이커머스 시스템 디자인 설계 및 개발 회고 보고서

📋 목차

1. 프로젝트 개요
2. 시스템 아키텍처 설계
3. 핵심 시스템별 설계 결정사항
4. 기술적 의사결정, 트레이드오프
5. 개발 과정에서 도전과 해결책
6. 회고 및 학습내용

🎯 프로젝트 개요

구현한 핵심 시스템

본 프로젝트에서는 대용량 트래픽을 처리할 수 있는 이커머스 시스템의 핵심 기능들을 설계하고 구현했습니다.

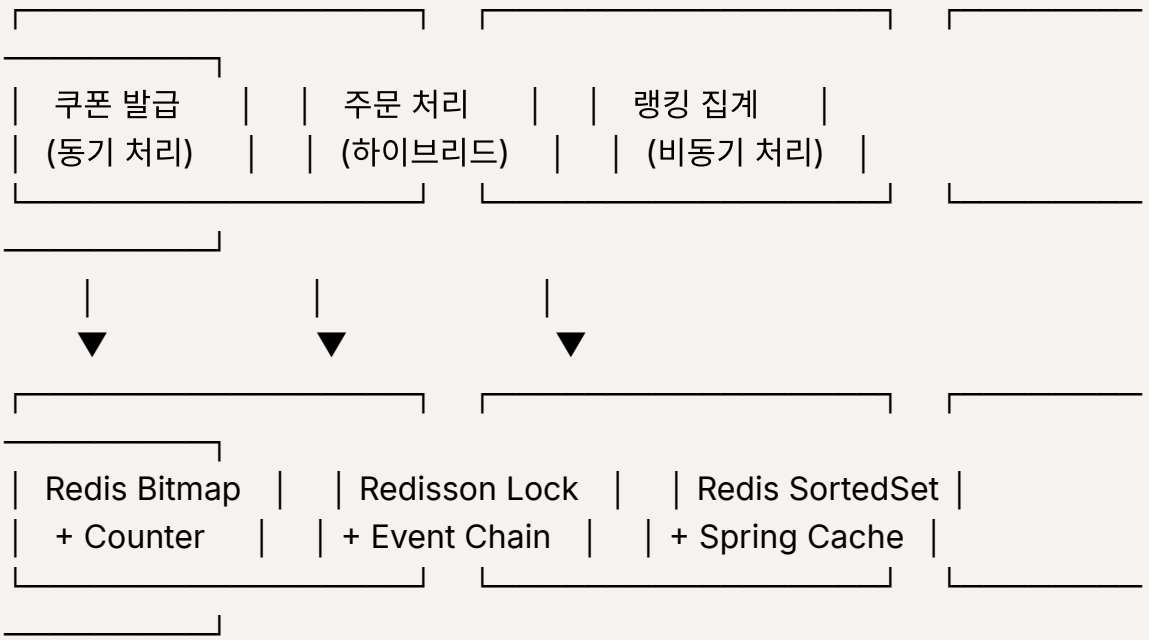
1. **선착순 쿠폰 발급 시스템** - Redis 기반 동시성 제어
2. **실시간 인기상품 랭킹 시스템** - Redis SortedSet 활용
3. **비동기 주문 처리 시스템** - Event-driven Architecture

설계 목표

- **동시성**: 대량의 동시 요청을 안전하게 처리
- **성능**: 빠른 응답 시간과 높은 처리량 확보
- **확장성**: 트래픽 증가에 대응 가능한 구조
- **일관성**: 데이터 정합성 보장

🏗️ 시스템 아키텍처 설계

전체 아키텍처 개념도



핵심 설계 원칙

1. 관심사 분리 (Separation of Concerns)

쿠폰 발급: 빠른 동시성 제어에 집중
주문 처리: 비즈니스 로직의 안전한 실행
랭킹 시스템: 실시간 데이터 집계와 조회 최적화

2. 비동기 처리를 통한 성능 최적화

주문 요청 → 즉시 응답 (동기)
↓
포인트 차감 → 결제 → 랭킹 업데이트 (비동기)

