# 와플스튜디오 Spring Seminar

세미나장: 정원식

2023.10.04.(수) 19:00

Week2

# Table of Contents

# Week1 과제 리뷰

- JPA 복습
- MultipleBagFetchException
- properties.hibernates.default_batch_size
- 요구사항 처리 DB vs Application
- @ManyToMany
- @Transactional 롤백 테스트

# Week1 과제 리뷰

## JPA 복습

- 자바 진영의 ORM 표준
- SQL 중심적 개발으로부터 벗어나, 객체 지향적 개발이 가능
- 그러나, 성능을 위해 실제로는 쿼리를 직접 짜야하는 경우도 있음. (대표적으로 N+1 이슈)

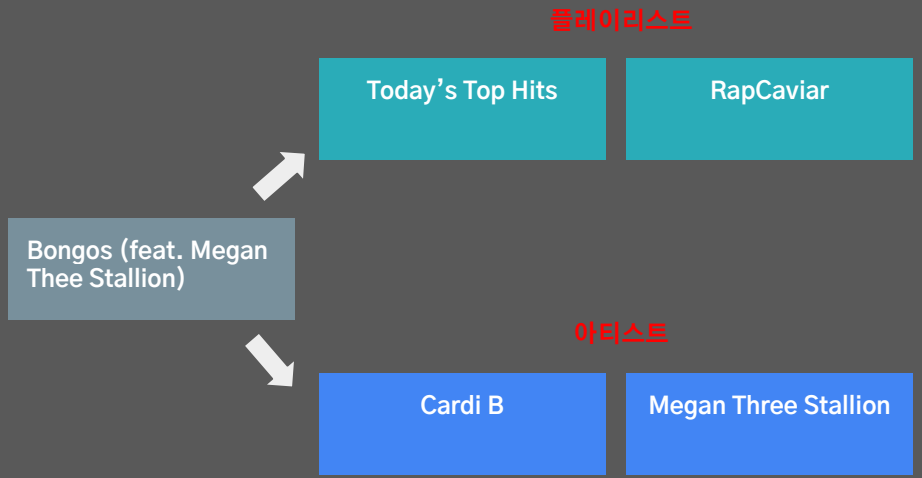
| @OneToOne, @ManyToOne | @OneToMany, @ManyToMany |
| --- | --- |
| default FetchType.EAGER | default FetchType.LAZY |
| EAGER Fetch시 조인해서 쿼리 | EAGER Fetch시 조인하지 않고 나눠 쿼리 |

# Week1 과제 리뷰

## MultipleBagFetchException (When)

- @OneToMany 관계를 두 번 이상 Fetch Join할 때 발생

```
@Entity(name = "songs")
class SongEntity(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0L,
    val title: String,
    val duration: Int,
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "album_id")
    val album: AlbumEntity,
    @OneToMany(mappedBy = "song")
    val artists: List<SongArtistEntity>,
    @OneToMany(mappedBy = "song")
    val playlists: List<PlaylistSongEntity>,
)
```



# Week1 과제 리뷰

## MultipleBagFetchException (Why)

- Bag
  - A Bag, similar to a *List*, is a collection that can contain duplicate elements. However, it is not in order.
  - Fetching two or more Bags at the same time on an *Entity* could form a Cartesian Product. Since a Bag doesn't have an order, Hibernate would not be able to map the right columns to the right entities. Hence, in this case, it throws a *MultipleBagFetchException*.

# Week1 과제 리뷰

## MultipleBagFetchException (Cartesian Product)

```
@Query("""
    SELECT s FROM songs s
    join fetch s.artists
    join fetch s.playlists
    WHERE s.id = :id
""")
fun findByIdWithJoinFetch(id: Long): SongEntity?
```

DB에서 카테시안 곱(N * M)을 응답

```
select s.id, s.title as song_title, a.name, p.title as playlist_title from songs s
left join playlist_songs ps on s.id = ps.song_id
inner join playlists p on ps.playlist_id = p.id
left join song_artists sa on s.id = sa.song_id
inner join artists a on sa.artist_id = a.id
where s.id =39;
```

DB에서 네 행을 응답

| ID | SONG_TITLE | NAME | PLAYLIST_TITLE |
|---|---|---|---|
| 39 | Bongos (feat. Megan Thee Stallion) | Cardi B | Today's Top Hits |
| 39 | Bongos (feat. Megan Thee Stallion) | Megan Thee Stallion | Today's Top Hits |
| 39 | Bongos (feat. Megan Thee Stallion) | Cardi B | RapCaviar |
| 39 | Bongos (feat. Megan Thee Stallion) | Megan Thee Stallion | RapCaviar |

(4 rows, 1 ms)

# Week1 과제 리뷰

## MultipleBagFetchException (How to solve 1)

```
@Entity(name = "songs")
class SongEntity(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0L,
    val title: String,
    val duration: Int,
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "album_id")
    val album: AlbumEntity,
    @OneToMany(mappedBy = "song")
    val artists: Set<SongArtistEntity>,
    @OneToMany(mappedBy = "song")
    val playlists: Set<PlaylistSongEntity>,
)
```

