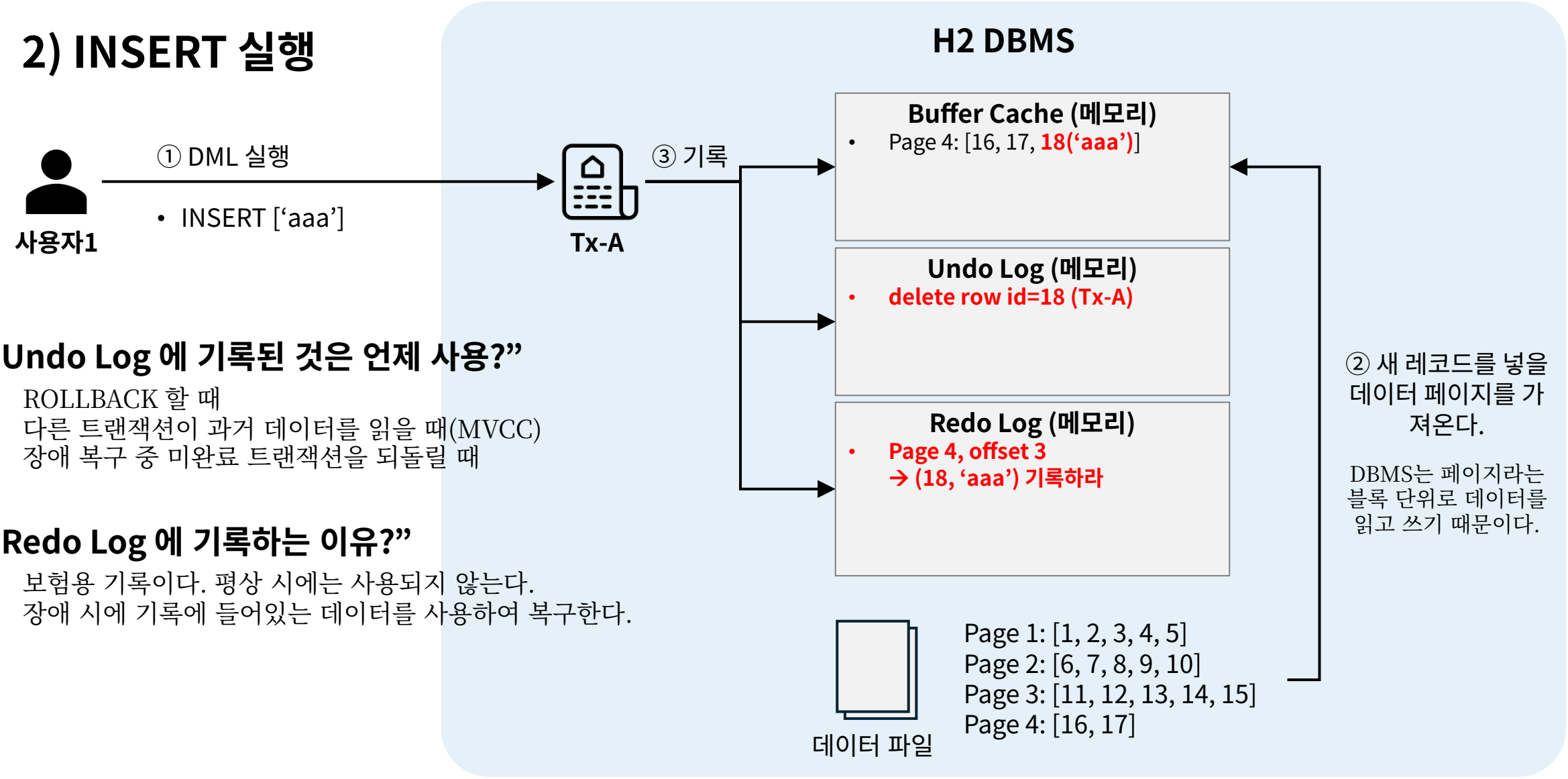
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

1) 구성 요소



8. 트랜잭션 적용하기 – 트랜잭션 구동원리

2) INSERT 실행



“Undo Log 에 기록된 것은 언제 사용?”

- ROLLBACK 할 때
- 다른 트랜잭션이 과거 데이터를 읽을 때(MVCC)
- 장애 복구 중 미완료 트랜잭션을 되돌릴 때

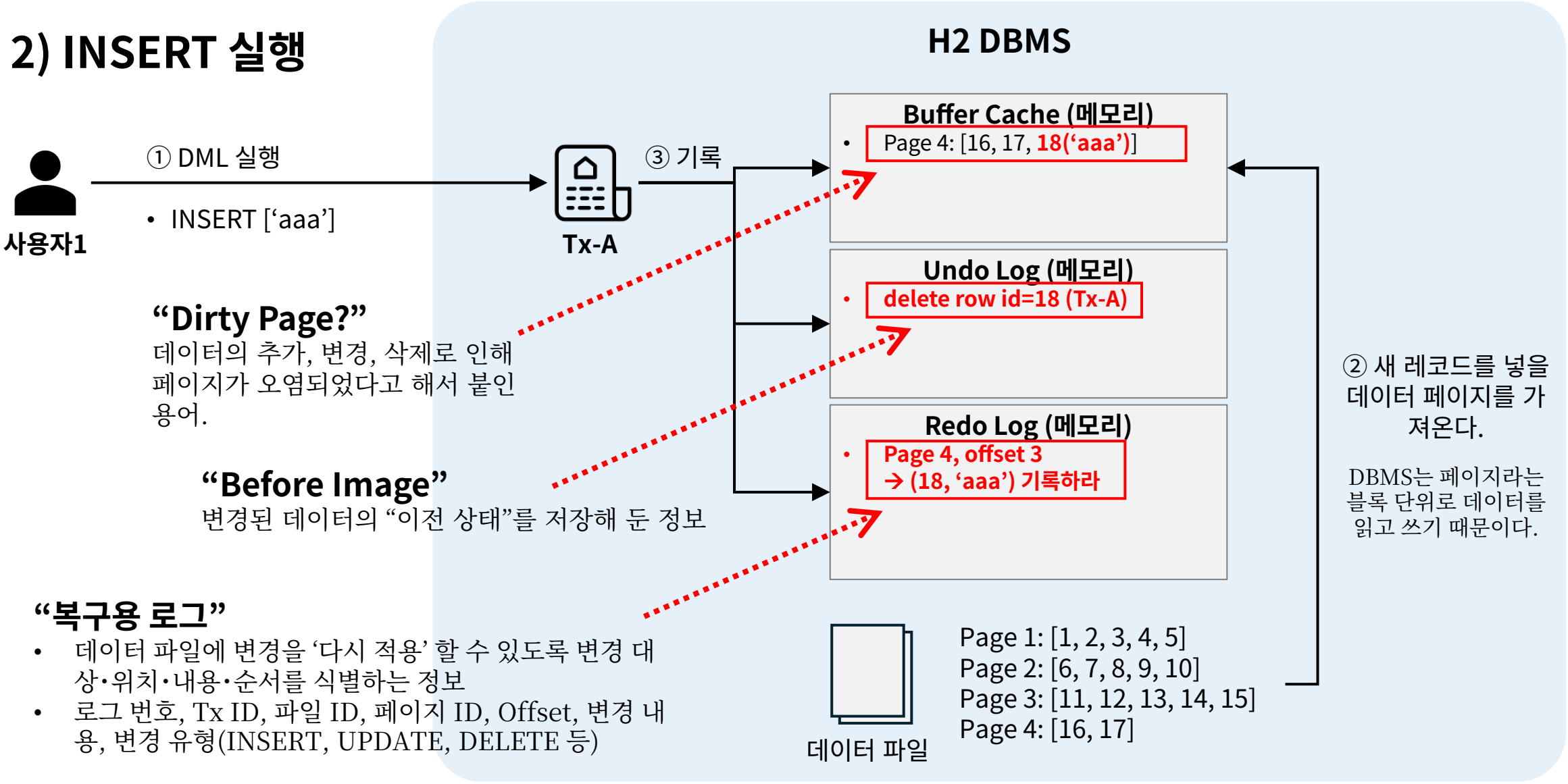
“Redo Log 에 기록하는 이유?”

- 보험용 기록이다. 평상 시에는 사용되지 않는다.
- 장애 시에 기록에 들어있는 데이터를 사용하여 복구한다.

MVCC(Multi-Version Concurrency Control) - 읽기는 과거 버전을 보고, 쓰기는 최신 버전을 쓰게 하여 읽기와 쓰기의 충돌을 피하는 구조.

8. 트랜잭션 적용하기 – 트랜잭션 구동원리

2) INSERT 실행



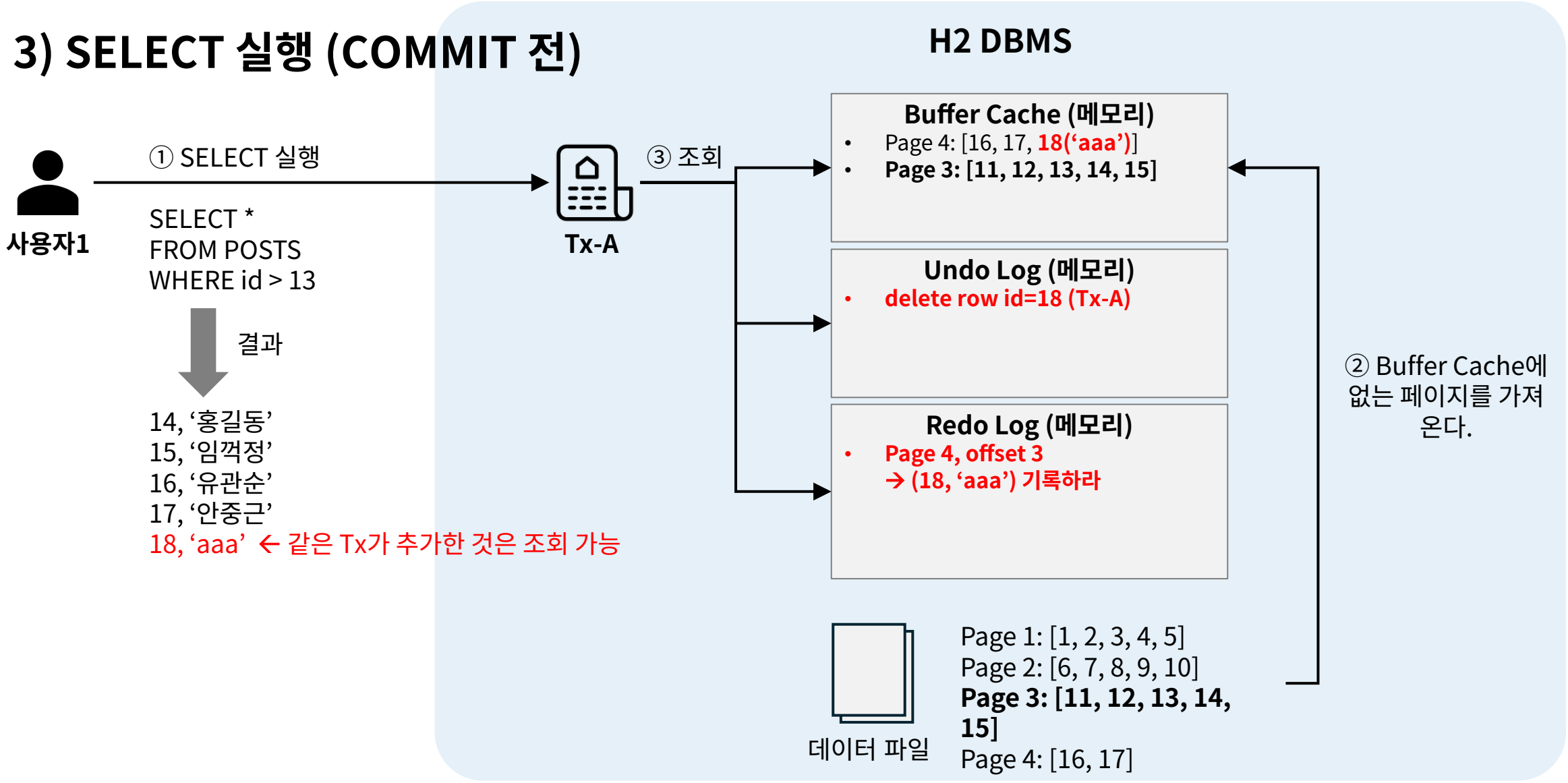
“Dirty Page?”
데이터의 추가, 변경, 삭제로 인해 페이지가 오염되었다고 해서 붙인 용어.

