AOP, 관점 지향 프로그래밍

**AOP**

AOP (Aspect-Oriented Programming)는 소프트웨어 개발에서 관심사의 분리를 위해 사용되는 프로그래밍 패러다임 중 하나입니다. 관심사의 분리란, 프로그램의 기능 구현과 이를 지원하는 부가 기능의 구현을 분리하는 것을 의미합니다.

**AOP는 핵심 기능 구현 코드에 영향을 주지 않으면서, 부가 기능을 추가할 수 있도록 해줍니다.** 이를 위해, AOP는 Cross-cutting concern(공통 관심사)라고 불리는 부가 기능들을 분리하여 구현하고, 이를 필요한 시점에 핵심 기능 구현 코드와 결합시킵니다.

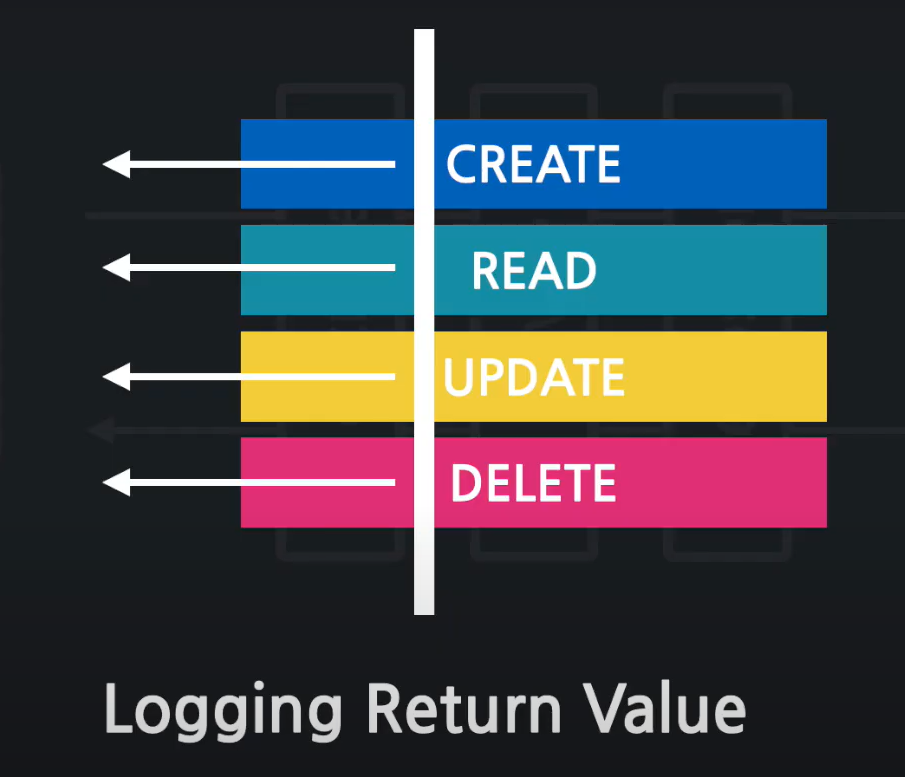
AOP의 구현 방법 중 하나는, 프로그램 코드에 직접 코드를 삽입하는 것이 아니라, 특정 Pointcut(적용 대상)과 Advice(부가 기능)을 정의하고, 이를 Aspect(관심 모듈)라는 단위로 묶어서 사용하는 것입니다. Aspect는 공통적으로 사용되는 부가 기능들을 모아놓은 것으로, 애플리케이션 전체에서 반복적으로 사용되는 기능들을 Aspect로 정의하고, 필요한 곳에서 적용할 수 있습니다.

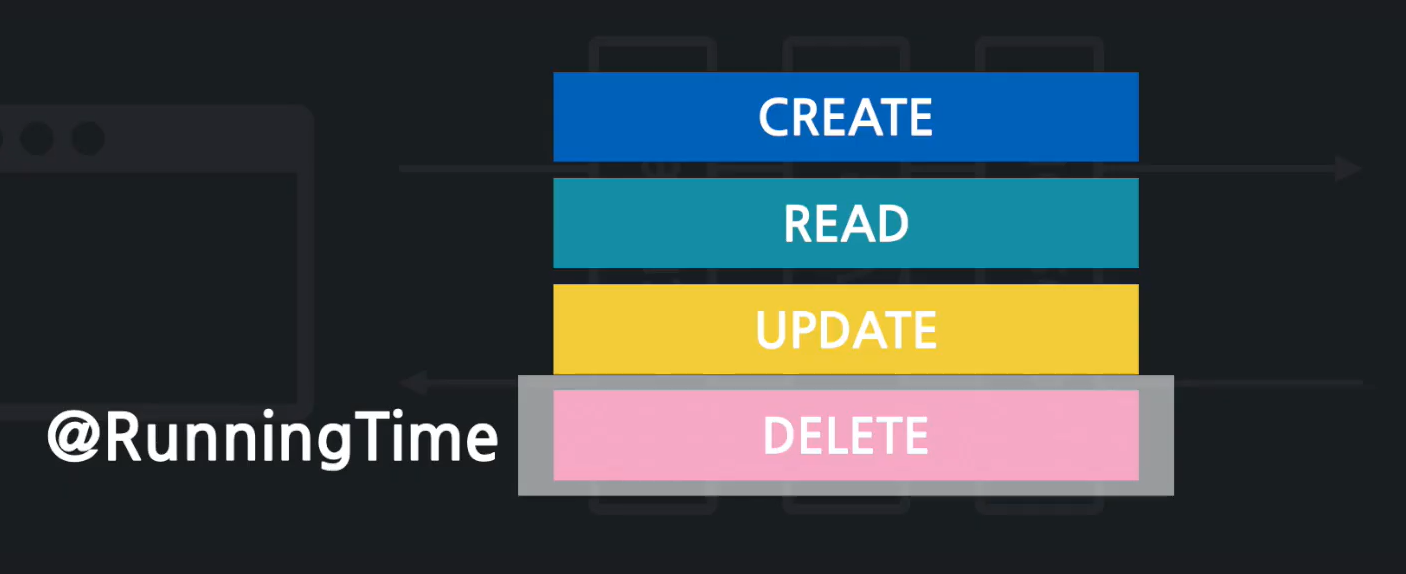
AOP는 **로깅, 보안, 트랜잭션 관리** 등 다양한 분야에서 활용되며, 코드의 가독성과 유지보수성을 높여줍니다. 또한, AOP를 사용하면, 중복 코드를 줄일 수 있고, 개발 생산성을 높일 수 있습니다.