- 순서가 상관 없다면 Set으로 선언
- 에러 자체는 사라진다.
- 만약 'Bongos'가 속한 플레이리스트가 1000개, 'Bongos'에 참여한 아티스트가 100명이라면 **DB에서 10만 행을 반환 -> 심각한 퍼포먼스 이슈가 발생할 수 있다.**

# Week1 과제 리뷰

## MultipleBagFetchException (How to solve 2)

- 쿼리를 쪼갠다
    - songRepository.findById(39)
    - playlistSongRepository.findBySongId(39)
    - songArtistRepository.findBySongId(39)
    - 싱글 쿼리가 무조건 좋은 것은 아님

# Week1 과제 리뷰

## properties.hibernate.default_batch_size

```yaml
spring:
  jpa:
    properties:
      hibernate:
        default_batch_fetch_size: 1000
```

application.yaml

- 쿼리를 IN 절로 모아주는 기능
- N+1 이슈를 해결하는 또 다른 방법임.

```
songRepository.findAllById(listOf(1L, 20L, 40L))
    .forEach { println(it.album.title) }
```

```
Hibernate: select s1_0.id,s1_0.album_id,s1_0.duration,s1_0.title from songs s1_0 where s1_0.id in (?,?,?)
Hibernate: select a1_0.id,a2_0.id,a2_0.name,a1_0.image,a1_0.title from albums a1_0 left join artists a2_0 on a2_0.id=a1_0.artist_id where
 array_contains(?,a1_0.id)
```

# Week1 과제 리뷰

## 요구사항 처리 DB vs Application

- 플레이리스트 그룹 조회시, 연관된 플레이리스트가 없다면 결과에서 제외
  - 방법 1: INNER JOIN으로 필터링을 DB에 맡김
  - 방법 2: OUTER JOIN으로 가져온 후에, 어플리케이션에서 empty 체크 후 필터링
- 곡 조회시, 제목 길이가 짧은 순으로 정렬
  - 방법 1:  Order By LENGH(s.title)로 DB에 맡김
  - 방법 2: 어플리케이션에서 sortBy { s.title } 함수 사용

# Week1 과제 리뷰

## @ManyToMany

```
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(
        name = "playlist_songs",
        joinColumns = [JoinColumn(name = "song_id")],
        inverseJoinColumns = [JoinColumn(name = "playlist_id")]
)
val playlistList: List<PlaylistEntity>,
```

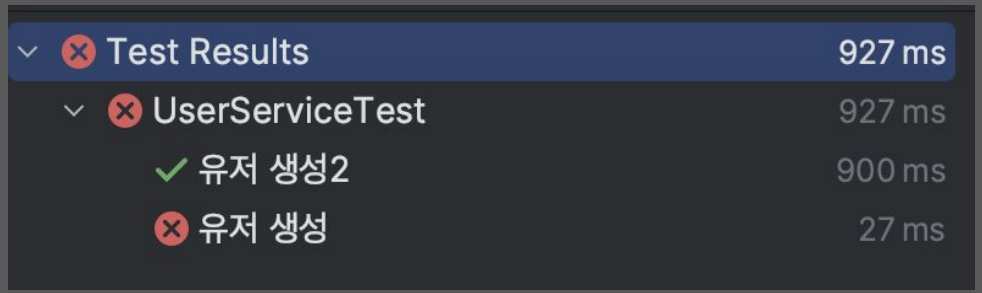
- 중간 테이블이 숨겨져 있어서 의식하지 못한 쿼리가 날라갈 수 있음.
- 중간 테이블이 두 개의 외래키만을 칼럼으로 갖기 때문에, 부가정보를 저장할 수 없음.

# Week1 과제 리뷰

## @Transactional 롤백 테스트

```kotlin
@SpringBootTest
class UserServiceTest @Autowired constructor(
    private val userService: UserService,
) {
    @Test
    fun `유저 생성`() {
        assertDoesNotThrow {
            userService.signUp(
                username = "wafflestudio",
                password = "spring",
                image = "https://wafflestudio.com/images/icon_intro.svg"
            )
        }
    }

    @Test
    fun `유저 생성2`() {
        assertDoesNotThrow {
            userService.signUp(
                username = "wafflestudio",
                password = "spring",
                image = "https://wafflestudio.com/images/icon_intro.svg"
            )
        }
    }
}
```