🔧 핵심 시스템별 설계 결정사항

1. 선착순 쿠폰 발급 시스템

설계 고민과 선택

문제: 대량의 동시 요청에서 정확한 수량 제어가 필요

기존 방식:

- DB 낙관적 락 : 간단하지만 충돌률 높음

수정 방식:

- Redis 기반 제어 : 높은 성능과 정확성 보장

최종 선택: Redis 하이브리드 방식

```
// 설계 결정의 핵심
@Transactional
public ResponseUserCoupon getCoupon(RequestUserCoupon requestUser
Coupon) {
    // 1. 비트맵으로 O(1) 중복 체크
    if (hasUserIssuedCouponBitmap(userId, couponId)) {
        throw new CustomException("쿠폰을 이미 발급받았음");
    }

    // 2. 원자적 카운터로 수량 제어
    Long currentCount = redisTemplate.opsForValue().increment(countKey);

    // 3. 발급 성공 시 데이터 저장
    setUserCouponIssuedBitmap(userId, couponId);
    saveIssuedTimestamp(userId, couponId);
}
```

핵심 설계 결정

1. Redis 데이터 구조 선택

```
비트맵 (Bitmap): 중복 발급 체크 (1 bit/user)
- 메모리 효율적: 100만 사용자 = 125KB
- O(1) 조회 성능

카운터 (Counter): 전체 발급 수량 관리
- 원자적 연산: INCR/DECR 보장
- 정확한 수량 제어

타임스탬프 (String): 배치 처리용 메타데이터
```

```
- 발급 시간 기록
- 효율적인 배치 추출
```

2. 실시간 인기상품 랭킹 시스템

설계 고민과 선택

문제: 실시간으로 변화하는 상품 인기도를 효율적으로 관리하고 조회

고려사항:

- 실시간성: 주문 즉시 랭킹 반영
- 조회 성능: 빠른 TOP N 조회
- 기간별 분리: 일간/주간 랭킹

최종 설계: Redis SortedSet + Spring Cache 조합

```
// 핵심 설계 로직
private void updatePopularityScore(Long productId, int quantity) {
    String dailyKey = getDailyPopularKey();    // popular:daily:20250821
    String weeklyKey = getWeeklyPopularKey();  // popular:weekly:2025:34

    // SortedSet으로 점수 누적
    redisTemplate.opsForZSet().incrementScore(dailyKey, productId.toString
(), quantity);
    redisTemplate.opsForZSet().incrementScore(weeklyKey, productId.toStri
ng(), quantity);

    // TTL 설정으로 자동 정리
    redisTemplate.expire(dailyKey, Duration.ofDays(1));
    redisTemplate.expire(weeklyKey, Duration.ofDays(7));
}
```

핵심 설계 결정

1. 이중 랭킹 시스템 설계

```
정적 랭킹 (TOP5):
- 데이터: DB sellQuantity (누적 판매량)
- 캐싱: Spring Cache (메모리)
```

```
- 특징: 안정적, 전체 기간 누적

동적 랭킹 (일간/주간):
- 데이터: Redis SortedSet (실시간 주문량)
- 업데이트: 주문 완료 시점
- 특징: 실시간, 기간별 트렌드
```

3. 비동기 주문 처리 시스템

설계 고민과 선택

문제: 주문 과정의 복잡한 비즈니스 로직을 안전하고 효율적으로 처리

고려사항:

- 재고 차감: 동시성 보장 필요
- 결제 처리: 외부 API 호출로 인한 지연
- 실패 시 보상: 복잡한 롤백 로직

최종 설계: Event-driven Architecture

```
// 주문 플로우 설계
1. 주문 생성 (동기) → OrderCreatedEvent
2. 포인트 차감 (비동기) → PointDeductedEvent
3. 결제 처리 (비동기) → PaymentCompletedEvent
4. 랭킹 업데이트 (비동기)
```

