바이브코딩을 활용한 스프링부트 프로그래밍

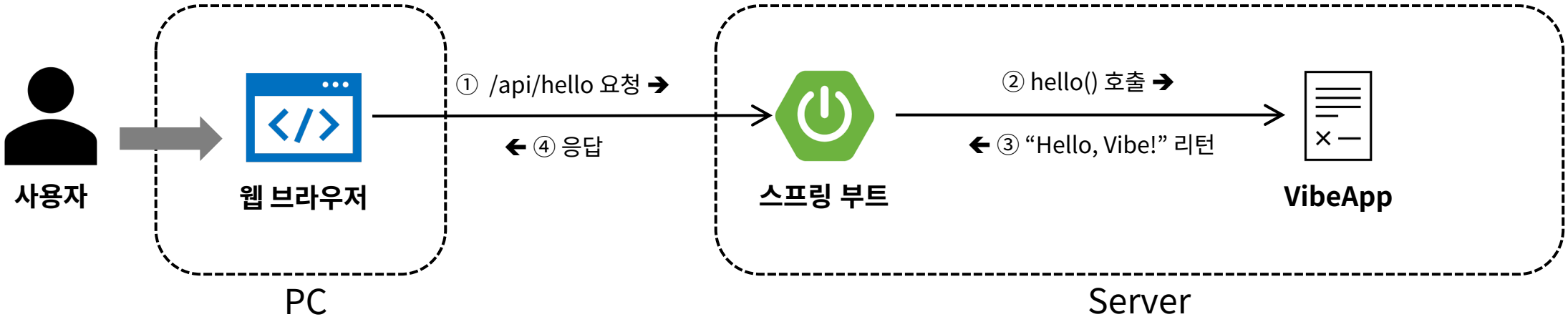
엄진영

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/
```

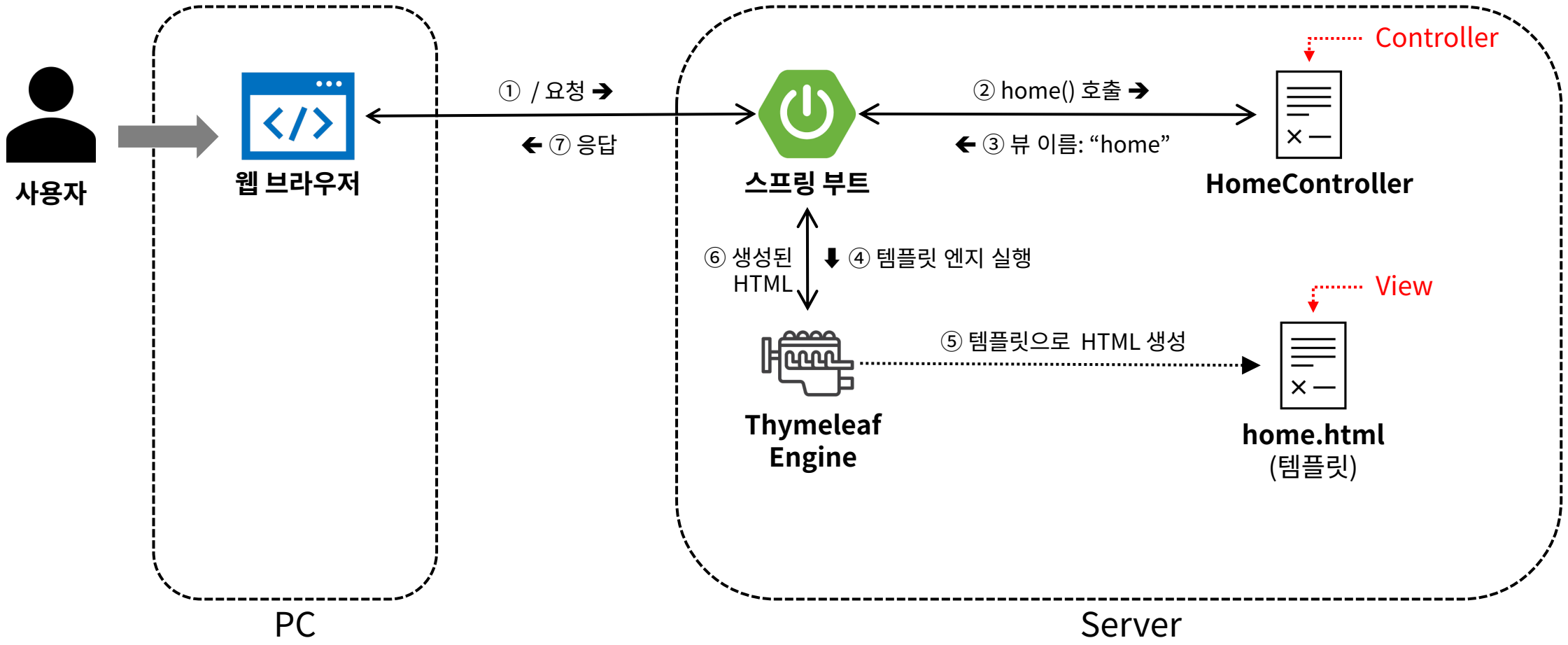
- **용도**
  - Gradle 빌드 결과물 및 임시 파일
- **특징:**
  - 빌드할 때마다 재생성됨
  - `./gradlew clean` 으로 폴더 삭제 가능
  - Git에 커밋하지 않음 (.gitignore 에 포함)

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

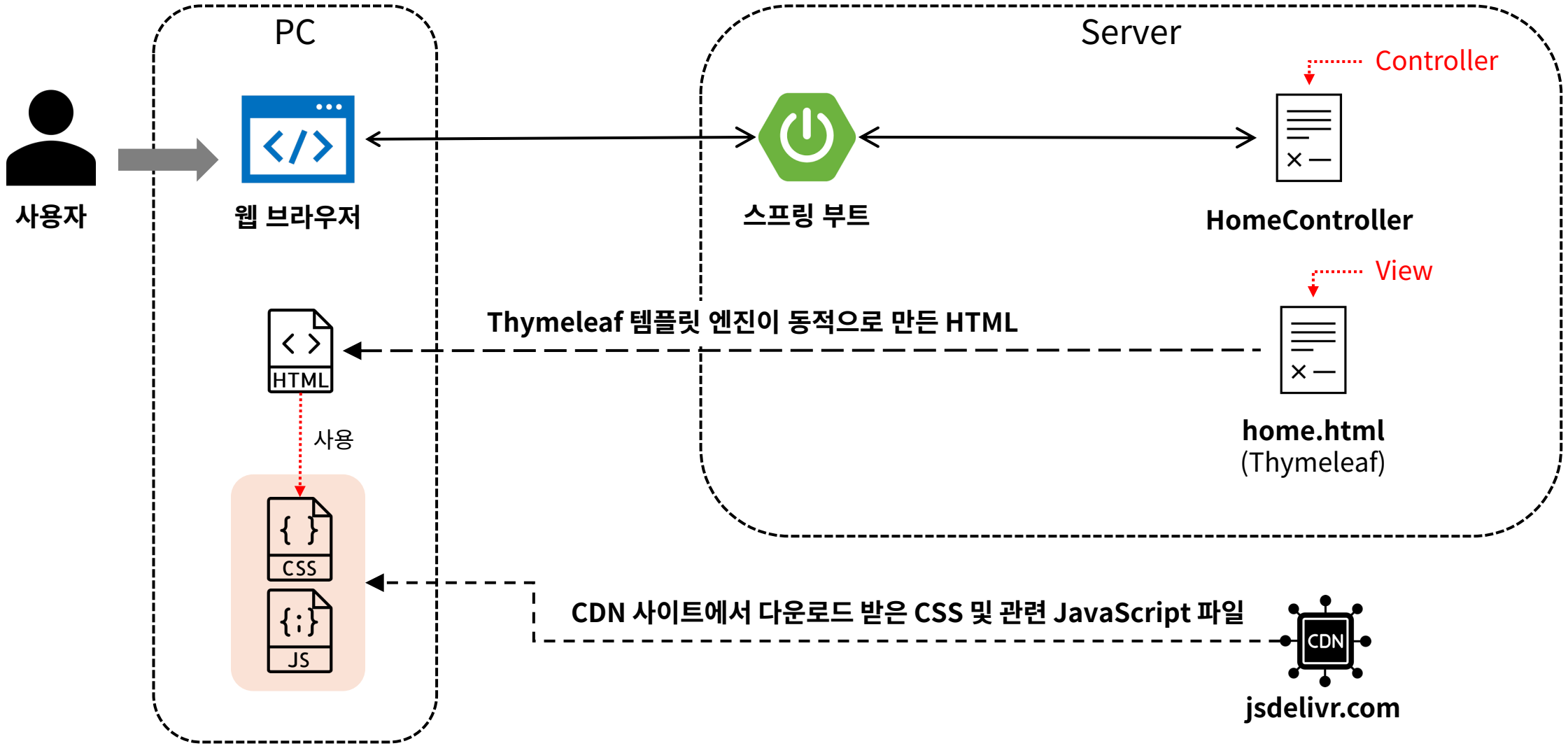


3. CSS 프레임워크 도입하기

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



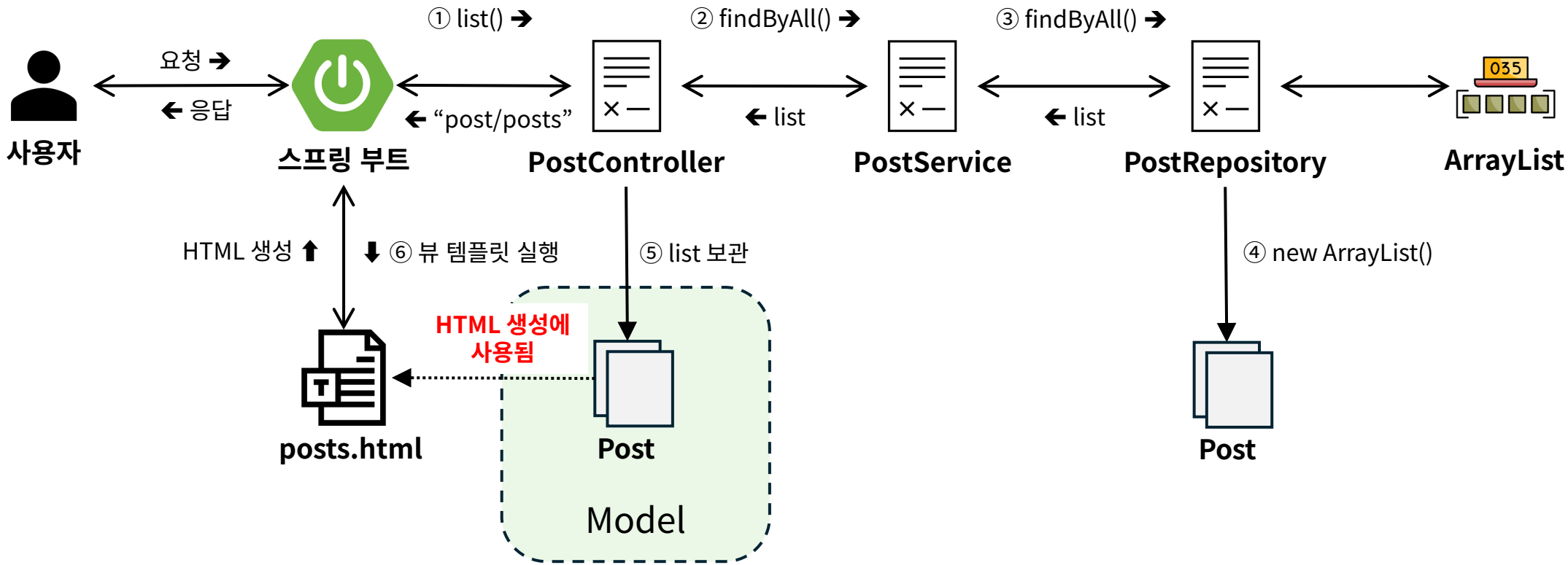
4. 게시글 CRUD 구현하기 (without DBMS)

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.

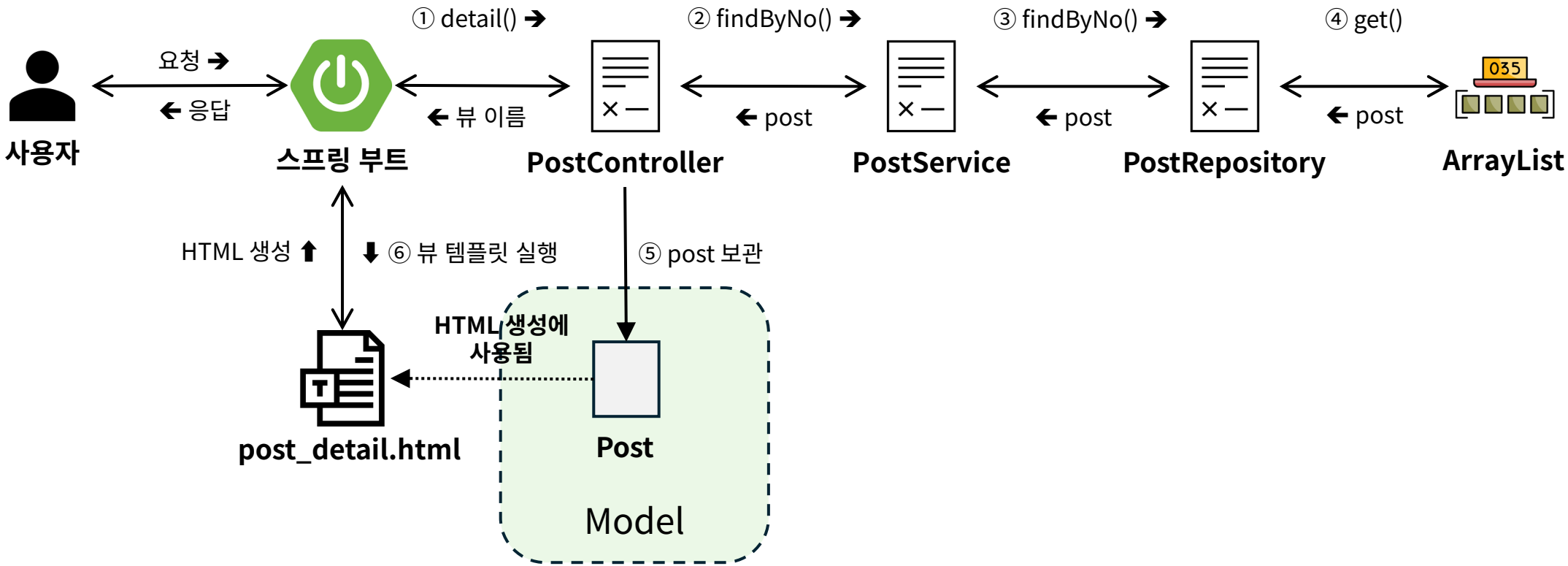
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 목록 조회)

URL: / posts



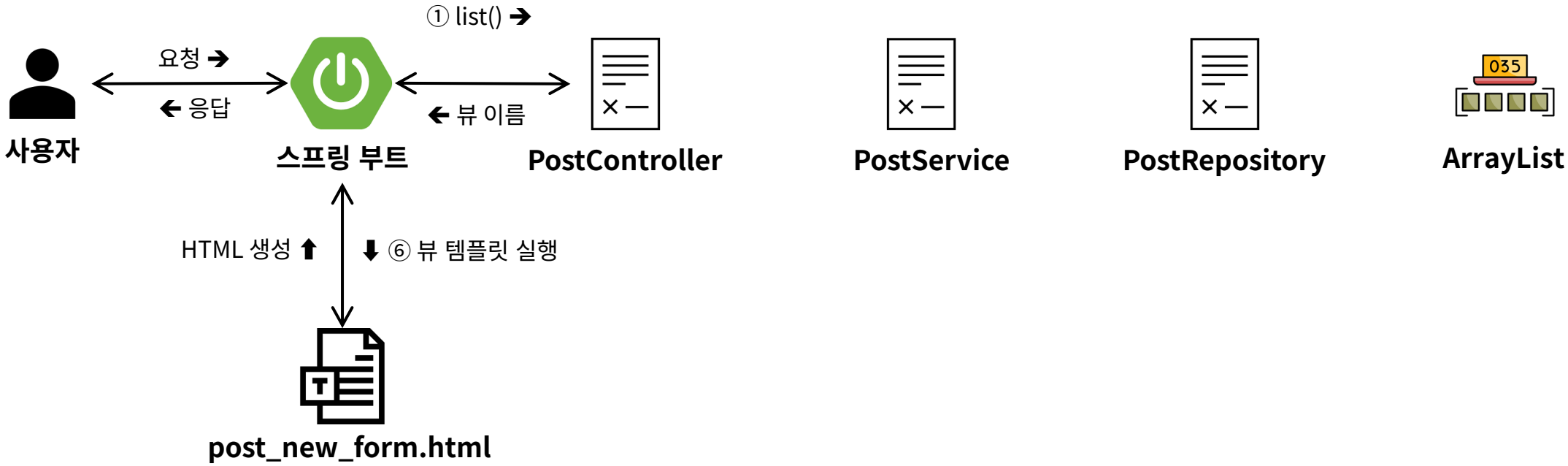
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 상세 조회)

URL: / posts/{no}



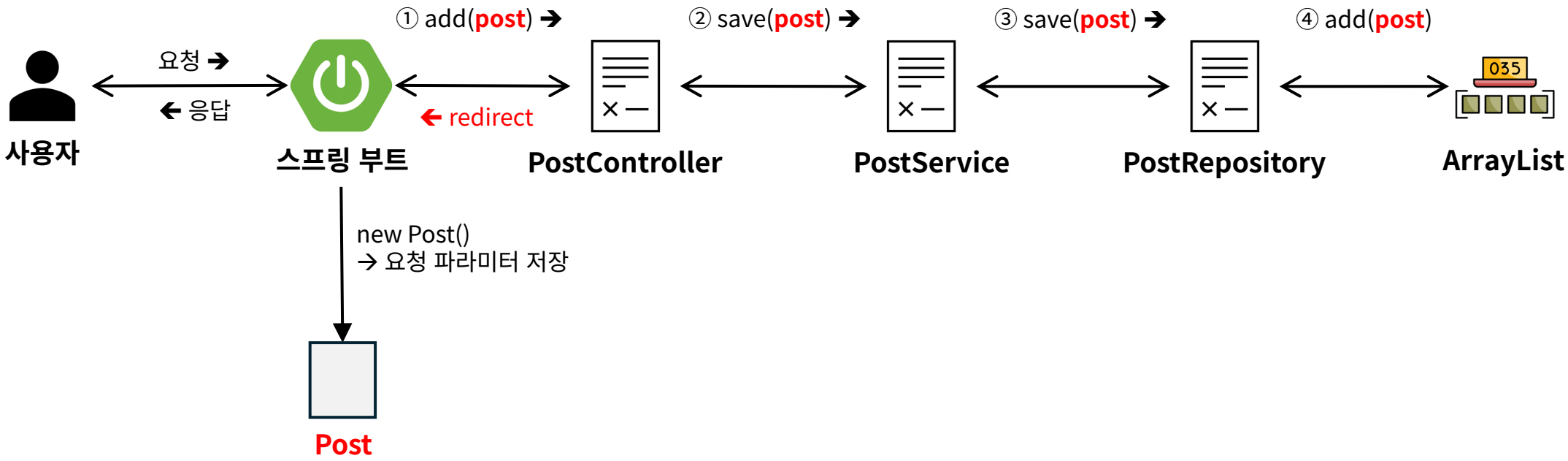
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 작성폼)

URL: / posts/new



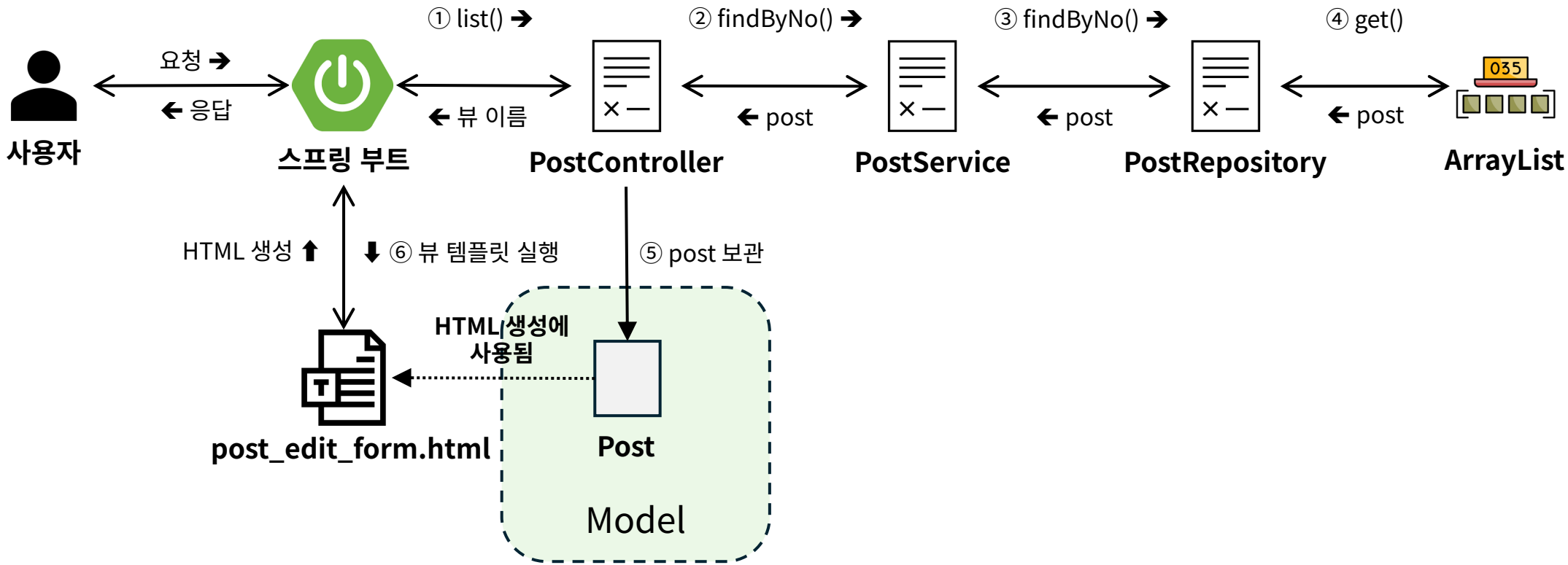
4. 게시글 CRUD 구현하기 – 아키텍처(새 게시글 등록)

URL: / posts/add



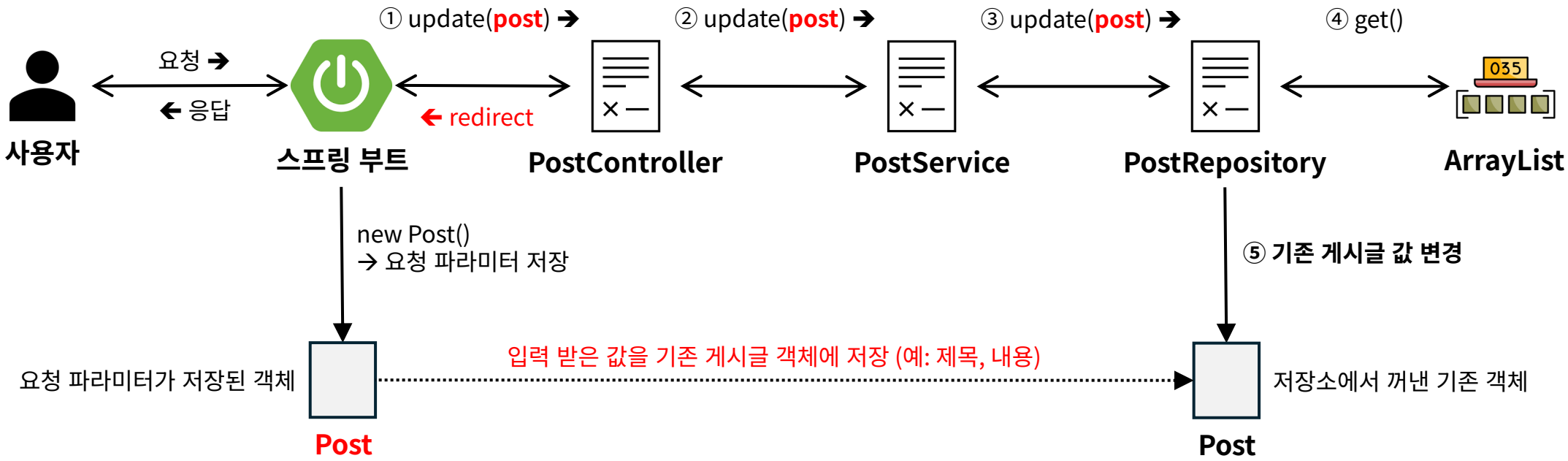
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 수정폼)

URL: / posts/{no}/edit



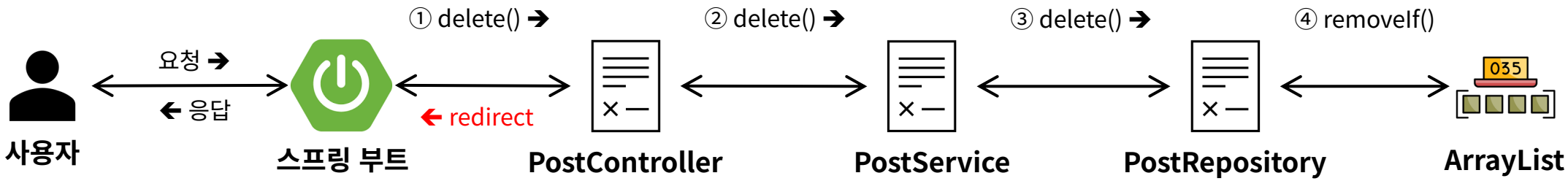
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 수정)

URL: / posts/{no}/save



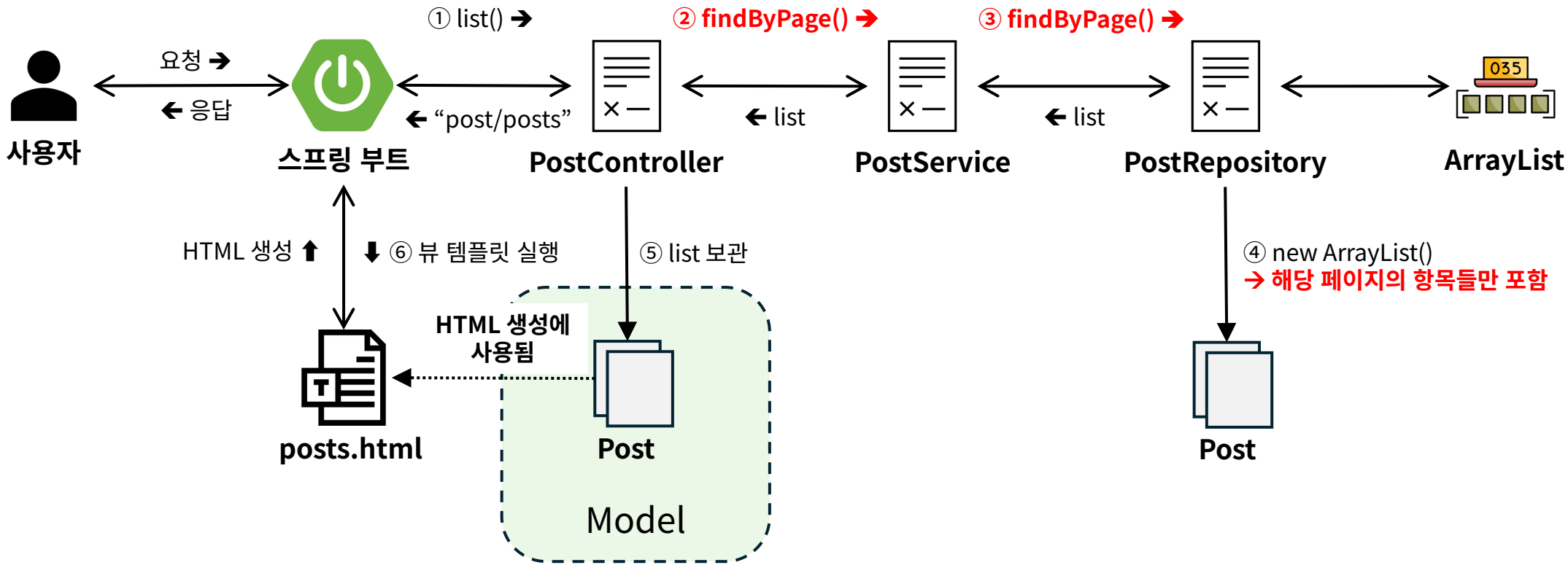
4. 게시글 CRUD 구현하기 – 아키텍처(게시글 삭제)

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 작업 시 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

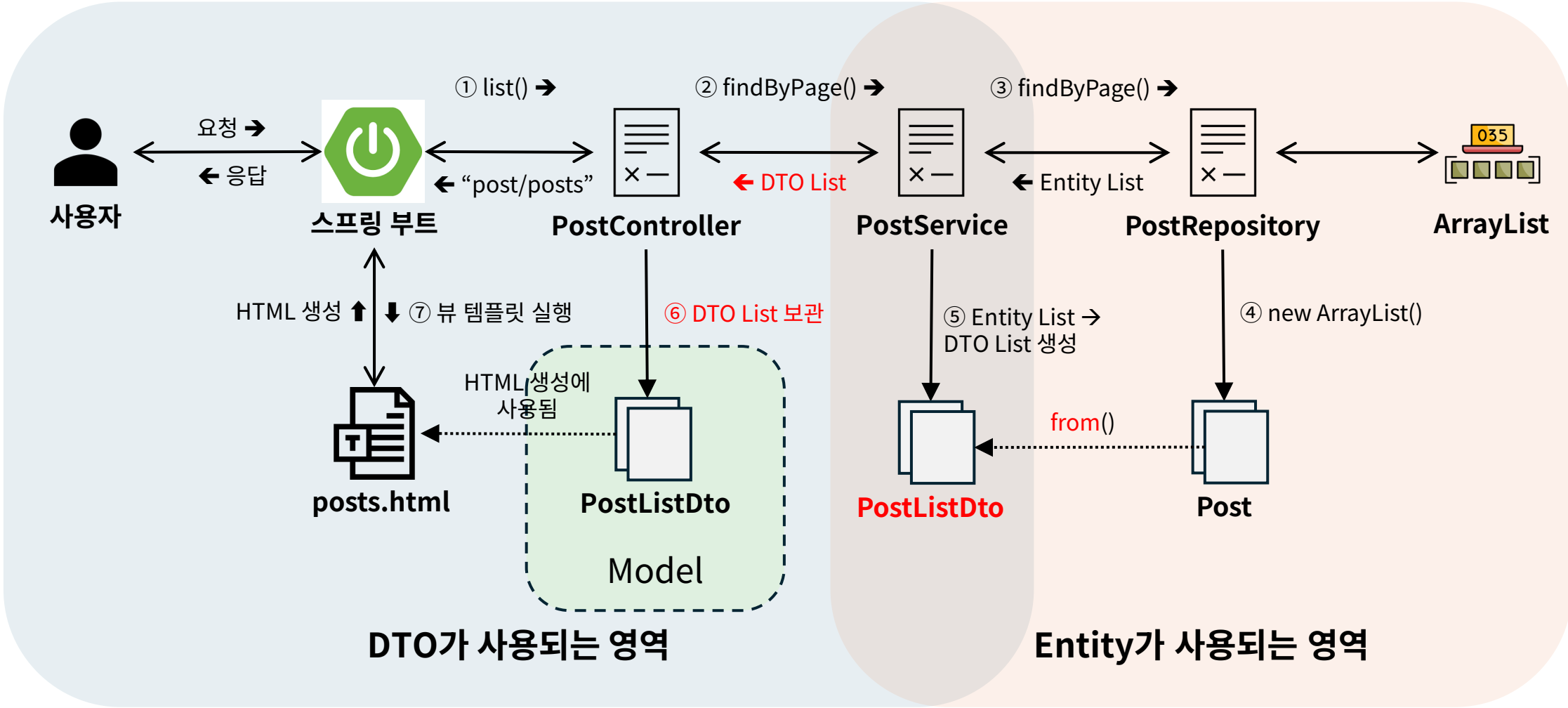
```
public class PostCreateDto {
    private String title;
    private String content;
}