대표적인 예로 @Transactional있다.

**AOP 주요 어노테이션**



댓글 서비스의 입출력값 확인을 위한 로깅 AOP작성,  


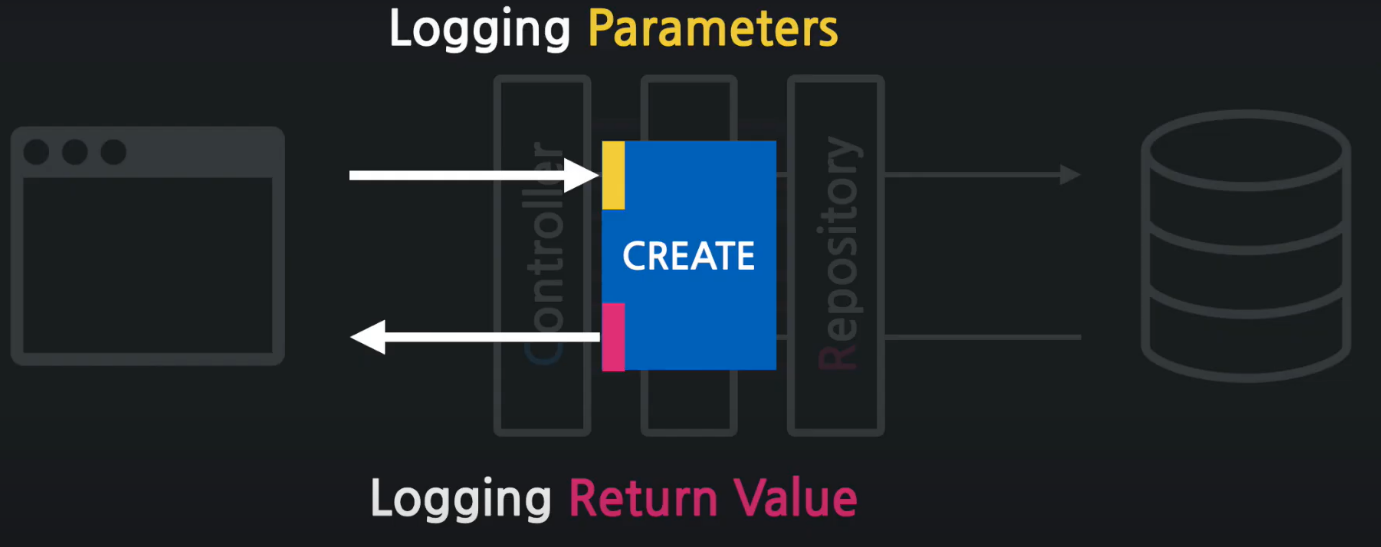
특정 메소드의 수행시간을 측정하는 AOP를 만들 예정이다.  


**테스트용 DB 설정**

application.properties에서  
postgreSQL 설정 주석 처리하고 H2 DB를 사용하자.

