

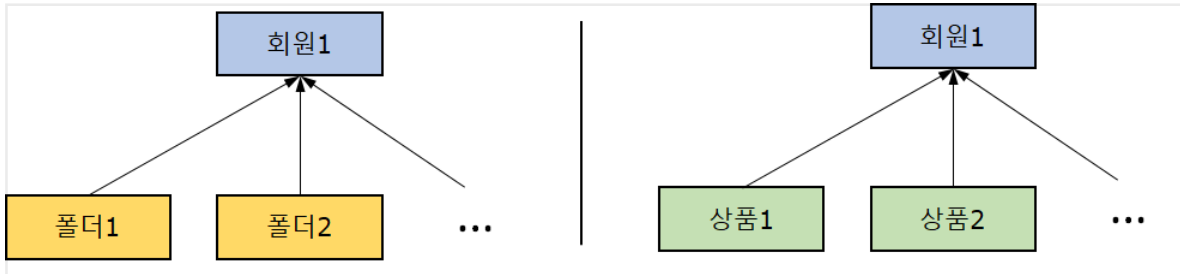
6월 4일

폴더테이블 설계

폴더 테이블에 필요한 정보

1. 폴더명: 회원이 등록한 폴더 이름 저장
2. 회원ID: 폴더를 등록한 회원의 ID 저장

A 회원이 생성한 폴더는 A 회원에게만 보여야함



기존 관계 설정 방법의 문제점

기존 관계설정 방법으로 구현한 Folder 객체

```
@Entity
public class Folder {
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    private Long id;














    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private Long userId;
}
```

- 폴더가 가지고 있는 userId가 "회원의 Id값" 이라는건 개발한 사람만 알 수있음
- 객체) 회원과 폴더의 관계

User	Folder
id	id
username	name
password	userId
email	createdAt
role	modifiedAt
oauth	

- DB) 회원과 폴더의 관계

USER	FOLDER
 ID bigint	 ID bigint
 EMAIL varchar(255)	 NAME varchar(255)
 OAUTH varchar(255)	 USER_ID bigint
 PASSWORD varchar(255)	 CREATED_AT timestamp
 ROLE varchar(255)	 MODIFIED_AT timestamp
 USERNAME varchar(255)	
 CREATED_AT timestamp	
 MODIFIED_AT timestamp	

- 회원 → 폴더 조회

1. user1 이 저장한 모든 폴더 조회

```
// 1. 로그인한 회원 (user1) 의 id 를 조회
Long userId = user1.getId();
// 2. userId 로 저장된 모든 folder 조회
List<Folder> folders = folderRepository.findAllByUserId(userId);
```

2. 이를 위해, Folder Repository 에 userId 를 기준으로 조회하는 함수 생성 필요

```
public interface FolderRepository extends JpaRepository<Folder, Long> {
    List<Folder> findAllByUserId(Long userId);
}
```

- 폴더 → 회원 조회

1. folder1 의 userId 로 회원 조회

```
// 1. folder1 의 userId 를 조회
Long userId = folder1.getUserId();
// 2. userId 로 저장된 회원 조회
User user = userRepository.findById(userId);
```

JPA 연관관계를 이용한 폴더 테이블 설계

회원 Entity 관점

- 회원 1명이 여러 개의 폴더를 가질 수 있음

- "@OneToMany" 로 설정

Java ▾

 복사

```
public class User {  
    @OneToMany  
    private List<Folder> folders;  
}
```

- 회원이 가진 폴더들을 조회

```
List<Folder> folders = user.getFolders();
```

폴더 Entity 관점

- 폴더 여러 개를 회원 1명이 가질 수 있음

- "@ManyToOne"