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
}
```

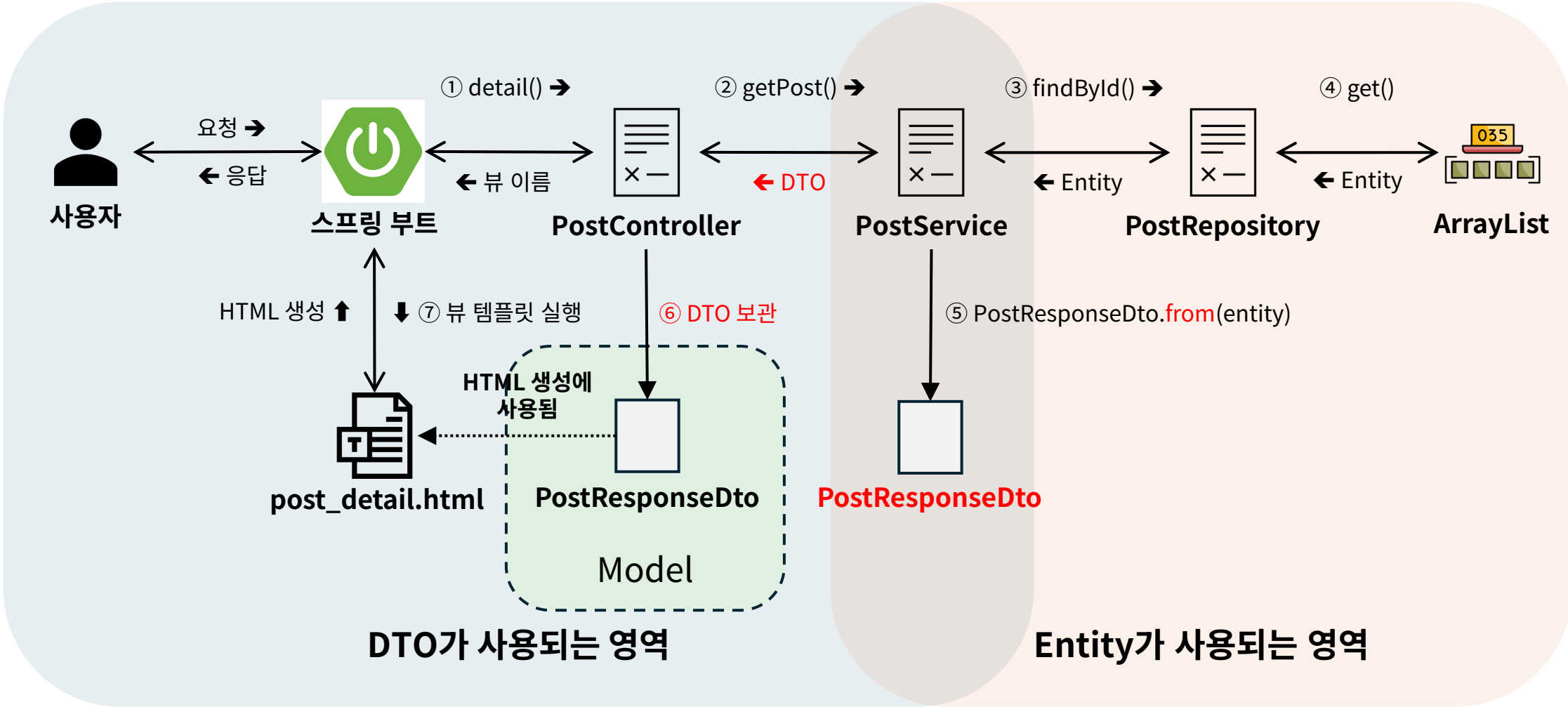
5. DTO 패턴 적용하기 – 아키텍처(게시글 목록 조회)

URL: / posts



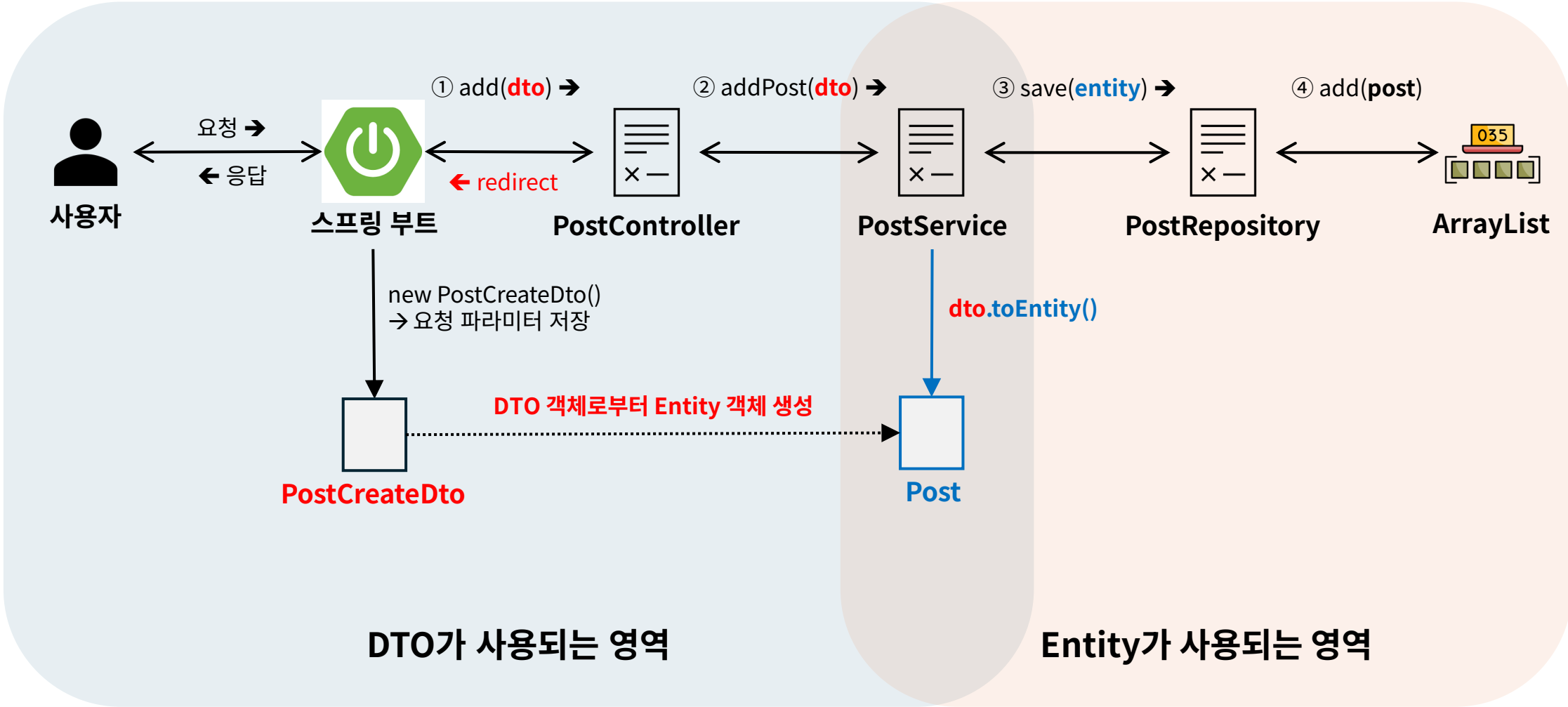
5. DTO 패턴 적용하기 – 아키텍처(게시글 상세 조회)

URL: / posts/{no}



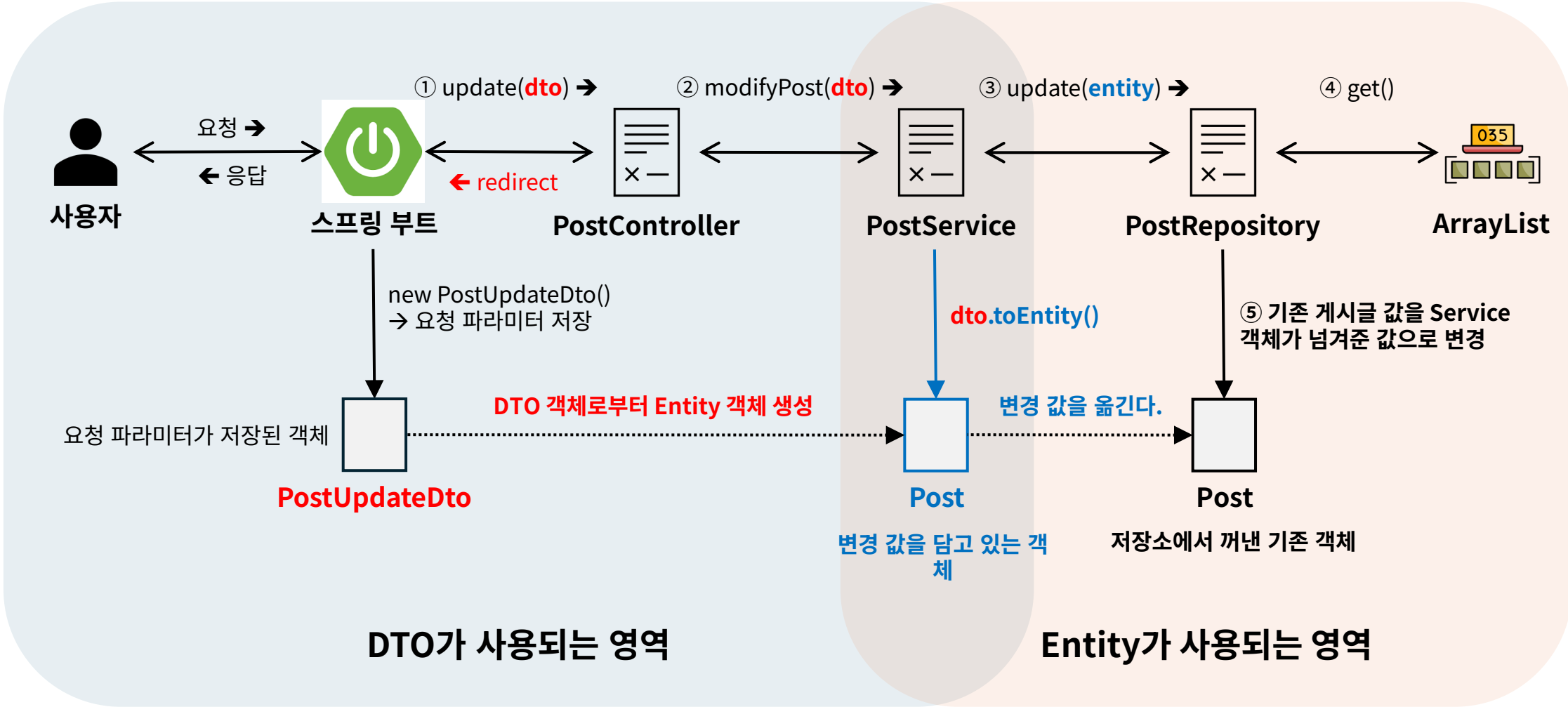
5. DTO 패턴 적용하기 – 아키텍처(새 게시글 등록)

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

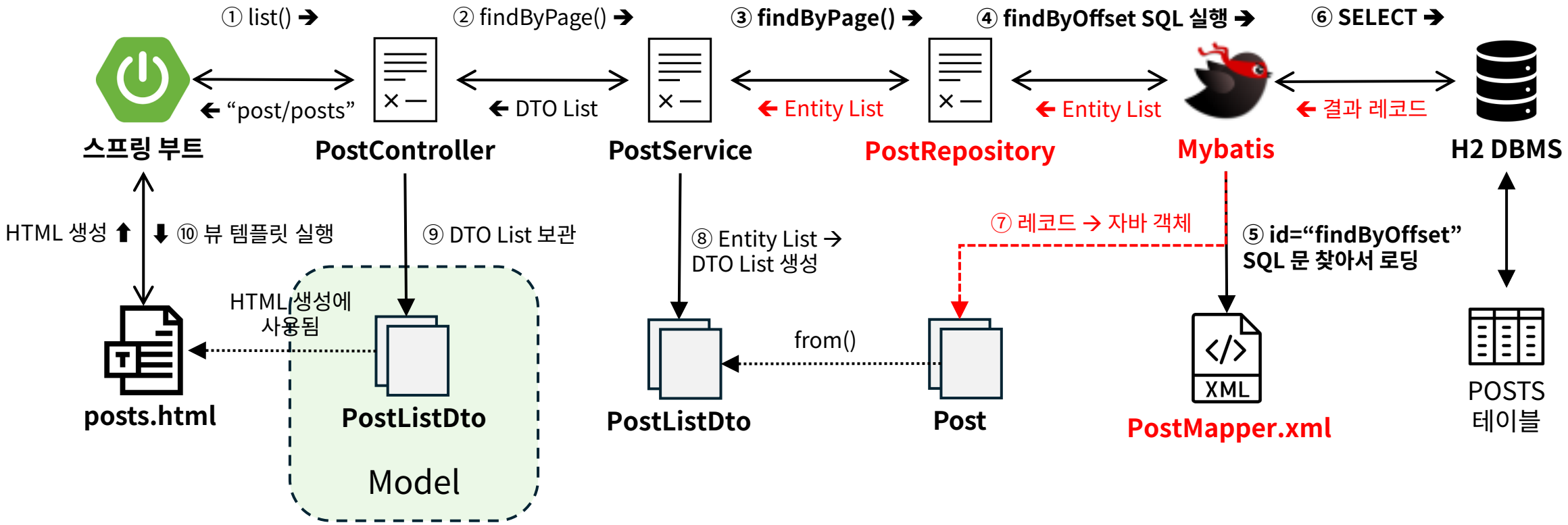
7. DBMS + Mybatis SQL Mapper 도입하기

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.

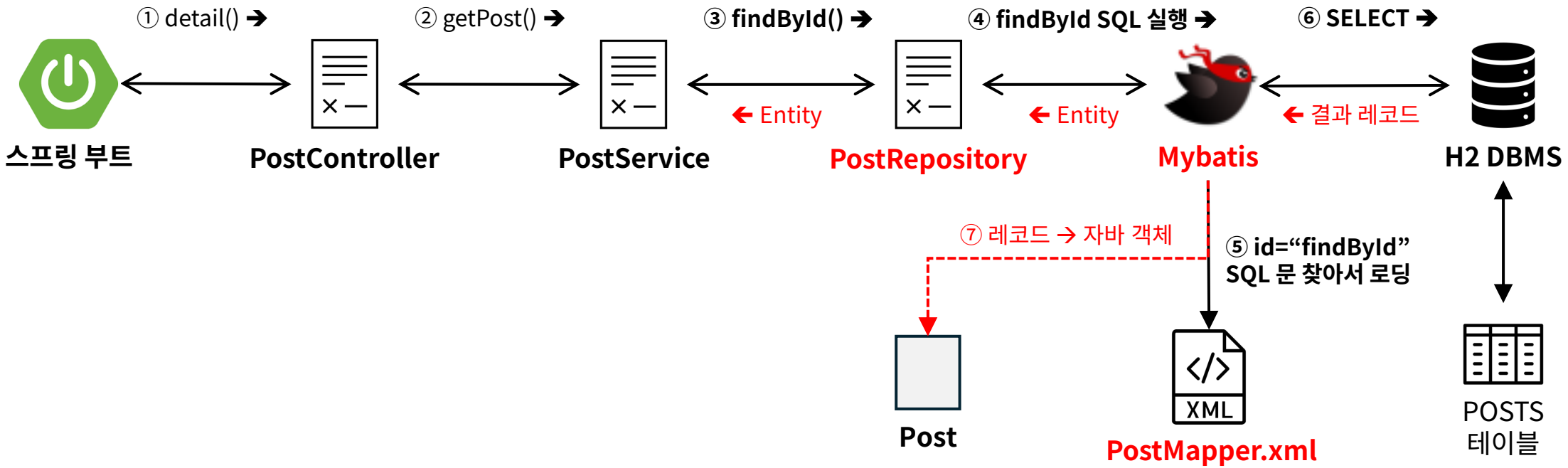
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 목록 조회)

URL: / posts



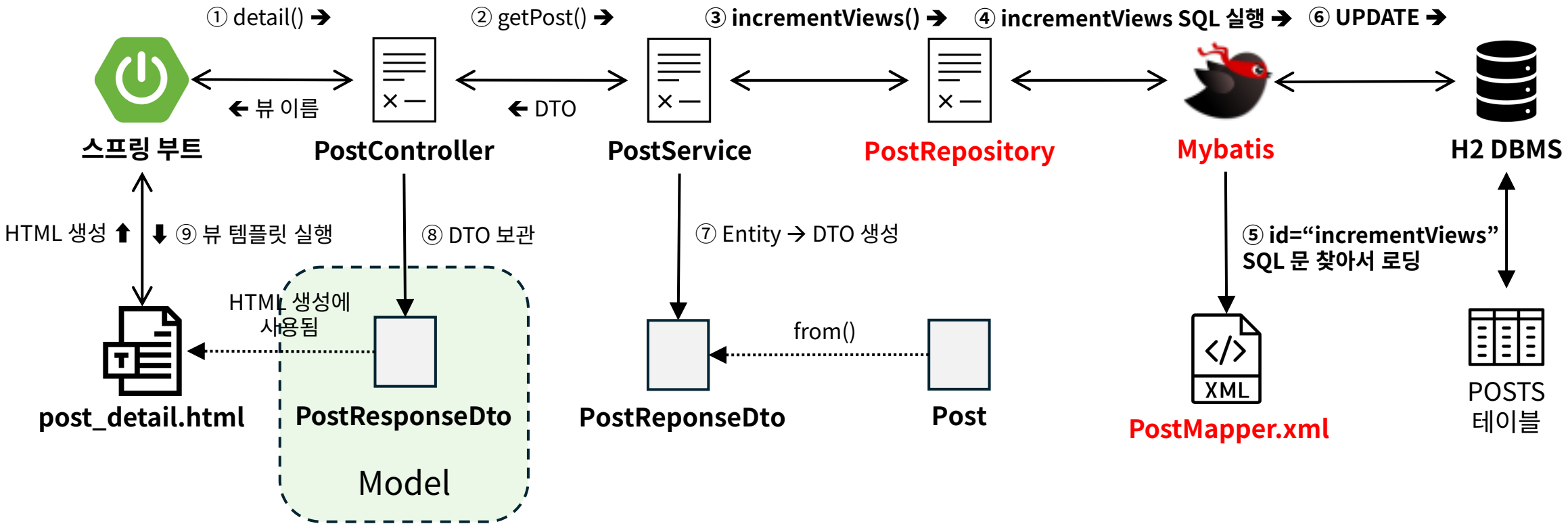
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 상세 조회)

URL: / posts/{no}



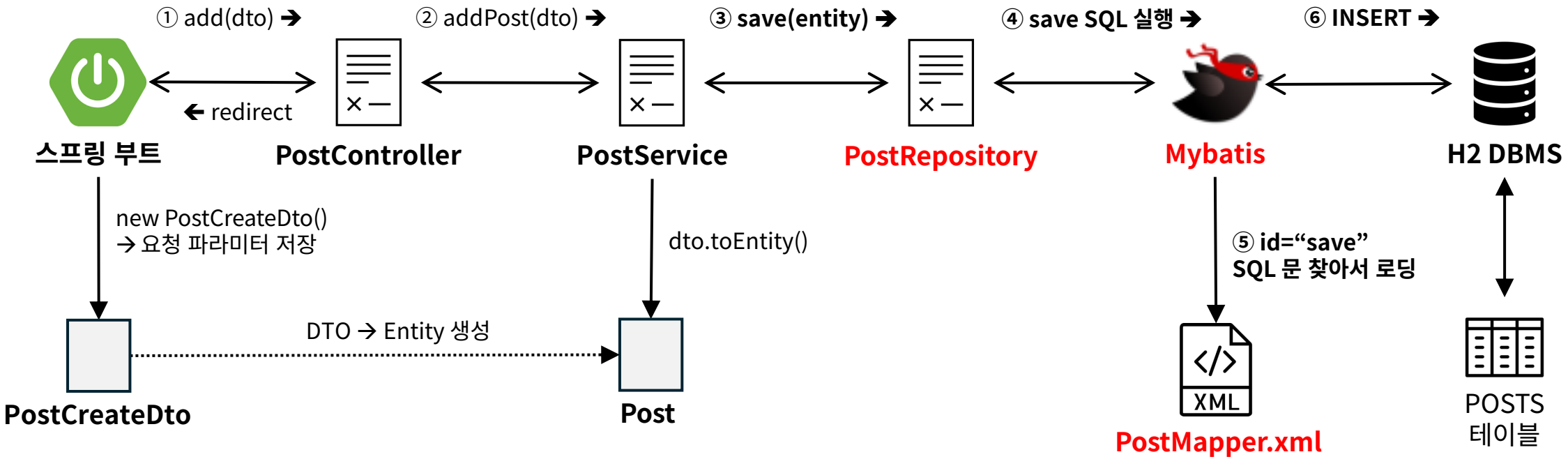
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 조회수 증가)

URL: / posts/{no}



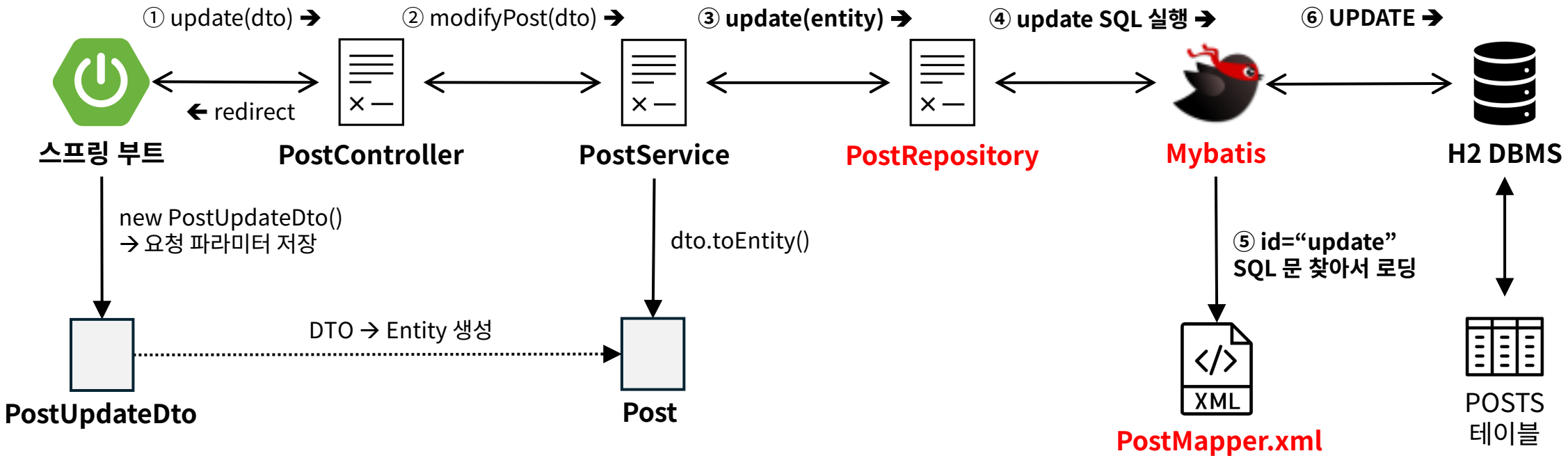
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(새 게시글 등록)

URL: / posts/add



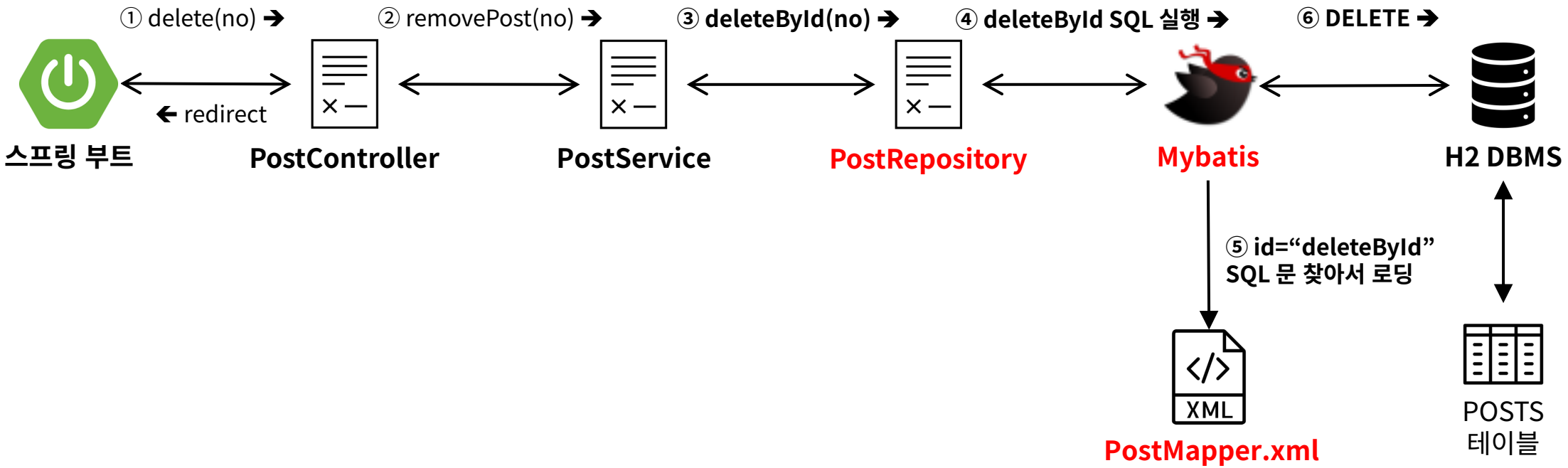
7. DBMS + Mybatis SQL Mapper 도입하기 – 아키텍처(게시글 수정)

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));
}
```