핵심 설계 결정

1. 동기/비동기 경계 설정

```
// 동기 처리: 즉시 응답이 필요한 핵심 로직
@Transactional(timeout = 10)
private Order createOrderCore(RequestOrder request) {
    // 재고 차감 - 반드시 성공해야 함
    productService.decreaseStock(request.productId(), request.requestQua
ntity());
    return orderRepository.save(order);
}
```

```
// 비동기 처리: 시간이 걸리지만 사용자 대기가 불필요한 로직
@EventListener
@Async("orderTaskExecutor")
public void handlePayment(PointDeductedEvent event) {
    // 결제 처리 - 시간이 걸려도 됨
    Payment result = paymentService.processPayment(...);
}
```

2. 분산 락을 통한 재고 관리

```
// Redisson 분산 락 적용
String lockKey = "product:stock:" + request.productId();
RLock lock = redissonClient.getLock(lockKey);

// 재시도 로직으로 안정성 확보
for (int i = 0; i < maxRetry; i++) {
    if (lock.tryLock(5, 5, TimeUnit.SECONDS)) {
        // 재고 차감 로직 실행
        break;
    }
    Thread.sleep(retryDelay);
}
```

3. 보상 트랜잭션 설계

```
// 실패 시 자동 롤백
private void handlePaymentFailure(PointDeductedEvent event) {
    pointService.refundPoints(event.getUserId(), event.getRequestPrice());
    productService.increaseStock(event.getProductId(), event.getRequestQu
antity());
    orderService.cancelOrder(event.getOrderId());
}
```

⚖️기술적 의사결정과 트레이드오프

1. 일관성 vs 성능

쿠폰 시스템의 선택

트레이드오프: 강한 일관성 vs 높은 성능

```
선택: Eventually Consistent
- Redis 우선 저장 → 즉시 응답
- 배치로 DB 동기화 → 최종 일관성 보장

장점: 빠른 응답 시간
단점: 일시적 불일치 가능성
```

의사결정 근거

```
1. 쿠폰 발급: 실시간성이 중요 (선착순)
2. 정산/통계: 정확성이 중요 (배치 처리)
3. 사용자 경험: 빠른 응답 필요

→ Redis + Batch 조합으로 두 마리 토끼 추적
```

2. 메모리 vs 저장공간

Redis 데이터 중복 저장

트레이드오프: 메모리 사용량 vs 조회 성능

```
결정: 3가지 데이터 구조 동시 사용
- Bitmap: 중복 체크용
- Counter: 수량 관리용
- Timestamp: 배치 처리용
```

3. 동기 vs 비동기

주문 처리 플로우 설계

트레이드오프: 응답 시간 vs 처리 완료 시간

```
동기 영역: 주문 생성 + 재고 차감
- 사용자 즉시 응답 필요
- 실패 시 명확한 에러 처리
```

비동기 영역: 결제 + 포인트 + 랭킹
- 처리 시간이 오래 걸림
- 사용자 대기 불필요

🚧 개발 과정에서의 도전과 해결책

쿠폰 시스템 개발 과정

DB 낙관적 락만 사용 :

- 100명 동시 요청 시 **99명이 재시도** 해야 함
- 재시도 로직 복잡성 증가
- 반복적인 DB접근

수정 과정

2차 시도:

- Redis Counter 도입

문제점: 중복 발급 체크 부재

3차 시도:

- Redis Bitmap 추가

배치 처리 문제: 전체 비트맵 스캔 비효율

최종 해결:

타임스탬프 키 추가

학습 내용

1. **점진적 개선의 중요성**: 단계별로 문제를 해결하며 최적해 도출
2. **트레이드오프 이해**: 성능과 복잡도 사이의 균형점 찾기
3. **실제 테스트의 중요성**: 동시성 테스트로 문제점 발견