Java ▾

 복사

```
public class Folder{  
    @ManyToOne  
    private User user;  
}
```

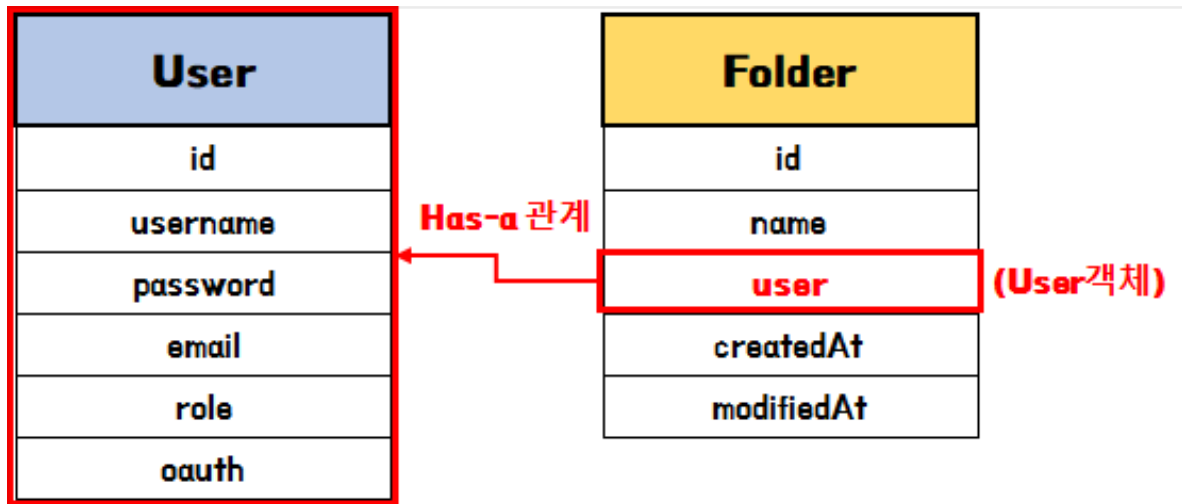
- 폴더를 소유한 회원을 조회

```
folder.getUser();
```

객체의 관계를 맺어주면, DB 의 관계 설정 맺어줌

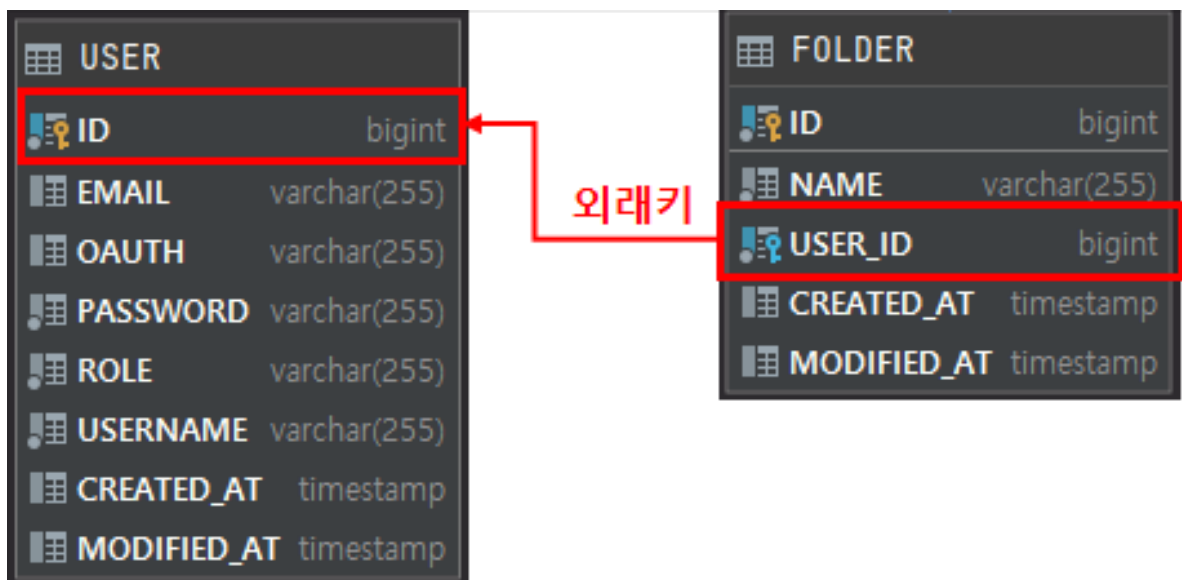
- 객체) 회원과 폴더의 관계

폴더를 소유한 회원 id 가 아닌 객체를 저장



Tip. Has-a 관계 : 객체가 객체를 가지고 있는 관계

- DB) 회원과 폴더의 관계
외래키를 통한 관계 형성



Tip. 외래키: Folder 테이블에 User_Id가 User 테이블과 관계가 있다는 걸 표시해줌

JPA 연관관계 Column 설정 방법

```
@ManyToOne
@JoinColumn(name = "USER_ID", nullable = false)
private User user;
```