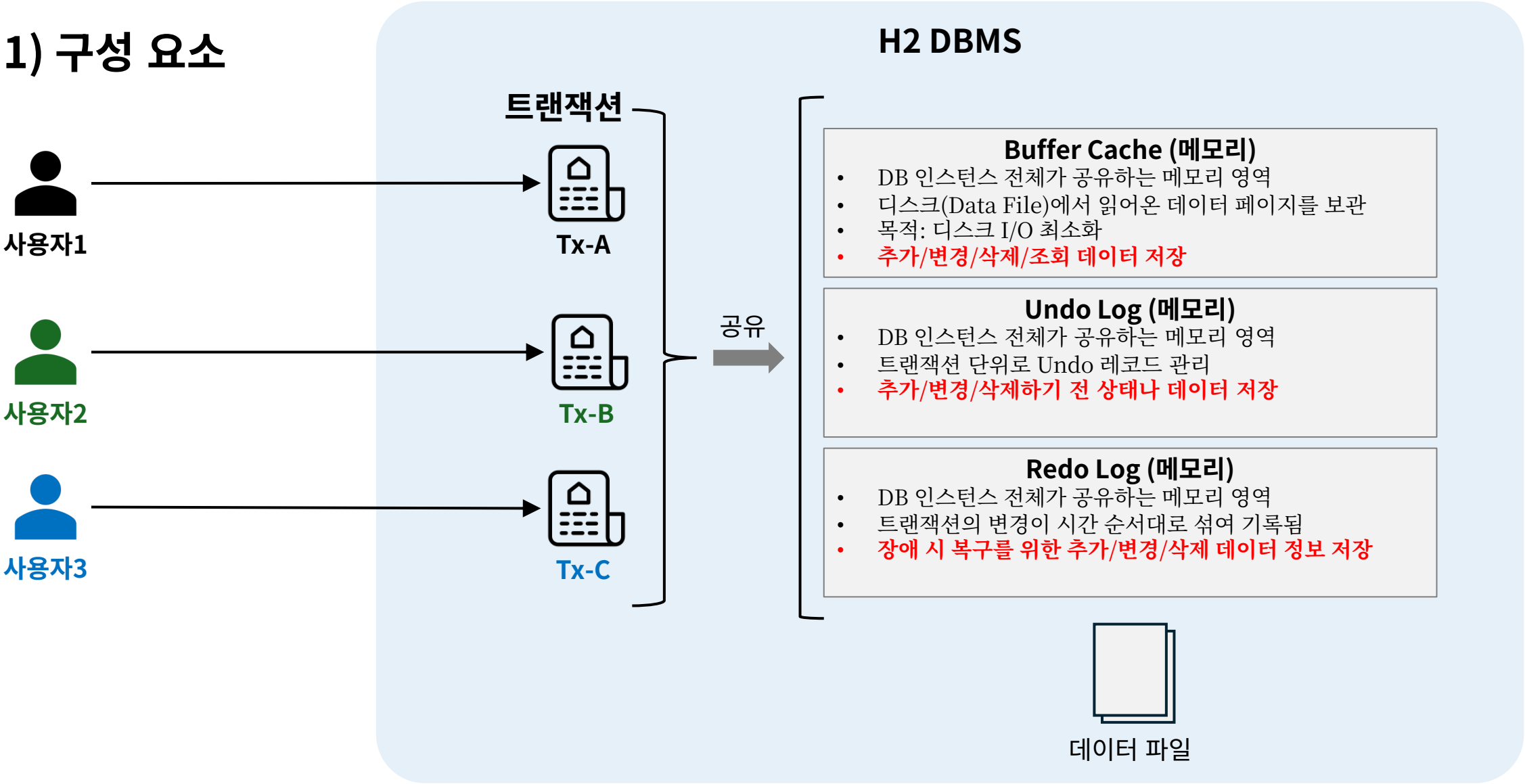
8. 트랜잭션 적용하기 – ACID 특성

Durability(지속성) 예시 코드:

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

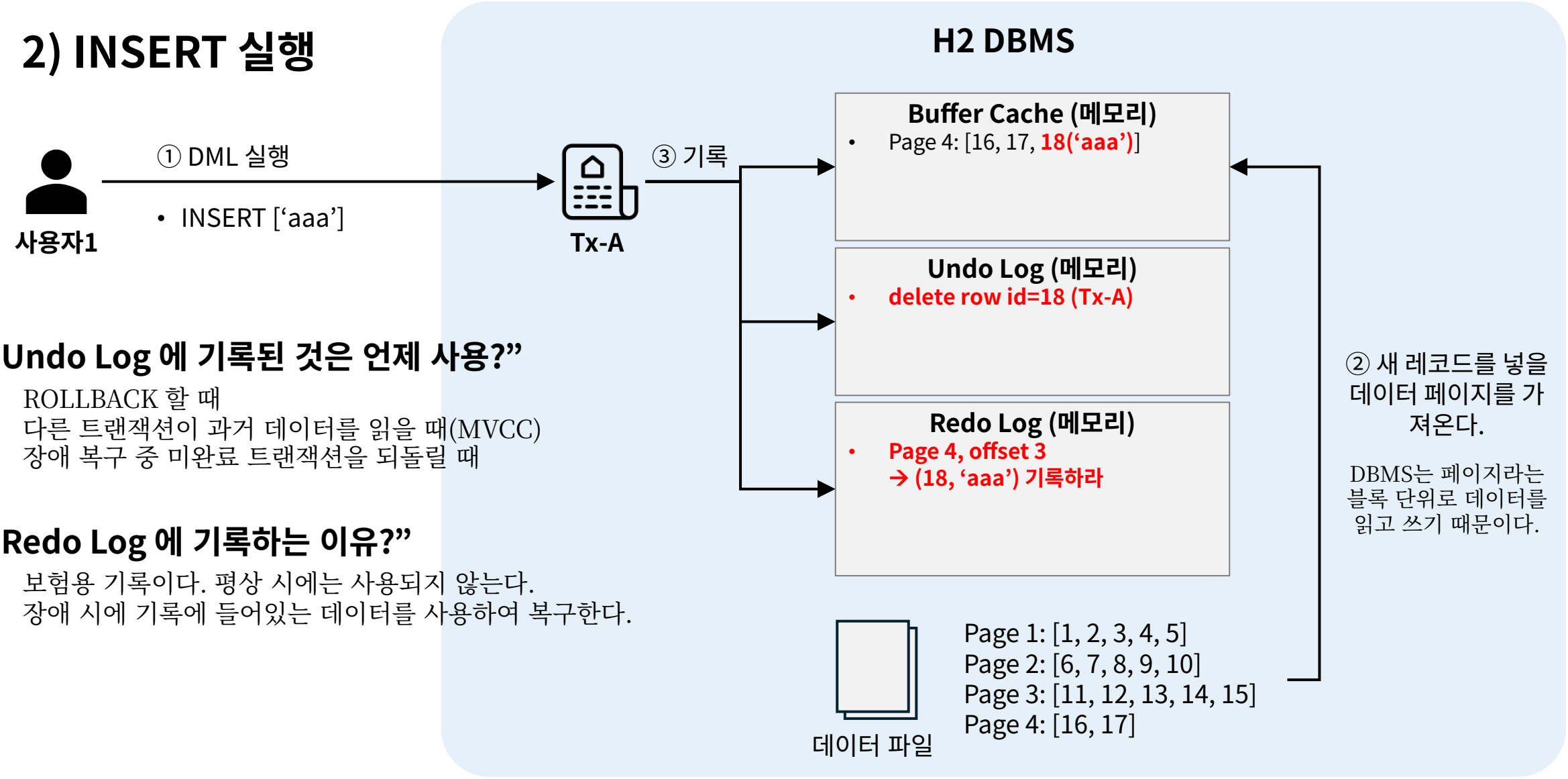
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

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

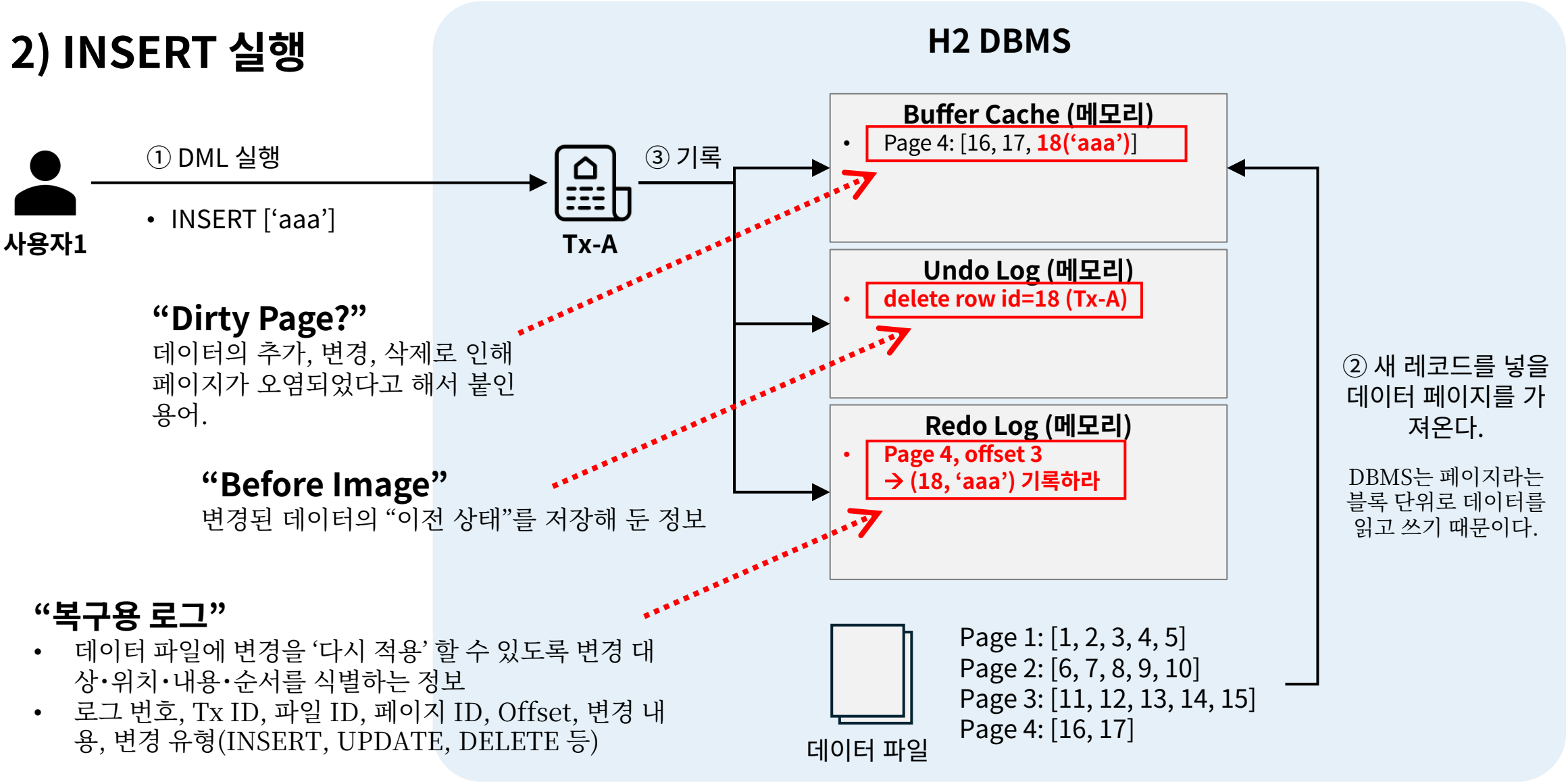
“Redo Log 에 기록하는 이유?”

- 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



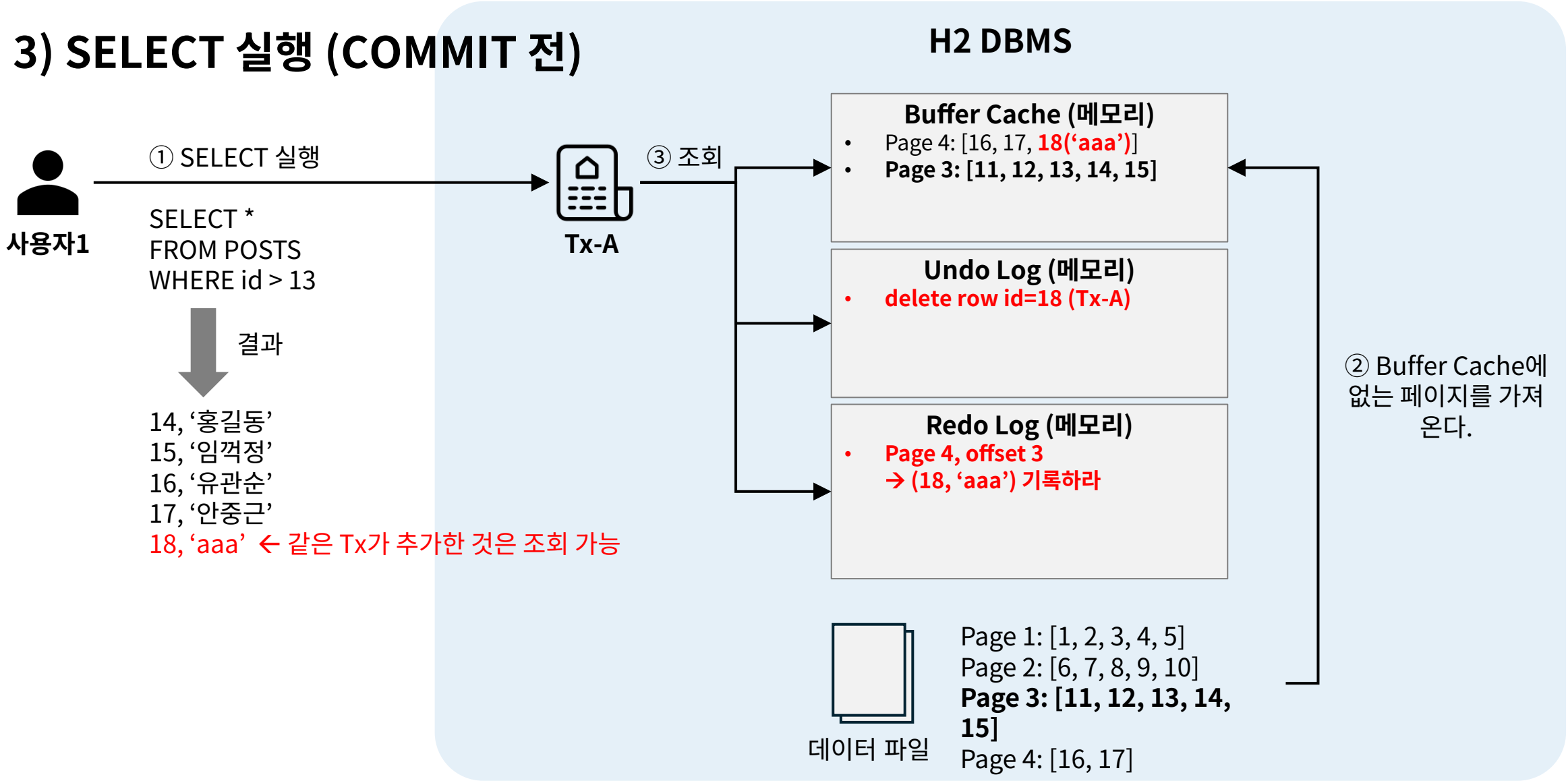
“Dirty Page?”
데이터의 추가, 변경, 삭제로 인해 페이지가 오염되었다고 해서 붙인 용어.