**댓글 생성 입출력 로깅**

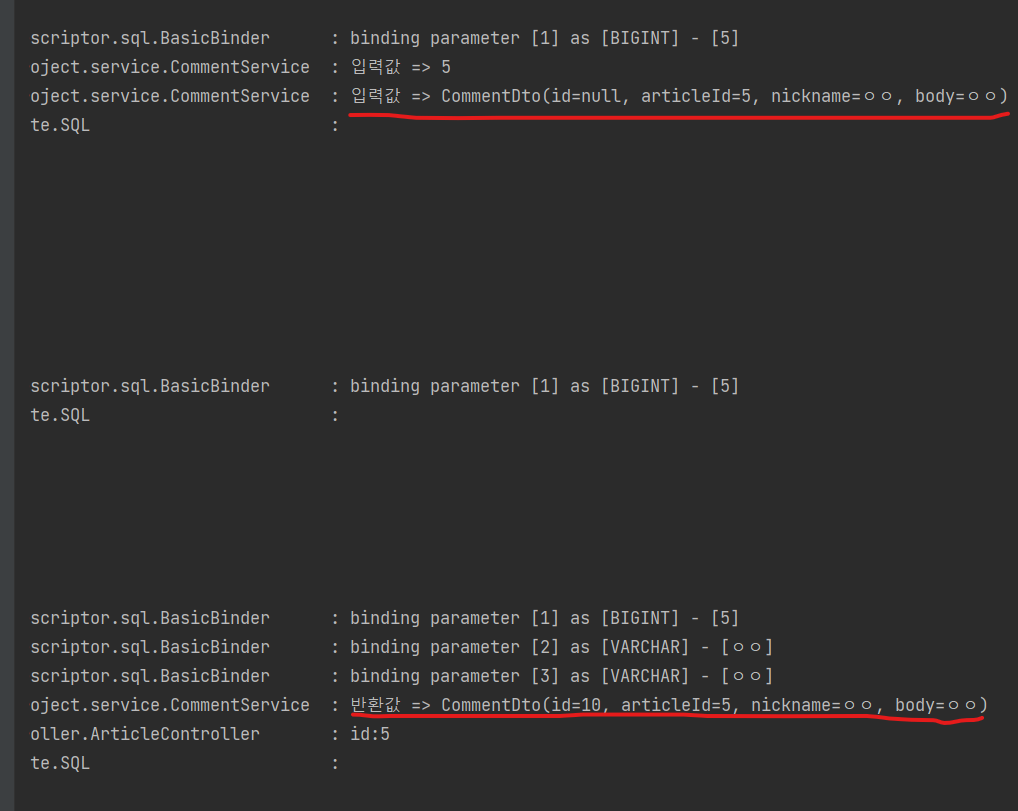
**CommentService**

  
create 메소드에 log를 찍어보자.  
입력값 : 파라미터로 받는 articleId와 dto  
출력값 : CommentDto

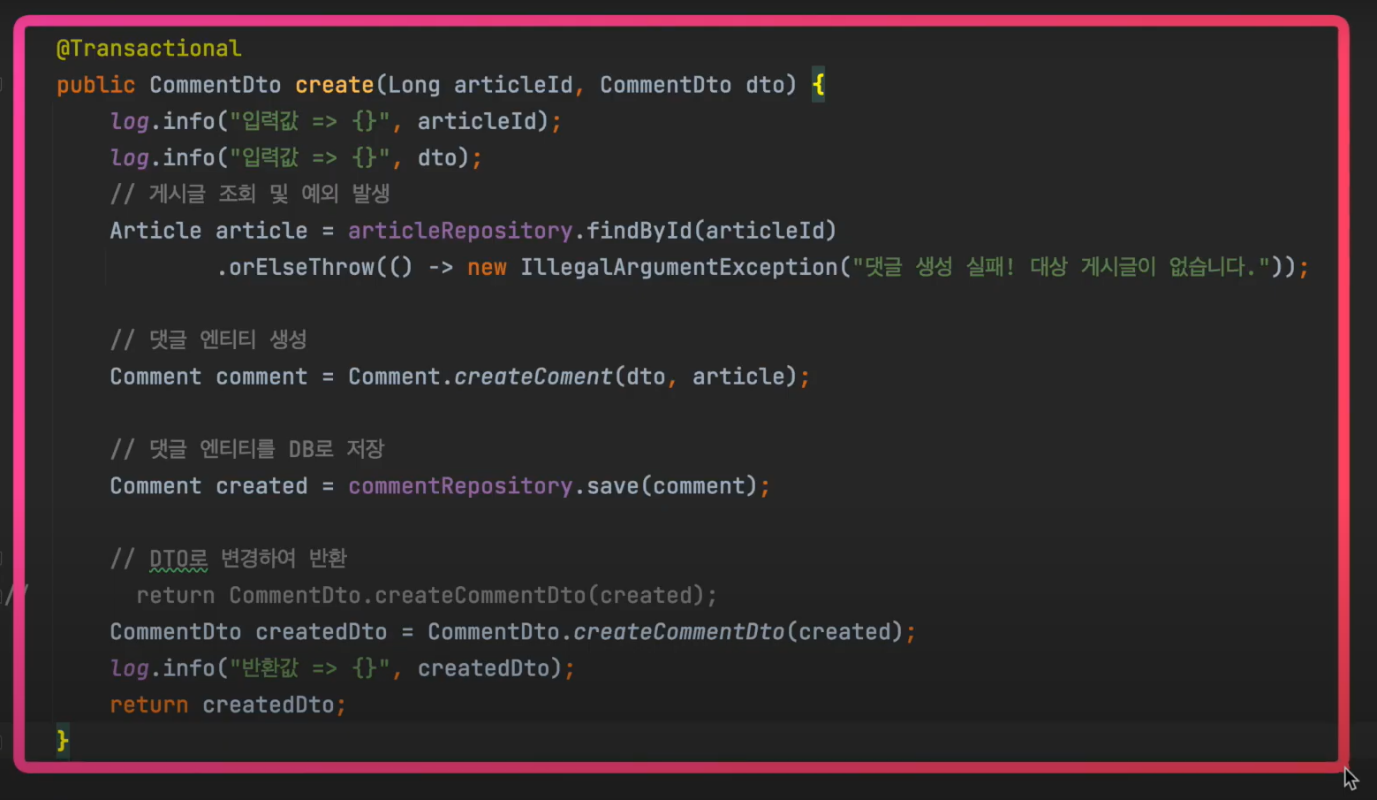
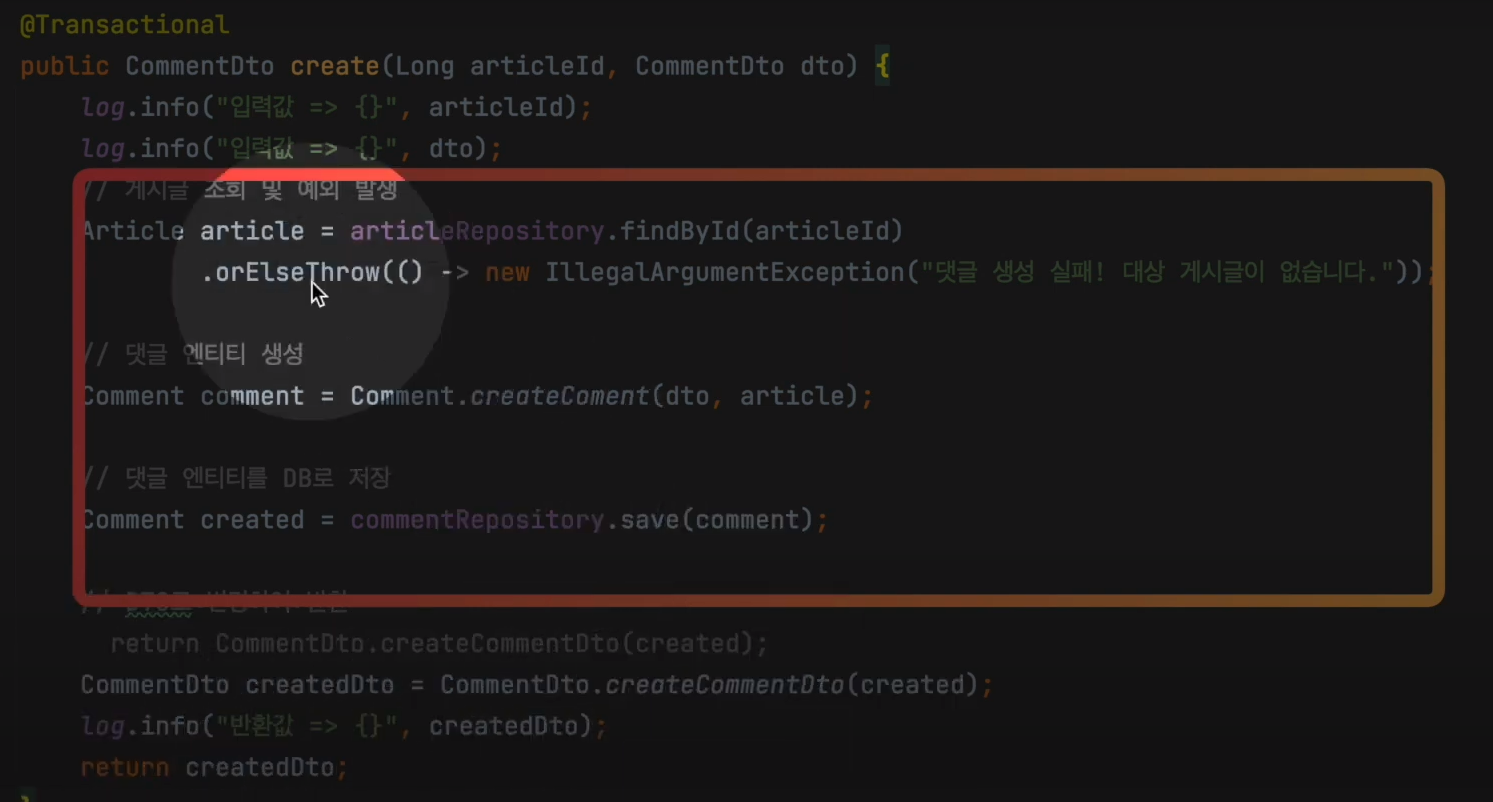
log.info("입력값 => {}",articleId);

log.info("입력값 => {}", dto);

log.info("반환값 => {}", createDto);



**기존코드의 문제점**

  
create 메소드의 핵심기능은 안쪽에서 실행되는 코드이다.  
log.info()는 핵심기능이 아니다.  
이렇게 핵심기능과 부가적인 기능 코드가 섞여있기 때문에 코드가 난해할 수 있다는 단점이 있다.  
  
또한 로깅 코드가 create 메소드말고 다른 메소드에서도 사용될 수 있기 때문에 코드의 중복이 발생한다.