“Before Image”
변경된 데이터의 “이전 상태”를 저장해 둔 정보

“복구용 로그”
- 데이터 파일에 변경을 ‘다시 적용’ 할 수 있도록 변경 대상·위치·내용·순서를 식별하는 정보
- 로그 번호, Tx ID, 파일 ID, 페이지 ID, Offset, 변경 내용, 변경 유형(INSERT, UPDATE, DELETE 등)

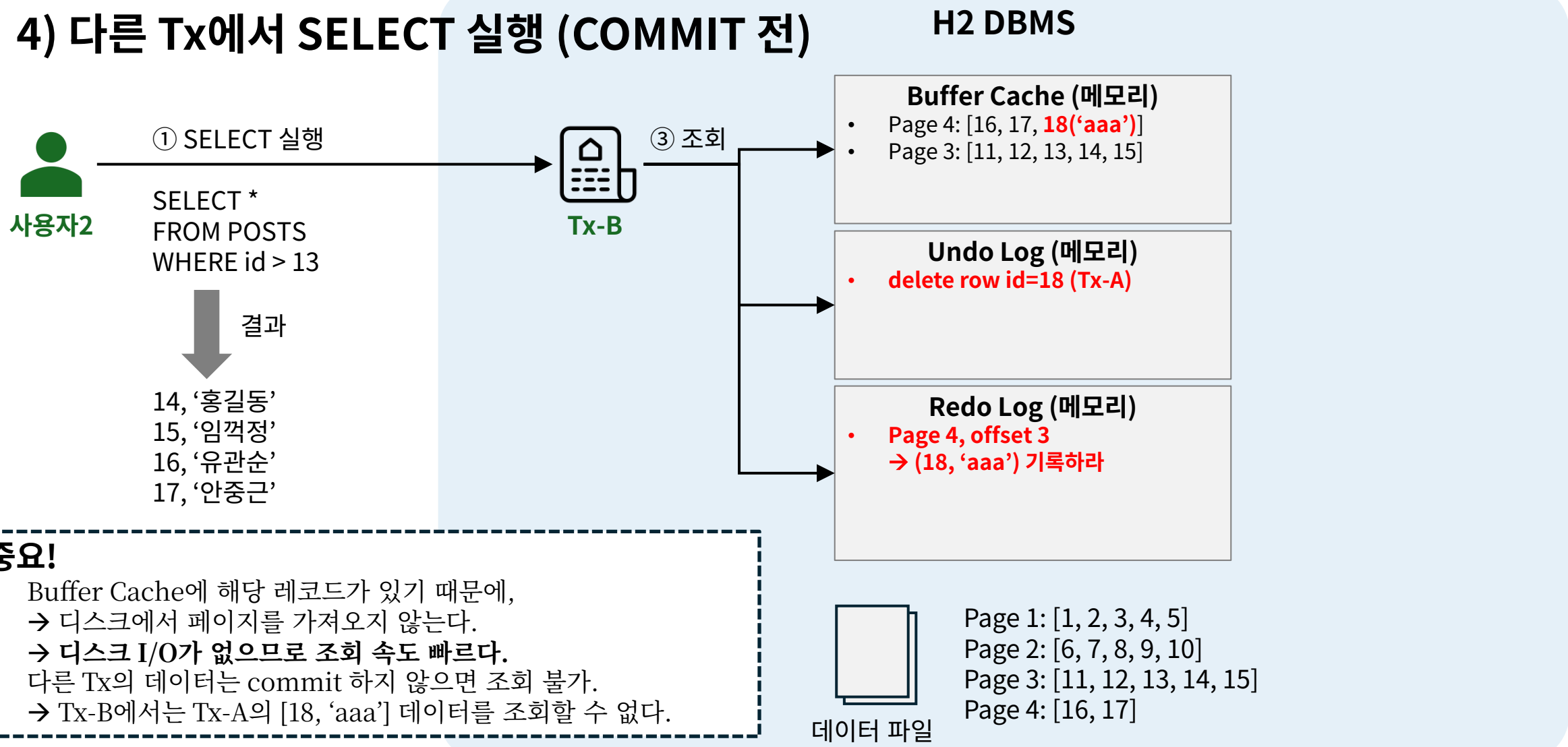
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

3) SELECT 실행 (COMMIT 전)



8. 트랜잭션 적용하기 – 트랜잭션 구동원리

4) 다른 Tx에서 SELECT 실행 (COMMIT 전)

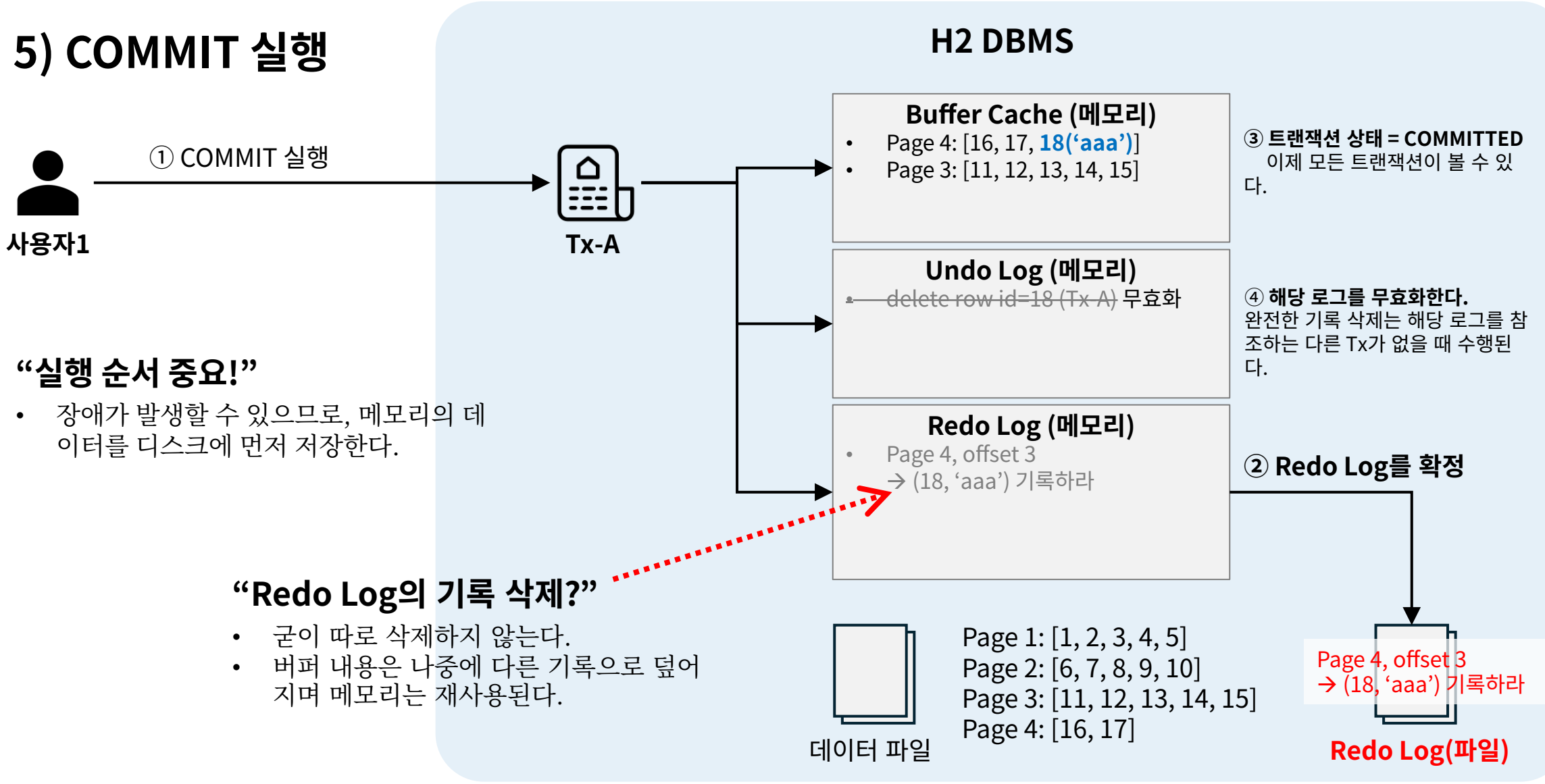


중요!

- Buffer Cache에 해당 레코드가 있기 때문에,
  → 디스크에서 페이지를 가져오지 않는다.
  → **디스크 I/O가 없으므로 조회 속도 빠르다.**
- 다른 Tx의 데이터는 commit 하지 않으면 조회 불가.
  → Tx-B에서는 Tx-A의 [18, 'aaa'] 데이터를 조회할 수 없다.

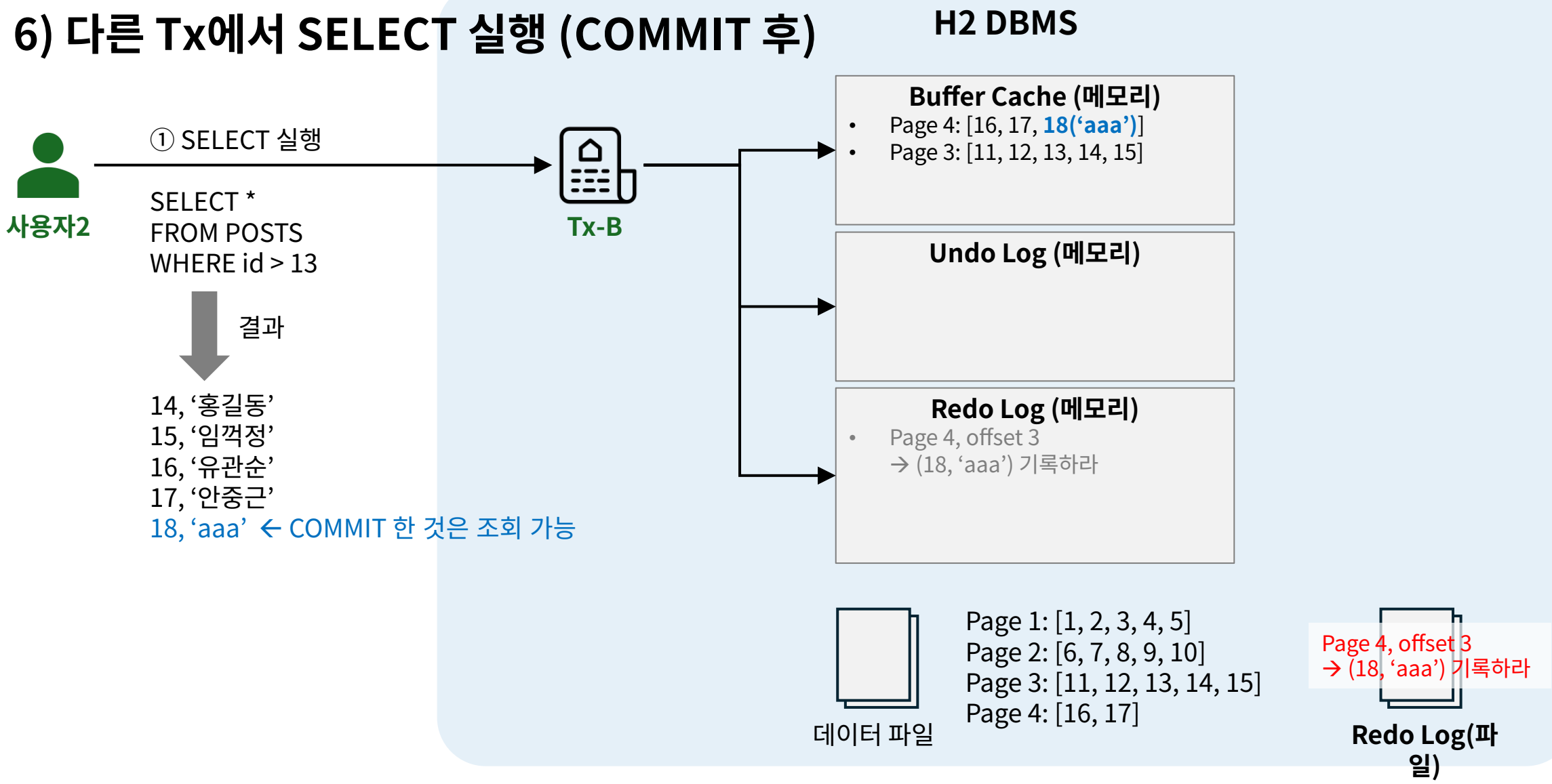
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

5) COMMIT 실행



8. 트랜잭션 적용하기 – 트랜잭션 구동원리

6) 다른 Tx에서 SELECT 실행 (COMMIT 후)