“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)

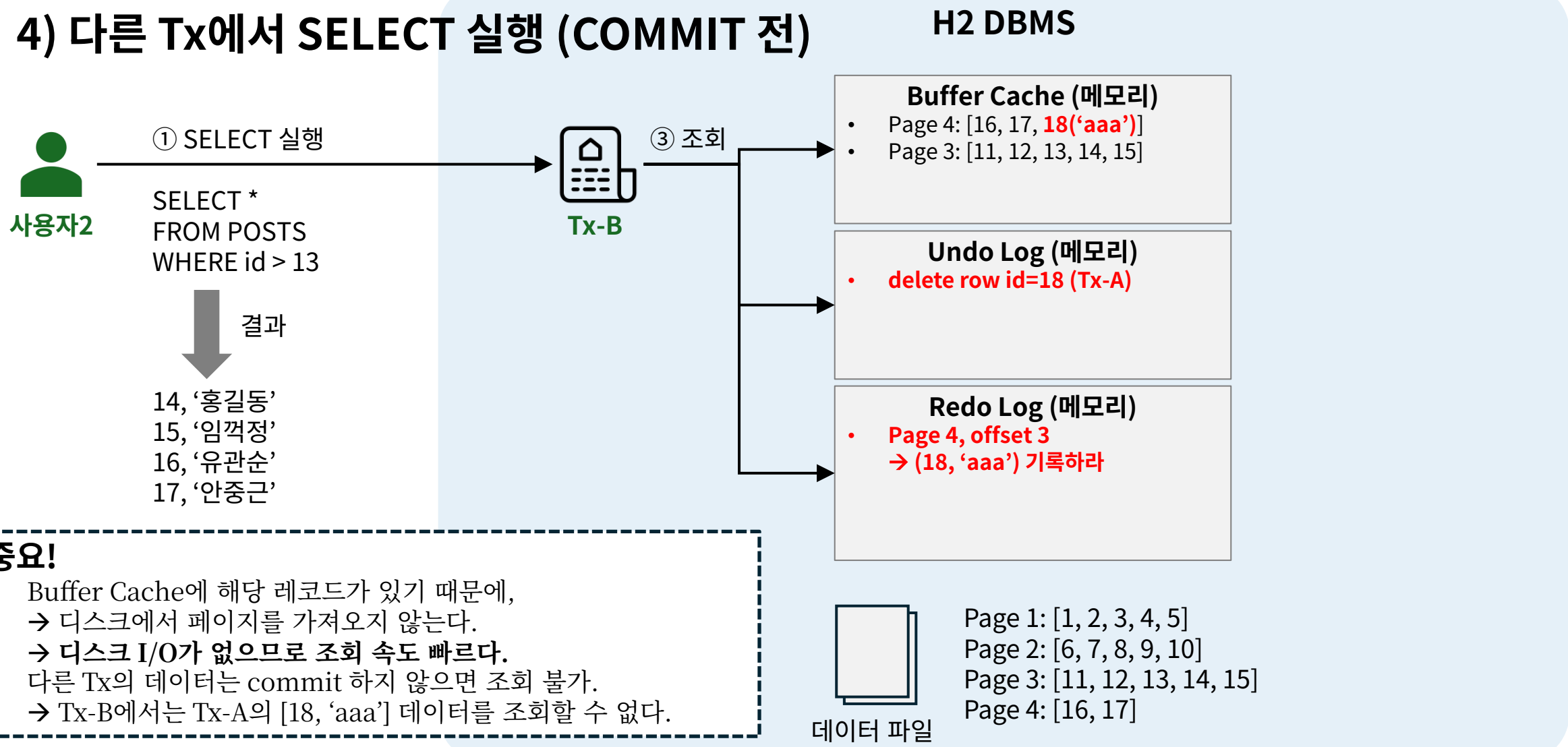
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

3) SELECT 실행 (COMMIT 전)



8. 트랜잭션 적용하기 – 트랜잭션 구동원리

4) 다른 Tx에서 SELECT 실행 (COMMIT 전)

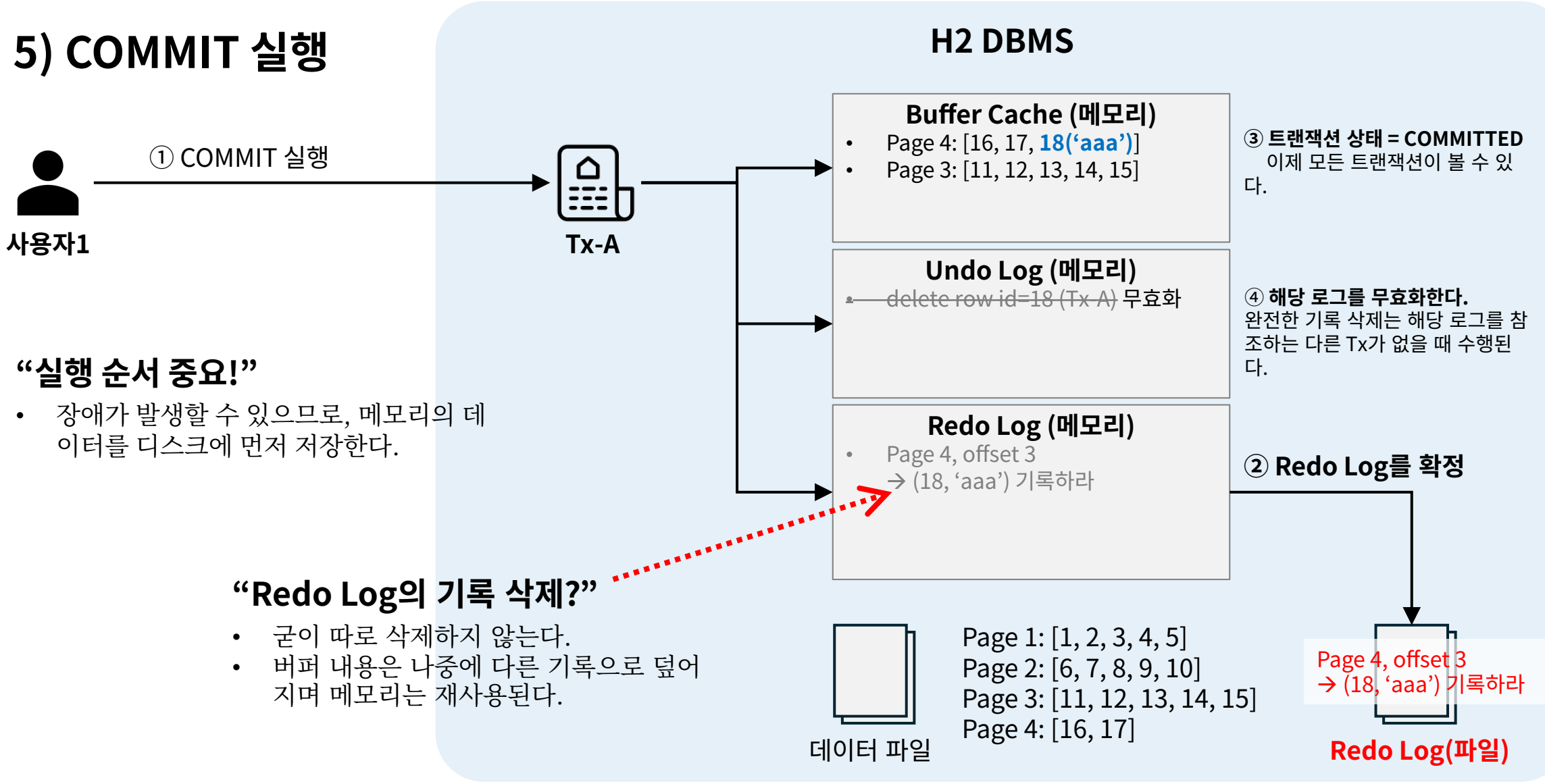


중요!

- 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.

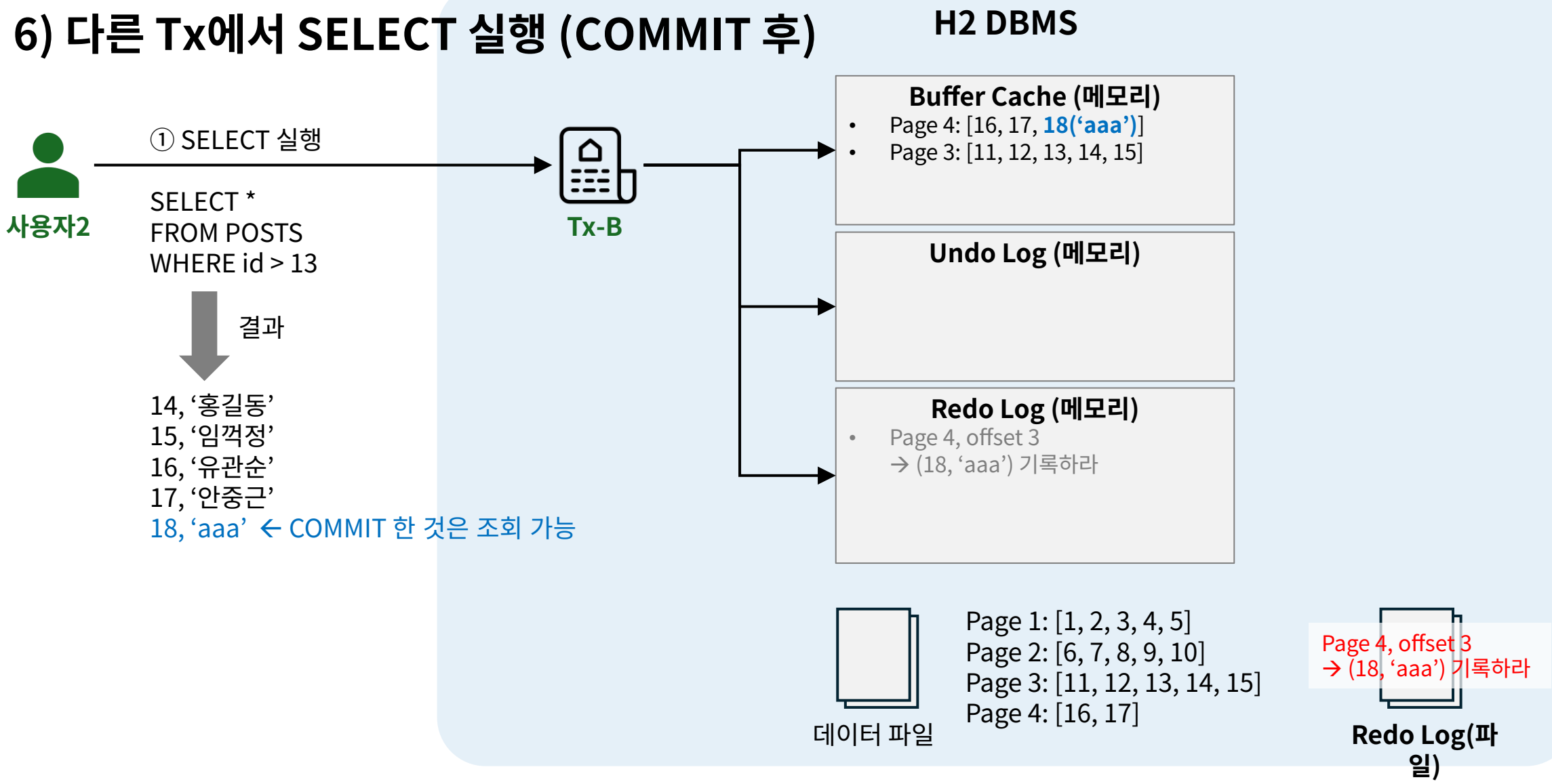
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

5) COMMIT 실행



8. 트랜잭션 적용하기 – 트랜잭션 구동원리

6) 다른 Tx에서 SELECT 실행 (COMMIT 후)

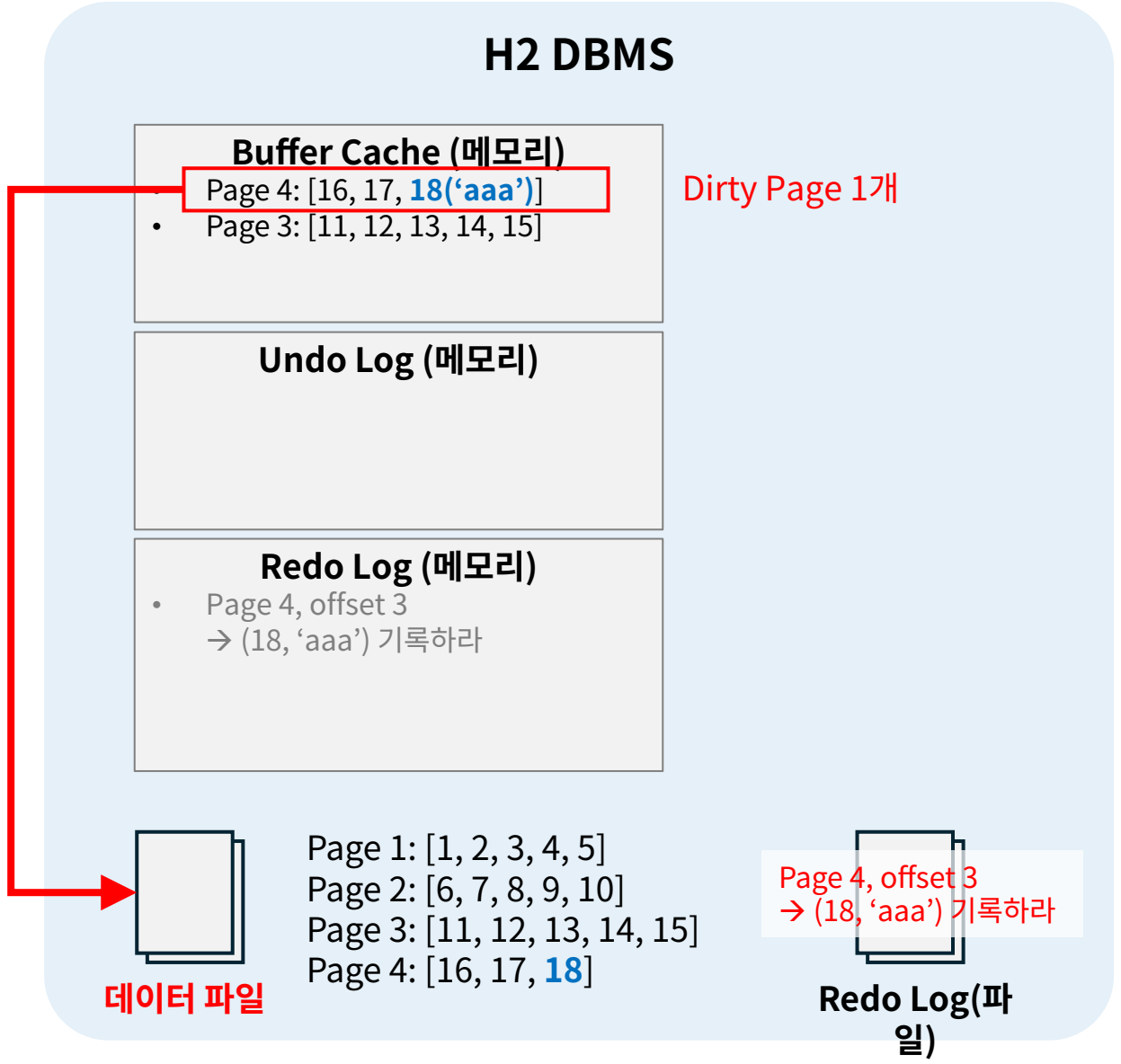


8. 트랜잭션 적용하기 – 트랜잭션 구동원리

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



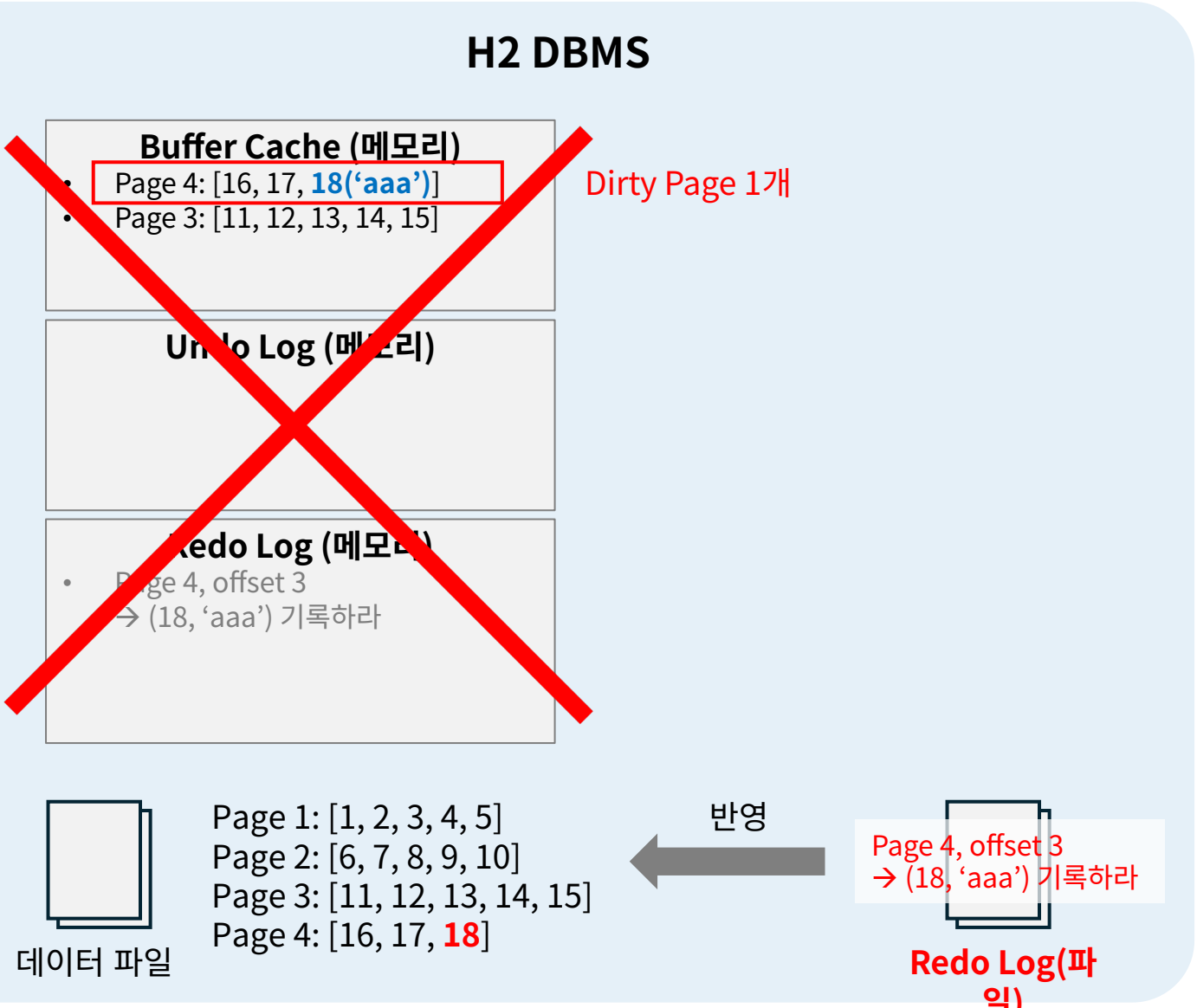
8. 트랜잭션 적용하기 – 트랜잭션 구동원리

8) 장애 발생!

“시나리오: COMMIT 직후 서버 다운”

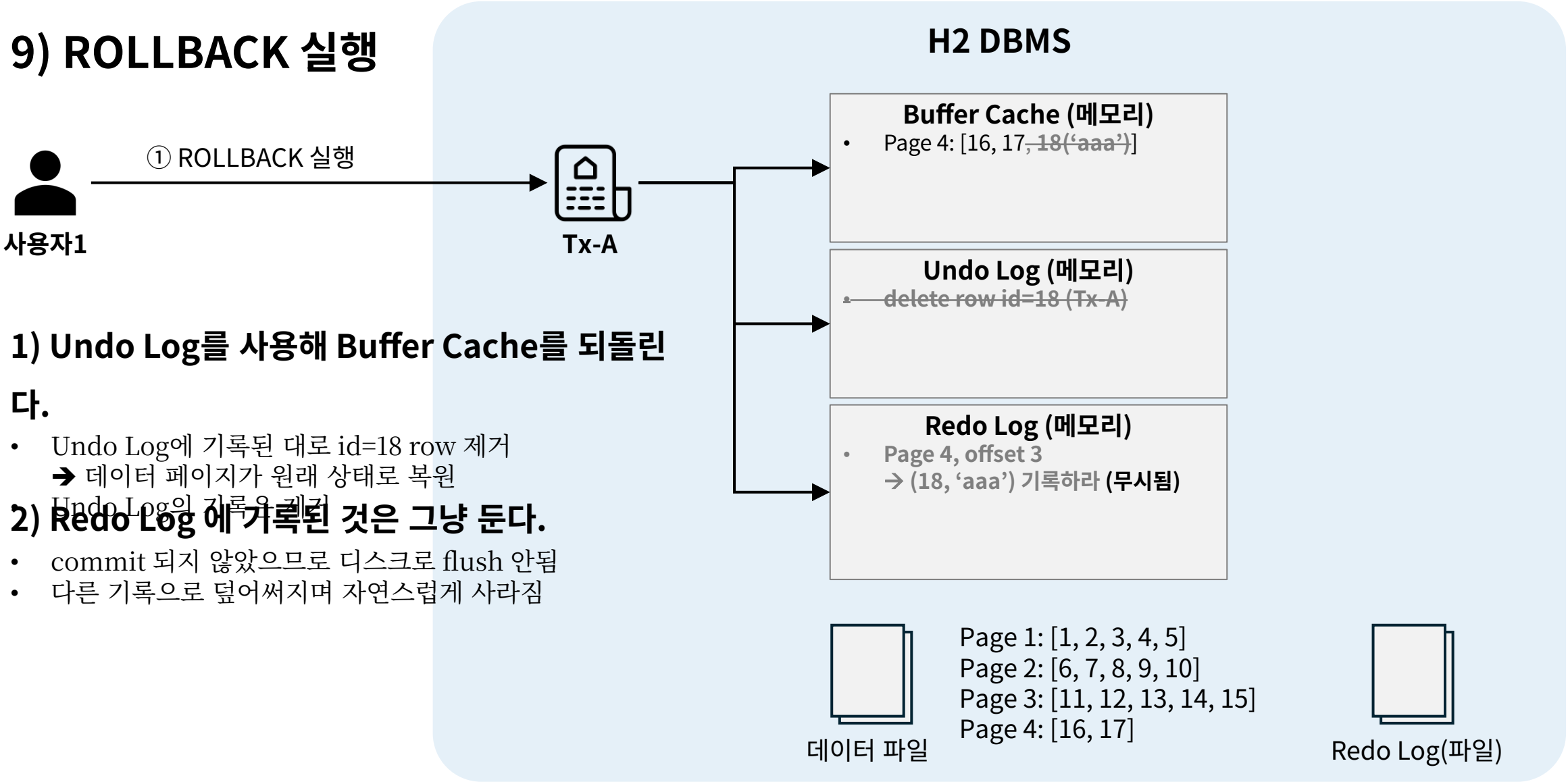
① INSERT 1000개
② COMMIT (Redo Log만 디스크에 저장)
③ 사용자에게 “성공” 응답
④ 서버 다운!
- Buffer Cache 손실
- Dirty Page 손실
- Data File에는 기록 안됨

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.

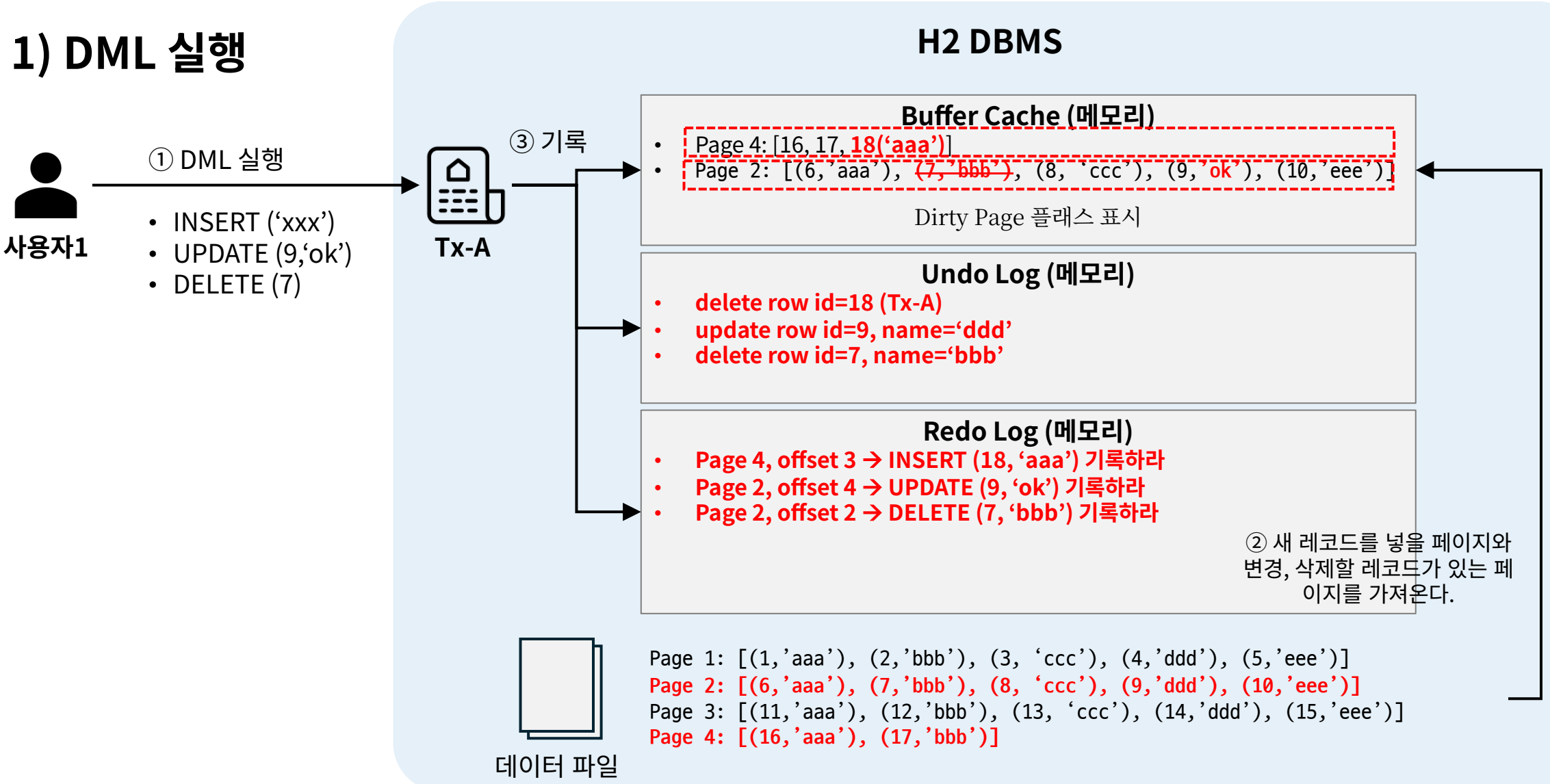
- Undo Log에 기록된 대로 id=18 row 제거
  ➔ 데이터 페이지가 원래 상태로 복원
- Undo Log의 기록은 제거

