

JPA Lock

울타리 프로젝트에서의 동시성 문제 해결 사례

CONTENTS

01

동시성 문제란?

02

JPA Lock 종류

03

울타리 프로젝트에서의 동시성 문제 해결 사례

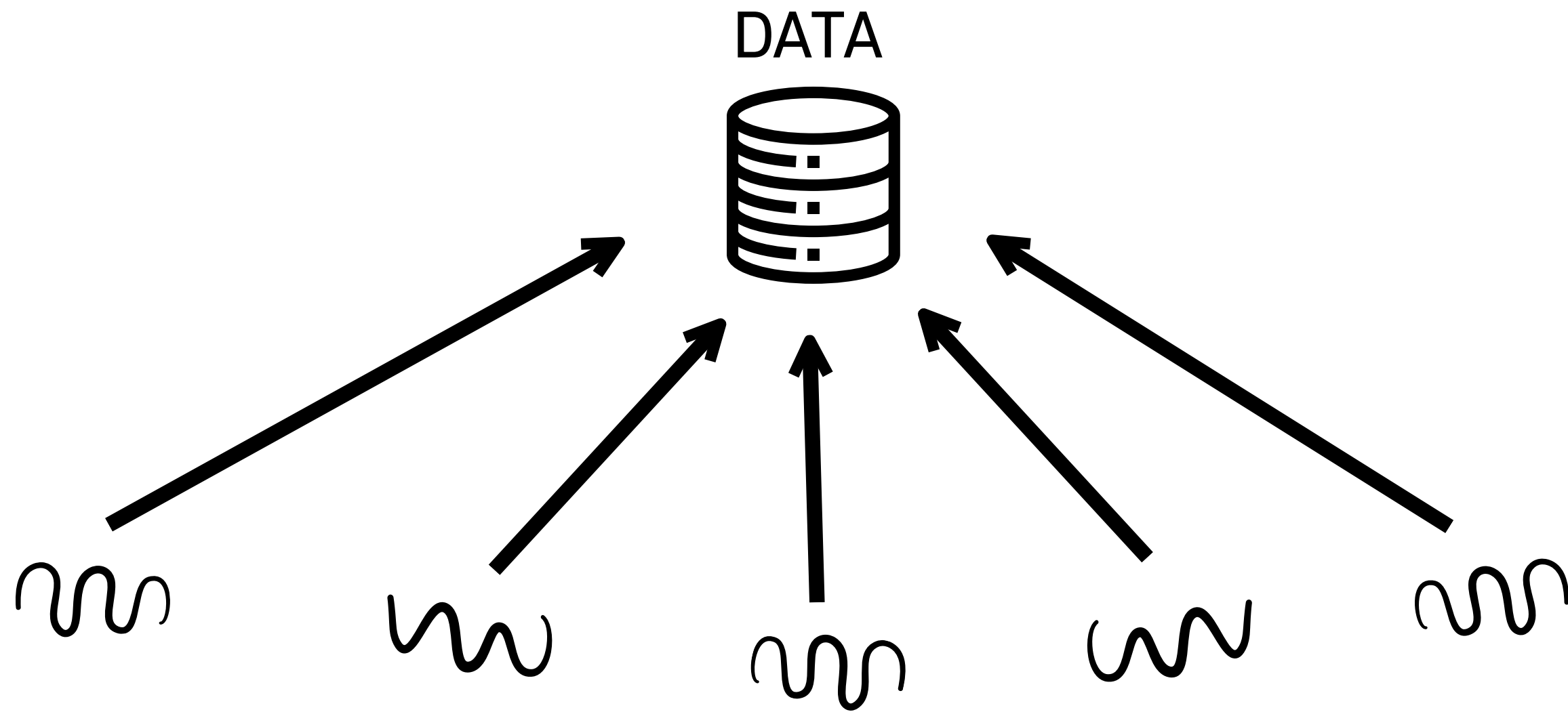
04

Q/A

CHAPTER 1

동시성 문제란?

동시성 문제란?