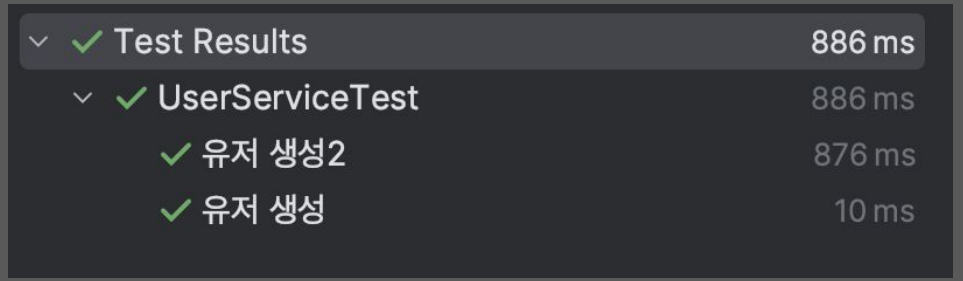

- username이 중복되어 두 테스트 중 한 테스트는 실패한다.(순서 보장 X)
- 테스트 간에 데이터베이스는 **격리되지 않는다.**

# Week1 과제 리뷰

## @Transactional 롤백 테스트

```kotlin
@Transactional
@SpringBootTest
class UserServiceTest @Autowired constructor(
    private val userService: UserService,
) {
    @Test
    fun `유저 생성`() {
        assertDoesNotThrow {
            userService.signUp(
                username = "wafflestudio",
                password = "spring",
                image = "https://wafflestudio.com/images/icon_intro.svg"
            )
        }
    }

    @Test
    fun `유저 생성2`() {
        assertDoesNotThrow {
            userService.signUp(
                username = "wafflestudio",
                password = "spring",
                image = "https://wafflestudio.com/images/icon_intro.svg"
            )
        }
    }
}
```




- @Transactional 어노테이션을 적용하면, 테스트 메소드가 끝나면 DB 수정이 롤백 된다.
- 따라서 두 테스트 모두 성공.

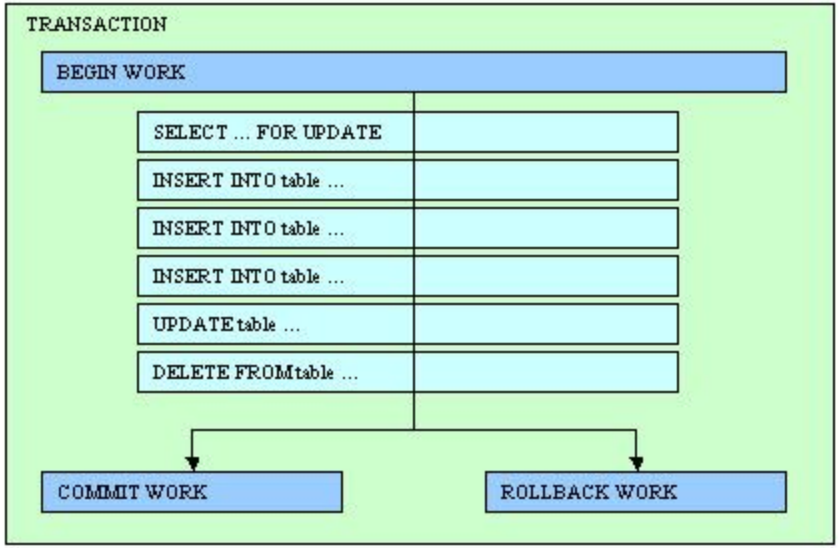
# AOP (Aspect Oriented Programming)

- 소스코드상 다른 부분에서 계속 반복해서 쓰는 부가적인 코드가 있다
- 이를 흩어진 관심사라고 함
- 흩어진 관심사를 모듈화하는 것이 AOP
- 대표적인 예로 @Transactional

# AOP (Aspect Oriented Programming)

## Transaction

- 데이터베이스의 상태를 변화시키는 하나의 논리적 기능을 수행하기 위한 작업 단위
- ACID(원자성, 일관성, 독립성, 지속성)



# AOP (Aspect Oriented Programming)

## @Transactional

```kotlin
@Transactional
override fun create(playlistId: Long, userId: Long) {
    if (playlistRepository.findById(playlistId).isEmpty) {
        throw PlaylistNotFoundException()
    }

    if (exists(playlistId = playlistId, userId = userId)) {
        throw PlaylistAlreadyLikedException()
    }

    playlistLikeRepository.save(
        PlaylistLikeEntity(
            playlistId = playlistId,
            userId = userId
        )
    )
}
```

=

```kotlin
override fun create(playlistId: Long, userId: Long) {
    transaction.begin()

    try {
        if (playlistRepository.findById(playlistId).isEmpty) {
            throw PlaylistNotFoundException()
        }

        if (exists(playlistId = playlistId, userId = userId)) {
            throw PlaylistAlreadyLikedException()
        }

        playlistLikeRepository.save(
            PlaylistLikeEntity(
                playlistId = playlistId,
                userId = userId
            )
        )

        transaction.commit()
    } catch (e : Exception) {
        transaction.rollback()
    }
}
```