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

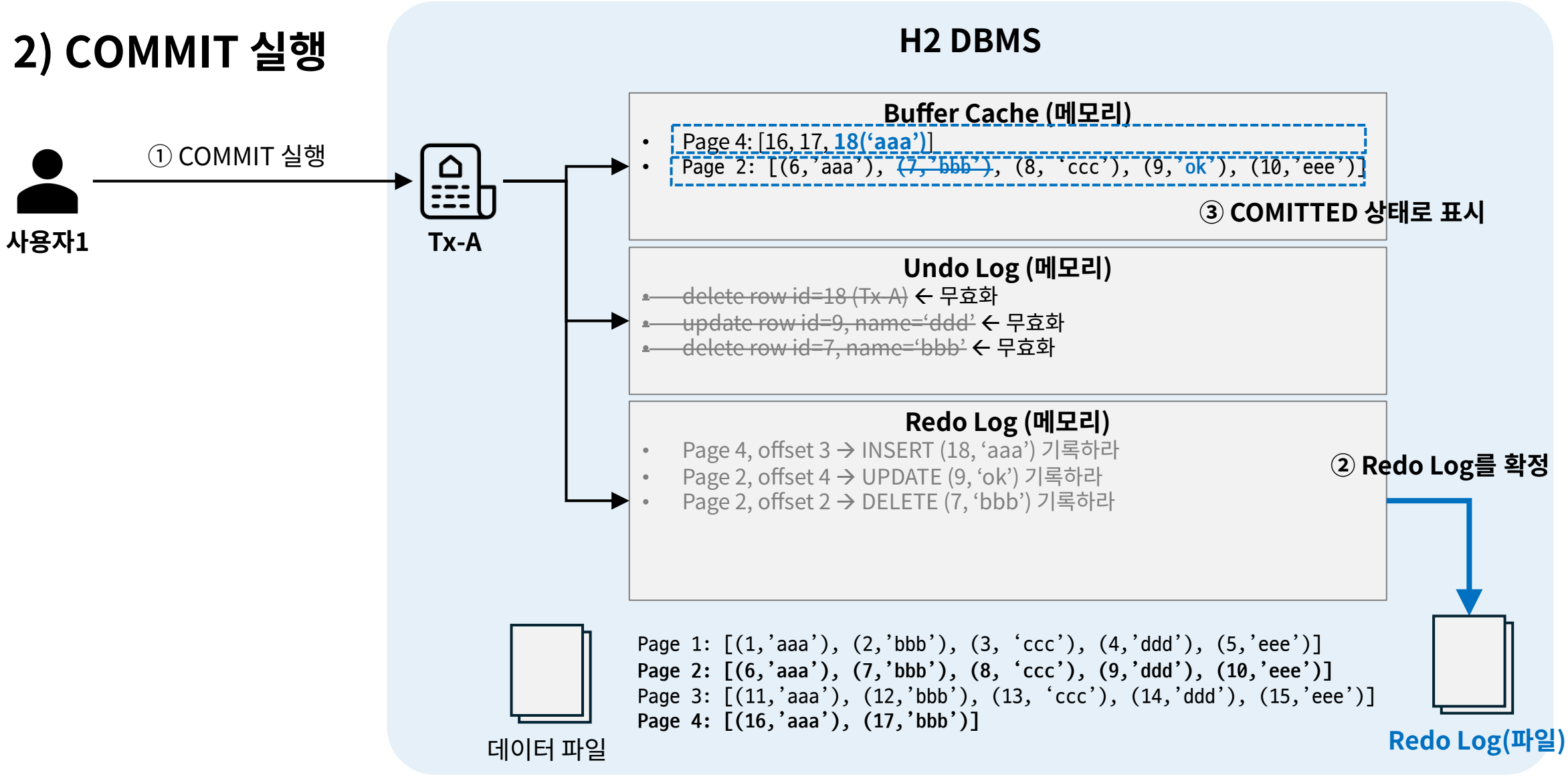
1) DML 실행



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

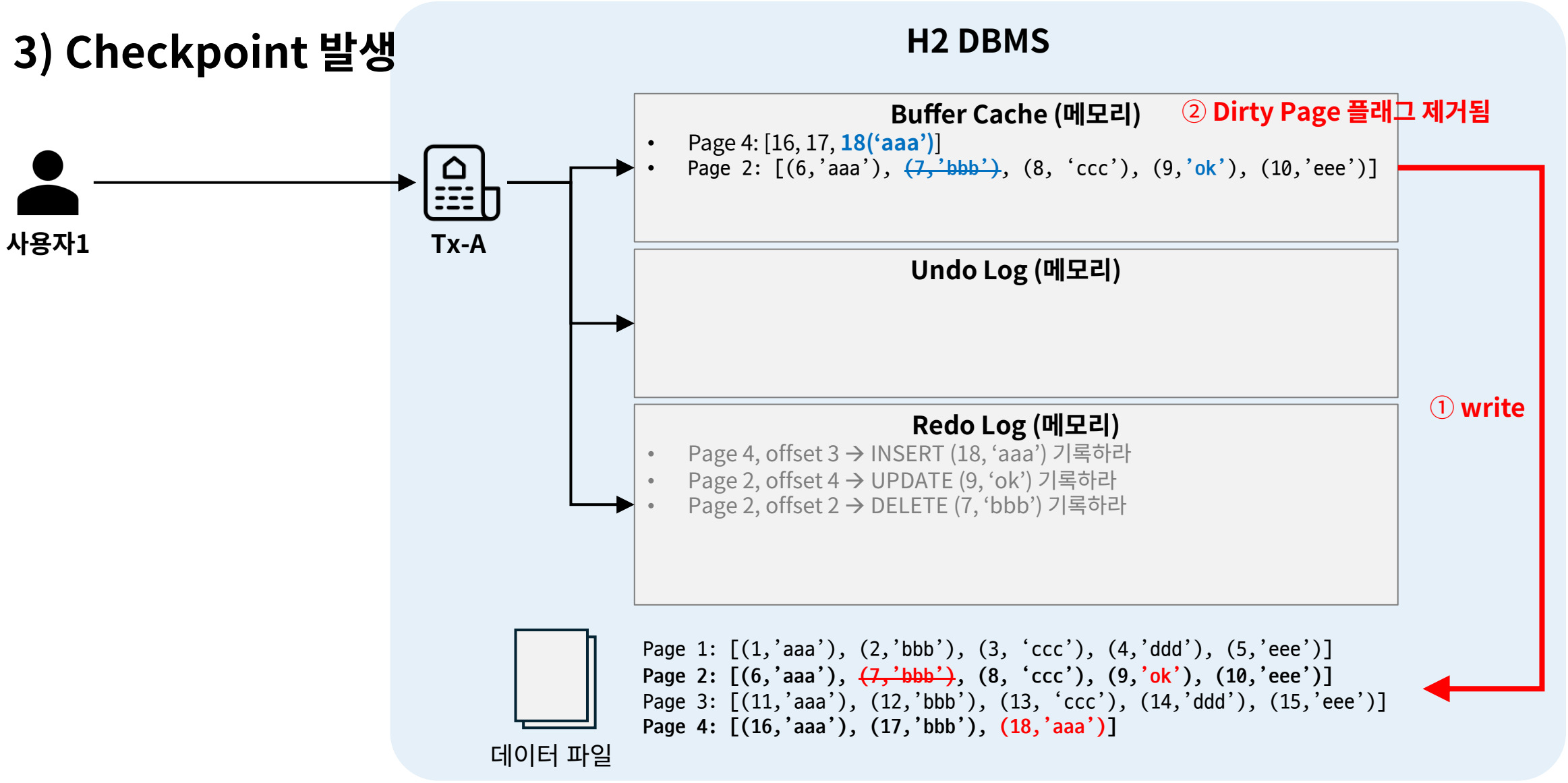
2) COMMIT 실행



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

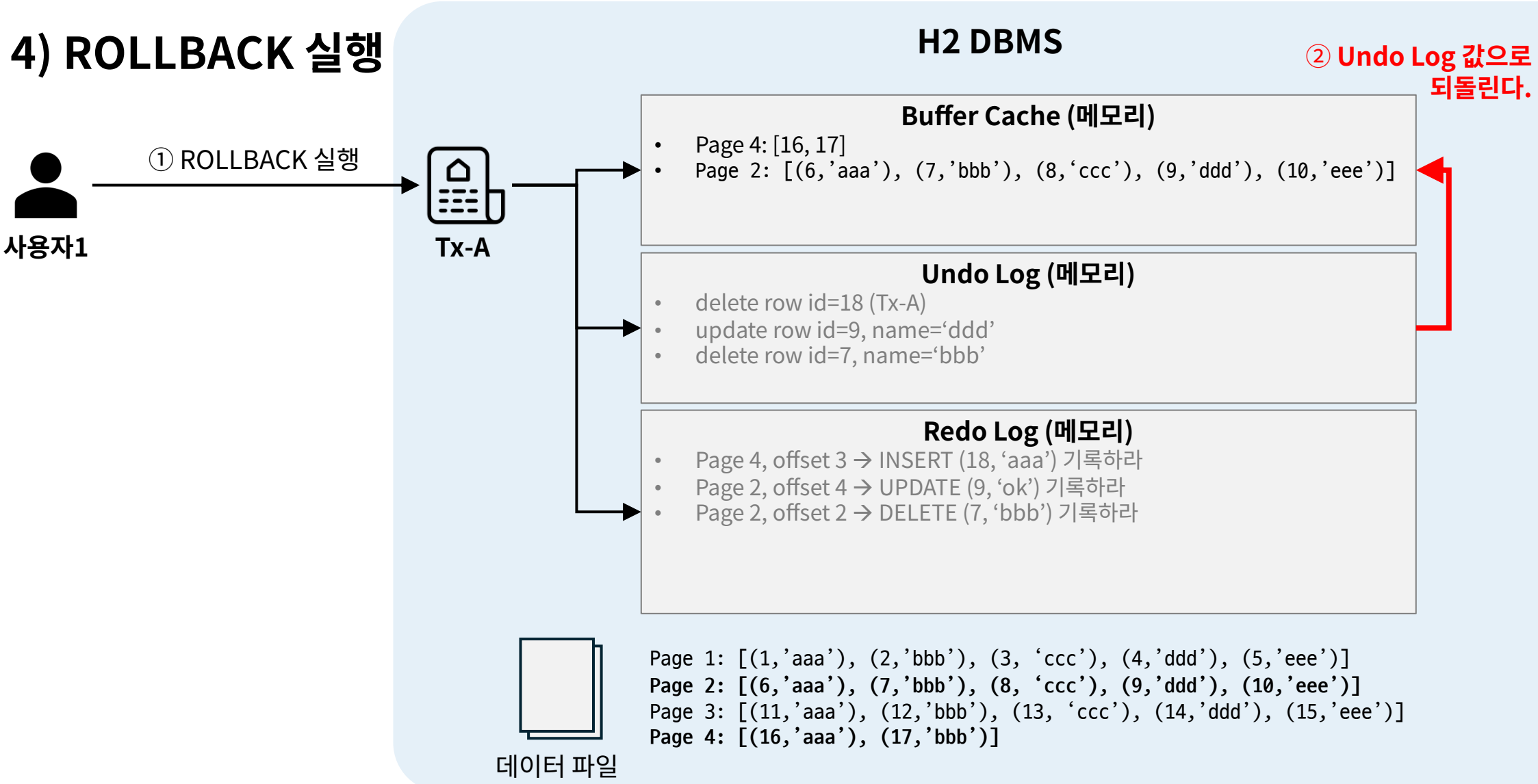
3) Checkpoint 발생



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	복잡한 SQL 리포트/통계 중심 DB 튜닝이 중요한 시스템	CRUD 중심 애플리케이션 도메인 로직이 중요한 서비스 생산성이 중요한 프로젝트

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”