- @JoinColumn 내 속성값 설정
 - name: 외래키 명
 - nullable: 외래키 null 허용여부
 - false (default)
 - ex) 폴더가 회원에 의해서만 만들어 질때. User 값이 필수
 - true
 - ex) 공용폴더의 경우 회원 가입 없이 내용 생성이 가능하니, 폴더의 user 객체를 null

로 설정

폴더 생성 기능 구현

- 요구사항
 1. 회원별 폴더를 추가
 2. 폴더를 추가할 때 1~N개를 한번에 추가 가능

✕

폴더 추가하기

폴더를 추가해서 관심상품을 관리해보세요!

신발

휴대폰

반찬거리

+

추가하기

3. 회원별 저장한 폴더들이 조회 되어야함

전체

반찬

음료수

유기농

군것질

과자

+

위 요구사항에 따른 API 설계

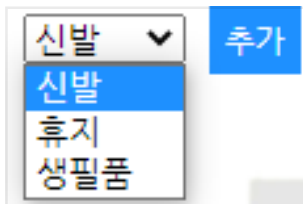
폴더 생성 및 조회

Aa 설명	::: API	≡ 입력	≡ 출력
회원의 폴더 생성	POST /api/folders	folderNames (JSON) : 생성할 폴더명들	folders (JSON) : 생성된 폴더의 정보들
회원의 폴더 조회	GET /		index.html model 추가 → folders

폴더 생성 기능 구현

요구사항

1. 관심상품에 폴더를 0~N개 설정 가능
 2. 관심상품이 등록되는 시점에는 어느 폴도에도 저장되지 않는다
 3. 관심상품 별로 기생성했던 폴더를 선택해 추가 할 수 있다.
- 폴더 전체 조회 및 선택



- 폴더와 상품의 연관 관계
- 상품 1개에 여러개 폴더 저장 가능
 - 폴더 1개에 여러개 상품 저장 가능
- 즉, 상품 : 폴더 = N : N

요구사항에 따른 API 설계

관심상품에 폴더 추가

Aa 설명	::: API	≡ 입력	≡ 출력
폴더 전체 조회	GET /api/folders		List<Folder>
폴더 추가	POST /api/products/{productId}/folder	{productId}: 관심상품 ID [Form 형태] folderId: 추가할 폴더 ID	폴더가 추가된 관심상품 ID
상품 조회 시 폴더정보 추가	GET /api/products		product 정보에 folderList 추가

API 사용시간 측정 방법

API 사용시간

Controller 에 요청이 들어온시간에서 응답이 나간 시간

(Controller 에 응답이 나간시간 - Controller 에 요청이 들어온 시간)

Ex) Controller 에 요청이 들어온 시간 : 9시 10분 30초

Controller 에 응답이 나간 시간 : 9시 10분 33초

사용시간 : 3초

예제코드

```
class Scratch {  
    public static void main(String[] args) {  
        // 측정 시작 시간  
        long startTime = System.currentTimeMillis();  
  
        // 함수 수행  
        long output = sumFromOneTo( input: 2_000_000_000);  
  
        // 측정 종료 시간  
        long endTime = System.currentTimeMillis();  
  
        long runTime = endTime - startTime;  
        System.out.println("소요시간: " + runTime);  
    }  
  
    private static long sumFromOneTo(long input) {  
        long output = 0;  
  
        for (int i = 1; i < input; ++i) {  
            output = output + i;  
        }  
  
        return output;  
    }  
}
```

회원별 총 API 사용시간 저장

API 사용시간을 저장할 테이블 설계

ApiUseTime 테이블 ...			
Aa 컬럼명	≡ 컬럼타입	≡ 중복허용	≡ 설명
<u>id</u>	Long	X	테이블 ID (PK)
<u>user_id</u>	Long	X	회원 ID (FK)
<u>totalTime</u>	Long	O	API 총 사용시간

회원별 API 총 사용시간 조회 (관리자용)