# AOP (Aspect Oriented Programming)

## Proxy

PlaylistController → PlaylistLikeService (Proxy) → PlaylistLikeService

```
19      @Transactional
20      override fun create(playlistId: Long, userId: Long) {   playlistId:
            if (playlistRepository.findById(playlistId).isEmpty) {   playlis
22              throw PlaylistNotFoundException()
23          }
24
25          if (exists(playlistId = playlistId, userId = userId)) {
26              throw PlaylistAlreadyLikedException()
27          }
28
29          playlistLikeRepository.save(
30              PlaylistLikeEntity(
31                  playlistId = playlistId,
32                  userId = userId
33              )
```

```
create:21, PlaylistLikeServiceImpl (com.wafflestudio.seminar.spring2023.playlist.service)
invoke0:-1, NativeMethodAccessorImpl (jdk.internal.reflect)
invoke:77, NativeMethodAccessorImpl (jdk.internal.reflect)
invoke:43, DelegatingMethodAccessorImpl (jdk.internal.reflect)
invoke:568, Method (java.lang.reflect)
invokeJoinpointUsingReflection:343, AopUtils (org.springframework.aop.support)
invokeJoinpoint:196, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceed:163, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceed:756, CglibAopProxy$CglibMethodInvocation (org.springframework.aop.framework)
proceedWithInvocation:123, TransactionInterceptor$1 (org.springframework.transaction.interceptor)
invokeWithinTransaction:391, TransactionAspectSupport (org.springframework.transaction.interceptor)
invoke:119, TransactionInterceptor (org.springframework.transaction.interceptor)
proceed:184, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceed:756, CglibAopProxy$CglibMethodInvocation (org.springframework.aop.framework)
intercept:708, CglibAopProxy$DynamicAdvisedInterceptor (org.springframework.aop.framework)
create:-1, PlaylistLikeServiceImpl$$SpringCGLIB$$0 (com.wafflestudio.seminar.spring2023.playlist.service)
likePlaylist:53, PlaylistController (com.wafflestudio.seminar.spring2023.playlist.controller)
```

# AOP (Aspect Oriented Programming)

## @Transactional 테스트

- @Test, @Transactional 같이 사용시 테스트 완료 후 롤백이 적용된다.

```
@Transactional
@SpringBootTest
class SongServiceTest @Autowired constructor(
    private val songService: SongService,
    private val queryCounter: QueryCounter,
) {
```

# Configuration

## @Bean을 수동으로 등록할 때 사용

```
@EnableConfigurationProperties(CacheProperties::class)
@Configuration
class CacheConfig(
    private val cacheProperties: CacheProperties,
) {

    @Bean
    fun cache(): Caffeine<Any, Any> {
        return Caffeine.newBuilder()
            .maximumSize(cacheProperties.size)
            .expireAfterWrite(cacheProperties.ttl)
    }
}
```

# Configuration

## @ConfiguationProperties

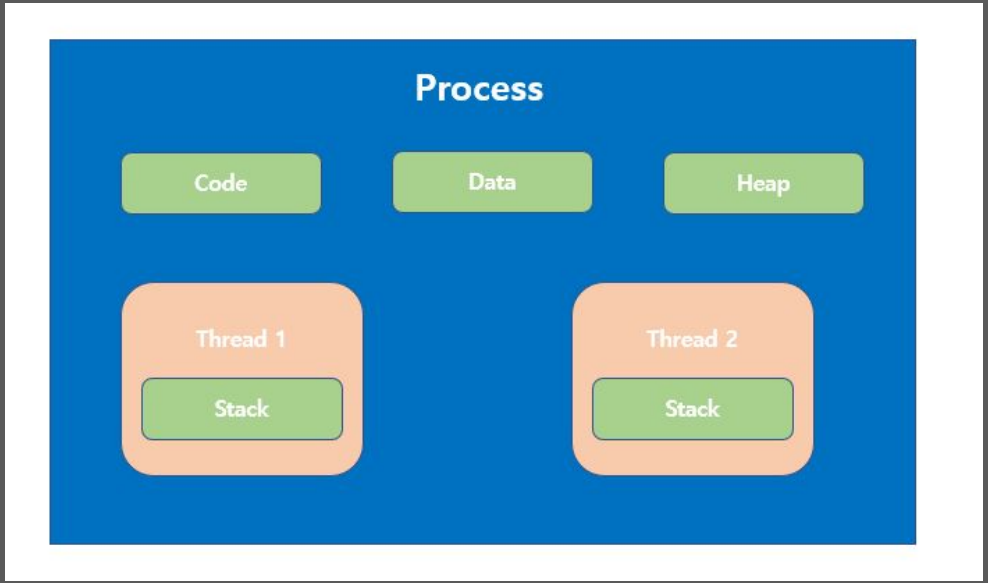
```
@ConfigurationProperties("cache")
data class CacheProperties(
    val ttl: Duration,
    val size: Long,
)
```

```
cache:
  ttl: 10s
  size: 100
```

```
@EnableConfigurationProperties(CacheProperties::class)
@Configuration
class CacheConfig(
    private val cacheProperties: CacheProperties,
) {
```