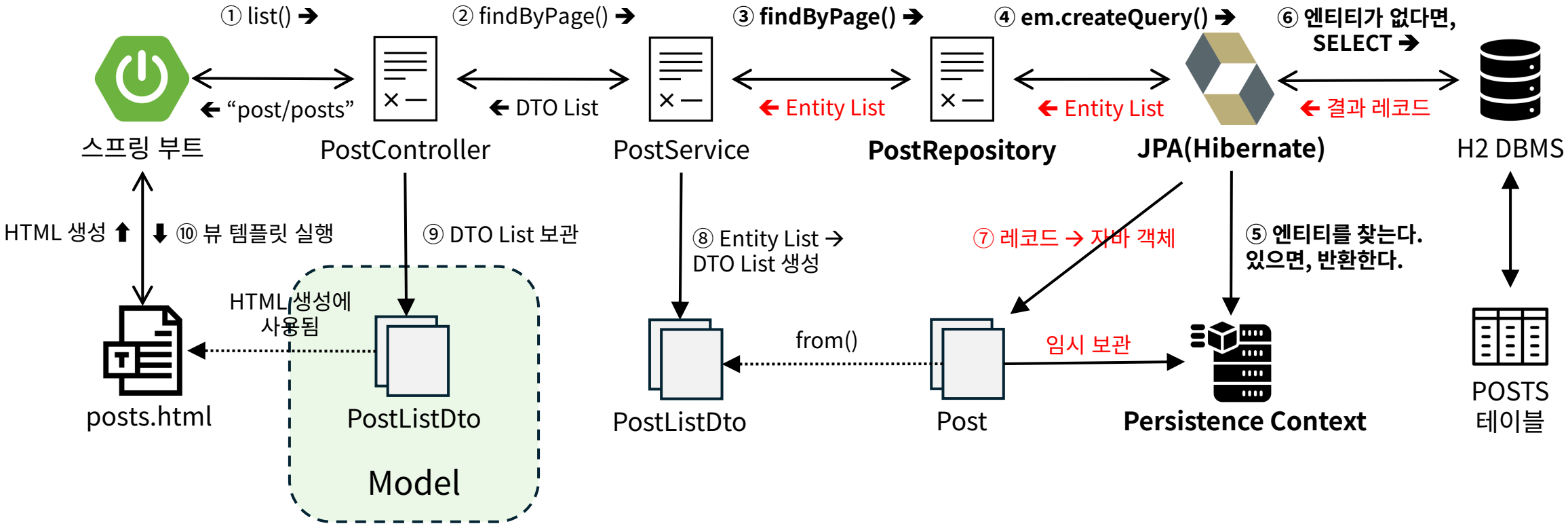
9. SQL Mapper를 ORM으로 바꾸기 – 스프링부트 스타터

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

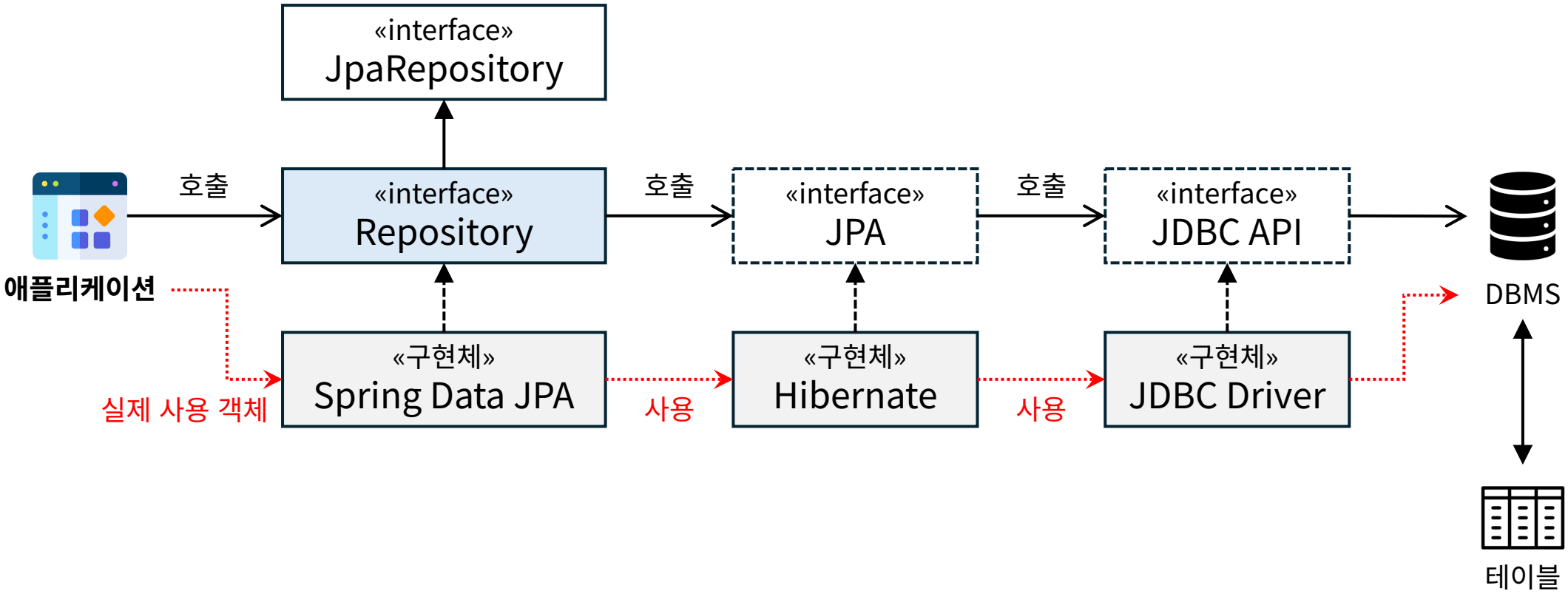
```java
@Service
public class PostService {
    @PersistenceContext
    private EntityManager em;

    @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

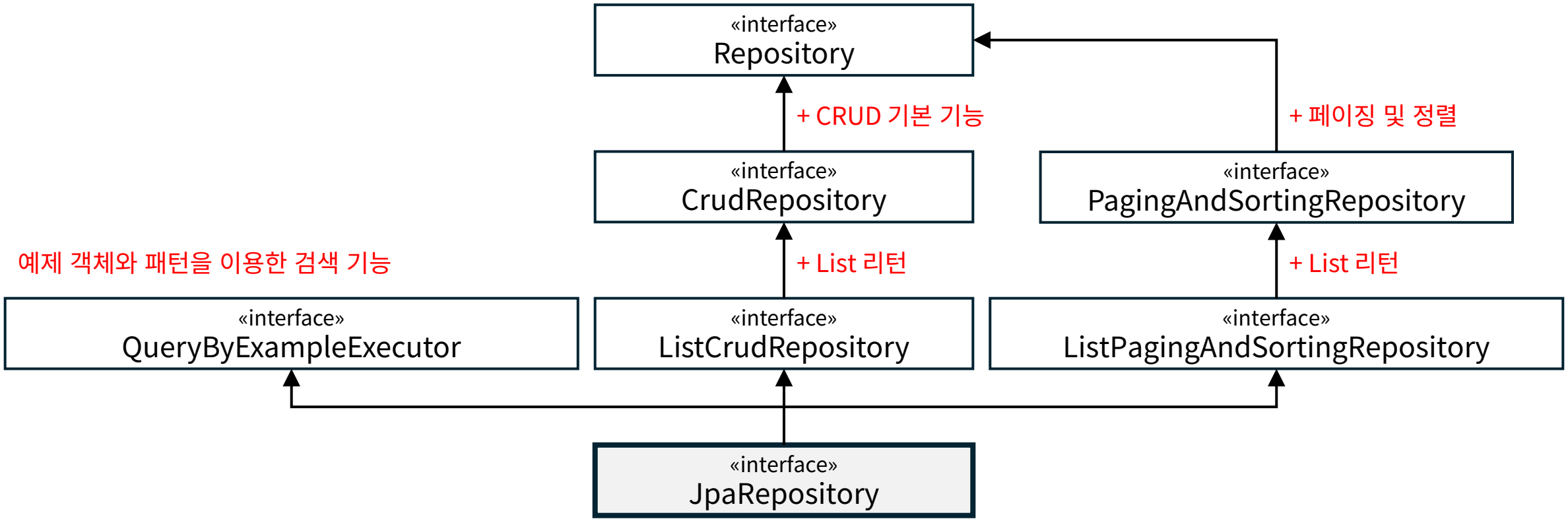


10. Spring Data JPA로 리팩토링하기 – 순수 JPA vs Spring Data JPA

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:

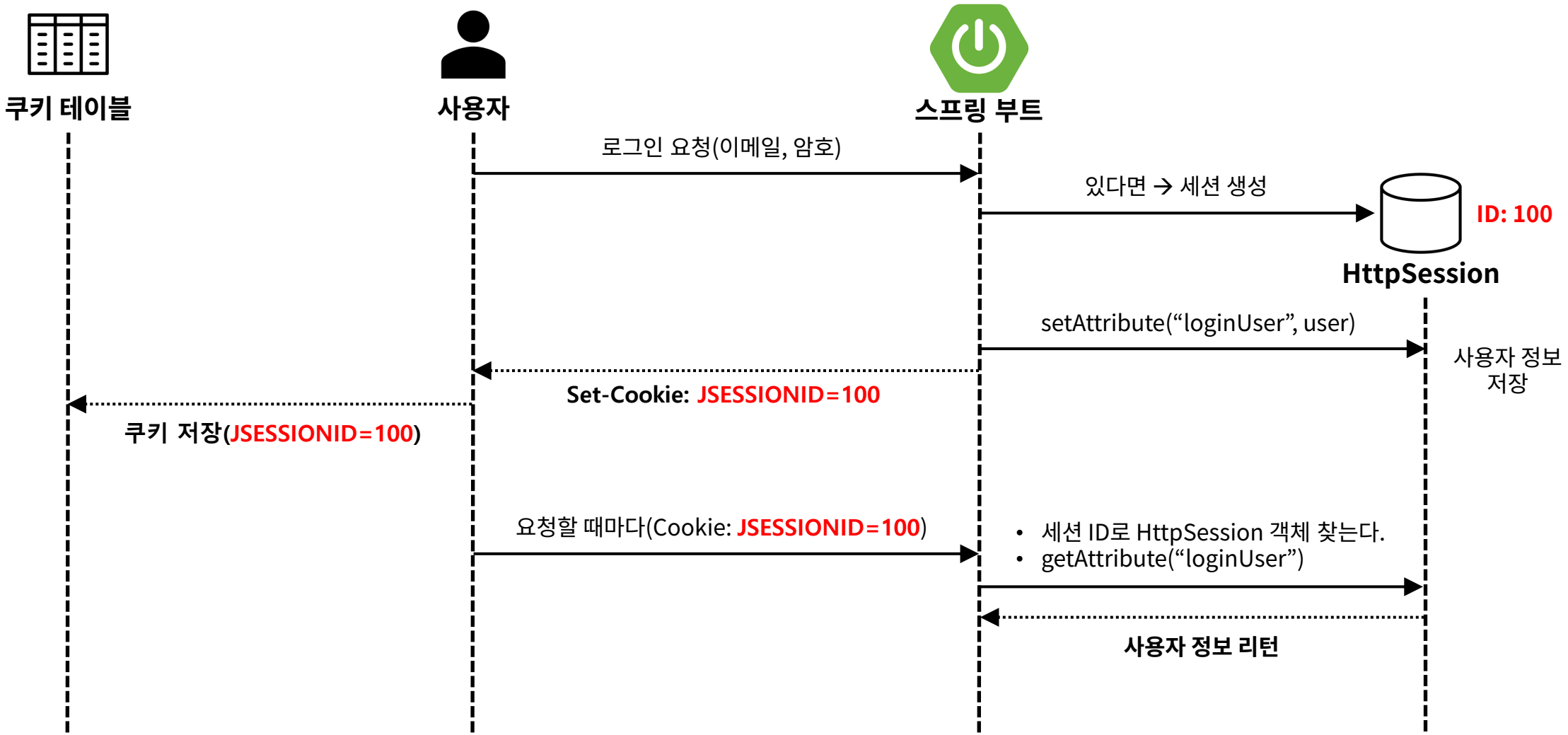
```
List<Post> results = queryFactory
    .selectFrom(post)
    .where(
        post.title.contains(keyword)
            .or(post.content.contains(keyword))
    )
    .orderBy(post.createdAt.desc())
    .limit(10)
    .fetch();
```

11. 세션 기반 사용자 인증하기(without Spring Security)

학습 목표

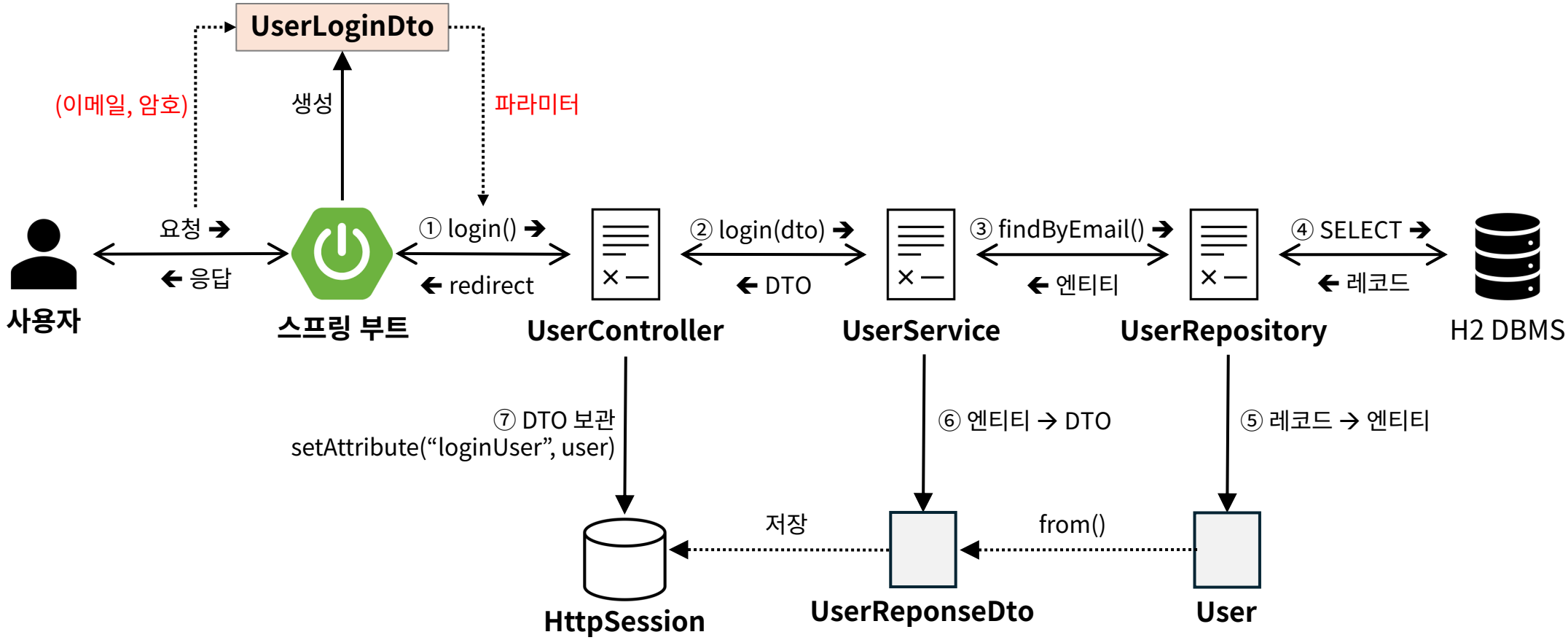
- **인증(Authentication)**과 **인가(Authorization)**의 차이를 설명하고, **HTTP 무상태성**으로 인한 세션 필요성과 **쿠키-세션 관계**를 이해한다.
- 세션 기반 로그인 처리 흐름(로그인 → 세션 생성 → 인증 → 로그아웃)을 단계별로 설명할 수 있다.
- **HttpSession**을 이용해 **로그인 상태를 저장·조회·삭제**하고, 사용자 정보를 세션에 안전하게 저장할 수 있다.
- 비밀번호를 암호화하여 저장하고 검증하는 방법을 구현할 수 있다.
- **Interceptor(또는 Filter)**를 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

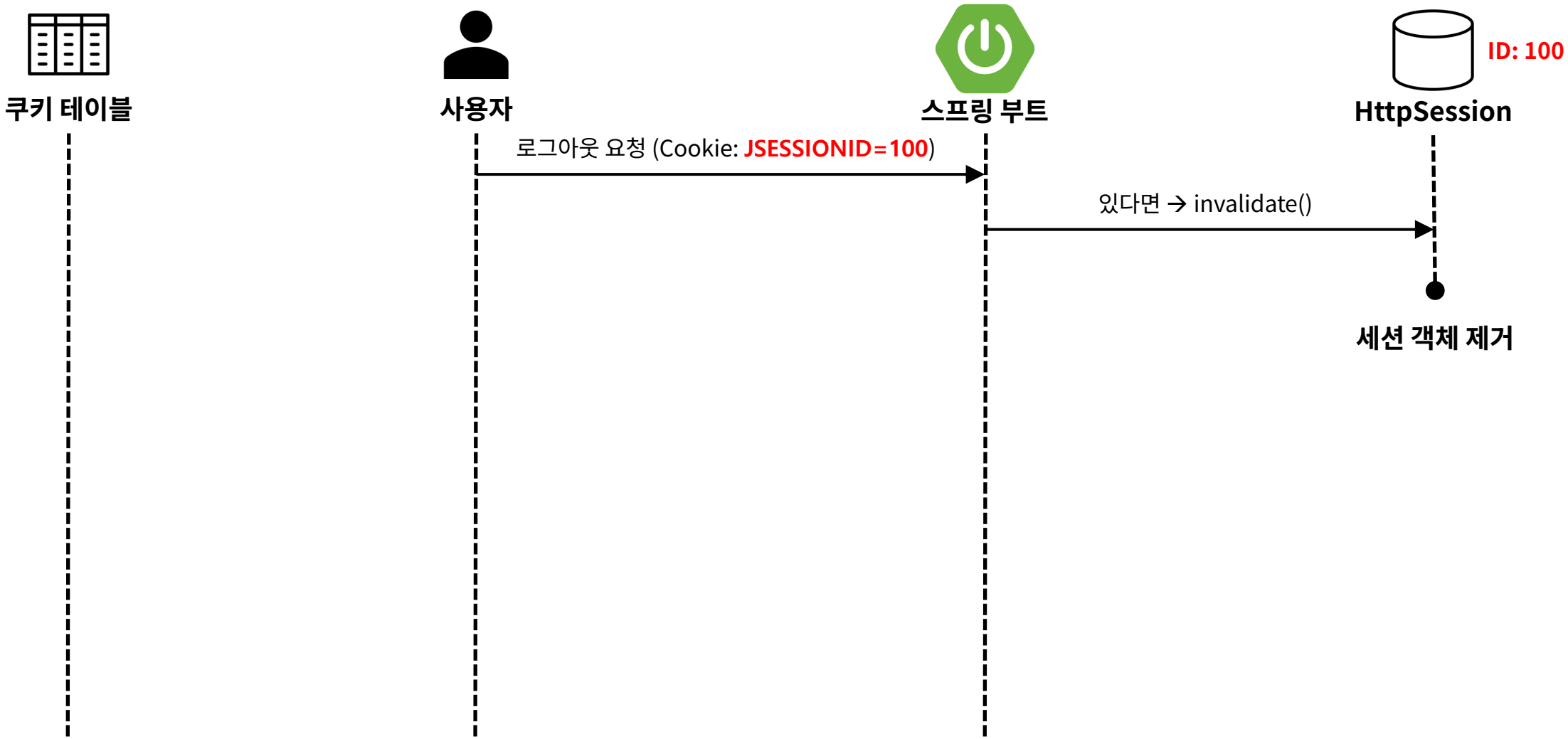


11. 세션 기반 사용자 인증하기 – 아키텍처(로그인 하기)

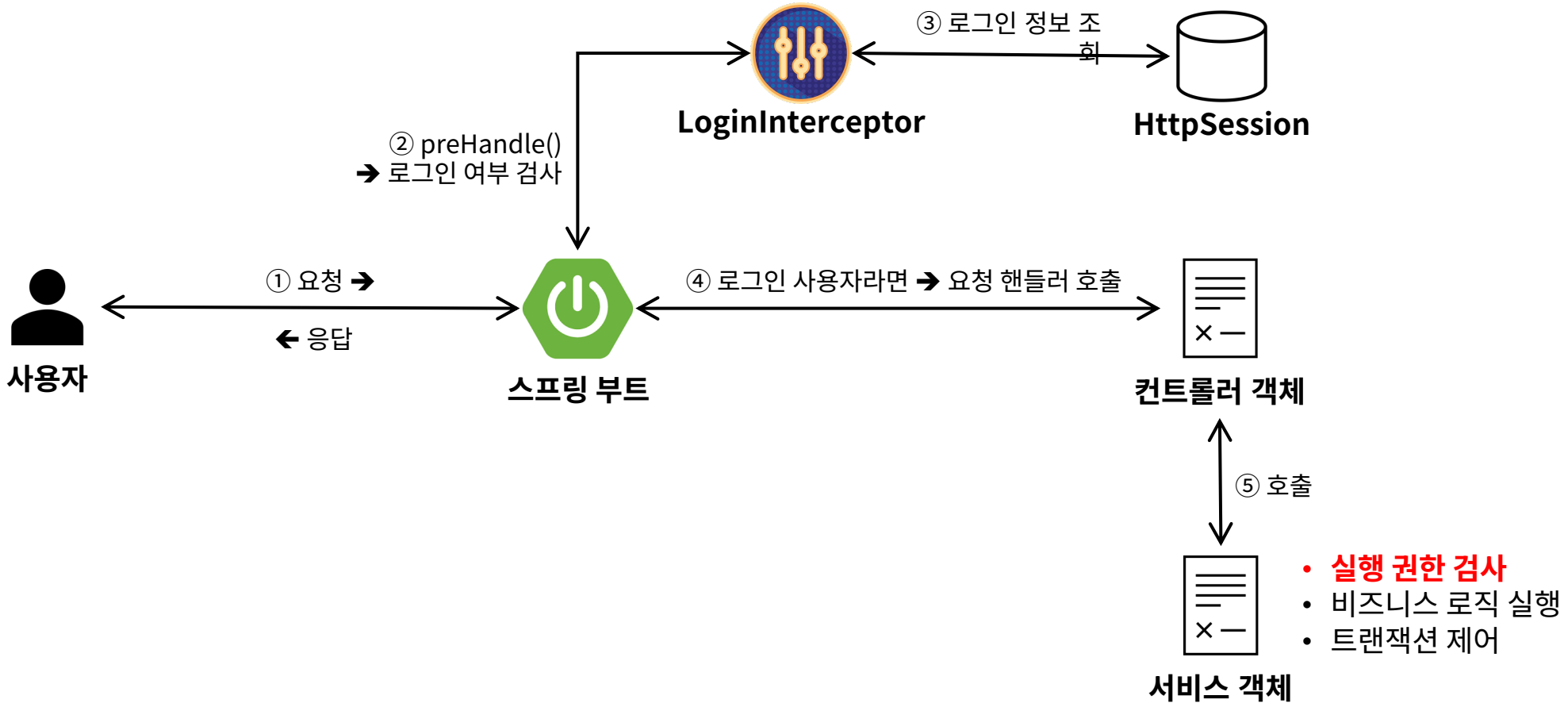
URL: /login



11. 세션 기반 사용자 인증하기 – 로그아웃 처리 흐름

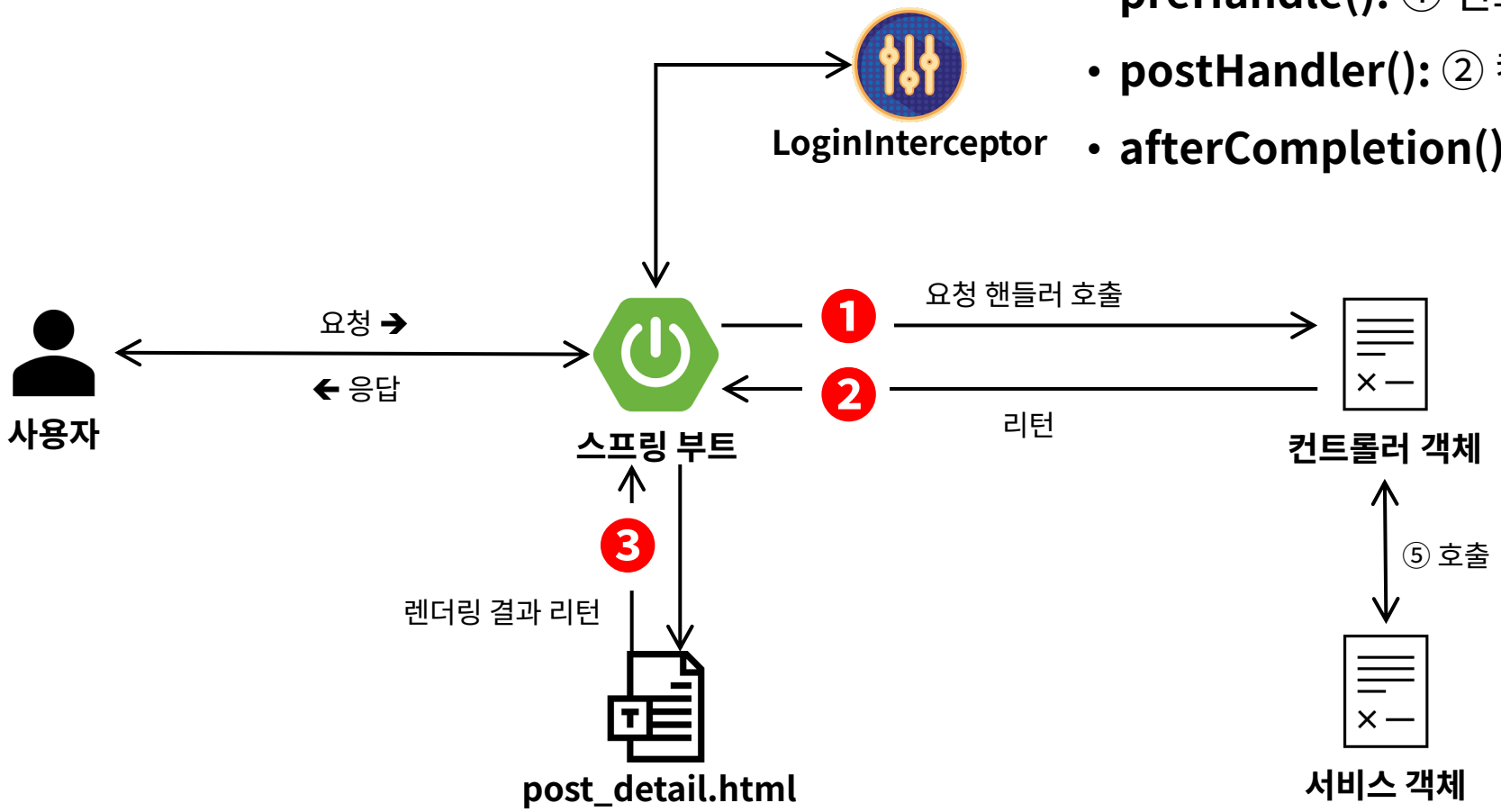


11. 세션 기반 사용자 인증하기 – 인가(Authorization)



11. 세션 기반 사용자 인증하기 – Interceptor 동작 원리

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



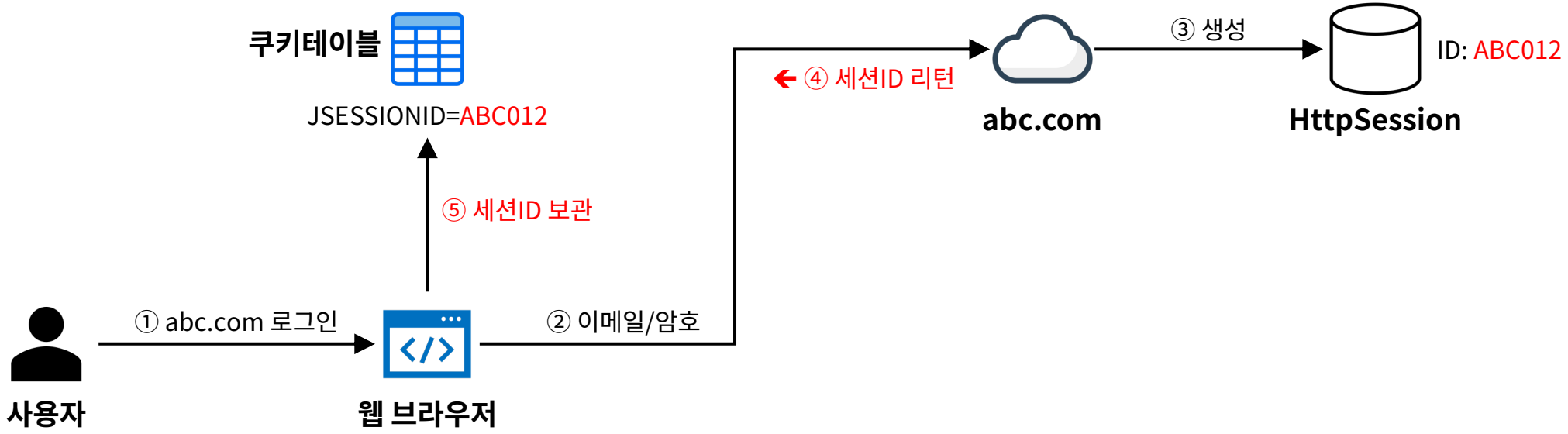
12. Spring Security 도입하기

학습 목표

- **Spring Security의 역할**과 수동 구현 대비 **도입 효과**를 설명할 수 있다.
- Spring Security의 **FilterChain 아키텍처**와 **주요 필터들**(인증, 인가, CSRF 등)**의 역할**을 이해한다.
- Spring Security 기본 설정 시 **자동 적용되는 보안 기능** (폼 로그인, CSRF 보호, 세션 관리)을 확인하고 설명할 수 있다.
- Interceptor를 통한 세션 기반 인증과 Spring Security FilterChain 기반 **인증 흐름을 비교하고 차이점을 설명**할 수 있다.
- **UserDetails**와 **UserDetailsService**를 구현하여 **사용자 인증 로직을 Spring Security에 통합**하고, 비밀번호 자동 검증을 이해한다.
- SecurityFilterChain을 설정하여 **URL 기반 접근 제어**(Authorization)와 로그인/로그아웃 흐름을 구성할 수 있다.
- **CSRF 보호의 필요성을 이해**하고, **Thymeleaf 폼에서 CSRF 토큰이 자동 적용되는 것**을 확인할 수 있다.
- 바이브코딩을 통해 기존 세션 인증 코드(Interceptor, 수동 검증)를 Spring Security 기반으로 리팩토링하고, 코드 감소 및 보안 강화 효과를 확인한다.

"Spring Security는
직접 구현했던 인증·인가 로직을 검증된 표준 구조로 대체하여
코드량은 줄이고 보안 수준은 높여주는 프레임워크다."

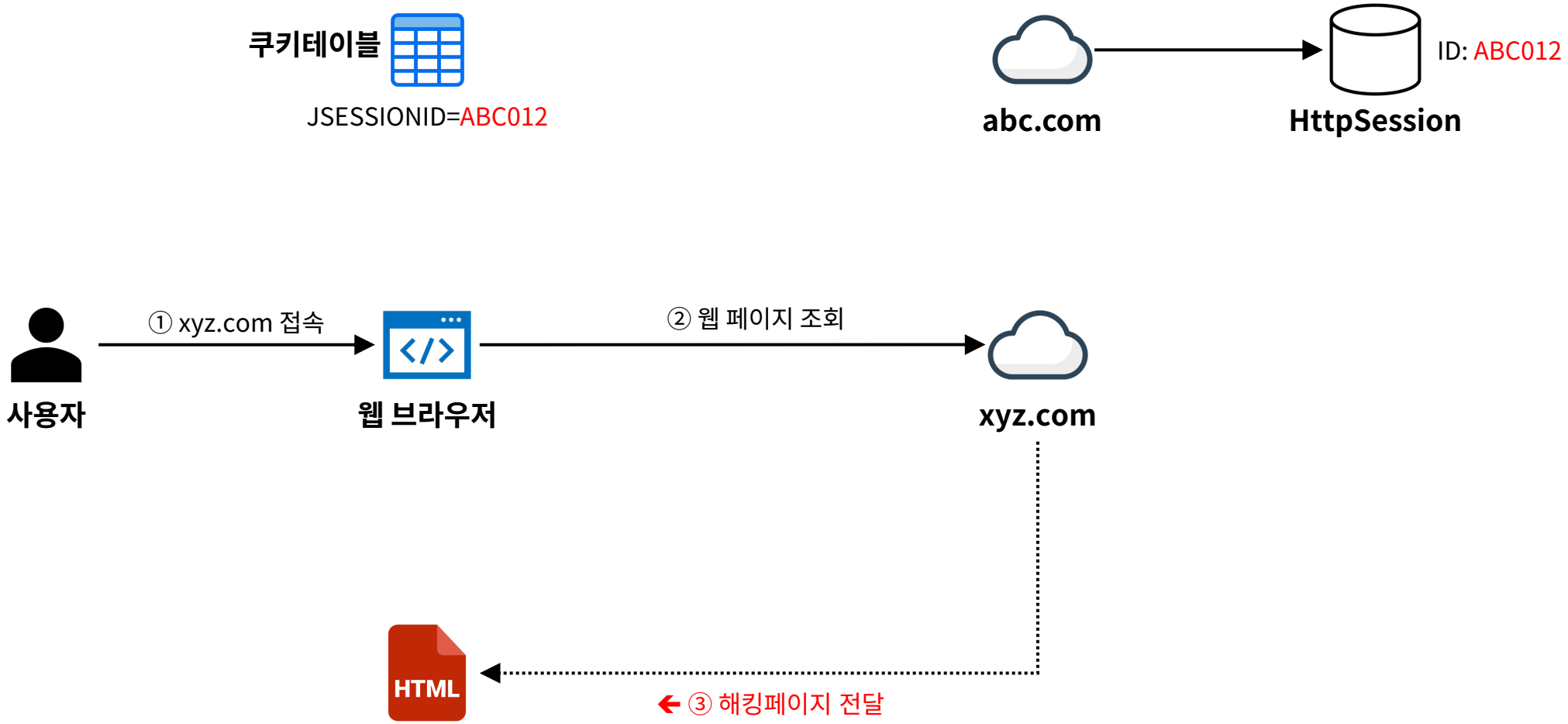
12. Spring Security 도입하기 – CSRF 공격 체험



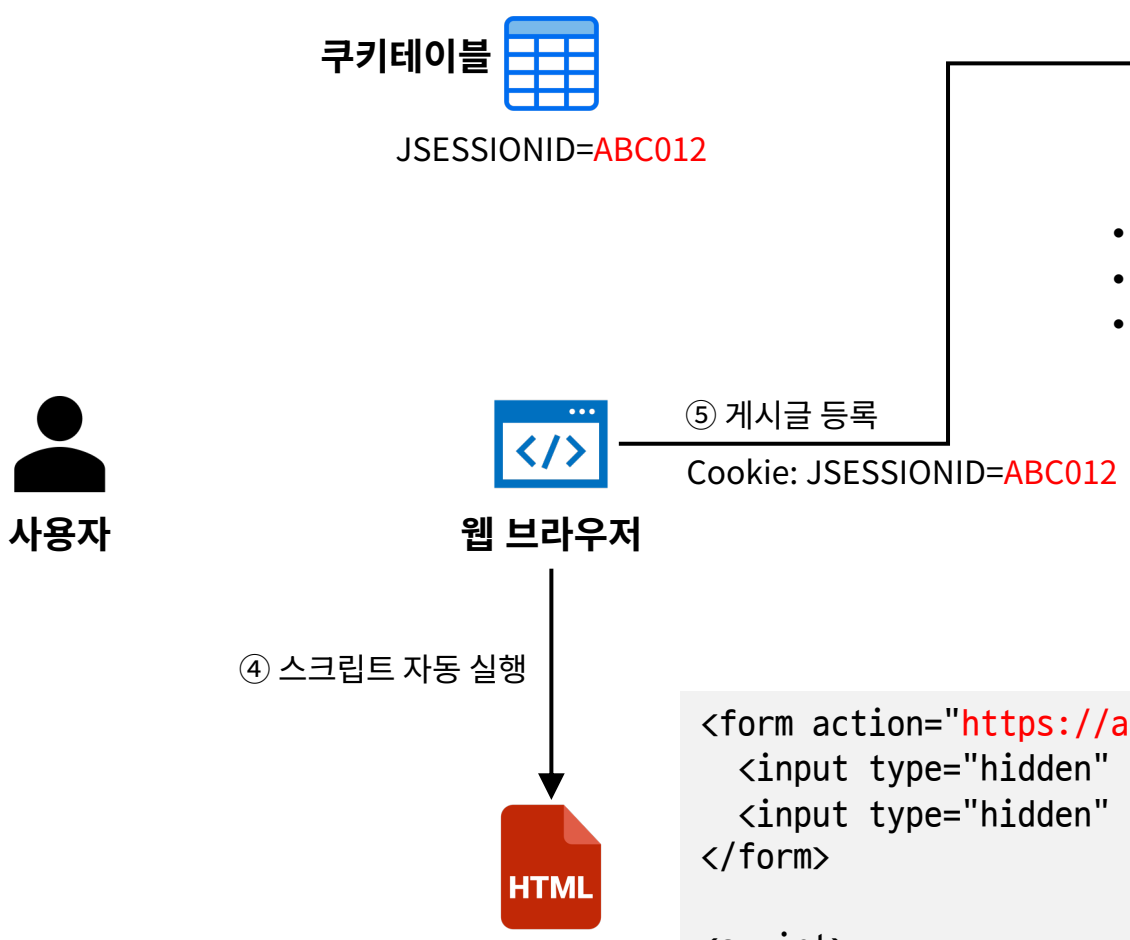
CSRF 용어

- Cross-Site → 사용자가 접속한 다른 사이트에서
- Request → HTTP 요청을
- Forgery → 사용자의 의도와 다르게 위조해서 보낸다.

12. Spring Security 도입하기 – CSRF 공격 체험



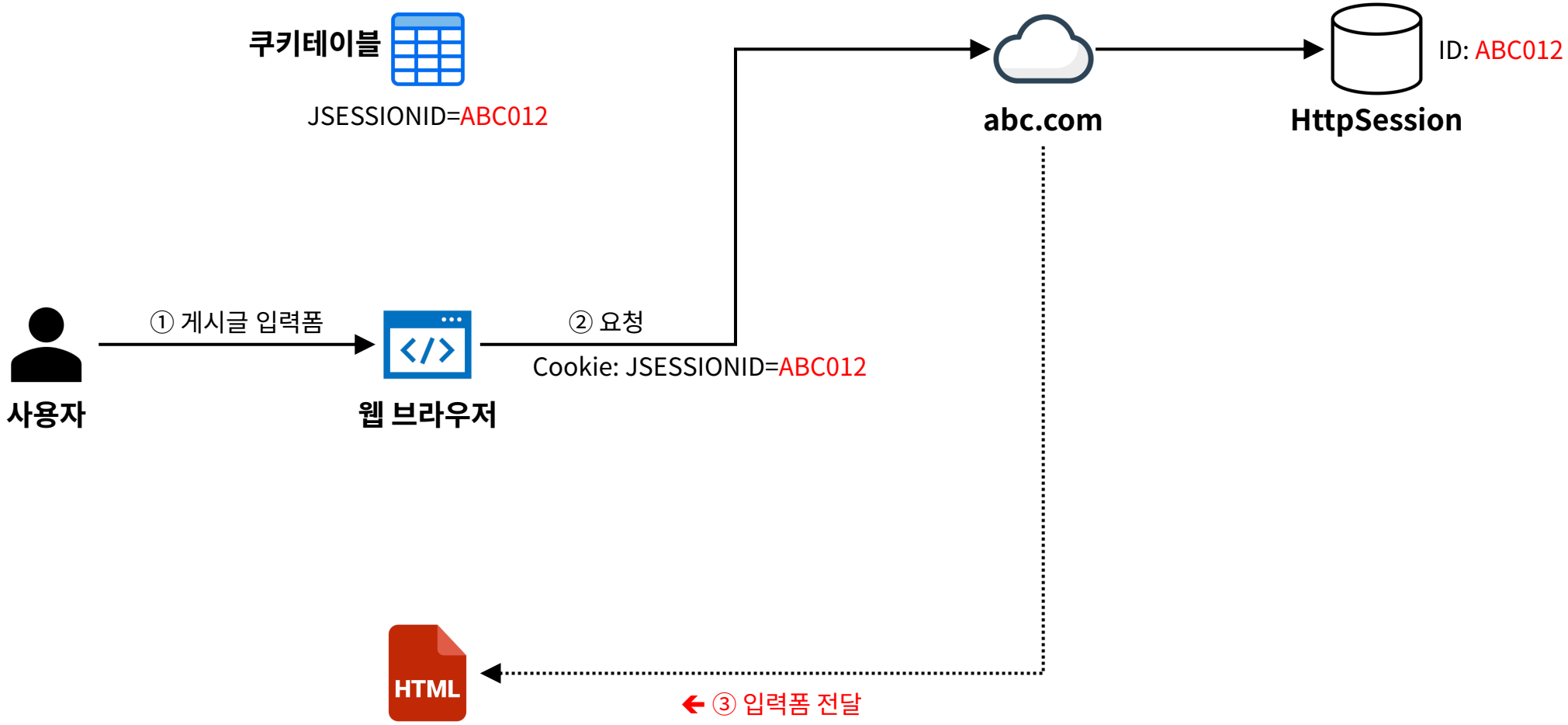
12. Spring Security 도입하기 – CSRF 공격 체험



- 서버는 쿠키로 전송된 **세션ID로 사용할 세션 객체를 결정**한다.
- 세션에 **저장된 로그인 정보를 사용**하여 요청을 처리한다.
- 즉, 이전에 로그인한 사용자의 게시글로 간주하여 등록한다.