멀티쓰레드 환경에서 공유 데이터에 대해 동시에 **수정 작업**을 수행하는 경우에 발생

DB에서의 동시성 문제와 Lock

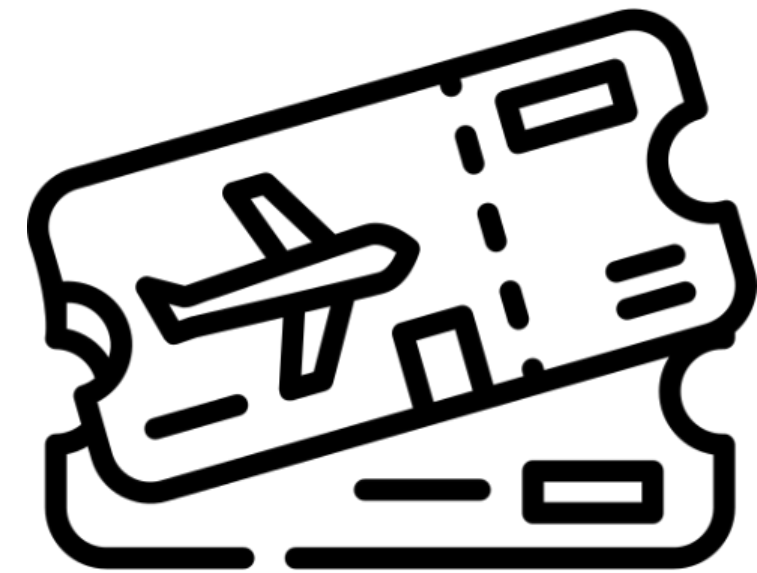
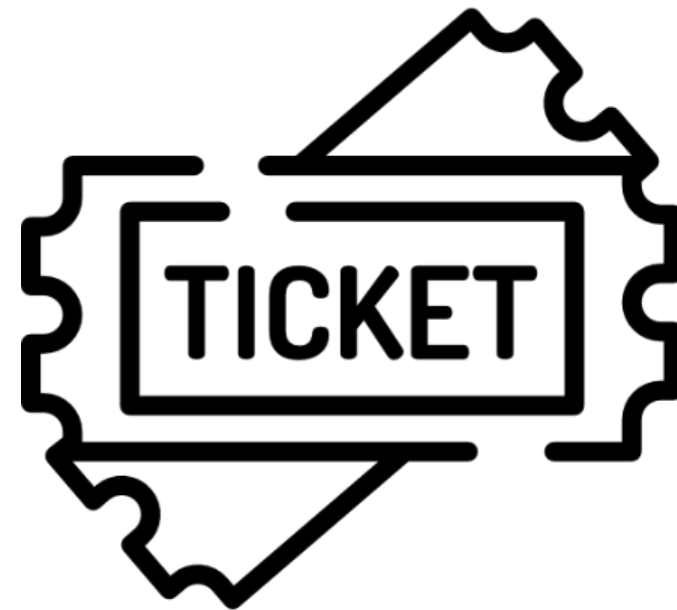
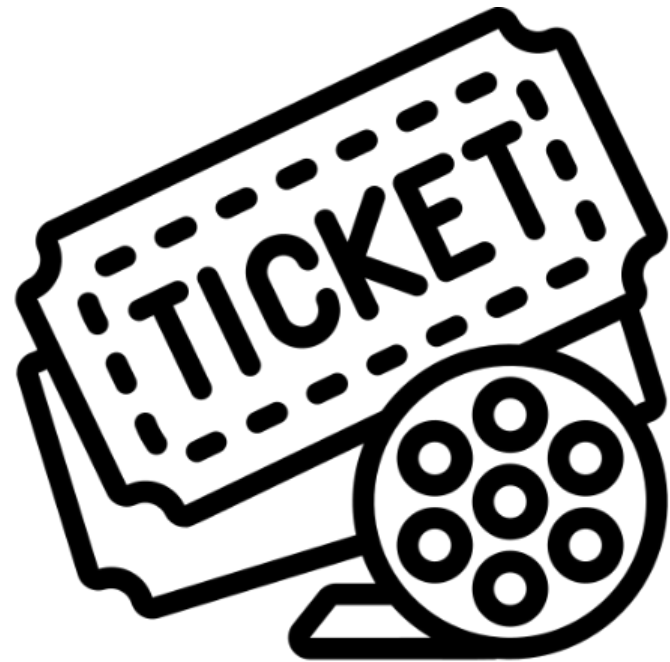
대부분의 DBMS에서 트랜잭션이 동시에 실행되는 것을 허용하는데,
이때 트랜잭션간 격리 수준에 따라 동시성 문제가 발생할 수 있다.
이때 Lock은 트랜잭션 격리 수준을 구현하는 하나의 방법이다.