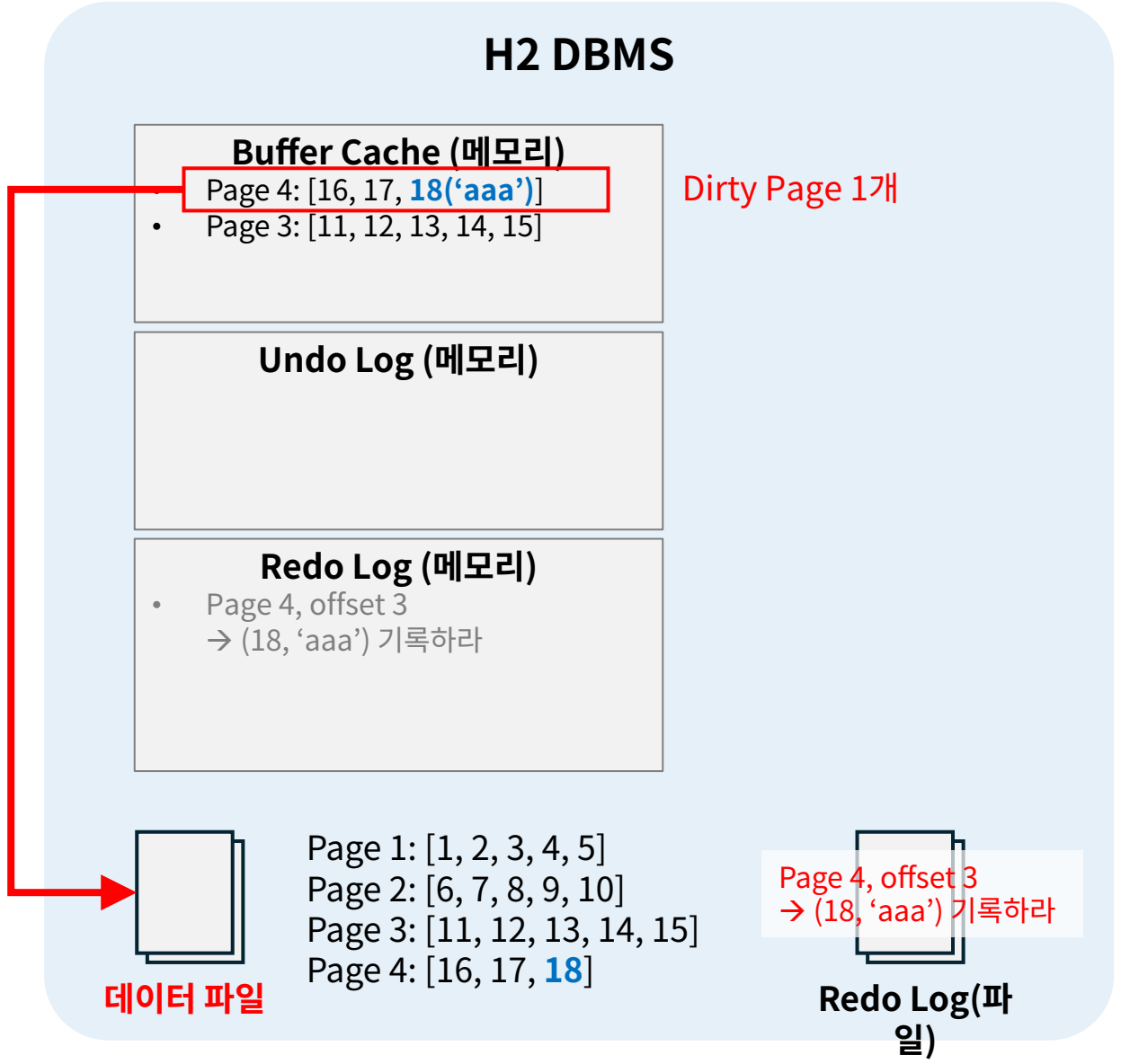


8. 트랜잭션 적용하기 – 트랜잭션 구동원리

7) Checkpoint 발생!

“Checkpoint 란?”

- **Dirty Pages**를 **디스크에 기록**하고 “여기까지는 안전해요!” 라고 표시하는 것.
- **목적:**
  - ① Redo Log 크기 관리
  - ② 복구 시간 단축
  - ③ 메모리 확보
- **발생 시점:**
  - ① 주기적 – 오래된 Dirty Pages부터 기록, Buffer Cache 공간 확보
  - ② Buffer Cache 사용량이 임계값을 초과
  - ③ Redo Log 파일이 가득 찼을 때
  - ④ 명시적 Checkpoint(수동)
  - ⑤ 정상 종료 시(Shutdown)
  - ⑥ 장애 발생 대비 - Background Writer Thread가 오래된 Dirty Page 스캔하여 조금씩 디스크에 기록



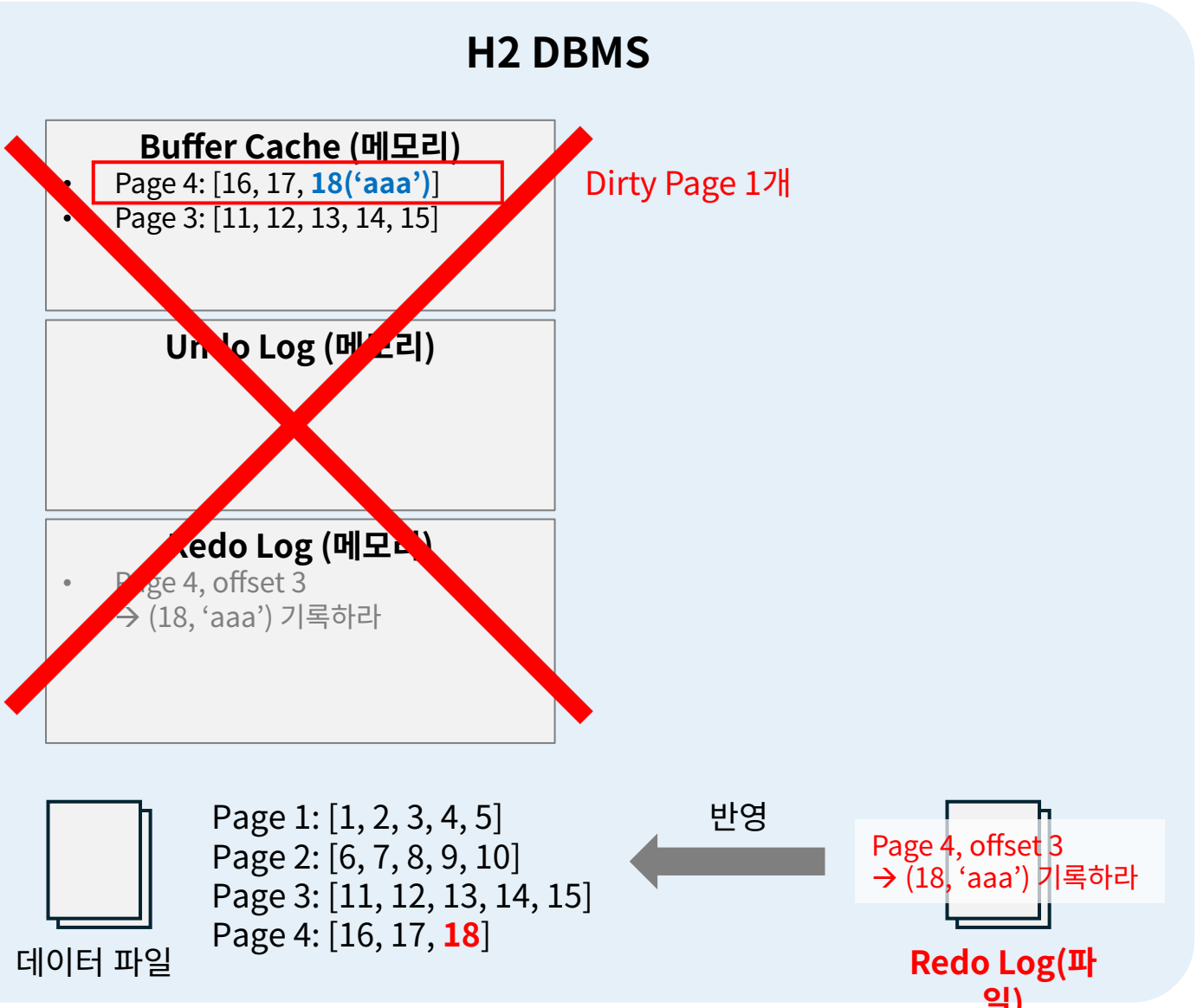
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

8) 장애 발생!

“시나리오: COMMIT 직후 서버 다운”

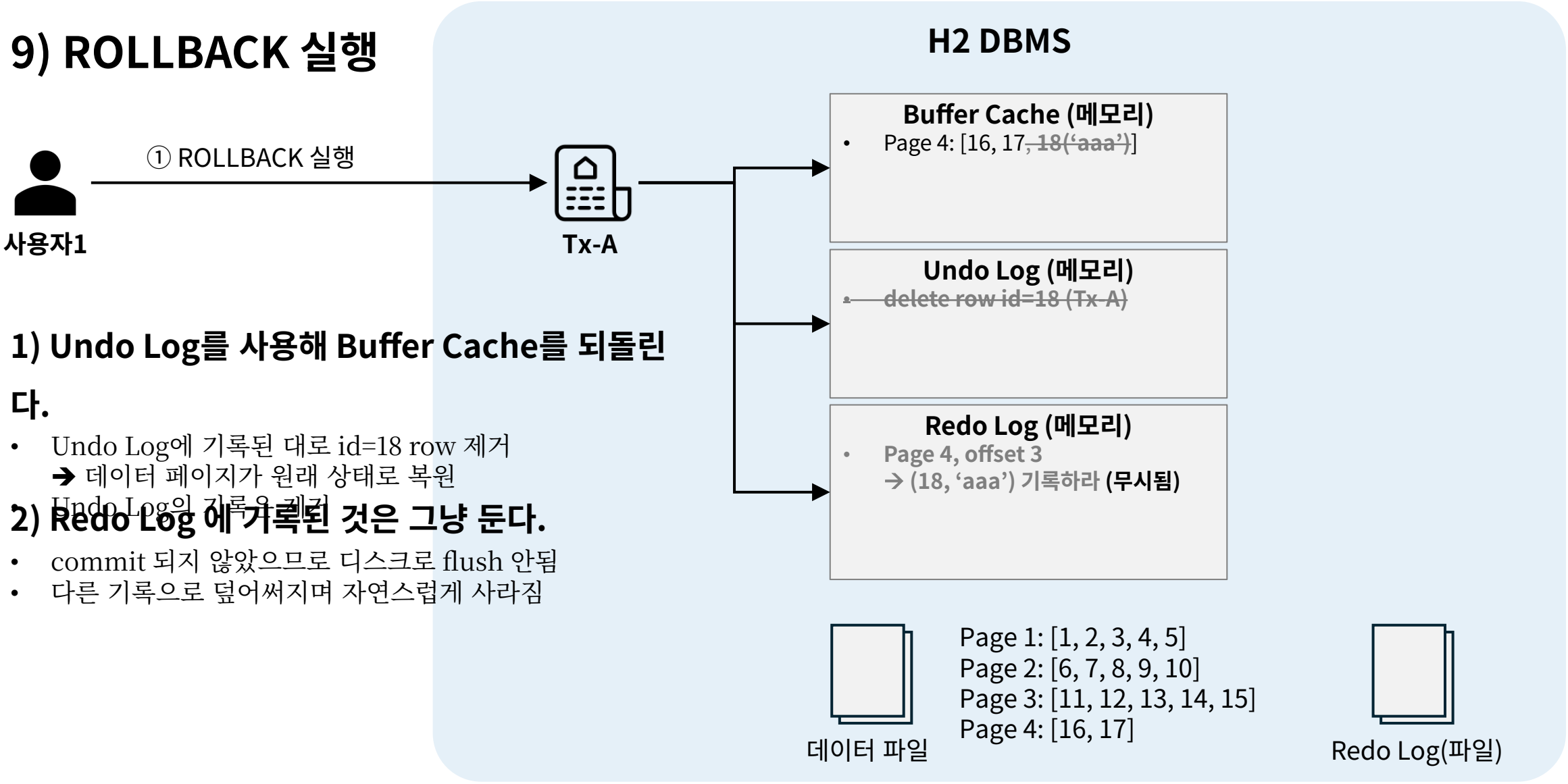
① INSERT 1000개
② COMMIT (Redo Log만 디스크에 저장)
③ 사용자에게 “성공” 응답
④ 서버 다운!
- Buffer Cache 손실
- Dirty Page 손실
- Data File에는 기록 안됨

⑤ **서버 재시작**
- **Redo Log 파일 읽기**
- **“아, INSERT 1000개 했구나!”**
- **Redo Log 재실행**
- **데이터 복구 완료!**



8. 트랜잭션 적용하기 – 트랜잭션 구동원리

9) ROLLBACK 실행



1) Undo Log를 사용해 Buffer Cache를 되돌린다.