```
<form action="https://abc.com/posts/add" method="POST">
  <input type="hidden" name=”title" value=" 하하하">
  <input type="hidden" name=”content" value=”해킹해킹!!">
</form>

<script>
  document.forms[0].submit();
</script>
```

12. Spring Security 도입하기 – CSRF 공격 방어



12. Spring Security 도입하기 – CSRF 공격 방어

- 서버는 입력폼을 보낼 때 함께 보낸 **CSRF 방지 코드 값을 확인**한다.
- **보낸 코드 값과 일치할 때**만 정상 로그인 사용자 요청으로 간주한다.


```
<form action="/posts/add" method="POST">
 <input type="hidden" name="_csrf" value="랜덤값">
  제목: <input type=”text" name=”title">
  내용: <textarea name=”content"></textarea>
  <button>등록</button>
</form>
```

12. Spring Security 도입하기 – 역할

- 웹 애플리케이션의 **인증**(Authentication)과 **인가**(Authorization)를 표준화된 방식으로 처리해 주는 **보안 프레임워크**이다.
- 하는 일:
  - 로그인 / 로그아웃 처리
  - 인증 여부 판단
  - 권한(Role) 기반 접근 제어
  - 세션 관리
  - CSRF 공격 방어
  - 비밀번호 암호화 및 검증
  - 보안 관련 예외 처리
- 즉, **보안과 관련된 공통 문제를 전담하는 인프라 계층**이다.

12. Spring Security 도입하기 – 도입 효과

수동 구현 방식:

"인증과 인가를 애플리케이션 코드로 직접 구현"

한계:

① 코드 중복과 산재
- 인증 체크 로직이 Interceptor, Controller, Utility 클래스 등에 흩어짐
- 유지보수 난이도 증가

② 보안 취약점 발생 가능성
- 세션 고정(Session Fixation) 미대응, CSRF 미처리, 인증 실패 처리 미흡
- 안전하지 않음

③ 확장성 부족
- Role 기반 인가 추가 어려움
- API / 관리자 / 사용자 분리 어려움
- JWT, OAuth 같은 확장 불가능

Spring Security 사용:

"인증과 인가를 프레임워크 레벨에서 자동 처리"

도입 효과:

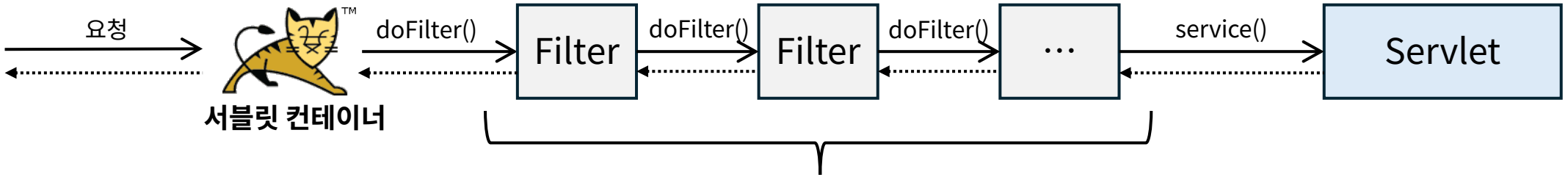
① 인증·인가 책임 분리
- 보안은 보안 프레임워크가 처리

② 표준화된 보안 흐름 제공
- FilterChain 기반 인증, SecurityContext로 인증 정보 관리, 검증된 패턴과 구조
- 개발자가 실수할 여지를 줄임

③ 코드 감소 효과
- 설정 클래스 몇 개, UserDetailsService 구현
- 코드는 줄고 기능은 더 강력

④ 보안 기본값 제공
- 인증되지 않은 접근 차단, CSRF 보호 활성화, 세션 관리 정책 적용
- 아무 설정 안 해도 기본은 안전

12. Spring Security 도입하기 – 서블릿 API의 Filter



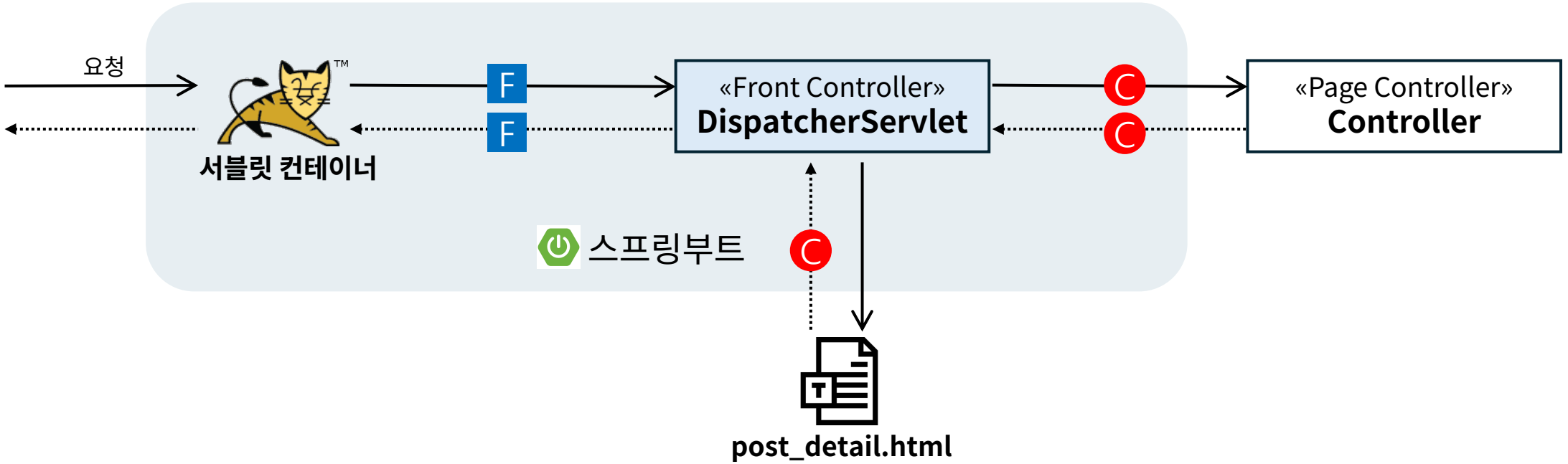
필터의 역할:

① HTTP 요청이 **서블릿에 도착하기 전에 전처리 수행**
- 요청을 계속 진행시킬지 결정(인증/인가 검사)
- 요청 인코딩 설정 (예: UTF-8)
- CORS 검사
- 로깅/모니터링 시작
- 요청 데이터 전처리 (Header 추가/변경, 파라미터 검증)

② 서블릿 실행 후 **HTTP 응답을 하기 전에 후처리 수행**
- 응답 로깅/성능 측정
- 응답 헤더 추가(보안 헤더, 캐시 제어 헤더)
- 응답 내용 가공(JSON 변환, HTML 후처리)
- 세션/컨텍스트 정리
- 예외 후처리

③ 필터의 설계 구조 상 기능 추가, 삭제가 자유롭다.

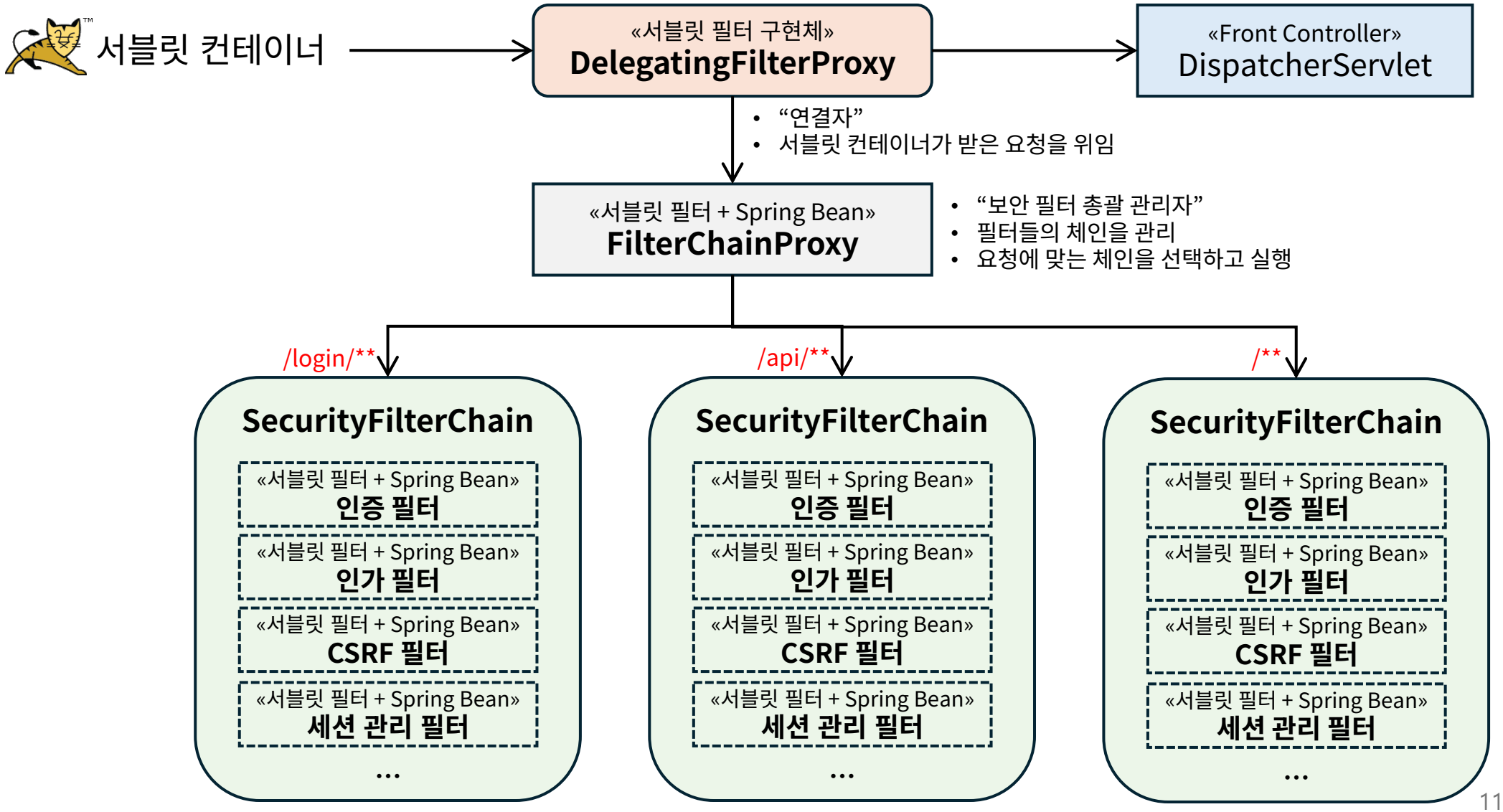
12. Spring Security 도입하기 – Filter vs Interceptor



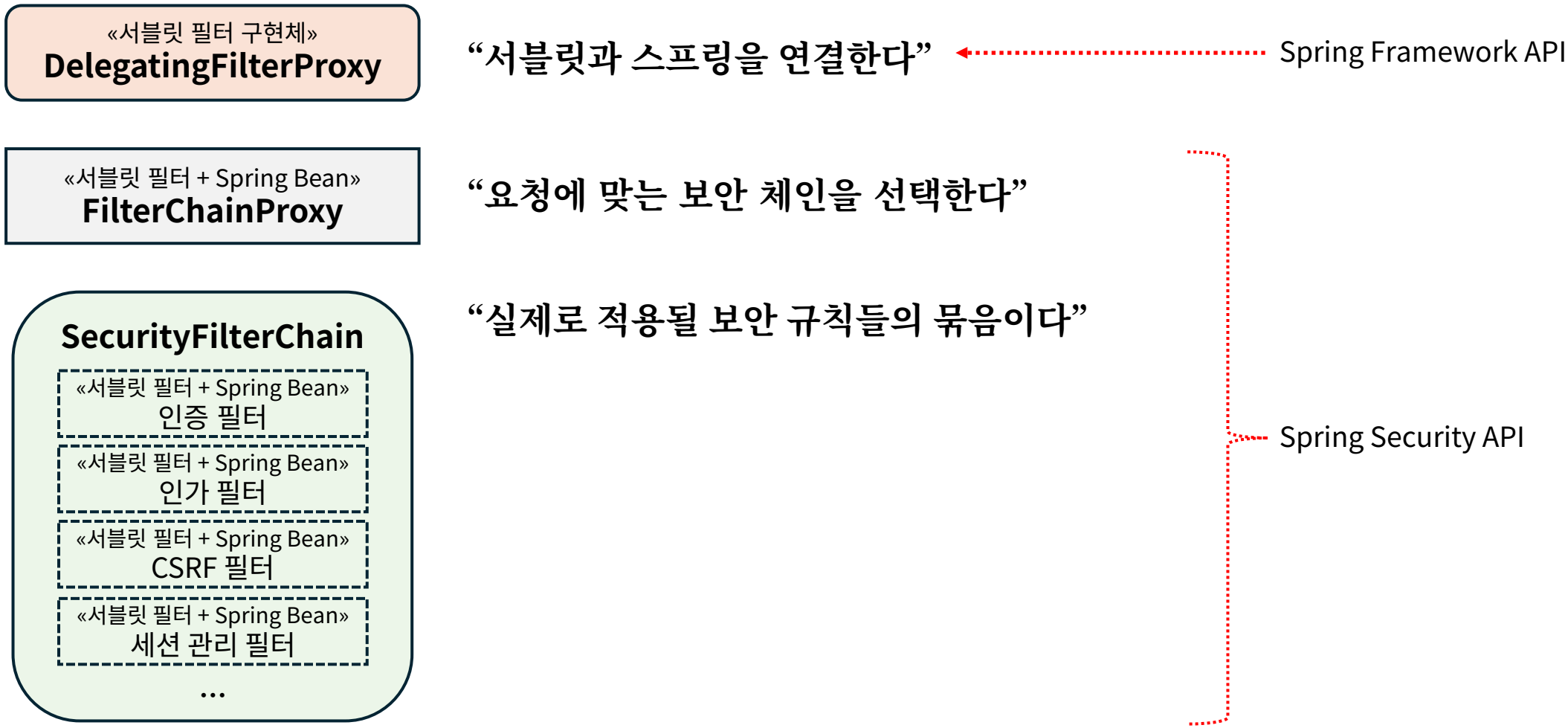
F **필터:**
서블릿 실행 전·후에 수행할 작업이 있을 경우에 사용한다.
Jakarta EE의 Servlet API 소속이다.

C **인터셉터:**
컨트롤러 실행 전·후에 수행할 작업이 있을 경우에 사용한다.
뷰 렌더링을 끝낸 후에 수행할 작업이 있을 경우에 사용한다.
Spring Framework API 소속이다.

12. Spring Security 도입하기 – FilterChain 아키텍처



12. Spring Security 도입하기 – FilterChain 아키텍처



12. Spring Security 도입하기 – FilterChain 아키텍처

왜 이렇게 복잡한 구조인가?

① Spring과 서블릿 컨테이너 분리

- DelegatingFilterProxy 필요

② URL 별 보안 정책 분리

- SecurityFilterChain 여러 개 필요

③ 확장성과 조합성

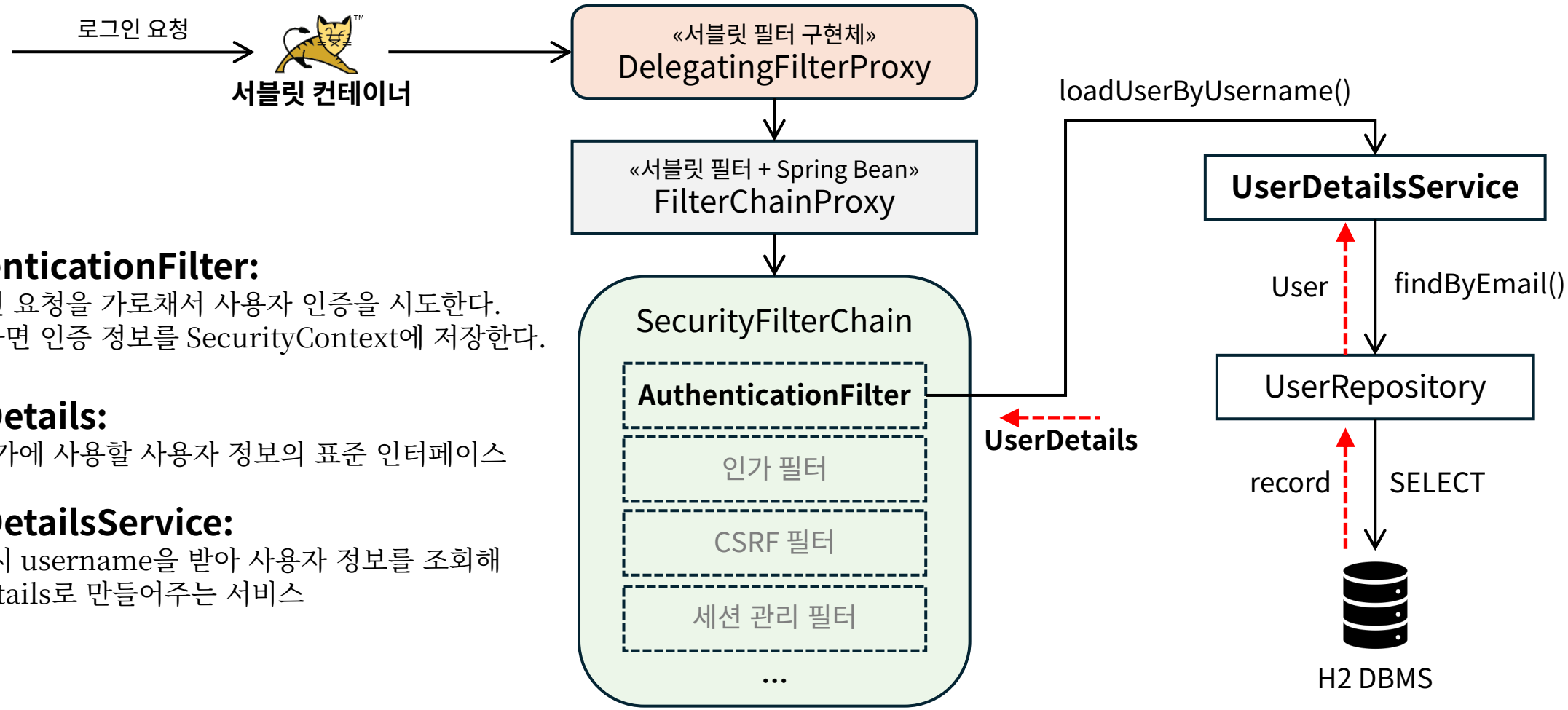
- 인증 방식, 인가 방식, CSRF, 세션 정책을 조합

✓ 즉, 대규모 시스템을 위한 구조

12. Spring Security 도입하기 – SecurityFilterChain 구성하기

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(Customizer.withDefaults())          <-- CsrfFilter 설정
            .httpBasic(Customizer.withDefaults())     <-- BasicAuthenticationFilter 설정
            .formLogin(Customizer.withDefaults())     <-- UsernamePasswordAuthenticationFilter 설정
            .authorizeHttpRequests((authorize) -> authorize
                .anyRequest().authenticated()
            );                                        <-- AuthorizationFilter 설정

        return http.build();
    }

}
```

12. Spring Security 도입하기 – 로그인 흐름과 주요 역할자



AuthenticationFilter:
- 로그인 요청을 가로채서 사용자 인증을 시도한다.
- 성공하면 인증 정보를 SecurityContext에 저장한다.

UserDetails:
인증·인가에 사용할 사용자 정보의 표준 인터페이스

UserDetailsService:
로그인 시 username을 받아 사용자 정보를 조회해 UserDetails로 만들어주는 서비스

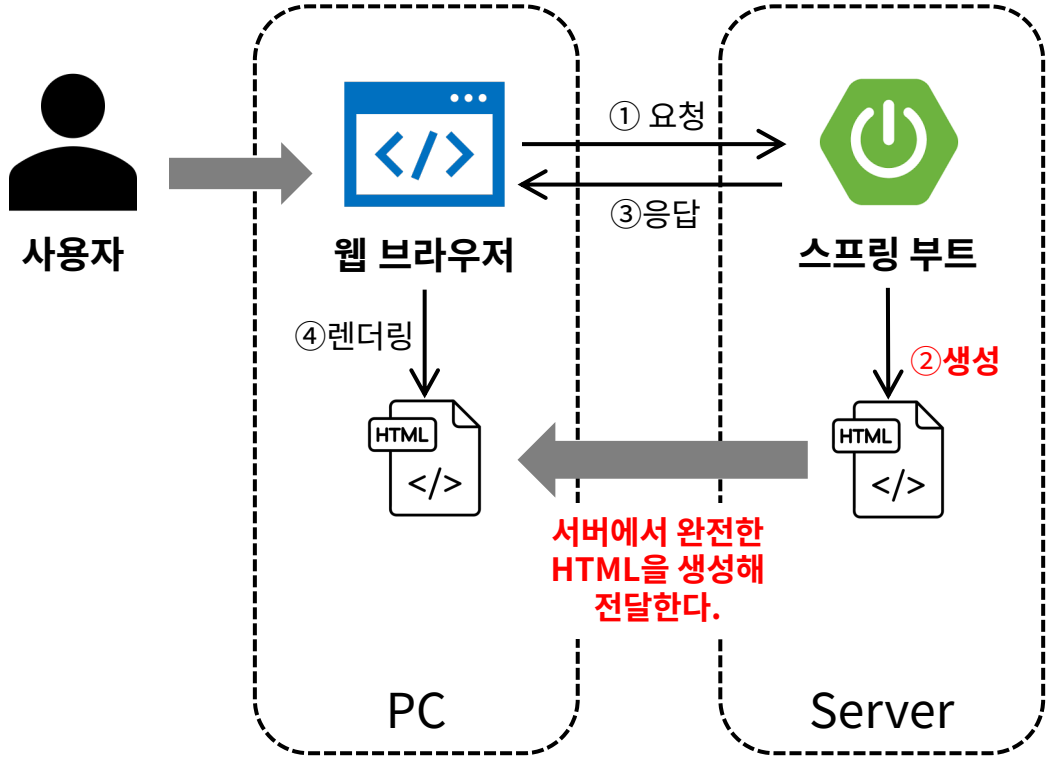
13. SSR 방식에서 CSR 방식으로 전환하기

학습 목표

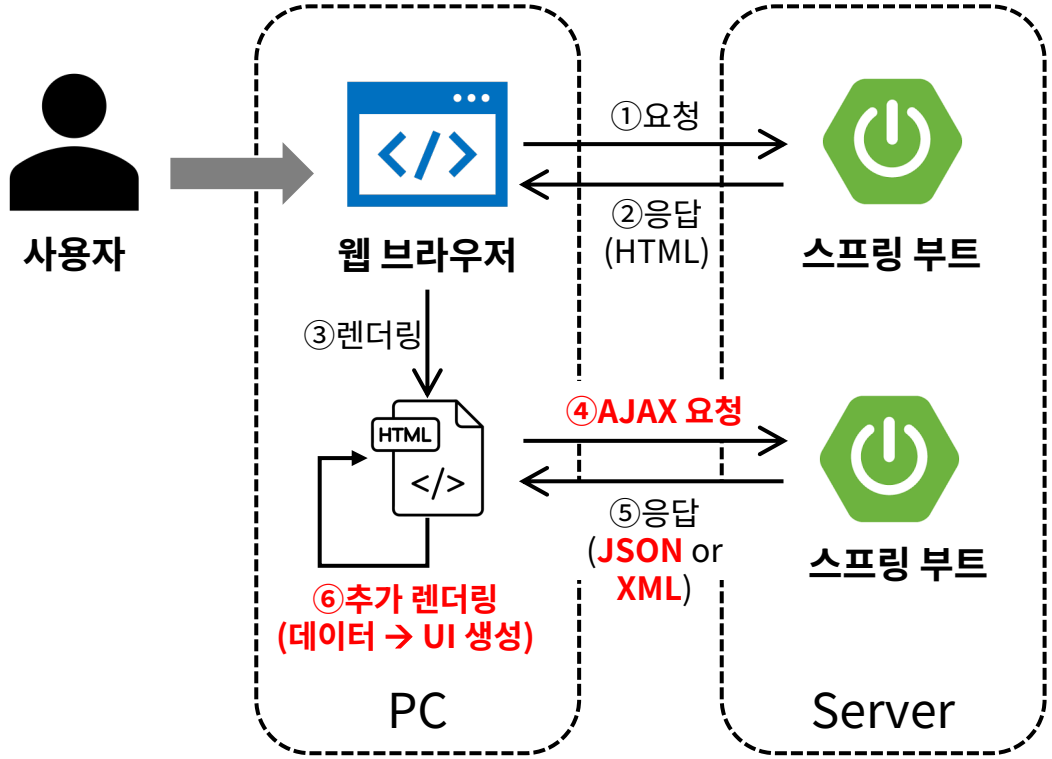
- 서버 사이드 렌더링(Thymeleaf)과 클라이언트 사이드 렌더링(JavaScript)의 차이와 장단점을 설명할 수 있다.
- REST API 설계 원칙(리소스 중심, HTTP 메서드, 상태 코드)을 이해하고 올바르게 적용할 수 있다.
- @RestController와 ResponseEntity를 활용하여 JSON 기반 REST API를 구현할 수 있다.
- @RestControllerAdvice를 사용하여 API 예외를 JSON 에러 응답으로 처리할 수 있다.
- 클라이언트(JavaScript)에서 Fetch API를 사용하여 REST API를 호출하고 동적으로 화면을 렌더링할 수 있다.
- 단일 서버 내에서 API와 정적 파일을 함께 제공하는 구조를 이해하고, 완전 분리 구조(Backend 서버 + Frontend 서버)와의 차이를 설명할 수 있다.
- CSR 환경에서 세션 쿠키 방식의 한계를 이해하고, API 인증에 적합한 토큰 기반 방식의 필요성을 설명할 수 있다.
- 바이브코딩을 통해 기존 Thymeleaf 기반 MVC 구조를 REST API + JavaScript 구조로 리팩토링한다.