2 Phase Locking

MVCC
(Multi-Version Concurrency Control)

예시

티켓 종류별 선착순으로 티켓을 발행하는 서비스



Movie Ticket Type

티켓 발행 가능수 : 1

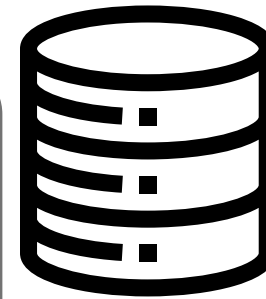
동시에 실행

Transaction 1

SELECT * FROM ticket_type
티켓 발행 가능수 1로 조회



UPDATE ticket_type
SET remaining_ticket_count = 0



Transaction 2

SELECT * FROM ticket_type
티켓 발행 가능수 1로 조회

UPDATE ticket_type
SET remaining_ticket_count = 0



예상 : 1개 발행

실제 : 2개 발행

의도와 다른 결과

```

@Getter
@Entity
@Table(name = "ticket_type")
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class TicketType {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private int remainingTicketCount;

    1 usage
    public TicketType(String name, int remainingTicketCount) {
        this.name = name;
        this.remainingTicketCount = remainingTicketCount;
    }

    1 usage
    public void decreaseRemainingTicketCount() {
        this.remainingTicketCount--;
    }
}

```

```

@Getter
public class Ticket {
    private TicketType ticketType;

    private LocalDateTime createdAt;

    1 usage
    public Ticket(TicketType ticketType) {
        this.ticketType = ticketType;
        this.createdAt = LocalDateTime.now();
    }

    @Override
    public String toString() {
        return createdAt + "에 발행된 " + ticketType.getName() + " 티켓";
    }
}

```



```
@Service
@Transactional
@RequiredArgsConstructor
public class TicketService {
    private final TicketTypeRepository ticketTypeRepository;

    1 usage
    public Ticket issueTicket(String ticketTypeName) {
        // Ticket Type 이름으로 TicketType 조회
        TicketType ticketType = ticketTypeRepository.findByName(ticketTypeName)
            .orElseThrow(EntityNotFoundException::new);

        // Ticket 발행 가능 개수가 0보다 작으면 Ticket 발행 X
        if (ticketType.getRemainingTicketCount() <= 0) {
            return null;
        }

        // ticket 발행 가능 개수를 -1하고, 새로 발행한 Ticket 반환
        ticketType.decreaseRemainingTicketCount();
        return new Ticket(ticketType);
    }
}
```

```

@Test
@DisplayName("동시에 10명이 티켓을 발행하면, 티켓 최대 발행 개수까지만 발행한다.")
void concurrentTest() throws InterruptedException {
    // given
    final int THREADS = 10;

    // 최대 5개까지 발행 가능한 Movie라는 TicketType을 저장한다.
    TicketType movieTicketType = new TicketType( name: "Movie", remainingTicketCount: 5);
    ticketTypeRepository.saveAndFlush(movieTicketType);

    ExecutorService executorService = Executors.newFixedThreadPool(THREADS);
    CountDownLatch latch = new CountDownLatch(THREADS);

    // when
    // Movie 타입의 ticket을 동시에 10번 발행 시도한다.
    for (int i = 1; i <= THREADS; i++) {
        executorService.execute(() -> {
            Ticket newTicket = ticketService.issueTicket( ticketTypeName: "Movie");
            System.out.println(newTicket);
            latch.countDown();
        });
    }
    latch.await();

    // then
    // Movie Ticket Type의 발행 가능 티켓 수가 0이어야 한다.
    TicketType ticketType = ticketTypeRepository.findByName("Movie").get();
    assertThat(ticketType.getRemainingTicketCount()).isZero();
}