- application.yaml에 커스텀화할 프로퍼티 명시
- properties를  자동주입 받을 수 있음

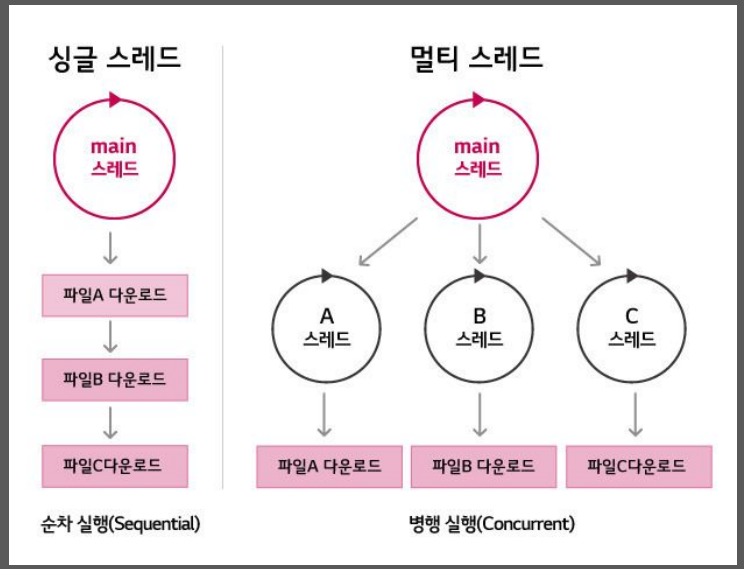
# 스레드




- 쓰레드(Thread)란 프로세스 내에서 실행되는 흐름의 단위 혹은 CPU 스케줄링의 기본 단위
- 프로세스 내에서 Code, Data, Heap 영역을 공유한다.

# 스레드

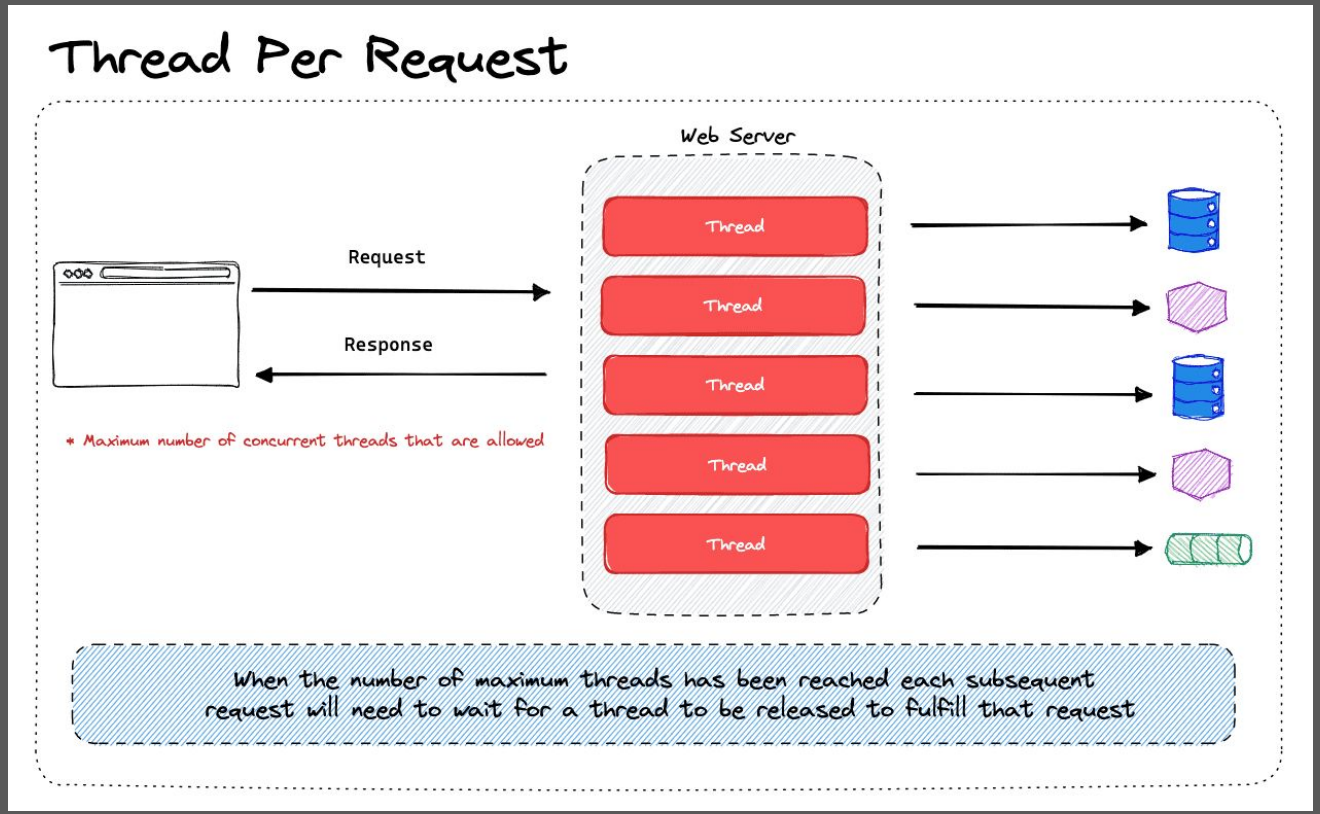
## 멀티 스레딩




- 하나의 프로세스를 다수의 실행 단위로 구분하여 자원을 공유하고, 자원의 생성과 관리의 중복성을 최소화하여 수행 능력을 향상시키는 것을 멀티쓰레딩이라고 한다.
- 하나의 프로그램에 동시에 여러개의 일을 수행할수 있도록 해주는 것이다.
- 시스템 자원 소모가 감소. 프로세스를 생성하는 system call이 줄어들기 때문