13. SSR 방식에서 CSR 방식으로 전환하기 – SSR vs CSR

- **SSR**(Server-Side Rendering) 방식은 **서버에서 완성된 HTML을 생성**하여 브라우저로 전송한다. 일반적으로 사용자가 페이지를 요청할 때마다 서버가 HTML 응답을 생성한다.
- **CSR**(Client-Side Rendering) 방식은 최소한의 HTML과 JavaScript 번들을 받은 후, 브라우저에서 실행된 자바스크립트가 **서버로부터 데이터(예: JSON)를 가져와** 동적으로 DOM을 조작해 **화면을 구성**한다.



SSR(Server-Side Rendering)



CSR(Client-Side Rendering)

13. SSR 방식에서 CSR 방식으로 전환하기 – SSR vs CSR

SSR 방식:

- ✅ 초기 로딩 빠름
- ✅ SEO 유리
- ✅ JavaScript 비활성화 환경 지원
- ❌ 서버 부하
- ❌ 페이지 전환마다 새로고침

CSR 방식:

- ✅ 부드러운 UX
- ✅ 서버 부하 감소
- ✅ 모바일 앱과 동일 API
- ❌ 초기 로딩 느림
- ❌ SEO 불리
- ❌ JavaScript 필수

SEO:

검색 엔진 최적화(Search Engine Optimization)의 약자다.
웹사이트가 구글·네이버 같은 검색 엔진에서 더 잘 검색되고, 더 위에 노출되도록 만드는 모든 기술과 전략을 말한다.

SSR에서는 응답할 때 완전한 HTML을 보내기 때문에 검색 엔진이 웹페이지의 내용을 잘 이해할 수 있다.

CSR에서는 응답할 때 최소 HTML을 보내기 때문에 검색 엔진에 웹 페이지의 내용을 바로 파악할 수 없다.

13. SSR 방식에서 CSR 방식으로 전환하기 – REST API 설계 원칙

- REST는 “REpresentational State Transfer”의 약자다.
- 자원의 현재 **상태**를 특정 **표현**(JSON, XML 등)으로 만들어 클라이언트에게 **전달**한다는 의미다.
- 원칙
  - 1) 리소스 중심 설계
  - 2) 올바른 HTTP 메서드 사용
  - 3) 적절한 HTTP 상태코드 활용

13. SSR 방식에서 CSR 방식으로 전환하기 – 원칙1) 리소스 중심 설계

- **URL은 "무엇을(자원)"**을 나타내고, **HTTP 메서드는 "어떻게(동작)"**을 나타냄

❌ 잘못된 설계(동사 중심):

```
GET /getPost?id=1
POST /createPost
POST /updatePost
POST /deletePost
GET /searchPostsByTitle?title=Spring
POST /likePost?id=1
```

- 모든 동작을 URL에 표현 (동사 사용)
- HTTP 메서드를 제대로 활용 안함
- 확장성 떨어짐
- RESTful하지 않음

✅ 올바른 설계(리소스 중심):

```
GET /posts/1              # 게시글 1번 조회
POST /posts               # 게시글 생성
PUT /posts/1              # 게시글 1번 전체 수정
PATCH /posts/1            # 게시글 1번 부분 수정
DELETE /posts/1           # 게시글 1번 삭제
GET /posts?title=Spring   # 제목으로 검색
POST /posts/1/likes       # 게시글 1번에 좋아요
```

- URL은 명사(리소스)만 사용
- HTTP 메서드로 동작 구분
- 직관적이고 예측 가능
- RESTful 원칙 준수

13. SSR 방식에서 CSR 방식으로 전환하기 – 원칙1) 리소스 중심 설계(계속)

리소스 설계 패턴 예시:

1) 컬렉션과 단일 리소스

컬렉션 (복수형):

```
GET    /posts          # 게시글 목록
POST   /posts          # 게시글 생성
```

단일 리소스 (복수형 + ID):

```
GET    /posts/1        # 게시글 1번
PUT    /posts/1        # 게시글 1번 수정
DELETE /posts/1        # 게시글 1번 삭제
```

2) 계층적 리소스(중첩)

게시글의 댓글:

```
GET    /posts/1/comments          # 게시글 1번의 댓글 목록
POST   /posts/1/comments          # 게시글 1번에 댓글 작성
GET    /posts/1/comments/5        # 게시글 1번의 댓글 5번
PUT    /posts/1/comments/5        # 댓글 5번 수정
DELETE /posts/1/comments/5        # 댓글 5번 삭제
```

사용자의 게시글:

```
GET    /users/홍길동/posts        # 홍길동의 게시글 목록
GET    /users/10/posts            # 사용자 10번의 게시글 목록
```

13. SSR 방식에서 CSR 방식으로 전환하기 – 원칙1) 리소스 중심 설계(계속)

리소스 설계 패턴 예시:

3) 쿼리 파라미터 활용

필터링:

```
GET /posts?status=published
GET /posts?author=홍길동
```

정렬:

```
GET /posts?sort=createdAt,desc
GET /posts?sort=title,asc
```

페이징:

```
GET /posts?page=1&size=10
```

검색:

```
GET /posts?search=Spring
```

조합:

```
GET /posts?status=published&author=홍길동
&sort=createdAt,desc&page=1&size=10
```

2) 특수한 동작(예외적으로 동사 허용)

복잡한 연산이나 RPC 스타일이 더 명확한 경우:

```
POST /posts/1/publish      # 게시글 발행
POST /posts/1/archive      # 게시글 보관
POST /auth/login           # 로그인
POST /auth/logout          # 로그아웃
POST /password/reset       # 비밀번호 재설정
POST /emails/send          # 이메일 발송
```

13. SSR 방식에서 CSR 방식으로 전환하기 – 원칙2) 올바른 HTTP 메서드 사용

HTTP 메서드 종류와 의미:

- **GET**: 리소스 조회 (Read)
- **POST**: 리소스 생성 (Create)
- **PUT**: 리소스 전체 수정 (Update - 전체 교체)
- **PATCH**: 리소스 부분 수정 (Update - 일부 변경)
- **DELETE**: 리소스 삭제 (Delete)
- **HEAD**: GET과 동일하지만 Body 없이 헤더만
- **OPTIONS**: 지원하는 메서드 확인 (CORS preflight)

GET – 리소스 조회

- 서버 상태를 변경하지 않음 (안전함, Safe)
- 같은 요청을 여러 번 해도 결과 동일 (멱등성, Idempotent)
- 캐싱 가능
- 잘못 사용한 예 - GET으로 상태 변경
  - GET /posts/1/delete
  - GET /posts/1/like
  - GET /posts/create?title=...

POST – 리소스 생성

- 새로운 리소스 생성
- 서버 상태 변경 (안전하지 않음)
- 멱등성 없음 (같은 요청 여러 번 → 여러 리소스 생성)

PUT – 리소스 전체 수정

- 리소스 전체를 새로운 것으로 대체
- 멱등성 있음 (같은 요청 여러 번 → 결과 동일)
- 모든 필드를 포함해야 함
- 주의:
  - 모든 필드를 보내야 함
  - 보내지 않는 필드는 null 또는 기본값으로 설정됨
  - 부분 수정이 필요하면 PATCH 사용

PATCH – 리소스 부분 수정

- 리소스의 일부만 수정
- 멱등성 있을 수도, 없을 수도 (구현에 따라)
- PUT과 달리 일부 필드만 보내도 됨

DELETE – 리소스 삭제

- 리소스 삭제
- 멱등성 있음 (같은 요청 여러 번 → 결과 동일)

13. SSR 방식에서 CSR 방식으로 전환하기 – 원칙3) 적절한 HTTP 상태 사용

상태 코드 분류:

- 1xx (정보): 요청을 받았으며 프로세스 계속 진행
- 2xx (성공): 요청을 성공적으로 처리
- 3xx (리다이렉션): 추가 작업 조치 필요
- 4xx (클라이언트 오류): 요청 오류
- 5xx (서버 오류): 서버 처리 오류

2xx – 성공

- 200 OK (성공)
- 201 Created (생성됨)
- 204 No Content (성공, 응답 바디 없음)

3xx – 리다이렉션

- 301 Moved Permanently (영구 이동)
- 304 Not Modified (캐시 사용)

4xx – 클라이언트 오류

- 400 Bad Request (잘못된 요청)
- 401 Unauthorized (인증 필요)
- 403 Forbidden (권한 없음)
- 404 Not Found (리소스 없음)
- 409 Conflict (충돌)
- 422 Unprocessable Entity (처리 불가)

5xx – 서버 오류

- 500 Internal Server Error (서버 오류)
- 503 Service Unavailable (서비스 이용 불가)
- 204 No Content (성공, 응답 바디 없음)

14. 토큰(JWT) 기반 인증 방식으로 전환하기

학습 목표

- 세션 기반 인증과 토큰 기반 인증의 차이를 이해하고, CSR 환경에서 **JWT 도입이 필요한 이유**를 설명할 수 있다.
- **JWT의 구조**와 **서명 기반 무결성 검증 원리**를 설명할 수 있다.
- 로그인 시 **JWT를 생성·발급하는 인증 흐름**을 단계별로 설명할 수 있다.
- **클라이언트에서 JWT를 저장하고 전송**하는 방법을 구현할 수 있다.
- JWT 기반 인증을 위해 Spring Security를 설정할 수 있다.
- JWT 기반 인증에서 **세션 관리 정책의 변화와 CSRF 처리 방식의 차이**를 설명할 수 있다.
- CSRF, 세션 고정, 서버 확장성 관점에서 **JWT 기반 인증의 장점**을 설명할 수 있다.
- JWT 기반 인증의 **한계와 보안 고려사항**을 이해한다.
- 바이브코딩을 통해 기존 세션 기반 인증 구조를 JWT 기반 인증 구조로 점진적으로 리팩토링하고 테스트할 수 있다.

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – 세션 기반 인증

세션 기반 인증 로그인 흐름:

1. 사용자 로그인 → 서버
2. 서버: 인증 성공 → 세션 생성 → 사용자 정보 보관
3. 서버 → 클라이언트: Set-Cookie: JSESSIONID=abc123
4. 클라이언트: 쿠키 저장

사용자 식별:

1. 클라이언트 → 서버: Cookie: JSESSIONID=abc123 (자동 전송)
2. 서버: 세션 저장소에서 abc123 조회
3. 서버: 사용자 정보 확인
4. 서버: 요청 처리 → 응답

저장 위치:

- 서버: 세션 데이터 (예: 메모리/Redis/DB)
- 클라이언트: 세션 ID만 (쿠키)

전송 방식:

- 브라우저가 요청할 때 자동으로 쿠키 포함
- 개발자가 명시적으로 안 해도 됨

상태 관리:

- Stateful (상태 저장)
- 서버가 세션 정보 저장
- 서버 메모리 사용

장점:

- ✅ 즉시 세션 무효화 가능 (로그아웃 시)
- ✅ 서버에서 세션 제어 가능
- ✅ 클라이언트에 민감 정보 노출 안함

단점:

- ❌ 서버 메모리 사용 (확장성 제약)
- ❌ 분산 환경에서 세션 공유 복잡 (Redis 필요)
- ❌ 모바일 앱 대응 어려움 (쿠키 관리)
- ❌ CORS 환경에서 쿠키 전송 제약

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – 토큰(JWT) 기반 인증

JWT 기반 인증 로그인 흐름:

1. 사용자 로그인 → 서버
2. 서버: 인증 성공 → JWT 생성 (서명포함)
3. 서버 → 클라이언트: { "token": "eyJhbGc..." }
4. 클라이언트: 토큰 저장 (localStorage/쿠키)

사용자 식별:

1. 클라이언트 → 서버: Authorization: Bearer eyJhbGc... (요청 헤더)
2. 서버: JWT 서명 검증 (비밀키 사용)
3. 서버: 토큰에서 사용자 정보 추출
4. 서버: 요청 처리 → 응답

저장 위치:

- 서버: 저장 안함 (비밀키만 보관)
- 클라이언트: JWT 전체 (localStorage/쿠키)

전송 방식:

- 요청 헤더로 명시적 전송: Authorization: Bearer {token}
- 개발자가 명시적으로 추가해야 함

상태 관리:

- Stateless (무상태)
- 서버가 토큰 저장 안함
- 메모리 사용 안함
- 토큰 자체에 모든 정보 포함 (self-contained)

장점:

- ✅ 서버 메모리 사용 안함 (확장성 우수)
- ✅ 분산 환경 적합 (어떤 서버든 검증 가능)
- ✅ 모바일 앱 대응 쉬움
- ✅ CORS 제약 없음 (헤더 전송)
- ✅ 마이크로서비스 환경 적합

단점:

- ❌ 토큰 즉시 무효화 어려움 (만료까지 유효)
- ❌ Payload 노출 (Base64 인코딩, 암호화 아님)
- ❌ 토큰 크기 큼 (매 요청마다 전송)
- ❌ 비밀키 유출 시 위험

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – 세션 vs 토큰(JWT)

구분	세션 기반	JWT 기반
상태	Stateful	Stateless
서버 저장	세션 데이터 저장	저장 안함
클라이언트 저장	세션 ID(쿠키)	JWT 전체
전송 방식	쿠키 (자동)	요청 헤더 (명시)
서버 메모리	사용	사용 안함
확장성	세션 공유 필요	독립 검증
모바일 대응	어려움	쉬움
CORS	제약 있음	제약 없음
무효화	즉시 가능	어려움
보안	서버 제어	토큰 탈취 위험

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – CSR과 토큰(JWT) 기반 인증

CSR 환경에서 세션의 문제점:

1. **CORS 제약**
   → 클라이언트/서버의 CORS 설정 필요 (복잡)
2. **모바일 앱 대응 어려움**
   → 모바일 네이티브 앱에서 쿠키 관리 복잡
3. **분산 환경 복잡도**
   → 서버 여러 대인 경우 세션 공유 필요 (복잡도 증가)
4. **API 설계 철학 불일치**
   → REST API는 Stateless 방식 (상태 저장 안함)
   → 세션 방식은 Stateful (서버가 세션 저장)

토큰 기반 인증이 필요한 이유 :

1. **CORS 문제 해결**
   → 헤더 기반 전송 (CORS 설정 간단)
2. **멀티 플랫폼 대응**
   → 하나의 API 서버에서 토큰(JWT) 발급 및 검증
   → 다양한 클라이언트에서 자신의 방식으로 토큰 저장
   → 모두가 동일한 방식(요청 헤더)으로 토큰 전송
3. **Stateless 아키텍처**
   → 서버가 상태 저장 안함, 토큰 자체에 정보 포함
4. **API 우선 설계**
   → Stateless, JSON 기반, 표준화된 인증 방식
   → API 중심 설계에 최적
5. **마이크로서비스에 대응**
   → Auth Service: JWT 발급
   → Post Service: JWT 검증 (독립적)
   → User Service: JWT 검증 (독립적)

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – CSR 환경에서 세션 vs 토큰(JWT)

구분	세션 기반	JWT 기반
CORS	복잡 (credentials)	간단 (요청 헤더)
모바일	어려움	쉬움
확장성	Redis 필요	독립 검증
API 철학	Stateful (불일치)	Stateless (일치)
구현 복잡도	높음	낮음
표준화	낮음	높음

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 특징

JWT (JSON Web Token)

- JSON 기반의 토큰
- 자체 포함적 (self-contained)
- URL-safe (URL에 사용 가능)
- 표준화 (RFC 7519)

JSON 기반 토큰:

JWT는 인증 정보를 JSON 객체 형태로 표현한 토큰이다.

```
{
  "sub": "user123",
  "role": "USER",
  "exp": 1710000000
}
```

- JSON 데이터를 Base64URL로 인코딩하여 문자열 형태로 전송한다.
- 언어/플랫폼 독립적 → Java, JavaScript, Python 모두에서 처리 가능
- 구조가 명확하고 읽기 쉬움
- 파싱이 간단함
- 실무:
  - 식별자, 권한, 만료 시간만 넣는다.
  - 비밀번호나 민감한 정보는 넣지 않는다.

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 특징

JWT (JSON Web Token)

- JSON 기반의 토큰
- 자체 포함적 (self-contained)
- URL-safe (URL에 사용 가능)
- 표준화 (RFC 7519)

자체 포함적 (self-contained):

JWT 하나만으로 인증에 필요한 정보를 모두 담고 있다.

```
{
  "sub": "user123",          ← 사용자 식별
  "role": "USER",            ← 권한
  "exp": 1710000000          ← 만료 정보
}
```

- 서버가 상태를 저장하지 않음 (Stateless)
- 서버 확장(Scale-out)에 유리
- 로드밸런서 뒤에서 서버 자유롭게 확장 가능
- 실무:
  - ✅ 서버 재시작에도 로그인 상태 유지
  - ❌ 토큰이 길어짐
  - ❌ 토큰 폐기가 어려움 → Refresh Token 필요

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 특징

JWT (JSON Web Token)

- JSON 기반의 토큰
- 자체 포함적 (self-contained)
- URL-safe (URL에 사용 가능)
- 표준화 (RFC 7519)

URL-safe (URL에 사용 가능):

JWT는 URL, HTTP Header, 쿠키에 안전하게 포함될 수 있는 문자열이다.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

- Base64URL 인코딩 사용 → +, /, = 같은 URL 위험 문자가 없다.
- Authorization Header에 바로 사용 가능

  예) `Authorization: Bearer <JWT>`
- URL 파라미터나 쿠키에도 사용 가능
- 실무:
  - ✅ 요청 Header로 전달하는 것을 권장
  - ❌ URL 파라미터 전달은 보안상 비권장 (로그 노출)

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 특징

JWT (JSON Web Token)

- JSON 기반의 토큰
- 자체 포함적 (self-contained)
- URL-safe (URL에 사용 가능)
- 표준화 (RFC 7519)

표준화 (RFC 7519):

JWT는 공식 인터넷 표준 문서(RFC 7519)로 정의된 토큰 규격이다.

- 형식, 필드 이름, 서명 방식이 표준으로 명시됨
- 벤터에 종속되지 않음
- 구현체가 다양함 → 서로 호환 가능

  → Java: jjwt, nimbus-jose-jwt

  → Node: jsonwebtoken
- 실무:
  - ✅ 특정 프레임워크에 묶이지 않음
  - ✅ 마이크로서비스 간 인증 공유에 적합

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 구조

JWT = Header.Payload.Signature

Header - 토큰 타입과 서명 알고리즘 (Base64URL 인코딩)

Payload – 실제 전달할 데이터 (Claims; Base64URL 인코딩)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJob25nZ2lsZG9uZyIsInJvbGUiOiJVU0VSIiwia
WF0IjoxNjQwMDAwMDAwLCJleHAiOjE2NDAwMDM2MDB9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6
yJV_adQssw5c
```

Signature

Header와 Payload의 무결성 보장(변조 방지)

"Header.Payload" , "비밀키" → (서명 알고리즘으로 생성)

- 암호화 되지 않음(단순 Base64URL 인코딩)
- 누구나 디코딩 가능

- ✅ 서버만 알고 있음 (application.properties)
- ✅ 256 비트 이상 권장, 환경 변수로 관리, 주기적 교체 권장
- ❌ 코드에 하드코딩 절대 금지, Git에 커밋 금지, 너무 짧으면 위험
- Header나 Payload를 조작 → Signature와 맞지 않음 → 변조 확인!

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 구조

Header:

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

- **alg (Algorithm): 서명 알고리즘**
  - HS256: HMAC SHA-256 (대칭키)
  - RS256: RSA SHA-256 (비대칭키)
  - ES256: ECDSA SHA-256 (비대칭키)
- **typ (Type): 토큰 타입**
  - 항상 “JWT”

- **대칭키 vs 비대칭키**
  - 대칭키: 같은 키로 서명/검증
    → 빠름, 간단
  - 비대칭키: 다른 키로 서명/검증
    → 느림, 복잡, 공개키 배포 가능

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 구조

Payload:

1) Registered Claims (표준으로 정의된 클레임)

```
{
    "iss": "https://auth.example.com",
    "sub": "honggildongg",
    "aud": "https://api.example.com",
    "exp": 1640003600,
    "nbf": 1640000000,
    "iat": 1640000000,
    "jti": "abc123"
}
```

- **iss** (Issuer): 발급자
- **sub** (Subject): 주제, 보통 사용자 ID
- **aud** (Audience): 대상
- **exp** (Expiration Time): 만료 시간
- **nbf** (Not Before): 이 시간 이전에는 무효
- **iat** (Issued At): 발급 시간
- **jti** (JWT ID): 토큰 고유 ID

주요 클레임:

- **sub**
  - 사용자 식별자
  - 보통 username 또는 user ID
  - 필수는 아니지만 거의 항상 사용
- **exp**
  - Unix Timestamp (초 단위)
  - 1640003600 = 2021-12-21 01:00:00 UTC
  - 이 시간 이후 토큰 무효
  - 필수로 사용 권장 (보안)
- **iat**
  - 토큰 발급 시간
  - 토큰 나이 계산에 사용

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – JWT 구조

2) Public Claims (공개 클레임)

```
{
  "https://example.com/claims/role": "admin"
}
```

충돌 방지를 위해 URI 형식 사용

```
{
  "role": "admin"
}
```

실무에서는 간단히 표현

3) Private Claims (비공개 클레임)

```
{
  "username": "홍길동",
  "role": "USER",
  "department": "개발팀",
  "permissions": ["READ", "WRITE"]
}
```

- 당사자간 합의된 클레임

Payload 주의 사항!

- ❌ **절대 넣으면 안되는 것**
  - 비밀번호
  - 주민 등록 번호
  - 신용 카드 번호
  - 기타 민감 정보
- ✅ **넣어도 되는 것**
  - 사용자 ID (username)
  - 역할 (role)
  - 권한 (permissions)
  - 만료 시간 (exp)
- Payload는 Base64URL 인코딩 됨
- 암호 아님 → 누구나 디코딩 가능

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – Spring Security의 JWT

Spring Security에서 JWT 생성 방식:

```
{
 "sub": "honggildongg",
 "iat": 1640000000,
 "exp": 1640003600,
 "iss": "https://myapp.com",
 "username": "홍길동",
 "role": "USER",
 "authorities": ["READ", "WRITE"]
}
```

Registered Claims

Private Claims

- Registered Claims와 Private Claims를 함께 사용
- Public Claims는 거의 사용하지 않음

주로 사용하는 Registered Claims:

- ✅ sub (Subject): 거의 필수
- ✅ iat (Issued At): 권장
- ✅ exp (Expiration): 필수
- △ iss (Issuer): 선택적 (멀티 서비스 환경에서 유용)
- △ aud (Audience): 선택적
- ❌ nbf (Not Before): 거의 안 씀
- ❌ jti (JWT ID): 거의 안 씀

일반적인 Registered Claims:

- username: 사용자 이름 (표시용)
- role: 역할 (USER, ADMIN 등)
- authorities: 권한 목록
- email: 이메일 (선택)
- department: 부서 (선택)

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – 로그인 흐름

Spring Security + JWT 로그인 흐름:

1. **클라이언트 → 서버:** POST /api/auth/login
2. **SecurityFilterChain:**
   - /api/login → permitAll() 설정
   - 인증 없이 Controller로 통과
3. **UserController.login():**
   - AuthenticationManager.authenticate() 호출
4. **Spring Security 인증:**
   - UserDetailsService.loadUserByUsername()
   - PasswordEncoder.matches()
   - Authentication 객체 생성
5. **JwtTokenProvider.createToken():**
   - Authentication → JWT 생성
   - Header.Payload.Signature
6. **응답:** {token: "..."}
7. **클라이언트:** localStorage에 토큰 저장

핵심 컴포넌트:

- **SecurityConfig →** Spring Security 설정
- **AuthenticationManager**
  → 사용자 인증을 담당하는 핵심 인터페이스
  → 로그인 시 사용자 검증에 사용
- **UserDetailsService →** 사용자 조회
- **PasswordEncoder →** 비밀번호 검증
- **UserController →** 로그인 엔드포인트
- **JwtProvider →** JWT 생성

14. 토큰(JWT) 기반 인증 방식으로 전환하기 – SpringSecurity 핵심 설정

```
// SecurityConfig.java
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
    return http
        .csrf(csrf -> csrf.disable()) // CSRF 비활성화
        .sessionManagement(session ->
            session.sessionCreationPolicy(
                SessionCreationPolicy.STATELESS)) // 세션 사용 안함
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/auth/**")
                .permitAll() // 로그인 허용
            .anyRequest().authenticated()) // 나머지 인증 필요
        .formLogin(form -> form.disable()) // 폼 로그인 비활성화
        .build();
}
```

Spring Security 필터 통과:

1. **SecurityContextPersistenceFilter**
   - SecurityContext 생성/복원
   - 현재는 비어있음 (인증 전)
2. **LogoutFilter**
   - 로그아웃 요청인지 확인
   - /api/auth/login이므로 통과
3. **UsernamePasswordAuthenticationFilter**
   - 기본 폼 로그인 필터
   - 비활성화 상태 (우리는 사용 안함)
4. **AuthorizationFilter**
   - URL 접근 권한 확인
   - /api/login → permitAll() 설정
   - ✅ 인증 없이 통과!