```

Test Results	834ms	2024-02-18T19:15:05.839413에 발행된 Movie 티켓
TicketServiceTest	834ms	2024-02-18T19:15:05.838997에 발행된 Movie 티켓
동시에 10명이 티켓을 발행하면, 티켓 최대 발행 개수까지만 발행한다.	834ms	2024-02-18T19:15:05.839003에 발행된 Movie 티켓
		2024-02-18T19:15:05.838997에 발행된 Movie 티켓
		2024-02-18T19:15:05.839219에 발행된 Movie 티켓
		2024-02-18T19:15:05.839088에 발행된 Movie 티켓
		2024-02-18T19:15:05.839275에 발행된 Movie 티켓
		2024-02-18T19:15:05.839040에 발행된 Movie 티켓
		2024-02-18T19:15:05.838996에 발행된 Movie 티켓
		2024-02-18T19:15:05.839118에 발행된 Movie 티켓

expected: 0
but was: 4

org.opentest4j.AssertionFailedError:
expected: 0
but was: 4

5개 발행을 기대했지만, 실제로는 10개가 발행
티켓 발행 가능 개수도 0이 아니라 4로 저장
즉, 동시성 문제 발생

CHAPTER 02

JPA Lock 종류

Pessimistic Lock

비관적 락

트랜잭션이 충돌한다고 가정하고
일단 락을 거는 방법

DB에서 제공하는 Lock 기능 사용

Optimistic Lock

낙관적 락

트랜잭션 대부분은 충돌이 발생하지
않는다는 것을 가정하는 방법

JPA가 제공하는 버전 관리 기능 사용

Pessimistic Lock

비관적 락

READ

Shared Lock

데이터를 반복 읽기만 하고
수정하지 않는 용도로
락을 걸때 사용

WRITE

Exclusive Lock

데이터를 수정할 용도로
락을 걸때 사용
락을 획득해야 수정이 가능

Movie Ticket Type

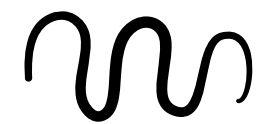
티켓 발행 가능수 : 1

동시에 실행

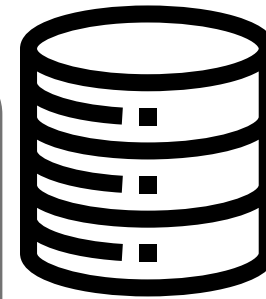
Transaction 1



SELECT * FROM ticket_type for update
티켓 발행 가능수 1로 조회



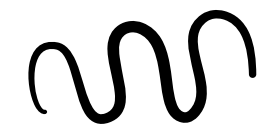
UPDATE ticket_type
SET remaining_ticket_count = 0



Transaction 2

SELECT * FROM ticket_type for update
티켓 발행 가능수 0으로 조회

티켓 발행 X



예상 : 1개 발행

실제 : 1개 발행

Optimistic Lock

낙관적 락

JPA에서 제공하는 기능으로 Entity에 Version 정보를 추가로 저장하여
변경 시점의 Version이 최신인 경우만 변경을 하는 동시성 제어 방법
(변경 후에 Version이 올라감)

Movie Ticket Type

티켓 발행 가능수 : 1

동시에 실행

Transaction 1

SELECT * FROM ticket_type

티켓 발행 가능수 1로 조회

Version 1

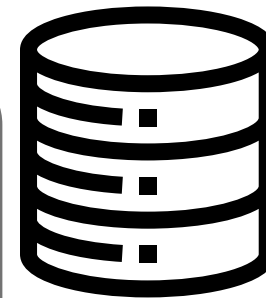


UPDATE ticket_type

SET remaining_ticket_count = 0,

version = 2

WHERE version = 1



Transaction 2

SELECT * FROM ticket_type

티켓 발행 가능수 1로 조회

Version 1

UPDATE ticket_type

SET remaining_ticket_count = 0,

version = 2

WHERE version = 1

최신 Version이 2이므로 실패

예상 : 1개 발행

실제 : 1개 발행

Locking

To specify the lock mode to be used, you can use the `@Lock` annotation on query methods, as shown in the following example:

Example 1. Defining lock metadata on query methods

```
interface UserRepository extends Repository<User, Long> {  
    // Plain query method  
    @Lock(LockModeType.READ)  
    List<User> findByLastname(String lastname);  
}
```

JAVA

This method declaration causes the query being triggered to be equipped with a `LockModeType` of `READ`. You can also define locking for CRUD methods by redeclaring them in your repository interface and adding the `@Lock` annotation, as shown in the following example:

Example 2. Defining lock metadata on CRUD methods

```
interface UserRepository extends Repository<User, Long> {  
    // Redclaration of a CRUD method  
    @Lock(LockModeType.READ)  
    List<User> findAll();  
}
```