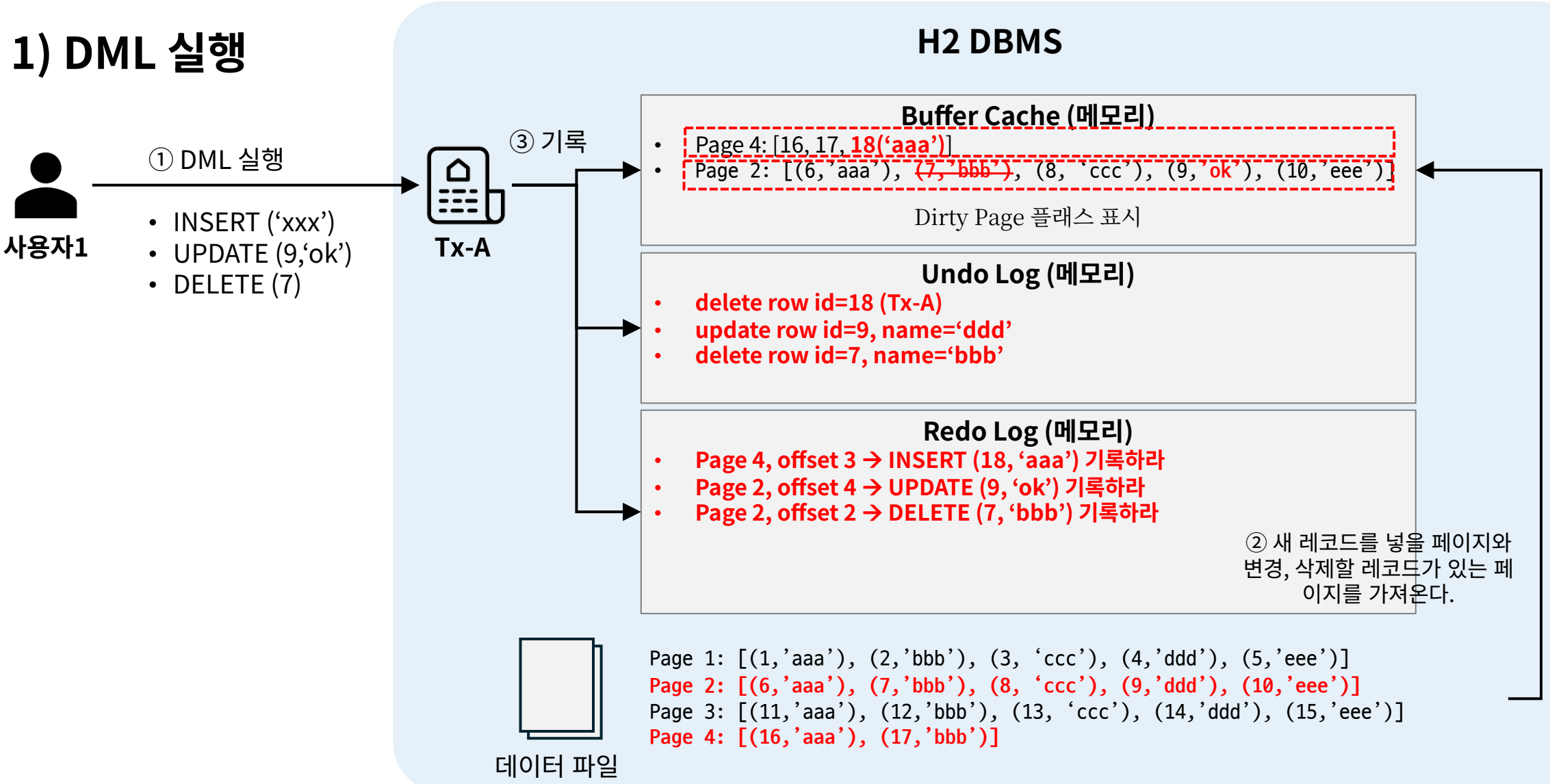
- Undo Log에 기록된 대로 id=18 row 제거
  ➔ 데이터 페이지가 원래 상태로 복원
- Undo Log의 기록은 제거

2) Redo Log 에 기록된 것은 그냥 둔다.

- commit 되지 않았으므로 디스크로 flush 안됨
- 다른 기록으로 덮어써지며 자연스럽게 사라짐

8. 트랜잭션 적용하기 – 트랜잭션 활용 예시

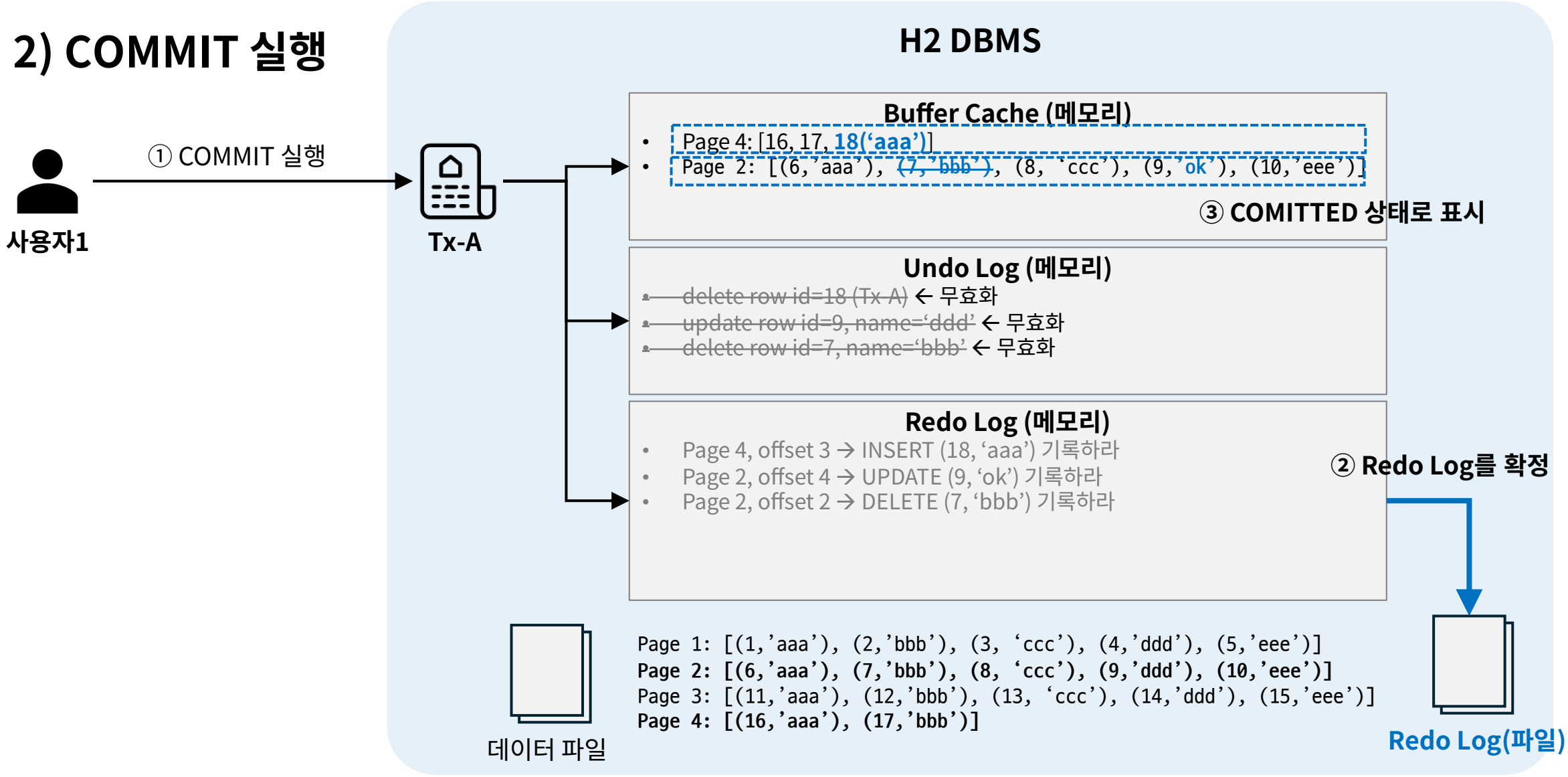
1) DML 실행



MVCC(Multi-Version Concurrency Control) - 읽기는 과거 버전을 보고, 쓰기는 최신 버전을 쓰게 하여 읽기와 쓰기의 충돌을 피하는 구조.

8. 트랜잭션 적용하기 – 트랜잭션 활용 예시

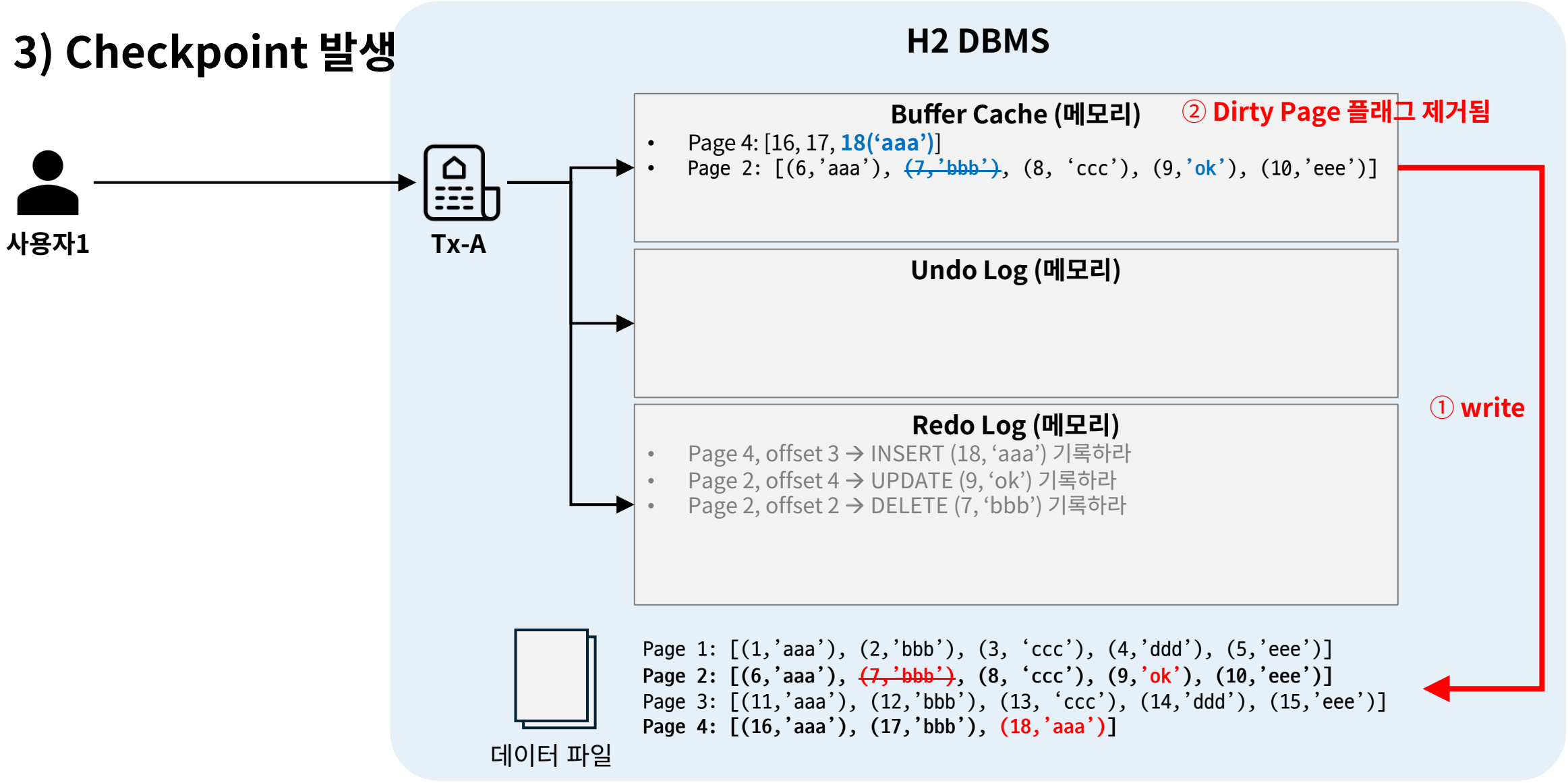
2) COMMIT 실행



MVCC(Multi-Version Concurrency Control) - 읽기는 과거 버전을 보고, 쓰기는 최신 버전을 쓰게 하여 읽기와 쓰기의 충돌을 피하는 구조.