이러한 문제를 AOP를 사용해 해결할 수 있다.

@Transactional

public CommentDto create(Long articleId, CommentDto dto) {

log.info("입력값 => {}", articleId);

log.info("입력값 => {}", dto);

// 게시글 조회 및 예외 발생

// .orElseThrow(() -> new IllegalArgumentException()) article이 없다면 예외발생시켜서 다음 코드가 실행되지 않는다.

Article article = articleRepository.findById(articleId)

.orElseThrow(() -> new IllegalArgumentException("댓글 생성 실패!! 대상 게시글이 없습니다."));

// 댓글 엔티티 생성

Comment comment = Comment.createComment(dto, article);

// 댓글 엔티티를 DB로 저장

Comment created = commentRepository.save(comment);

// DTO로 변환하여 반환

// return CommentDto.createCommentDto(created);

CommentDto createDto = CommentDto.createCommentDto(created);

log.info("반환값 => {}", createDto);

return createDto;

}

**입력값 로깅 AOP**

AOP 패키지 생성

**DebuggingAspect**

package com.example.firstproject.aop;

@Aspect // AOP 클래스 선언: 부가 기능을 주입하는 클래스

@Component // Ioc 컨테이너가 해당 객체를 생성 및 관리

@Slf4j

public class DebuggingAspect {

// 1.대상 메소드 선택: CommentService#create()

// 어느 대상을 타겟으로 해서 부가기능을 주입할 것인지.

@Pointcut("execution(\* com.example.firstproject.service.CommentService.create(..))")

// \* : public, return 타입 지정하는 곳

// create(..) 파라미터가 무엇이든 상관없다.

private void cut() {}

// 2.실행 시점 설정: cut()의 대상이(create 메소드) 수행되기 이전에 아래 부가기능이 수행된다.

@Before("cut()")

public void loggingArgs(JoinPoint joinPoint) { // JoinPoint: cut()의 대상 메소드

// 입력값 가져오기

Object[] args = joinPoint.getArgs();

// 클래스명

String className = joinPoint.getTarget().getClass().getSimpleName();

// 메소드명

String methodName = joinPoint.getSignature().getName();

// 입력값 로깅하기(아래 예시처럼)

// CommentService#create()의 입력값 => 5

// CommentService#create()의 입력값 => CommentDto(id=null,...)

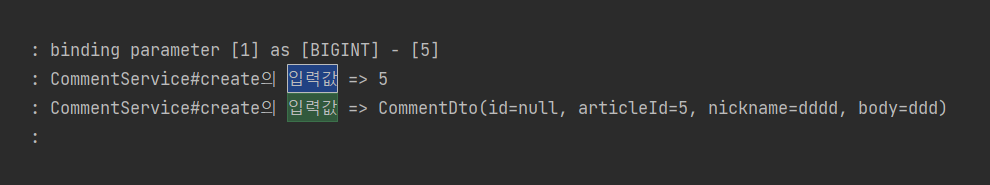
for(Object obj : args) { // forEach

log.info("{}#{}의 입력값 => {}", className,methodName,obj);

}

}

}

로그가 잘 나온당 😊  


**반환값 로깅 AOP**

// 실행 시점 설정 : cut()에 지정된 대상 호출 성공 후!

@AfterReturning(value = "cut()", returning = "returnObj")

public void loggingReturnValue(JoinPoint joinPoint, Object returnObj) { // 메소드의 리턴값

// 클래스명

String className = joinPoint.getTarget().getClass().getSimpleName();

// 메소드명

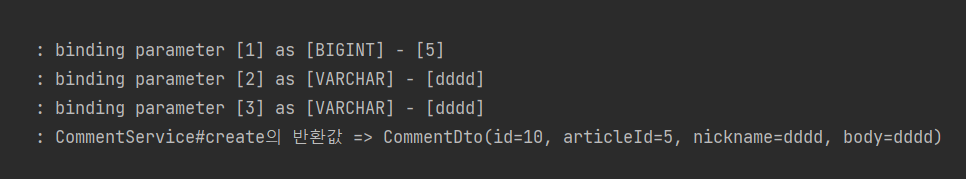
String methodName = joinPoint.getSignature().getName();