# 스레드

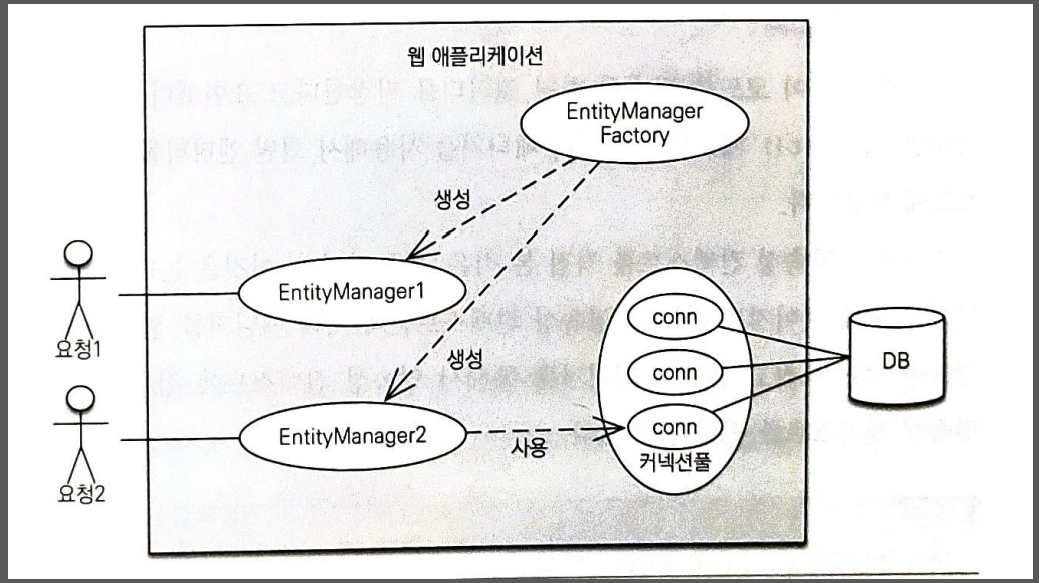
## 스프링 MVC의 스레드 모델




한 개의 스레드가 한 개의 클라이언트 요청을 처리한다.

# 스레드

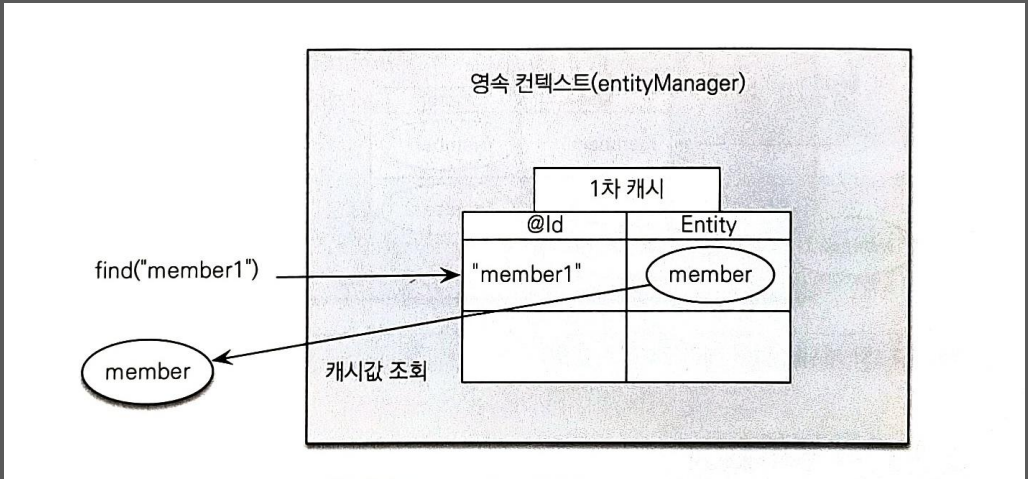
## Spring JPA의 스레드 모델




- 한 개의 요청 – 한 개의 스레드
- 한 개의 스레드 – 한 개의 엔티티매니저 (영속성 컨텍스트)
- 한 개의 스레드 – 한 개의 영속성 컨텍스트