2. 랭킹 시스템 개발 과정

캐싱 전략의 진화

1차 설계:

- 단순 DB 조회

문제점: 매 조회마다 DB 접근

2차 설계:

- Spring Cache 도입

문제점: 실시간성 부족 (주문 반영 지연)

3차 설계:

- Redis SortedSet 추가

최종 결과: 이중 랭킹 시스템

- **정적 TOP5**: Spring Cache 기반, 전체 누적 판매량 (안정적)
- **동적 랭킹**: Redis SortedSet 기반, 기간별 실시간 주문량 (트렌드)

학습 내용

1. **캐싱 계층화**: 용도별로 다른 캐시 전략 적용
2. **실시간성과 안정성**: 두 가지 요구사항을 동시에 만족
3. **데이터 구조 선택**: SortedSet의 강력함 체험

3. 비동기 처리 시스템 개발 과정

이벤트 체인 설계의 고민

초기 고민: 모든 로직을 동기 처리할지, 어디까지 비동기로 할지

의사결정 기준:

```
동기 처리 기준:
1. 사용자가 즉시 결과를 알아야 하는가?
2. 실패 시 명확한 에러 처리가 필요한가?
3. 다음 단계가 이 결과에 의존하는가?

비동기 처리 기준:
1. 처리 시간이 오래 걸리는가?
2. 외부 시스템 의존성이 있는가?
3. 사용자가 기다릴 필요가 없는가?
```

실패 처리 전략의 진화

초기: 단순 예외 발생
개선: 보상 트랜잭션 패턴 적용
최종: 세밀한 롤백 로직

학습 내용

1. **이벤트 체인 설계**: 각 단계별 책임 분리의 중요성
2. **보상 트랜잭션**: 분산 환경에서의 일관성 보장 방법
3. **비동기 처리의 복잡성**: 디버깅과 모니터링의 어려움

🎓 회고 및 학습 내용

주요 학습 내용

1. Redis 활용 능력 향상

학습 전: 단순 캐시 저장소로만 인식
학습 후: 다양한 데이터 구조의 특성과 활용법 이해

> Bitmap: 메모리 효율적인 불린 배열
> SortedSet: O(log n) 정렬된 집합
> Counter: 원자적 연산 보장

2. 동시성 제어 학습

- 낙관적 락 vs 비관적 락 vs 분산 락의 사용 시점
- Redis의 원자적 연산 활용법
- 동시성과 성능의 트레이드오프

3. 시스템 설계 관점의 변화

설계 전: 기능 구현에 집중
설계 후: 비기능적 요구사항(성능, 확장성, 일관성) 고려

> 단순 구현 → 요구사항 분석 → 트레이드오프 고려 → 최적 설계

4. 테스트 주도 개발의 중요성

동시성 테스트: 실제 문제 상황 재현을 통한 검증

아쉬웠던 점들

1. 초기 설계의 불완전성

문제: 단계별로 문제를 발견하며 수정하는 방식
개선방향: 초기에 더 많은 시나리오 고려 필요

2. 모니터링 부족

현재: 로그 기반 모니터링
필요: 메트릭 기반 실시간 모니터링

```
// 추가 필요한 모니터링
- Redis 메모리 사용량
- 이벤트 처리 지연 시간
- 배치 처리 성공률
```

3. 에러 처리의 일관성

문제: 시스템별로 다른 에러 처리 방식
개선방향: 통일된 에러 처리 전략 필요

잘했던 점들

1. 점진적 개선 접근

- 1차 → 2차 → 3차 → 최종 으로 단계별 개선
- 각 단계에서 명확한 문제점 파악 및 해결

2. 실제 테스트를 통한 검증

- 동시성 테스트로 실제 문제 발견
- 성능 테스트로 개선 효과 정량적 측정

3. 비즈니스 요구사항 우선 고려

- 기술적 완벽성보다 실제 요구사항에 맞는 설계
- 사용자 경험을 우선시한 동기/비동기 경계 설정