// 반환값 로깅

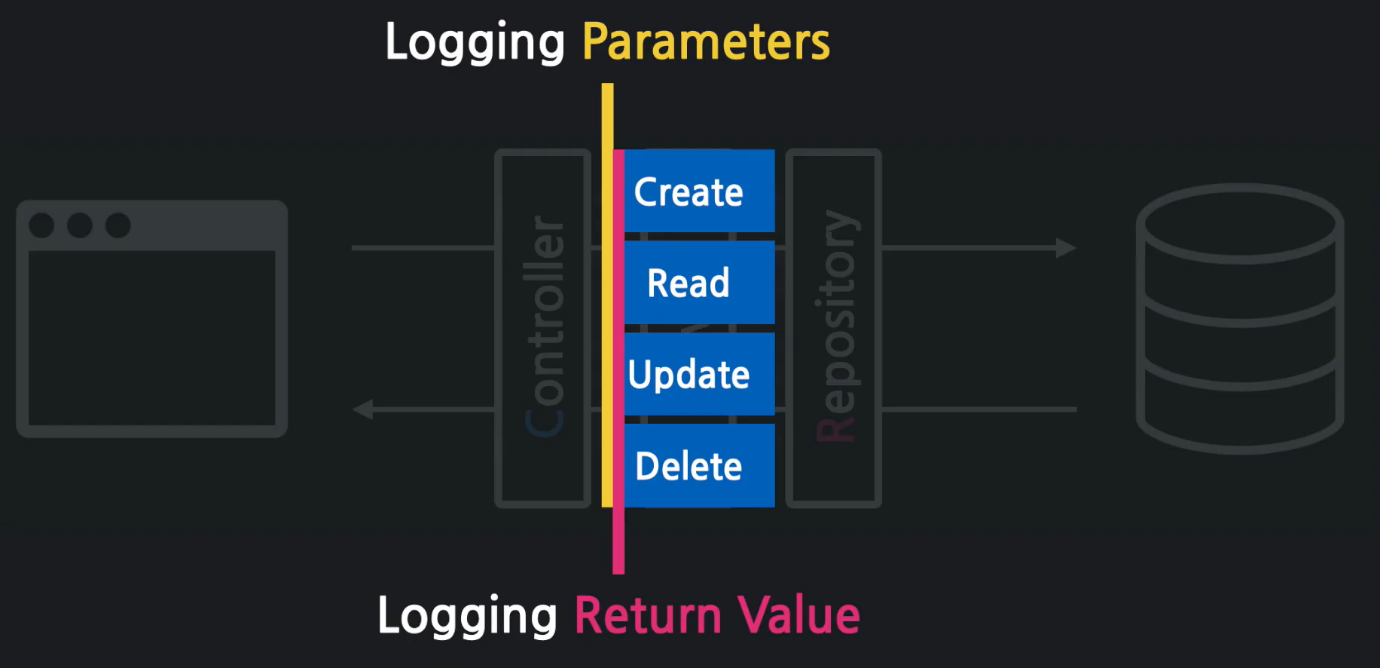
// CommentService#create()의 반환값 => CommentDto(id=10,...)

log.info("{}#{}의 반환값 => {}", className,methodName,returnObj);

}



**AOP 대상 범위 변경**

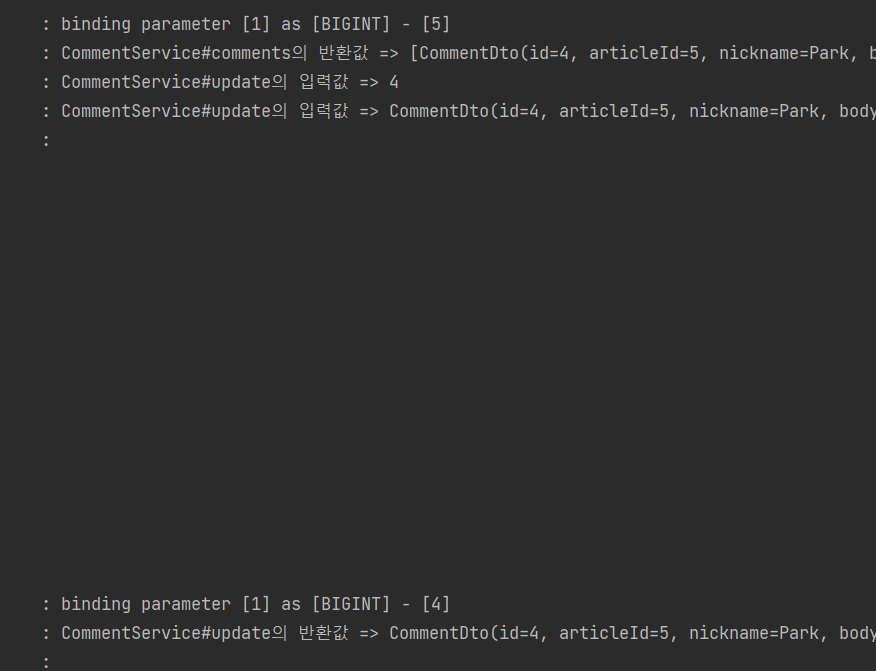


현재는 create 메소드만 설정되어 있기때문에  
생성할때만 log가 찍힌다.  
CommentService의 모든 메소드로 대상 범위를 변경해보자.

@Pointcut("execution(\* com.example.firstproject.service.CommentService.create(..))")