JAVA

```
package jakarta.persistence;
```

```
public enum LockModeType
{
    
        | Synonymous with OPTIMISTIC. OPTIMISTIC is to be preferred for new applications.
    READ,

    
        | Synonymous with OPTIMISTIC_FORCE_INCREMENT. OPTIMISTIC_FORCE_INCREMENT is to be preferred for new applications.
    WRITE,

    
        | Optimistic lock.
        | Since: 2.0
    OPTIMISTIC,
```

| Optimistic lock, with version update.

| Since: 2.0

OPTIMISTIC_FORCE_INCREMENT,

| Pessimistic read lock.

| Since: 2.0

PESSIMISTIC_READ,

| Pessimistic write lock.

| Since: 2.0

PESSIMISTIC_WRITE,

| Pessimistic write lock, with version update.

| Since: 2.0

PESSIMISTIC_FORCE_INCREMENT,

| No lock.

| Since: 2.0

NONE

}

Pessimistic Lock

```
public interface TicketTypeRepository extends JpaRepository<TicketType, Long> {  
    1 usage  
    @Lock(LockModeType.PESSIMISTIC_WRITE)  
    Optional<TicketType> findByName(String name);  
}
```

✓ Test Results	1sec 59 ms	2024-02-18T19:23:37.825793에 발행된 Movie 티켓
✓ TicketServiceTest	1sec 59 ms	2024-02-18T19:23:37.863204에 발행된 Movie 티켓
✓ 동시에 10명이 티켓을 발행하면, 티켓 최대 발행 개수까지만 발행한다.	1sec 59 ms	2024-02-18T19:23:37.867263에 발행된 Movie 티켓
		2024-02-18T19:23:37.901336에 발행된 Movie 티켓
		2024-02-18T19:23:37.910338에 발행된 Movie 티켓
		null
		null
		null
		null
		null

Optimistic Lock

```
@Getter
@Entity
@Table(name = "ticket_type")
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class TicketType {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private int remainingTicketCount;

    @Version
    private int version;

    1 usage
    public TicketType(String name, int remainingTicketCount) {
        this.name = name;
        this.remainingTicketCount = remainingTicketCount;
    }

    1 usage
    public void decreaseRemainingTicketCount() {
        this.remainingTicketCount--;
    }
}
```

```
public interface TicketTypeRepository extends JpaRepository<TicketType, Long> {
    // @Lock(LockModeType.OPTIMISTIC) // 생략 가능
    1 usage
    Optional<TicketType> findByName(String name);
}
```

Specifies the version field or property of an entity class that serves as its optimistic lock value. The version is used to ensure integrity when performing the merge operation and for optimistic concurrency control.

Only a single Version property or field should be used per class; applications that use more than one Version property or field will not be portable.

The Version property should be mapped to the primary table for the entity class; applications that map the Version property to a table other than the primary table will not be portable.

The following types are supported for version properties: int, Integer, short, Short, long, Long, java.sql.Timestamp.

1. 클래스당 1개의 필드만 가능
2. Primary table에서만 사용 가능
3. int, Integer, short, Short, long, Long, java.sql.Timestamp 타입 사용 가능

Test Results

TicketServiceTest

동시에 10명이 티켓을 발행하면, 티켓 최대 발행 개수까지만 발행한다.

expected: 0
but was: 4

```
2024-02-18T20:44:28.677284에 발행된 Movie 티켓  
Exception in thread "pool-2-thread-6" org.springframework.orm.ObjectOptimisticLockingFailureException Create breakpoint :  
    at org.springframework.orm.jpa.vendor.HibernateJpaDialect.convertHibernateAccessException(HibernateJpaDialect.java:205)  
    at org.springframework.orm.jpa.vendor.HibernateJpaDialect.translateExceptionIfPossible(HibernateJpaDialect.java:231)  
    at org.springframework.orm.jpa.JpaTransactionManager.doCommit(JpaTransactionManager.java:565)  
    at org.springframework.transaction.support.AbstractPlatformTransactionManager.processCommit(AbstractPlatformTransactionManager.java:770)  
    at org.springframework.transaction.support.AbstractPlatformTransactionManager.commit(AbstractPlatformTransactionManager.java:743)  
    at org.springframework.transaction.interceptor.TransactionAspectSupport.commitTransactionAfterReturning(TransactionAspectSupport.java:650)  
    at org.springframework.transaction.interceptor.TransactionAspectSupport.invokeWithinTransaction(TransactionAspectSupport.java:305)  
    at org.springframework.transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:119)  
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:184)  
    at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.proceed(CglibAopProxy.java:765)  
    at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:717)  
    at com.example.ticket.service.TicketService$$SpringCGLIB$$0.issueTicket(<generated>)
```