8. 트랜잭션 적용하기 – 트랜잭션 활용 예시

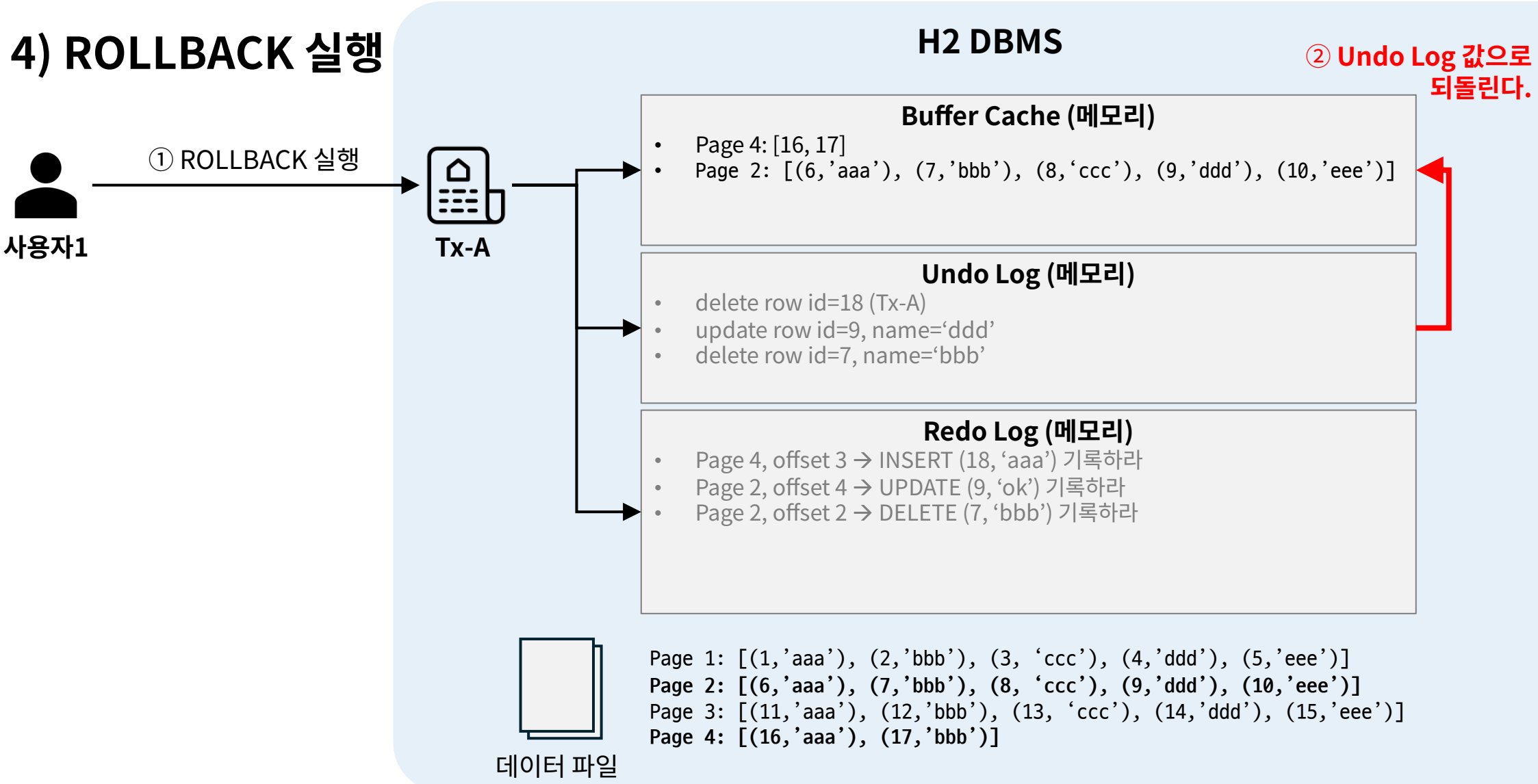
3) Checkpoint 발생



MVCC(Multi-Version Concurrency Control) - 읽기는 과거 버전을 보고, 쓰기는 최신 버전을 쓰게 하여 읽기와 쓰기의 충돌을 피하는 구조.

8. 트랜잭션 적용하기 – 트랜잭션 활용 예시

4) ROLLBACK 실행



MVCC(Multi-Version Concurrency Control) - 읽기는 과거 버전을 보고, 쓰기는 최신 버전을 쓰게 하여 읽기와 쓰기의 충돌을 피하는 구조.

9. SQL Mapper를 ORM으로 바꾸기

학습 목표

- SQL 중심 개발 방식의 한계를 경험하고 ORM 도입 필요성을 설명할 수 있다.
- ORM의 개념과 JPA 표준 스펙의 역할을 이해하고 설명할 수 있다.
- JPA 엔티티를 정의하고 핵심 구성 요소(Entity, EntityManager, 영속성 컨텍스트)의 역할과 객체-테이블 매핑을 설정할 수 있다.
- 엔티티의 생명주기(비영속, 영속, 준영속, 삭제)를 이해하고 각 상태별 동작을 설명할 수 있다.
- EntityManager를 직접 사용하여 Repository를 구현하고 기본 CRUD 기능을 수행할 수 있다.
- 변경 감지(Dirty Checking)와 flush 발생 시점의 동작 원리를 이해하고, JPA에서 트랜잭션 관리가 필수적인 이유를 설명할 수 있다.
- JPQL을 사용하여 엔티티 기반 조회 쿼리를 작성할 수 있다.
- 바이브코딩을 활용하여 순수 JPA 기반 Repository를 구현하고, 반복 코드 패턴을 인식하여 Spring Data JPA의 필요성을 설명할 수 있다.

9. SQL Mapper를 ORM으로 바꾸기 – ORM 이란?

ORM(Object-Relational Mapping)

- 객체와 관계형 데이터베이스를 **자동으로 연결(매핑)**해주는 기술
- **SQL Mapper 방식**:
  - Java 객체 → SQL → DB 테이블
  - **개발자가 SQL을 직접 작성**
- **ORM 방식**:
  - Java 객체 ←→ ORM ←→ DB 테이블
  - 객체를 저장하면 DB에 자동 INSERT
  - 객체를 조회하면 SELECT 결과를 객체로 자동 변환
  - **SQL ←→ 객체 변환을 ORM이 대신 처리**

9. SQL Mapper를 ORM으로 바꾸기 – SQL Mapper vs ORM

항목	SQL Mapper (예: Mybatis)	ORM (예: JPA)
구동 방식	Java 객체 → SQL → DB 테이블	Java 객체 ←→ ORM ←→ DB 테이블
SQL	직접 작성	자동 생성
제어권	개발자	ORM
객체 중심	X	O
생산성	중간	높음
복잡 쿼리	매우 강함	상대적 약함
사용처	CRUD 중심 애플리케이션 도메인 로직이 중요한 서비스 생산성이 중요한 프로젝트	복잡한 SQL 리포트/통계 중심 DB 튜닝이 중요한 시스템

9. SQL Mapper를 ORM으로 바꾸기 – JPA 란?

JPA(Java Persistence API)

- 자바에서 **ORM을 사용하기 위한 표준 인터페이스(규약)**이다
- JPA는 구현체가 아니다.
- “이렇게 ORM을 써야 한다”는 약속(표준 API의 규칙을 정의)이다.
- 구현체: JPA 규격을 실제로 구현한 ORM 라이브러리
  - Hibernate → 사실상 표준, 가장 널리 사용, Spring Boot 기본 선택, 기능 풍부, 커뮤니티 큼
  - EclipseLink → Oracle 주도, JPA 레퍼런스 구현체, 표준 구현 확인용, 학술용
  - OpenJPA → Apache 재단에서 구현, IBM WebSphere 계열에서 사용, 특정 기업/레거시 환경
- 구동:
  - Application Code ➔ JPA(인터페이스) ➔ Hibernate (구현체) ➔ JDBC ➔ Database
  - 참고: Application Code ➔ JDBC API(인터페이스) ➔ JDBC Driver(구현체) ➔ Database

9. SQL Mapper를 ORM으로 바꾸기 – JPA ORM을 쓰는 이유

장점

- SQL 작성량 대폭 감소
- 객체 중심 설계 가능
- DB 벤더 독립성
- 생산성 향상
- 도메인 모델 중심 개발

단점

- 내부 동작 이해 없으면 성능 문제
- 복잡한 쿼리는 오히려 어려움
- 학습 난이도 존재
- “마법처럼 쓰면 망한다”

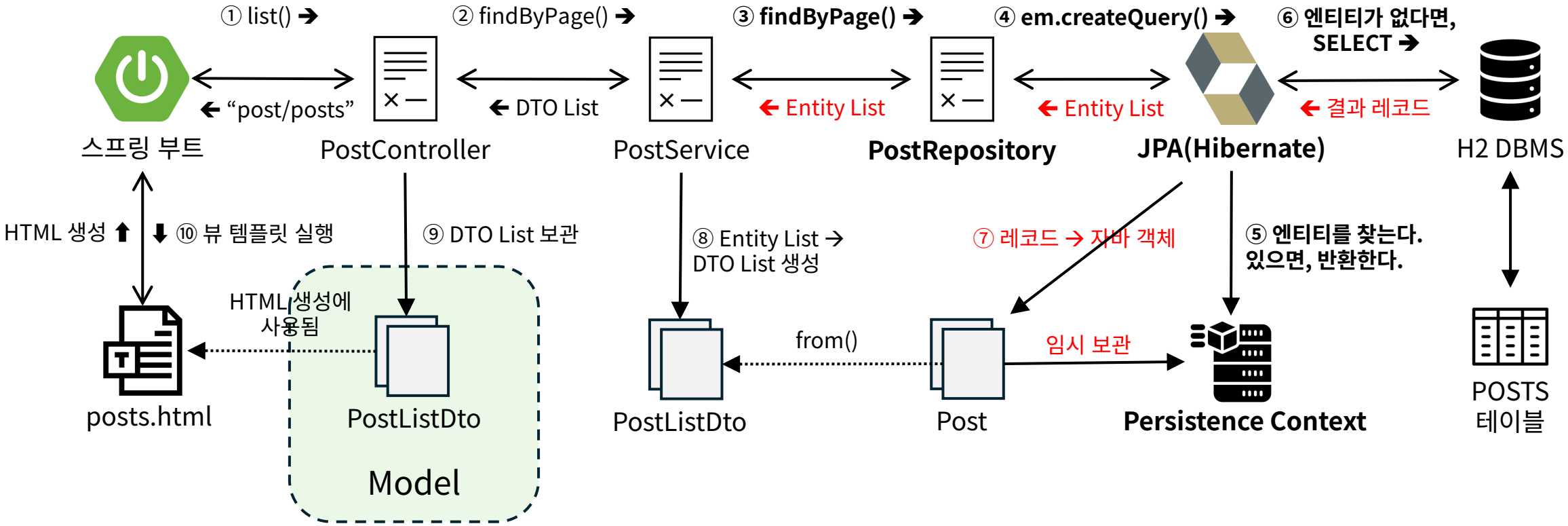
9. SQL Mapper를 ORM으로 바꾸기 – 스프링부트 스타터

spring-boot-starter-data-jpa

- JPA API 포함
- Hibernate 구현체 자동 포함
- EntityManager 자동 설정
- 트랜잭션 자동 구성
- “스프링에서 JPA를 쓴다” ➔ Hibernate 기반 JPA 사용
- 용어 정리
  - **JPA**: ORM 표준 API
  - **Hibernate**: JPA 구현체
  - **Spring Data JPA**: JPA를 쉽게 쓰게 해주는 스프링 프로젝트
  - **Spring-boot-starter-data-jpa**: 위 모든 걸 묶은 스프링부트 스타터

9. SQL Mapper를 ORM으로 바꾸기 – 아키텍처(게시글 목록 조회)

URL: / posts



9. SQL Mapper를 ORM으로 바꾸기 – Persistence Context

- 영속성 컨텍스트는 **트랜잭션 범위 내**에서 **엔티티 객체를 관리**하는 JPA의 메모리 공간이다.
- **트랜잭션 범위 내에서만 유효 → 트랜잭션 종료 시 캐시 초기화**

"Persistence Context = **Entity들의 대기실**"

9. SQL Mapper를 ORM으로 바꾸기 – Persistence Context 핵심 기능

- **객체지향과 관계형 DB 사이의 괴리 해결**
  - 동일설 보장(Identity): **같은 데이터(레코드) = 같은 객체**

```
// 매번 새 객체 생성 (Mybatis 방식)
Post post1 = new Post(1L, "제목");
Post post2 = new Post(1L, "제목");
post1 == post2;  // false (서로 다른 객체)
```

```
// 같은 ID면 같은 객체 반환 (JPA 방식)
Post post1 = em.find(Post.class, 1L);
Post post2 = em.find(Post.class, 1L);
post1 == post2;  // true (1차 캐시에서 같은 객체 반환)
```

- **성능 최적화**
  - SELECT 실행 ➔ DBMS에 질의 ➔ 결과 레코드로 엔티티 객체 생성 ➔ **캐시에 보관**
  - 같은 SELECT 실행 ➔ 캐시에 보관된 것을 사용 (**DBMS 질의 안함!**)

```
@Transactional
public void cacheExample() {

    // ① 첫 조회: DB에서 가져와서 1차 캐시에 저장
    Post post1 = em.find(Post.class, 1L);
    // SQL: SELECT * FROM posts WHERE no = 1

    // ② 두 번째 조회: 1차 캐시에서 반환 (SQL 실행 안함!)
    Post post2 = em.find(Post.class, 1L);

    System.out.println(post1 == post2);  // true
}
```

9. SQL Mapper를 ORM으로 바꾸기 – Persistence Context 핵심 기능(계속)

- **쓰기 지연 (Write-Behind)**
  - INSERT, UPDATE, DELETE 실행 시 **커밋 시점에 한 번에 실행**
  - 단, **IDENTITY 전략**(DB가 ID를 자동 생성)을 사용하는 컬럼이 있는 경우, **즉시 실행**한다.

```
@Transactional
public void writeDelay() {
    Post post1 = new Post();
    em.persist(post1);  // SQL 저장소에 등록만

    Post post2 = new Post();
    em.persist(post2);  // SQL 저장소에 등록만
    // 여기까지 INSERT SQL 실행 안됨

    // 커밋 시점에 한 번에 실행
    // INSERT INTO posts ...  (post1)
    // INSERT INTO posts ...  (post2)
}
```

- **변경 감지 (Dirty Checking)**
  - setter 호출 ➔ 트랜잭션 커밋할 때 **자동으로 UPDATE SQL 실행**

```
@Transactional
public void autoUpdate() {

    Post post = em.find(Post.class, 1L);
    // 스냅샷 저장: {title: "A", content: "B"}

    post.setTitle("A2");

    post.setContent("B2");

    // em.update(post); 이런 거 필요 없음!

    // 커밋 시:
    // 스냅샷과 비교 → 변경 감지 → UPDATE 자동 실행
}
```