회원별 API 총 사용시간 조회 API ...			
Aa Name	Method	URL	설명
API 총 사용시간 조회	GET	/api/use/time	회원별 API 총 사용시간 조회 (관리자용)

AOP

부가기능 모듈화의 필요성

- 핵심기능: 각 API 별 수행해야 할 비즈니스 로직 ex) 상품 키워드 검색, 관심상품 등

핵심기능1	핵심기능2	핵심기능3	...
-------	-------	-------	-----

- 부가기능: 핵심 기능을 보조하는 기능 ex) 회원 패턴 분석을 위한 로그 기록, API 수행시간 저장 등

부가기능1	부가기능1	부가기능1	...
핵심기능1	핵심기능2	핵심기능3	
부가기능1	부가기능1	부가기능1	

예제 코드(노란색 색칠된 부분이 부가기능)

```

// 측정 시작 시간
long startTime = System.currentTimeMillis();

try {
    // 핵심기능 수행
    // 로그인 되어 있는 회원 테이블의 ID
    Long userId = userDetails.getUser().getId();

    Product product = productService.createProduct(requestDto, userId);

    // 응답 보내기
    return product;
} finally {
    // 측정 종료 시간
    long endTime = System.currentTimeMillis();
    // 수행시간 = 종료 시간 - 시작 시간
    long runTime = endTime - startTime;
    // 수행시간을 DB 에 기록
    ...
}

```

-문제점

모든 '핵심기능'의 Controller에 '부가기능' 코드를 추가한다면?

ex) '핵심기능'이 100개 :100개의 '핵심기능' 모두에 동일한 내용의 코드 추가 필요

'핵심기능' 이 나중에 추가된다면

항상 '부가기능' 추가 가능성 생각해놔야함

'부가기능' 추가를 깜박한다면 ex) 일부 API 수행시간이 추가되지 않음 -> 신뢰성

이슈 발생

'핵심기능' 수정 시

같은 함수 내에 '핵심기능', '부가기능'이 섞여있어 '핵심기능'만 이해 하려해도

'부가기능' 까지 이해해야함

'부가기능' 변경이 필요하다면?

'핵심기능' 갯수만큼 '부가기능' 수정필요, 삭제도 마찬가지로

부가기능 모듈화

AOP(Aspect Oriented Programming)를 통해 부가기능 모듈화 가능

- '부가기능' 은 '핵심기능'이랑 관점(Aspect), 관심이 다르다

- 따라서 '부가기능'만 분리해서 부가기능 중심으로 설계, 구현 가능

핵심기능들

핵심기능1

핵심기능2

핵심기능3

...

부가기능

부가기능1

스프링이 제공하는 AOP

핵심기능

핵심기능1

핵심기능2

핵심기능3

...



부가기능

부가기능1
(어드바이스)

부가기능1 적용위치
(포인트컷)



부가기능1

핵심기능1

부가기능1

부가기능1

핵심기능2

부가기능1

부가기능1

핵심기능3

부가기능1

...