티켓이 1개 발행되었고, 나머지 시도에 대해서 OptimisticLockException 발생

티켓 발행 가능 수보다 더 많이 발행되지 않음을 보장

7. *OptimisticLockException*

When the persistence provider discovers optimistic locking conflicts on entities, it throws *OptimisticLockException*. We should be aware that due to the exception the active transaction is always marked for rollback.

It's good to know how we can react to *OptimisticLockException*. Conveniently, this exception contains a reference to the conflicting entity. **However, it's not mandatory for the persistence provider to supply it in every situation.** There is no guarantee that the object will be available.

There is a recommended way of handling the described exception, though. We should retrieve the entity again by reloading or refreshing, preferably in a new transaction. After that, we can try to update it once more.

낙관적 락 도중에 충돌이 발생하면 *OptimisticLockException*을 발생시킨다.

가급적이면 새로운 트랜잭션에서 Entity를 reloading이나 refreshing을 하고,
재시도하는 방법을 추천한다.

출처 : <https://www.baeldung.com/jpa-optimistic-locking>

03

CHAPTER

울타리 프로젝트에서의 동시성 문제 해결 사례

이번주 정식 출시 예정

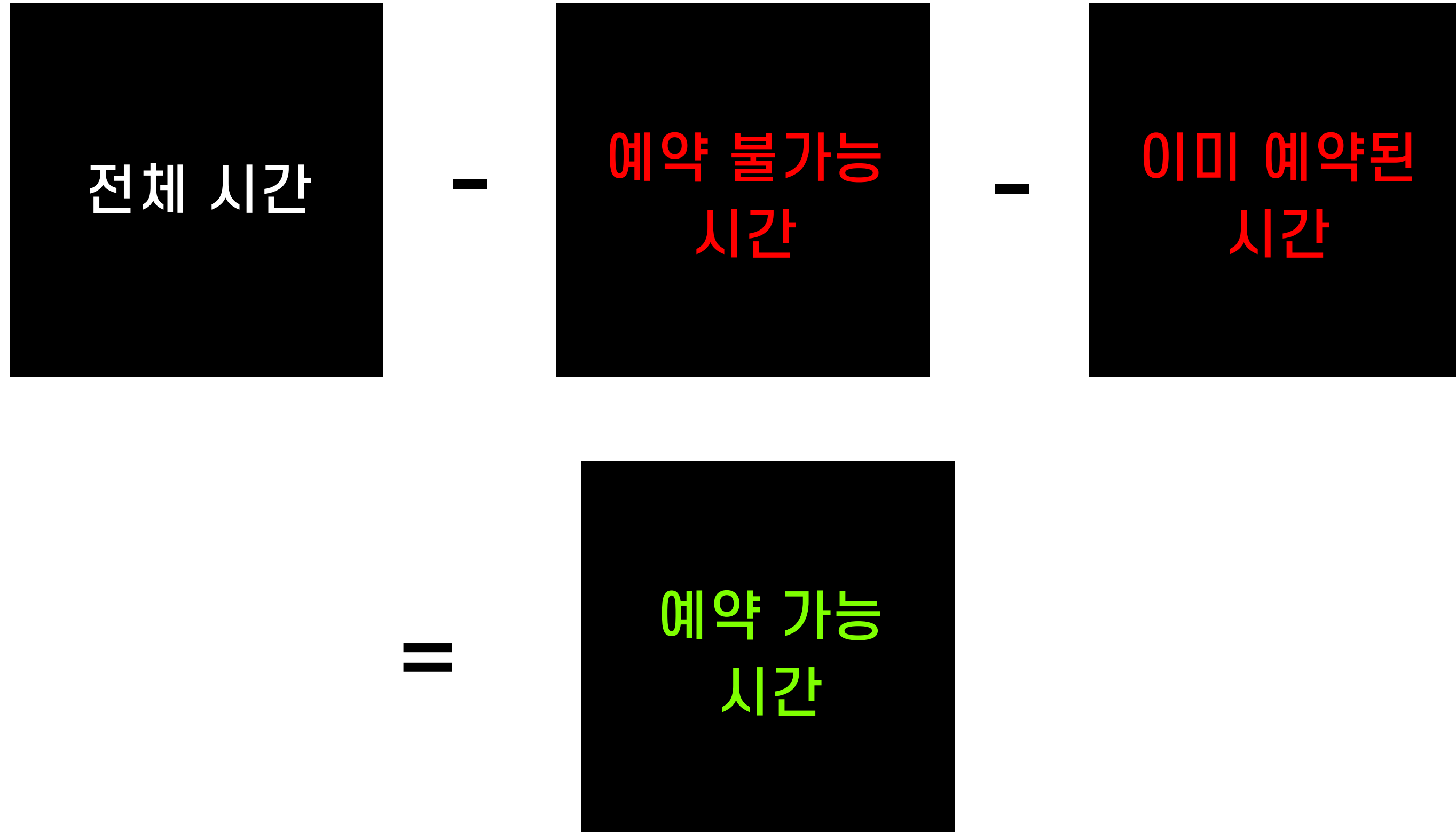


최고의 CLASS4 예약 서비스

<https://main.dmmioltqeirq3.amplifyapp.com/>

울타리 프로젝트에서
여러 사람이 동시에 예약했을때
동시성 문제가 발생하지 않나요?





예약 불가능 시간 => 주말, 공휴일, 수업 시간, 스케줄(현장 실습 등)