9. SQL Mapper를 ORM으로 바꾸기 – Persistence Context 핵심 기능(계속)

- **지연 로딩 (Lazy Loading)**
  - 연관된 엔티티를 실제로 사용할 때까지 조회를 미루는 것
  - 사용하지 않으면 영원히 조회 안함

```
// 즉시 로딩 예:
@Entity
public class Post {
    @Id
    private Long no;

    @ManyToOne(fetch = FetchType.EAGER) // 즉시 로딩
    private User author;
}

Post post = em.find(Post.class, 1L);
// - POST 조회 시 User도 함께 조회
// - JOIN 쿼리 한 번으로 데이터 모두 가져옴
// - author를 사용하지 않아도 무조건 조회
// - 조인 SQL 예:
//     SELECT p.*, u.* FROM posts p
//       LEFT JOIN users u ON p.author_id = u.id
//     WHERE p.no = 1
```

```
// 지연 로딩 예:
@Entity
public class Post {
    @Id
    private Long no;

    @ManyToOne(fetch = FetchType.LAZY)
    private User author;  // 필요할 때만 로딩
}

Post post = em.find(Post.class, 1L);
// - Post만 조회
//     SELECT * FROM posts WHERE no = 1

String authorName = post.getAuthor().getName();
// - 여기서 author 조회
//     SELECT * FROM users WHERE id = ?
```

9. SQL Mapper를 ORM으로 바꾸기 – 엔티티 생명주기와 엔티티 상태

비영속 (new)	영속 (managed)	준영속 (detached)	삭제 (removed)

`Post p = new Post();`
`p.setTitle("…");`
	`em.persist(post);`	`em.detach(post);`
		`// 트랜잭션 종료`
		`// em.close() 호출`
			`em.remove(post);`

• 영속성 컨텍스트와 관계가 없는 **순수 Java 객체**	• **1차 캐시에 저장됨** • 변경 감지 대상이 됨 • 트랜잭션 커밋 시 자동으로 DB 동기화	• 영속성 컨텍스트에서 분리 • 변경 감지 안됨 • **1차 캐시에서 제거됨**	• 삭제 예정 상태 • **트랜잭션 커밋 시 DELETE SQL 실행**

9. SQL Mapper를 ORM으로 바꾸기 – 기타 메서드

- **JPQL 조회**
  - 조건 조회 및 목록 조회, SQL 아니고 JPQL 이다. 엔티티 기준 쿼리다. 결과는 영속 상태다.

```
List<Post> posts = em.createQuery("select p from Post p", Post.class).getResultList();
```

- **merge() (주의해서 사용할 것!)**
  - 준영속 엔티티를 **다시 영속 상태로 복귀**시킨다.
  - **새로운 객체를 반환**한다. **모든 필드를 덮어쓴다**. **실무에서 update 용도로 사용하지 말라.**

```
Post merged = em.merge(detachedPost);
```

9. SQL Mapper를 ORM으로 바꾸기 – 엔티티의 영속성 컨텍스트 예시

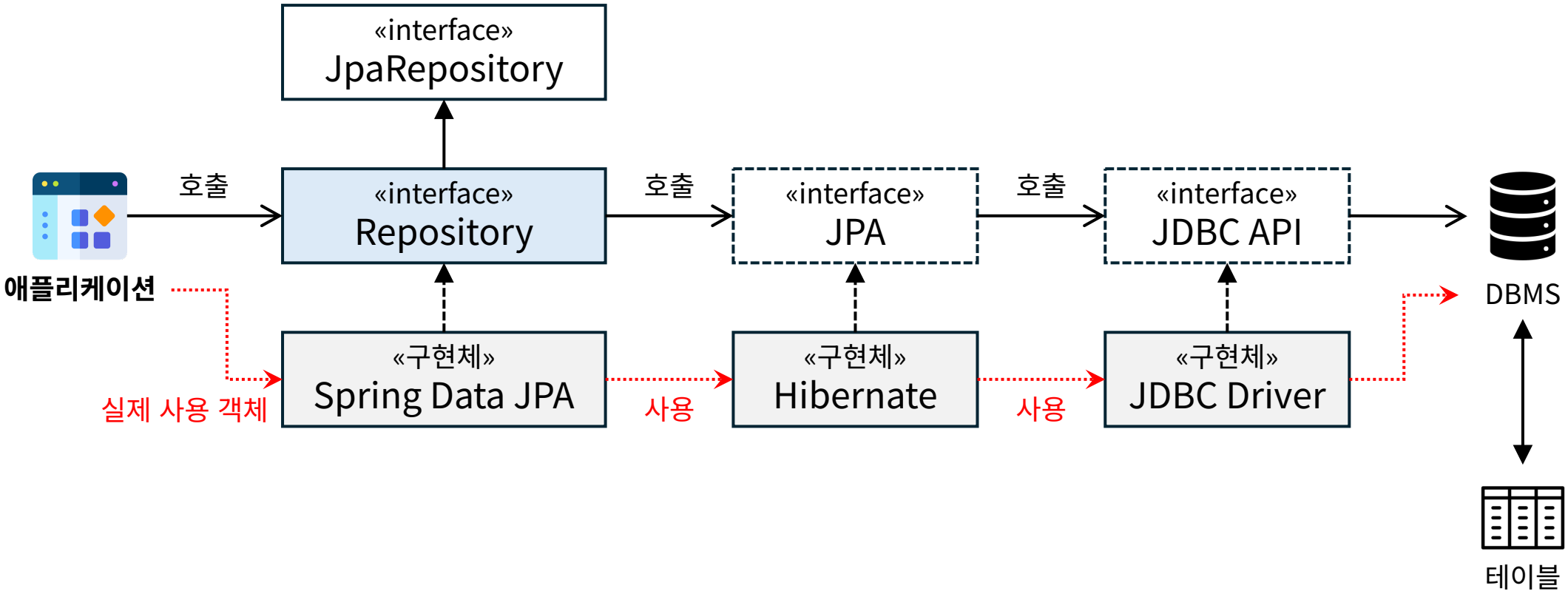
```java
@Service
public class PostService {
    @PersistenceContext
    private EntityManager em;

    @Transactional
    public void persistenceContextExample() {

        // 1. 비영속 → 영속
        Post newPost = new Post();  // 비영속
        newPost.setTitle("새 게시글");

        // 영속 (1차 캐시 등록)
        em.persist(newPost);

        // 2. 1차 캐시 확인
        Long id = newPost.getNo();
        Post cached = em.find(Post.class, id);
        // SQL 실행 안함!

        System.out.println(newPost == cached);  // true

        // 3. 변경 감지
        cached.setTitle("수정된 제목");
        // UPDATE SQL 자동 예약

        // 4. 준영속으로 만들기
        em.detach(cached);
        cached.setTitle("다시 수정");
        // 이건 UPDATE 안됨

        // 커밋 시: UPDATE (detach 전 변경만 반영)
    }
}
```

10. Spring Data JPA로 리팩토링하기

학습 목표

- Spring Data JPA의 **역할과 등장 배경**을 설명할 수 있다.
- Repository 추상화와 **인터페이스 기반 프로그래밍**을 이해하고, 기존 EntityManager 기반 Repository를 Spring Data JPA로 리팩토링할 수 있다.
- **JpaRepository 인터페이스를 활용**하여 기본 CRUD 기능을 구현할 수 있다.
- Query Method와 @Query를 활용하여 다양한 조회 쿼리를 작성할 수 있다.
- Pageable과 Sort를 사용하여 페이징과 정렬 기능을 구현할 수 있다.
- SQL 중심 사고에서 **도메인(엔티티) 중심 설계로 전환**하고, 비즈니스 로직을 엔티티에 배치할 수 있다.
- 트랜잭션과 Spring Data JPA의 연계 방식을 이해한다.
- 바이브코딩으로 점진적 리팩토링을 경험하고, Spring Data JPA의 장점과 한계를 설명할 수 있다.

10. Spring Data JPA로 리팩토링하기 – 계층 구조

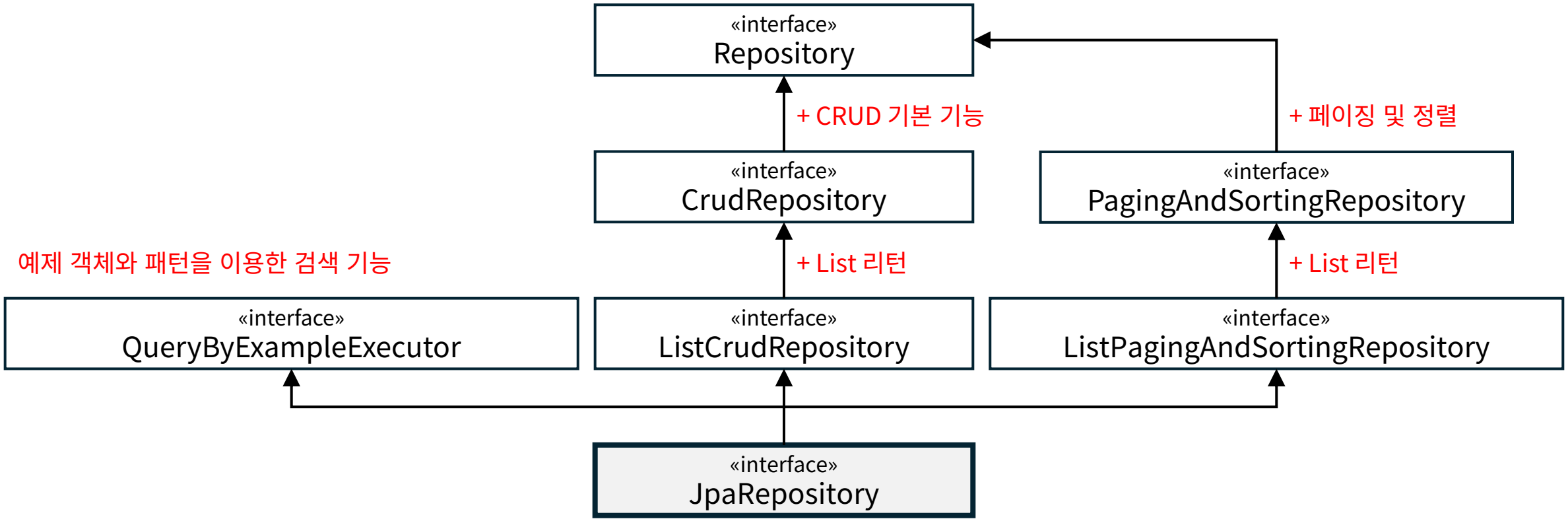


10. Spring Data JPA로 리팩토링하기 – 순수 JPA vs Spring Data JPA

구분	순수 JPA + Hibernate	Spring Data JPA
개념	EntityManager를 직접 사용하여 엔티티의 영속성, 쿼리, 트랜잭션을 직접 제어하는 방식	JPA(EntityManager)를 내부에서 사용하면서, Repository 인터페이스만 정의하면 구현을 자동으로 만들어주는 프레임워크
사용 계층	JPA 직접 사용	JPA 위의 추상화
Repository 구현	직접 작성	자동 생성
EntityManager	직접 사용	내부에서 사용
CRUD 코드	많음	거의 없음
쿼리 작성	JPQL 직접	메서드명 / JPQL / QueryDSL
학습 목적	JPA 원리 이해	생산성 향상
제어 수준	매우 높음	상대적으로 제한
용도	• JPA 내부 동작을 학습하는 단계 • 복잡한 영속성 컨텍스트 제어 필요 • Repository 동작을 세밀하게 통제해야 할 때 • 프레임워크 의존도를 최소화하고 싶을 때	• 일반적인 CRUD 중심 애플리케이션 • 빠른 개발 생산성 • 표준적인 Repository 패턴 • 팀 협업, 유지보수 중시

10. Spring Data JPA로 리팩토링하기 – JpaRepository

- JpaRepository는 Spring Data JPA의 **핵심 인터페이스**이다.
- JPA(EntityManager)를 직접 쓰지 않고도 엔티티의 CRUD·페이징·정렬을 표준 방식으로 제공하는 리포지토리 추상화 인터페이스이다.



10. Spring Data JPA로 리팩토링하기 – JpaRepository

«interface»
Repository

- Spring Data Repository 계층의 기준점이 되는 **마커 인터페이스**이다.
- 메서드는 없다. 단지 **자동으로 구현체를 생성해야 하는 대상임을 표시**하는 용도다.

«interface»
CrudRepository

- 엔티티의 기본 CRUD 기능을 제공한다.
- save(), findById(), findAll(), delete(), deleteById(), count(), existsById() 등

«interface»
PagingAndSortingRepository

- 대용량 데이터 처리를 위한 **페이징** 및 **정렬** 기능을 제공한다.
- findAll(Pageable pageable), findAll(Sort sort)

«interface»
JpaRepository

- JPA 영속성 컨텍스트 제어와 배치 처리, flush 제어, JPA 최적화 기능을 제공한다.
- flush(), saveAndFlush(), deleteAllBatch(), deleteAllByIdInBatch() 등
- **실무에서 주로 사용하는 인터페이스다.**

10. Spring Data JPA로 리팩토링하기 – JpaRepository 사용

- 리포지토리 인터페이스는 **JpaRepository를 상속** 받아 정의한다.

```
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long no;  // ← ID 타입

    private String title;
    private String content;
}

public interface PostRepository extends JpaRepository<Post, Long> {
}
```

엔티티 ID 타입

10. Spring Data JPA로 리팩토링하기 – JpaRepository 주요 메서드 사용법

save() – 저장 및 수정

```
Post post = new Post();
post.setTitle("제목");
postRepository.save(post);  // em.persist() → INSERT

Post existing = postRepository.findById(1L).orElseThrow();
existing.setTitle("수정된 제목");
postRepository.save(existing);  // em.merge() → SELECT + UPDATE
```

- 신규 데이터 판단 기준:
  - @Id 필드 값이 **null** → 신규
  - @Id 필드 값이 **0(primitive)** → 신규
  - @Id 필드 값이 **있으면** → 기존(merge)
- 기존 데이터 변경할 때:
  - 영속 상태일 때는 save() 생략 가능 → Dirty Checking 을 한다.
  - 즉 @Transactional 내에서는 save() 호출 불필요!