# 스레드

## 영속성 컨텍스트 복습

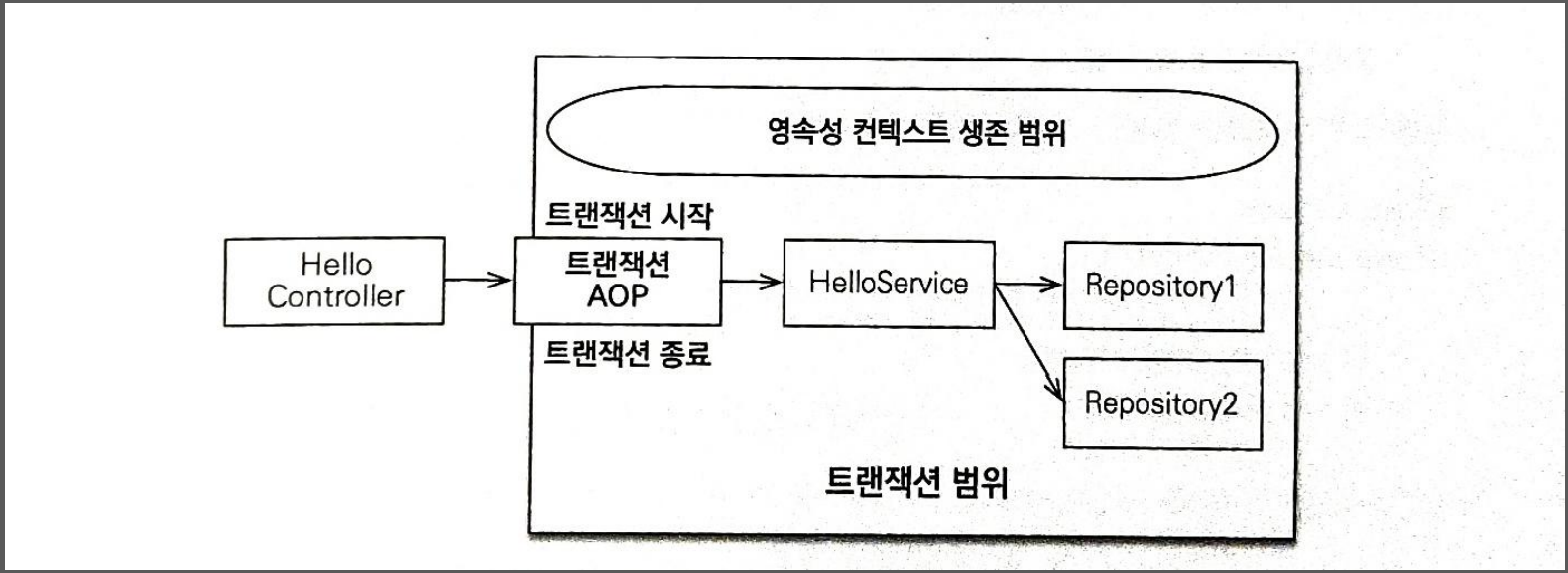



- 엔티티 캐싱
- 쓰기 지연
- 지연 로딩
- 변경 감지

현재까지 우리는 엔티티 캐싱, 지연 로딩을 사용했는데 모두 영속성 컨텍스트가 가능하게 해준 것.

# 스레드

## 영속성 컨텍스트의 생존 범위 (디폴트)



# 스레드

## 영속성 컨텍스트의 생존 범위 (OSIV)

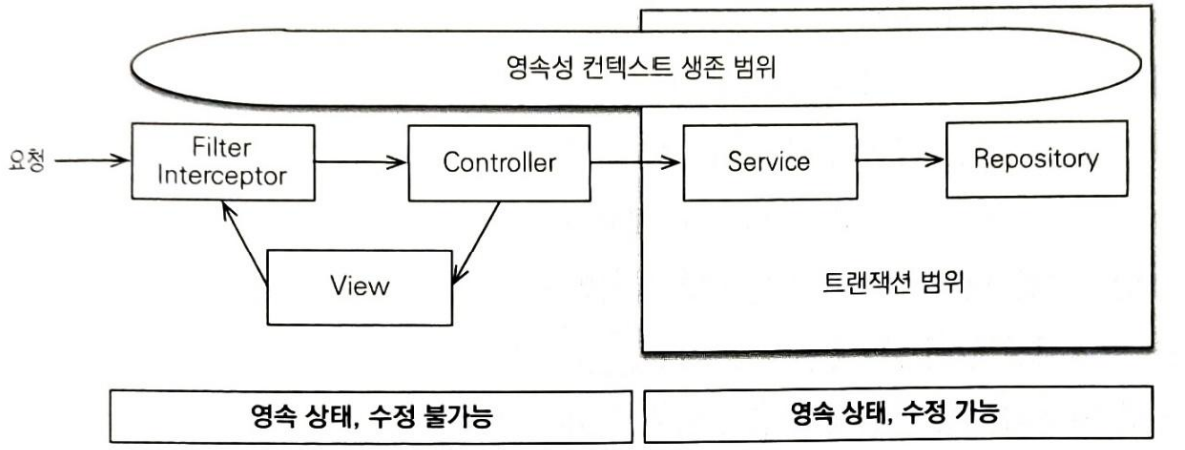



그림 13.7 스프링 OSIV – 비즈니스 계층 트랜잭션

```
2023-10-03T18:19:14.923+09:00  WARN 30995 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
```