예약 프로세스

예약 요청
유효성 검사

예약 날짜, 예약 테이블 수 등



해당 날짜의 예약 가능
시간 조회 후
예약 시간과 겹치지
않는지 확인



DB에
예약 정보 추가

문제점

해당 일자의 예약 가능 시간 = 전체 시간 - 예약 불가능 시간 - 이미 예약된 시간

즉, 예약 프로세스 도중에 예약 불가능 시간과 예약된 시간은 변경되면 X
해당 날짜의 예약 불가능 시간과 이미 예약된 시간에 Lock을 걸자



만약 예약 불가능 시간과 예약된 시간이 존재하지 않는 날이라면?
Lock을 걸기 위한 Record가 존재하지 않음


해결 방법




사용 가능 시간 테이블 추가



모든 날짜에 대해 사용 가능 시간 Record가 존재해야 함
(매일 자정에 Scheduler가 돌면서 사용 가능 시간 생성 및 정합성을 맞춤)


사용 가능 시간 테이블 Record에 Pessimistic Write Lock을 걸고 예약 진행

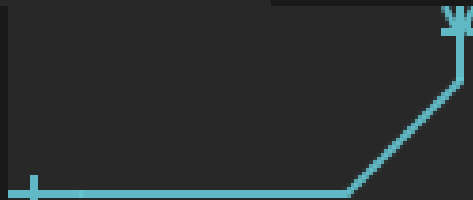
reservation 예약				
	id	예약 id	Domain	Type
	study_group_id	스터디그룹 id	Domain	Type
	date	날짜	Domain	Type
	range_bit	시간 구간 bit	Domain	Type
	canceled_at	예약 취소된 시간	Domain	Type

unavailable_time 예약 불가능 시간				
	id	예약 불가능 시간 id	Domain	Type
	discriminator	판별자	Domain	Type
	date	날짜	Domain	Type
	range_bit	시간 구간 bit	Domain	Type
	detail_reason	이유	Domain	Type

table_reservation 테이블 예약				
	id	테이블 예약 id	Domain	Type
	reservation_id	예약 id	Domain	Type
	table_info_id	테이블 id	Domain	Type

available_table_time 예약 가능 시간				
	id	시간 id	Domain	Type
	table_info_id	테이블 id	Domain	Type
	date	날짜	Domain	Type
	time_bit	시간 bit	Domain	Type

table_info 테이블 정보				
	id	테이블 id	Domain	Type
	type	타입	Domain	Type
	seat	좌석수	Domain	Type
	canceled_at	삭제된 시간	Domain	Type



CHAPTER 4

Q/A

THANK YOU