10. Spring Data JPA로 리팩토링하기 – JpaRepository 주요 메서드 사용법

findById() – 조회

```
// 예1) Optional 반환
Optional<Post> optionalPost = postRepository.findById(1L);

// 예2) 없으면 → 예외 발생
Post post = postRepository.findById(1L)
        .orElseThrow(() -> new EntityNotFoundException("Post not found"));

// 예3) 없으면 → null 리턴
Post post = postRepository.findById(1L)
        .orElse(null);

// 예4) 있는지 여부 검사
if (postRepository.findById(1L).isPresent()) {
    Post post = postRepository.findById(1L).get();
}
```

10. Spring Data JPA로 리팩토링하기 – JpaRepository 주요 메서드 사용법

findAll() – 전체 조회

```
// 기본
List<Post> posts = postRepository.findAll();

// 정렬
List<Post> posts = postRepository.findAll(Sort.by("no").descending());
List<Post> posts = postRepository.findAll(Sort.by("createdAt").descending());

// 여러 정렬 조건
Sort sort = Sort.by("createdAt")
                .descending()
                .and(Sort.by("title")
                .ascending());
List<Post> posts = postRepository.findAll(sort);
```

10. Spring Data JPA로 리팩토링하기 – JpaRepository 주요 메서드 사용법

페이징

```
// 페이지 요청 생성
Pageable pageable = PageRequest.of(
    0,       // 페이지 번호 (0부터 시작)
    10,      // 페이지 크기
    Sort.by("no").descending()   // 정렬 (선택)
);

// 페이지 조회
Page<Post> page = postRepository.findAll(pageable);
```

10. Spring Data JPA로 리팩토링하기 – JpaRepository 주요 메서드 사용법

deleteById() vs delete()

```
// 방법 1: ID로 삭제 (먼저 조회 후 삭제)
postRepository.deleteById(1L);
// SELECT * FROM posts WHERE no = 1
// DELETE FROM posts WHERE no = 1

// 방법 2: Entity로 삭제 (바로 삭제)
Post post = postRepository.findById(1L).orElseThrow();
postRepository.delete(post);
// DELETE FROM posts WHERE no = 1

// 효율적인 방법
@Modifying
@Query("DELETE FROM Post p WHERE p.no = :no")
void deleteByNo(@Param("no") Long no);
// DELETE FROM posts WHERE no = ? (조회 없이 바로 삭제)
```

10. Spring Data JPA로 리팩토링하기 – JpaRepository 주요 메서드 사용법

count() & existsById()

```
// 전체 개수
long count = postRepository.count();
// SELECT COUNT(*) FROM posts

// 존재 확인
boolean exists = postRepository.existsById(1L);
// SELECT COUNT(*) FROM posts WHERE no = 1

// 활용
if (!postRepository.existsById(id)) {
    throw new EntityNotFoundException();
}
```

10. Spring Data JPA로 리팩토링하기 – N + 1 데이터 조회 문제와 해결책

N+1 문제

```
// N+1 발생
List<Post> posts = postRepository.findAll();
for (Post post : posts) {
    System.out.println(post.getAuthor().getName()); // N번 추가 조회!
}
// SELECT * FROM posts -- 1번
// SELECT * FROM users WHERE id = ? -- N번

// 해결 1) Fetch Join
@Query("SELECT p FROM Post p JOIN FETCH p.author")
List<Post> findAllWithAuthor();

// 해결 2) @EntityGraph
@EntityGraph(attributePaths = {"author"})
List<Post> findAll();
// SELECT p.*, u.* FROM posts p LEFT JOIN users u ON p.author_id = u.id
// 쿼리 1번으로 모든 데이터 조회!
```

10. Spring Data JPA로 리팩토링하기 – Query Method

- **메서드 이름**만으로 **쿼리를 자동으로 생성**하는 기법이다.

```
public interface PostRepository extends JpaRepository<Post, Long> {

    List<Post> findByTitle(String title);

}
```

Spring Data JPA가 메서드 이름을 분석하여 쿼리를 자동 생성

➔ `SELECT p FROM Post p WHERE p.title = ?1`

10. Spring Data JPA로 리팩토링하기 – Query Method 이름 짓는 규칙

키워드	메서드명	JPQL
And	findByTitleAndContent	WHERE title = ? AND content = ?
Or	findByTitleOrContent	WHERE title = ? OR content = ?
Is, Equals	findByTitle	WHERE title = ?
Between	findByNoBetween	WHERE no BETWEEN ? AND ?
LessThan	findByNoLessThan	WHERE no < ?
GreaterThan	findByNoGreaterThan	WHERE no > ?
Like	findByTitleLike	WHERE title LIKE ?
Containing	findByTitleContaining	WHERE title LIKE %?%
StartingWith	findByTitleStartingWith	WHERE title LIKE ?%
EndingWith	findByTitleEndingWith	WHERE title LIKE %?
OrderBy	findByTitleOrderByNoDesc	ORDER BY no DESC
Not	findByTitleNot	WHERE title <> ?

10. Spring Data JPA로 리팩토링하기 – Query Method 이름 짓는 규칙(계속)

키워드	메서드명	JPQL
In	`findByNoIn(List<Long>)`	`WHERE no IN (?)`
NotIn	`findByNoNotIn(List<Long>)`	`WHERE no NOT IN (?)`
True/False	`findByDeletedTrue`	`WHERE deleted = true`
IsNull	`findByTitleIsNull`	`WHERE title IS NULL`
IsNotNull	`findByTitleIsNotNull`	`WHERE title IS NOT NULL`

10. Spring Data JPA로 리팩토링하기 – @Query

JPQL

```
@Query("SELECT p FROM Post p " +
        "WHERE p.title LIKE %:keyword% OR p.content LIKE %:keyword%")
List<Post> searchByKeyword(@Param("keyword") String keyword);
```

Native SQL

```
@Query(
    value = "SELECT * FROM posts WHERE created_at > NOW() - INTERVAL 7 DAY",
    nativeQuery = true)
List<Post> findRecentPosts();
```

10. Spring Data JPA로 리팩토링하기 – @Query

수정 쿼리

```
@Modifying
@Query("UPDATE Post p SET p.viewCount = p.viewCount + 1 WHERE p.no = :no")
void incrementViewCount(@Param("no") Long no);
```

DTO 프로젝션

```
@Query("SELECT new com.example.dto.PostListDto(p.no, p.title, p.createdAt) " +
            "FROM Post p ORDER BY p.no DESC")
List<PostListDto> findAllAsDto();
```

10. Spring Data JPA로 리팩토링하기 – QueryDSL

- 타입 안전한 방식으로 **SQL/JPQL을 자바 코드로** 작성할 수 있게 해주는 프레임워크이다.

기존 방식(문자열 쿼리)의 문제점:

- 오타나 필드명을 변경해도 컴파일 할 때 에러 안남.

```
@Query("SELECT p FROM Post p WHERE p.title = :title")
List<Post> findByTitle (@Param(”title") String title);
```

QueryDSL 방식(자바 코드)의 장점:

- 컴파일 타임에 오류 검출됨. IDE 자동 완성 기능을 활용할 수 있음.

```
QPost post = QPost.post;
List<Post> result = queryFactory
    .selectFrom(post)
    .where(post.title.eq(title))
    .fetch();
```

10. Spring Data JPA로 리팩토링하기 – 기존 방식들의 한계

Query Method의 한계:

- OR 조건 표현 어려움
- 동적 쿼리 불가능
- 검색 조건이 선택적으로 들어올 때?

```
List<Post> findByTitleContainingOrContentContaining(String keyword1, String keyword2);
```

10. Spring Data JPA로 리팩토링하기 – 기존 방식들의 한계

@Query의 한계:

- 문자열이라 오타 발견 어려움 → 컴파일은 성공, 런타임에 오류!
- 동적 쿼리 작성 복잡

```
@Query("SELECT p FROM Post p WHERE p.tittle = :title")  // tittle 오타!
List<Post> findByTitle(@Param("title") String title);


@Query("SELECT p FROM Post p WHERE " +
       "(:title IS NULL OR p.title LIKE %:title%) AND " +
       "(:content IS NULL OR p.content LIKE %:content%)")
List<Post> search(@Param("title") String title, @Param("content") String content);
```

10. Spring Data JPA로 리팩토링하기 – 기존 방식들의 한계

Criteria API의 한계:

- JPA 표준이지만 너무 복잡하고 가독성 최악
- 읽기 어렵고 유지보수 힘듦

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Post> query = cb.createQuery(Post.class);
Root<Post> post = query.from(Post.class);
query.select(post)
     .where(cb.equal(post.get("title"), title));
```

10. Spring Data JPA로 리팩토링하기 – QueryDSL 장점

타입 안전:

```
QPost post = QPost.post;
queryFactory
    .selectFrom(post)
    .where(post.title.eq(title))  // title 필드가 없으면 컴파일 에러!
    .fetch();
```

10. Spring Data JPA로 리팩토링하기 – QueryDSL 장점

동적 쿼리 간편:

```
BooleanBuilder builder = new BooleanBuilder();
if (title != null) builder.and(post.title.contains(title));
if (content != null) builder.and(post.content.contains(content));