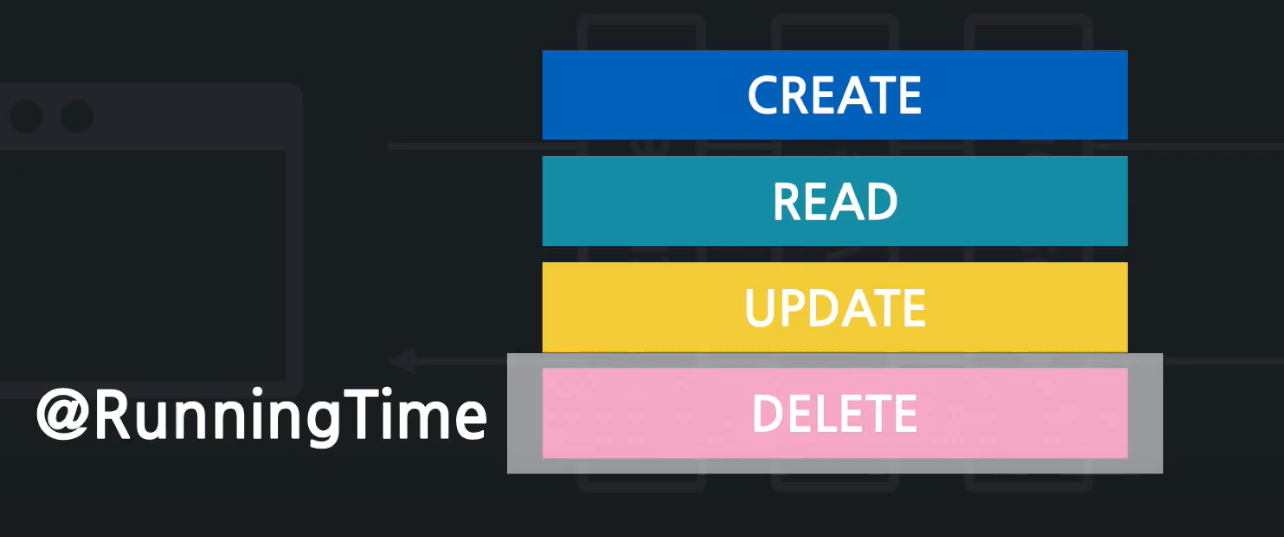
create -> \* 로 변경

@Pointcut("execution(\* com.example.firstproject.service.CommentService.\*(..))")

생성,수정도 다 log가 잘 찍힌다.  


CommentService에 update 메소드에는  
log를 찍는 코드가 없다.  
이렇게 AOP를 이용하면 부가기능을 핵심코드에 찔러 넣을 수가 있다.

**수행 시간 측정 AOP**

특정 메소드의 수행시간 측정하기.  


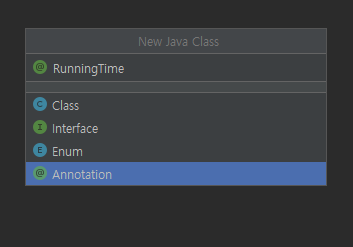
**@Target과 @Retention**

@Target과 @Retention은 Java에서 사용되는 어노테이션으로,  
컴파일러에서 다른 어노테이션을 처리하는 방법에 대한 추가 정보를 제공하는 데 사용됩니다.

* **@Target**은 어노테이션이 적용될 수 있는 **대상 요소 유형을 지정**하는 데 사용됩니다.  
  예를 들어, 어노테이션이 **클래스, 메서드 또는 필드**에 적용될 수 있는지 여부를 지정할 수 있습니다.
* **@Retention**은 어노테이션의 **보존 정책을 지정**하는 데 사용됩니다.  
  이는 어노테이션이 컴파일된 클래스 파일에서 **유지되는 기간을 제어**합니다.  
  @Retention의 **옵션으로는 SOURCE, CLASS, RUNTIME**이 있으며,  
  **RUNTIME을 지정하면 어노테이션이 실행 시에도 유지**됩니다.

annotation 패키지 생성.  
RunningTime 어노테이션 생성.

**RunningTime**



package com.example.firstproject.annotation;

@Target({ElementType.TYPE, ElementType.METHOD}) // 어노테이션 적용 대상(여러개 지정할때는 {중괄호}로 묶기)

@Retention(RetentionPolicy.RUNTIME) // 어노테이션 유지 기간

public @interface RunningTime { }

**PerformanceAspect**

RunningTime 어노테이션 사용할 AOP 클래스 생성.

package com.example.firstproject.aop;

@Aspect // AOP 클래스 선언

@Component // Ioc컨테이너에 등록

@Slf4j // 로깅

public class PerformanceAspect {

// 특정 어노테이션을 대상 지정

@Pointcut("@annotation(com.example.firstproject.annotation.RunningTime)")

private void enableRunningTime() {}

// 기본 패키지의 모든 메소드

@Pointcut("execution(\* com.example.firstproject..\*.\*(..))") // firstproject하위의 모든 메소드(..\*.\*), 필드 개수 상관없음(..)

private void cut() {}

// enableRunningTime,cut을 함께 사용하는 메소드

// 실행 시점 설정: 두 조건을 모두 만족하는 대상을 전후로 부가 기능을 삽입

@Around("cut() && enableRunningTime()")

public void loggingRunningTime(ProceedingJoinPoint joinPoint) throws Throwable { // Around JoinPoint는 ProceedingJoinPoint를 사용(대상을 실핼까지 진행할 수 있음)

// 메소드 수행 전, 시간 측정 시작

StopWatch stopWatch = new StopWatch(); // 스프링에서 제공하는 시간 측성 객체

stopWatch.start();

// 메소드를 수행

Object returningObj = joinPoint.proceed(); // 타겟팅된 대상을 수행, 예외는 throws 처리

// 메소드 수행 후, 측정 종료 및 로깅

stopWatch.stop();

// 메소드명

String methodName = joinPoint.getSignature().getName();

log.info("{}의 총 수행 시간 => {} sec", methodName, stopWatch.getTotalTimeSeconds());

}

}

이제 @RunningTime을 붙여서 테스트해보자.  
CommentApiController에 delete() @RunningTime추가.