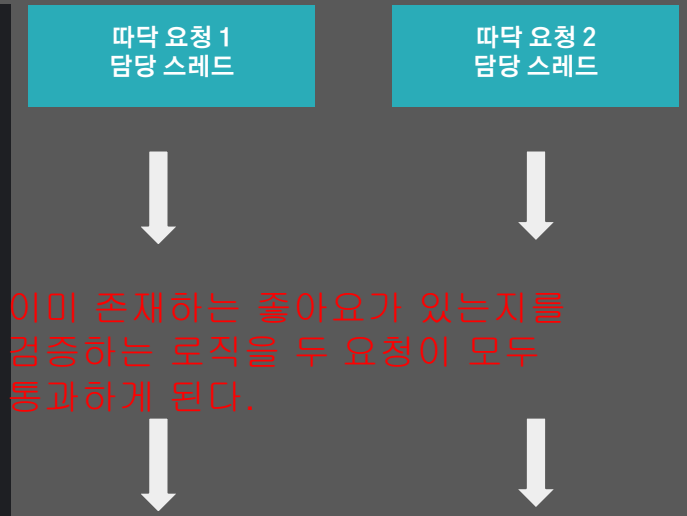
# 스레드

## ‘따닥’ 이슈 (클라이언트가 좋아요 요청을 동시에 2번 보내면?)

```
override fun create(playlistId: Long, userId: Long) {
    if (playlistRepository.findById(playlistId).isEmpty) {
        throw PlaylistNotFoundException()
    }

    if (exists(playlistId = playlistId, userId = userId)) {
        throw PlaylistAlreadyLikedException()
    }

    playlistLikeRepository.save(
        PlaylistLikeEntity(
            playlistId = playlistId,
            userId = userId
        )
    )
}
```

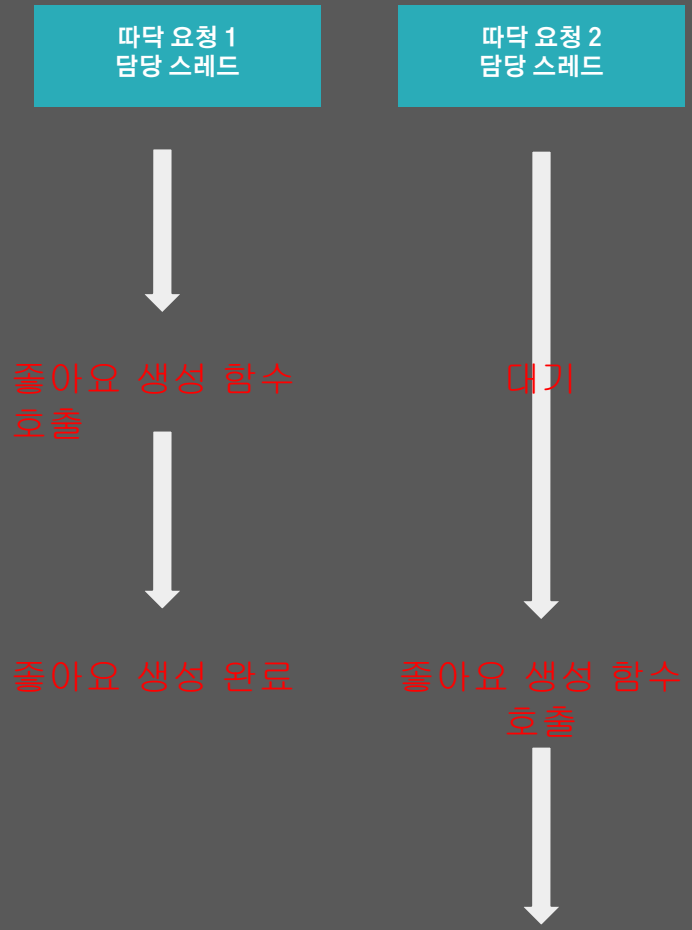


# 스레드

## ‘동기화’를 통한 따닥 이슈 해결

```
@Synchronized
override fun createSynchronized(playlistId: Long, userId: Long) {
    create(playlistId, userId)
}
```

- 동기화: 다수의 스레드가 하나의 공유 데이터 혹은 코드 블록에 동시에 접근하지 못하도록 막는 것.



# 스레드

Synchronized는 여러 서버 인스턴스간 공유가 불가능

# 스레드

## DB Unique Key

```
create table playlist_likes (
    id bigint auto_increment,
    playlist_id bigint not null,
    user_id bigint not null,
    primary key (id),
    unique (playlist_id, user_id)
);
```

중복 데이터 생성 방지를 어플리케이션이 아닌 DB에 위임

```
Caused by: org.h2.jdbc.JdbcSQLIntegrityConstraintViolationException: Unique index or primary key violation: "PUBLIC.CONSTRAINT_INDEX_9 ON
 PUBLIC.PLAYLIST_LIKES(PLAYLIST_ID NULLS FIRST, USER_ID NULLS FIRST) VALUES ( /* key:1 */ CAST(1 AS BIGINT), CAST(1 AS BIGINT))"; SQL statement:
insert into playlist_likes (playlist_id,user_id,id) values (?,?,default) [23505-214]
```

# Q & A