queryFactory.selectFrom(post)
    .where(builder)
    .fetch();
```

10. Spring Data JPA로 리팩토링하기 – QueryDSL 장점

가독성 좋음:

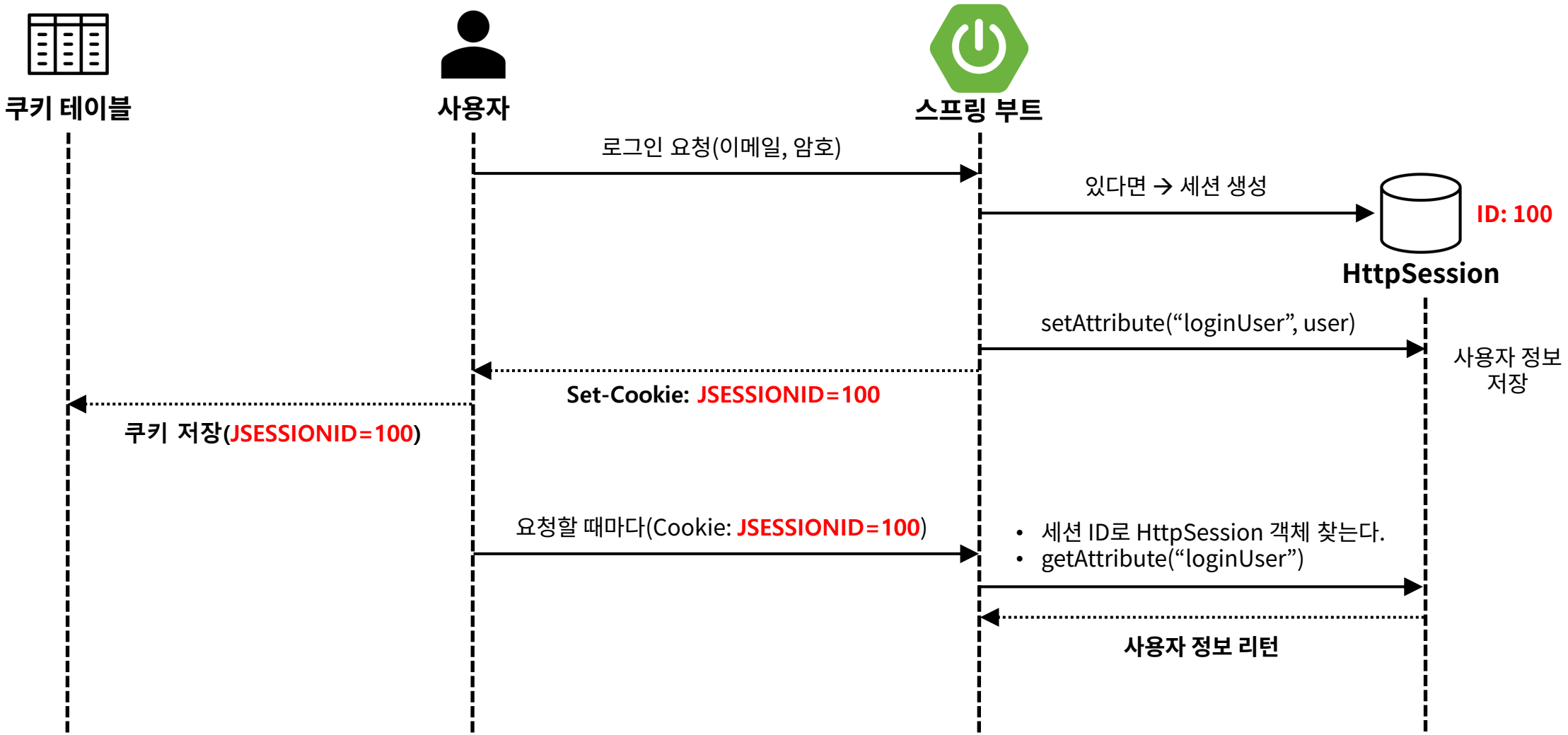
```
List<Post> results = queryFactory
    .selectFrom(post)
    .where(
        post.title.contains(keyword)
            .or(post.content.contains(keyword))
    )
    .orderBy(post.createdAt.desc())
    .limit(10)
    .fetch();
```

11. 세션 기반 사용자 인증하기(without Spring Security)

학습 목표

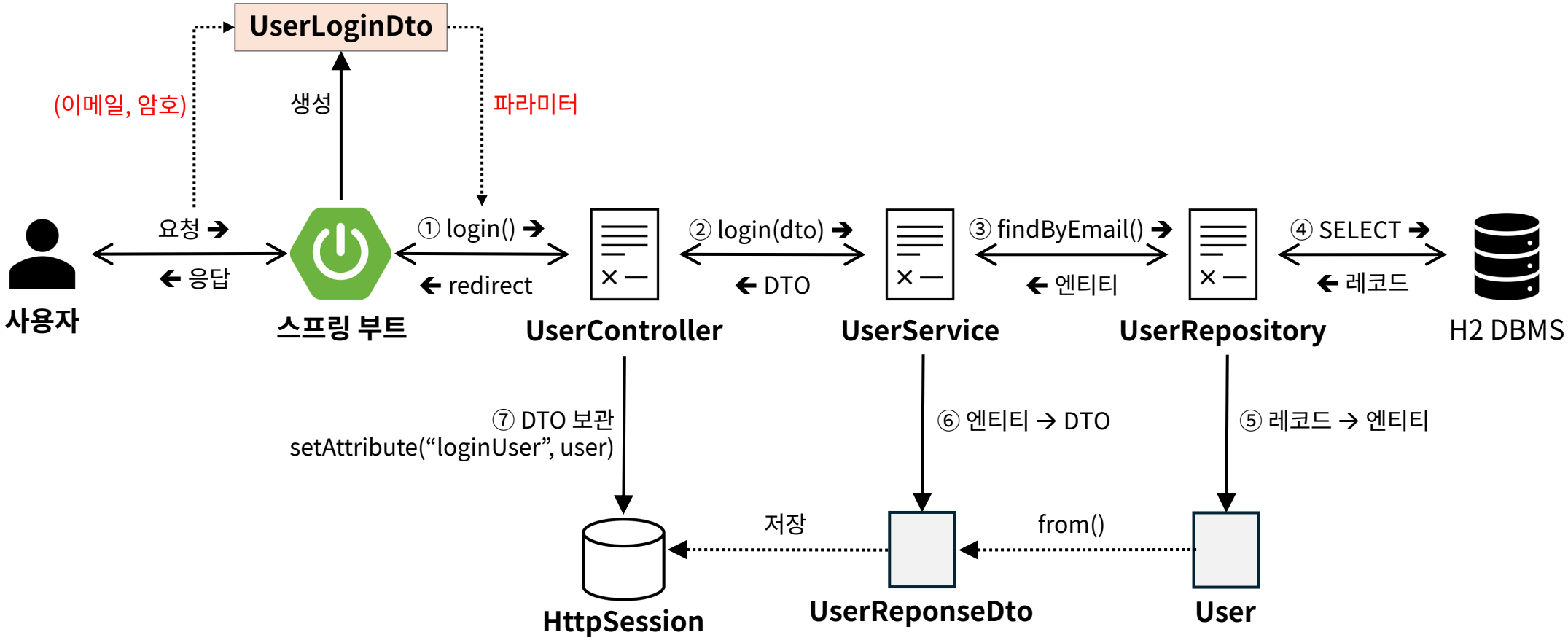
- **인증(Authentication)**과 **인가(Authorization)**의 차이를 설명하고, **HTTP 무상태성**으로 인한 세션 필요성과 **쿠키-세션 관계**를 이해한다.
- 세션 기반 로그인 처리 흐름(로그인 → 세션 생성 → 인증 → 로그아웃)을 단계별로 설명할 수 있다.
- **HttpSession**을 이용해 **로그인 상태를 저장·조회·삭제**하고, 사용자 정보를 세션에 안전하게 저장할 수 있다.
- 비밀번호를 암호화하여 저장하고 검증하는 방법을 구현할 수 있다.
- **Interceptor(또는 Filter)**를 활용하여 로그인 여부에 따른 **접근 제어를 중앙화**할 수 있다.
- 세션 기반 인증의 한계(확장성, 메모리)와 **주요 보안 이슈 (세션 하이재킹, CSRF)**를 설명할 수 있다.
- Spring Security 도입 이전 단계로서 수동 세션 인증을 구현하며, 표준 프레임워크의 필요성을 이해한다.
- 바이브코딩을 통해 로그인/로그아웃 및 인증 흐름을 구현하고, 요청-응답-세션 상태 변화를 디버깅으로 확인한다.

11. 세션 기반 사용자 인증하기 – 로그인 처리 흐름

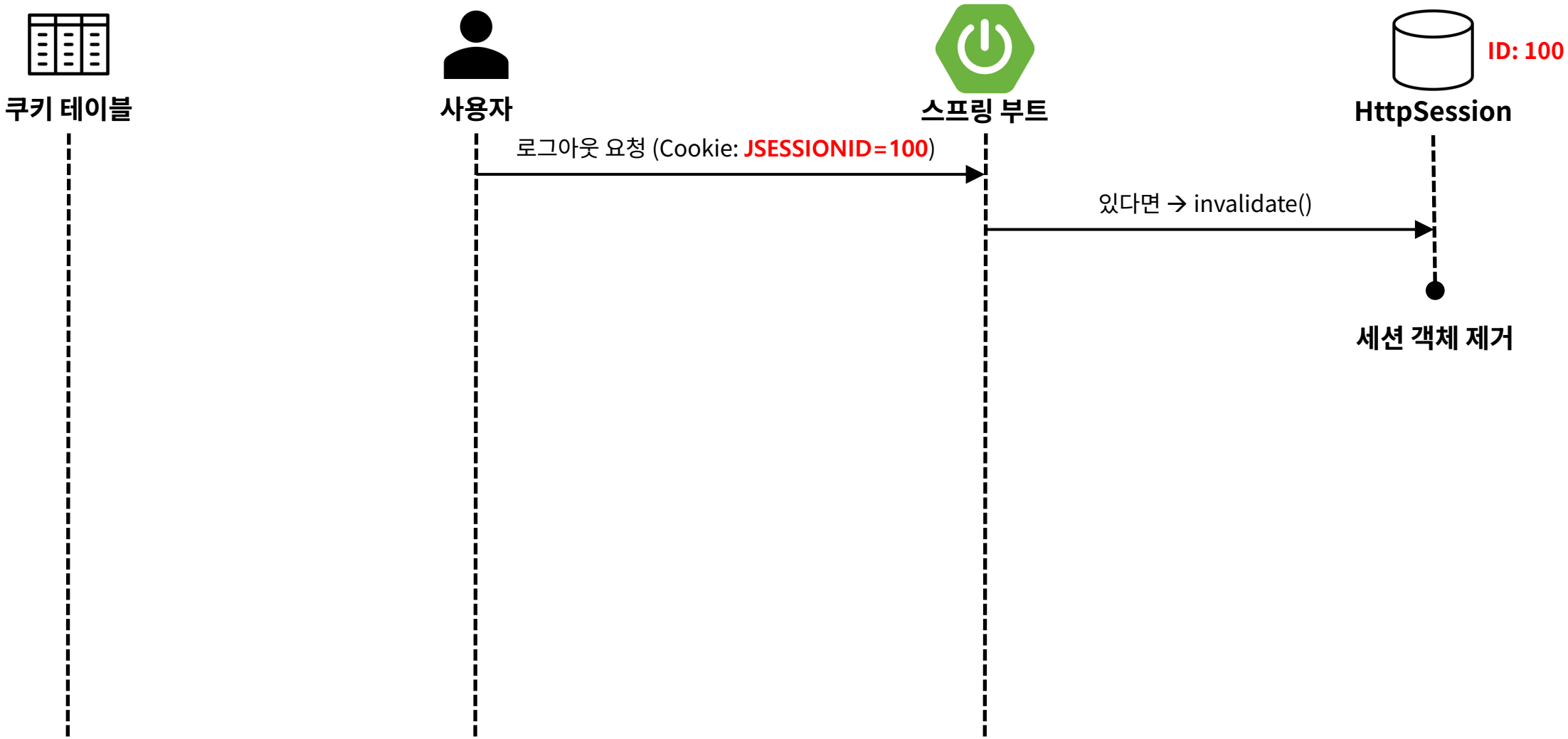


11. 세션 기반 사용자 인증하기 – 아키텍처(로그인 하기)

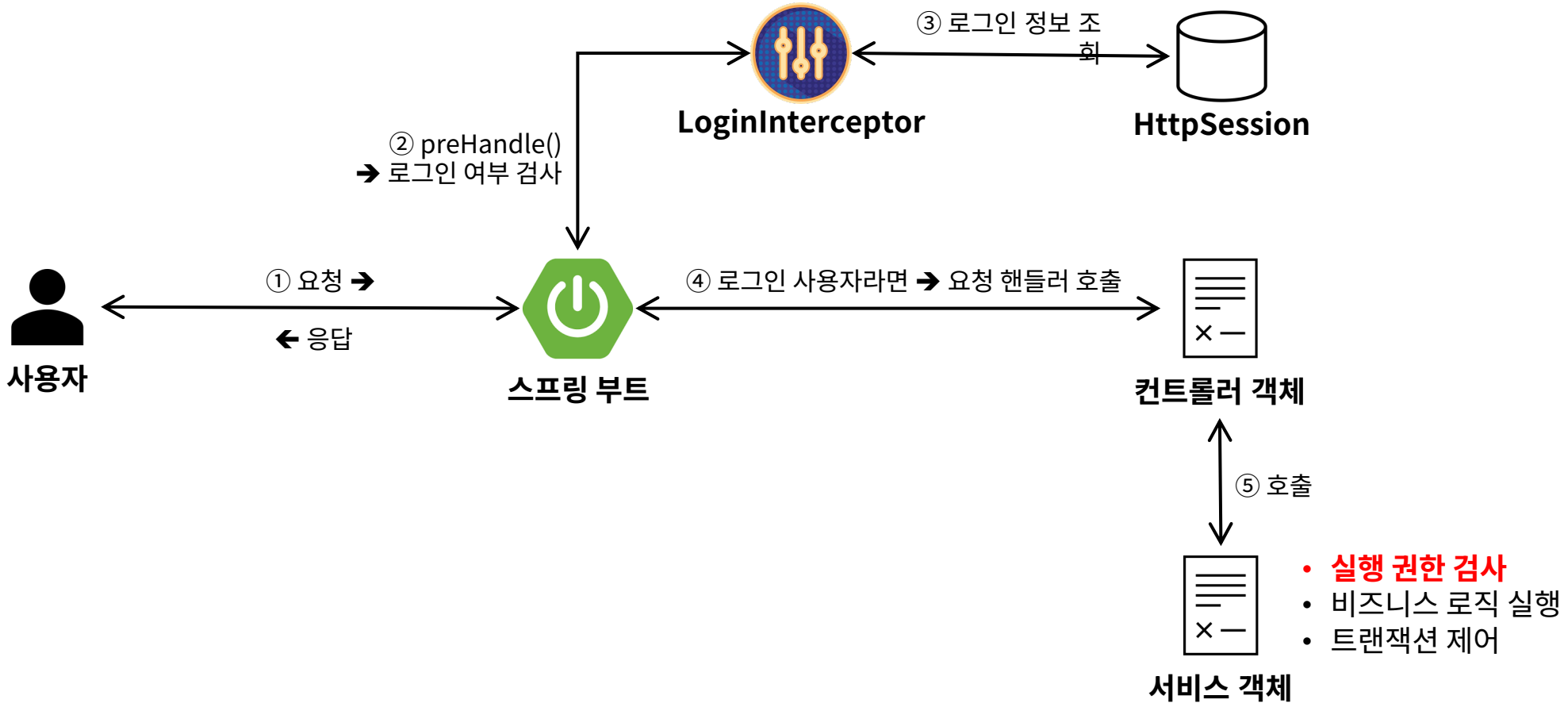
URL: /login



11. 세션 기반 사용자 인증하기 – 로그아웃 처리 흐름



11. 세션 기반 사용자 인증하기 – 인가(Authorization)



11. 세션 기반 사용자 인증하기 – Interceptor 동작 원리

- **preHandle():** ① 컨트롤러 실행 전
- **postHandler():** ② 컨트롤러 실행 후
- **afterCompletion():** ③ 뷰 렌더링 후