@RunningTime

@DeleteMapping("/api/comments/{id}")

public ResponseEntity<CommentDto> delete(@PathVariable Long id) {

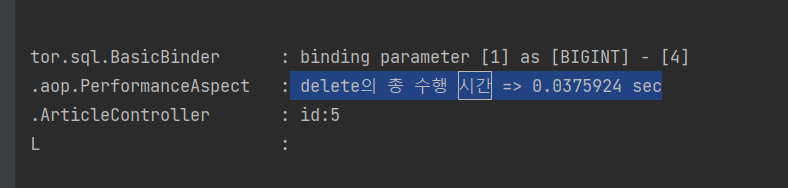
// 서비스에게 위임

CommentDto deletedDto = commentService.delete(id);

// 결과 응답

return ResponseEntity.status(HttpStatus.OK).body(deletedDto);

}

삭제하면 메소드의 수행 시간을 확인할 수 있다.  


**전체코드**

**DebuggingAspect**

package com.example.firstproject.aop;

@Aspect // AOP 클래스 선언: 부가 기능을 주입하는 클래스

@Component // Ioc 컨테이너가 해당 객체를 생성 및 관리

@Slf4j

public class DebuggingAspect {

// 1.대상 메소드 선택: CommentService#create()

// 어느 대상을 타겟으로 해서 부가기능을 주입할 것인지.

@Pointcut("execution(\* com.example.firstproject.service.CommentService.\*(..))")

// \* : public, return 타입 지정하는 곳

// create(..) 파라미터가 무엇이든 상관없다.

private void cut() {}

// 2.실행 시점 설정: cut()의 대상이(create 메소드) 수행되기 이전에 아래 부가기능이 수행된다.

@Before("cut()")

public void loggingArgs(JoinPoint joinPoint) { // JoinPoint: cut()의 대상 메소드

// 입력값 가져오기

Object[] args = joinPoint.getArgs();

// 클래스명

String className = joinPoint.getTarget().getClass().getSimpleName();

// 메소드명

String methodName = joinPoint.getSignature().getName();

// 입력값 로깅하기(아래 예시처럼)

// CommentService#create()의 입력값 => 5

// CommentService#create()의 입력값 => CommentDto(id=null,...)

for(Object obj : args) { // forEach

log.info("{}#{}의 입력값 => {}", className,methodName,obj);

}

}

// 실행 시점 설정 : cut()에 지정된 대상 호출 성공 후!

@AfterReturning(value = "cut()", returning = "returnObj")

public void loggingReturnValue(JoinPoint joinPoint, Object returnObj) { // 메소드의 리턴값

// returning = "returnObj"와 Object returnObj 변수명은 같아야한다.

// 클래스명

String className = joinPoint.getTarget().getClass().getSimpleName();

// 메소드명

String methodName = joinPoint.getSignature().getName();

// 반환값 로깅

// CommentService#create()의 반환값 => CommentDto(id=10,...)

log.info("{}#{}의 반환값 => {}", className,methodName,returnObj);

}

}

**PerformanceAspect**

package com.example.firstproject.aop;

@Aspect // AOP 클래스 선언

@Component // Ioc컨테이너에 등록

@Slf4j // 로깅

public class PerformanceAspect {

// 특정 어노테이션을 대상 지정

@Pointcut("@annotation(com.example.firstproject.annotation.RunningTime)")

private void enableRunningTime() {}

// 기본 패키지의 모든 메소드

@Pointcut("execution(\* com.example.firstproject..\*.\*(..))") // firstproject하위의 모든 메소드(..\*.\*), 필드 개수 상관없음(..)

private void cut() {}

// enableRunningTime,cut을 함께 사용하는 메소드

// 실행 시점 설정: 두 조건을 모두 만족하는 대상을 전후로 부가 기능을 삽입

@Around("cut() && enableRunningTime()")

public void loggingRunningTime(ProceedingJoinPoint joinPoint) throws Throwable { // Around JoinPoint는 ProceedingJoinPoint를 사용(대상을 실핼까지 진행할 수 있음)

// 메소드 수행 전, 시간 측정 시작

StopWatch stopWatch = new StopWatch(); // 스프링에서 제공하는 시간 측성 객체

stopWatch.start();

// 메소드를 수행

Object returningObj = joinPoint.proceed(); // 타겟팅된 대상을 수행, 예외는 throws 처리

// 메소드 수행 후, 측정 종료 및 로깅

stopWatch.stop();

// 메소드명

String methodName = joinPoint.getSignature().getName();

log.info("{}의 총 수행 시간 => {} sec", methodName, stopWatch.getTotalTimeSeconds());

}

}

``