부가기능은 필요한 부분에만 선택적 적용가능

어드바이스 : 부가기능

포인트컷 : 부가기능 적용위치

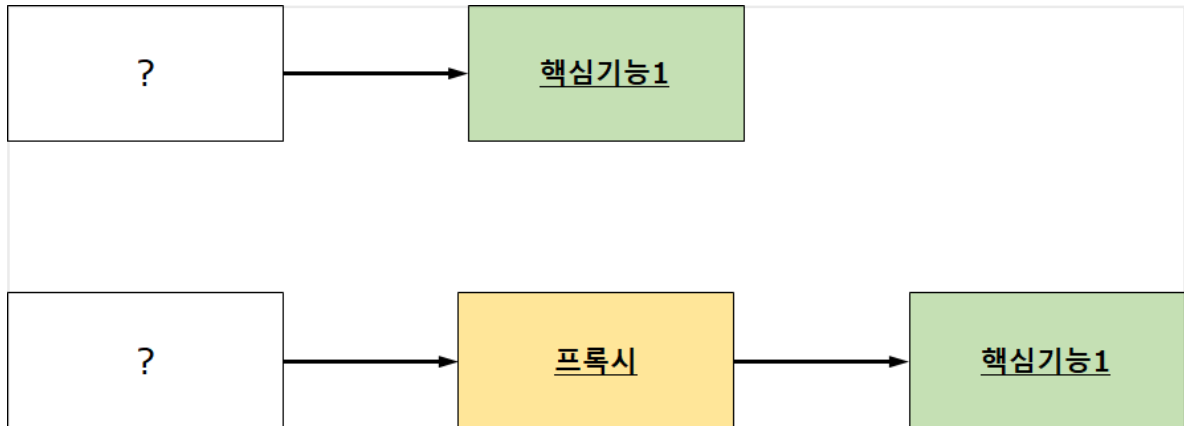
스프링 AOP 이해

스프링 AOP 동작 이해

- 개념적 이해

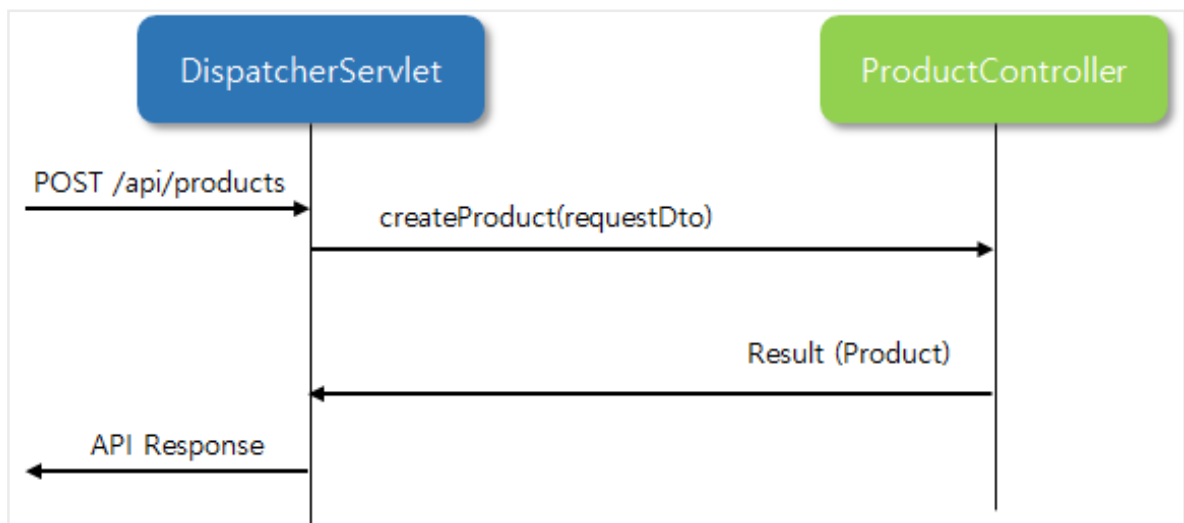


- 스프링 실제 동작

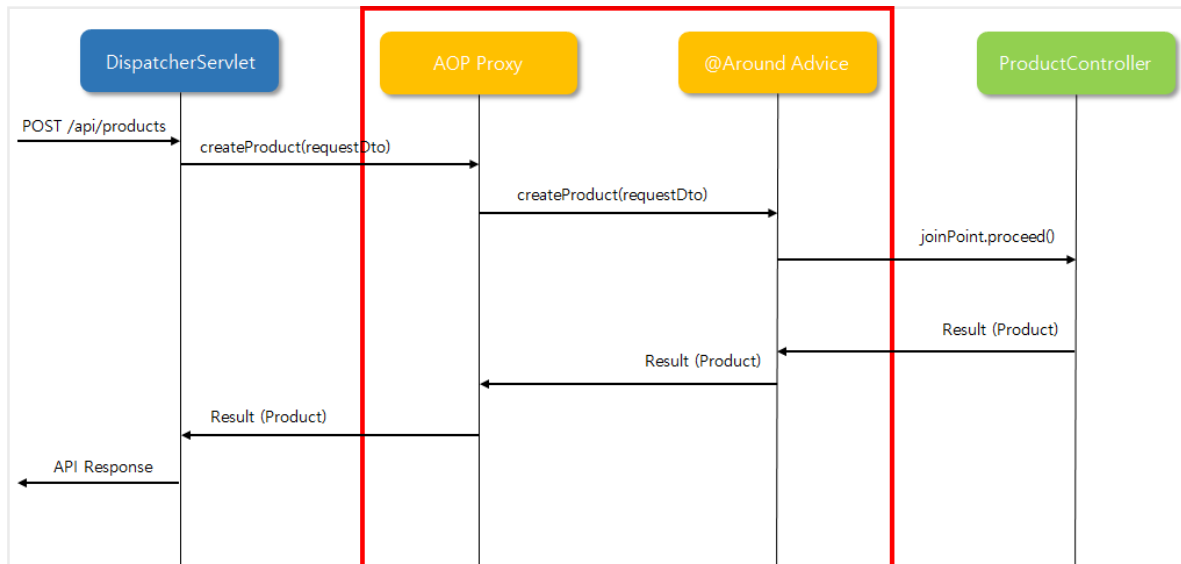


시퀀스 다이어그램

- AOP 적용 전



- AOP 적용 후



DispatcherServlet 이랑 ProductController 입장에 변화는 전혀 없음
 호출되는 함수의 input, output 이 완전 동일
 "joinPoint.proceed()"에 의해서 원래 호출하려한 함수, 인수(argument) 가 전달됨
 -> createProduct(requestDto)
 - 스프링 서버가 기동될 때
 핵심 기능 DI 시 프록시 객체를 중간에 삽입함

스프링 AOP 어노테이션

1. **@Aspect** : 스프링 빈(bean) 클래스에만 적용 가능
 사용을위해선 아래 코드 같은 형식으로 진행 필요

```

@Aspect
@Component
public class UseTimeAop {
  
```

2. 어드바이스 종류

@Around : '핵심기능' 수행 전과 후 (@Before + @After)
 @Before : '핵심기능' 호출 전 (ex. Client 의 입력값 Validation 수행)
 @After : '핵심기능' 수행 성공/실패 여부와 상관없이 언제나 동작 (try, catch 의 finally() 처럼 동작)
 @AfterReturning : '핵심기능' 호출 성공 시 (함수의 Return 값 사용 가능)
 @AfterThrowing : '핵심기능' 호출 실패 시. 즉, 예외(Exception)가 발생한 경우만 동작 (ex. 예외가 발생했을 때 개발자에게 email 이나 SMS 보냄)

3. 포인트 컷

포인트컷 Expression Language

◦ 포인트컷 Expression 형태

```
execution(modifiers-pattern? return-type-pattern declaring-type-pattern? method-name-pattern(param-pattern) throws-pattern?)
```

- ? 는 생략 가능

◦ 포인트컷 Expression 예제

```
@Around("execution(public * com.sparta.springcore.controller.*(..))")  
public Object execute(ProceedingJoinPoint joinPoint) throws Throwable { ... }
```

modifiers-pattern : public, private,*

return-type-pattern : void, String, List<String>,*

declaring-type-pattern :

클래스명 (패키지명 필요),

com.sparta.springcore.controller.* - controller 패키지의 모든

클래스에 적용

com.sparta.springcore.controller.. - controller 패키지 및 하위

패키지의 모든 클래스에 적용

method-name-pattern(param-pattern)

함수명 :

addFolders : addFolders() 함수에만 적용

add* : add 로 시작되는 모든 함수에 적용

파라미터 패턴 (param-pattern)

(com.sparta.springcore.dto.FolderRequestDto) -

FolderRequestDto 인수 (arguments) 만 적용

() - 인수 없음

(*) - 인수 1개 (타입 상관없음)

(..) - 인수 0~N개 (타입 상관없음)

@Pointcut

포인트컷 재사용 가능

포인트컷 결합(combine) 가능

```

@Component
@Aspect
public class Aspect {
    @Pointcut("execution(* com.sparta.springcore.controller.*.*(..))")
    private void forAllController() {}

    @Pointcut("execution(String com.sparta.springcore.controller.*.*())")
    private void forAllViewController() {}

    @Around("forAllContorller() && !forAllViewController")
    public void saveRestApiLog() {
        ...
    }

    @Around("forAllContorller()")
    public void saveAllApiLog() {
        ...
    }
}

```

Controller - Service - Repository 3계층에 맞춰 구현을 해줘야 각자 역할들
 간에 구분이 쉬워지고 내용이 훨씬 보여서 코드 섞임 등을 예방 할 수있다