

예상문제

대모듈	소모듈	기출문제	공부 주제	참고자료	관련 예상문제
Software Architect ure 핵심 (3문항 / 11점)	기반 솔루션 분류/정의				
	기반 솔루션 주요 기능/장단점	01. Software Architecture 핵심 > 기반 솔루션 주요기능/장단점	<ul style="list-style-type: none"> I/F 기반의 Data동기화 문제 → Data File에 대한 동기화? DB Data 동기화(EAI, Kafka) Kafka/Redis 등 Backing 서비스 Clustering 및 성능관련된 특징에 대한 이해: Pub/Sub, 비동기에 따른 순차 거래 특징, 가용성 처리 방식 	I/F 솔루션 카프카 구성도 : https://my-develop-note.tistory.com/266	<ul style="list-style-type: none"> 예상문제 #5 - I/F 기반 Data 동기화 - Software Architect - KTDS KMS 예상문제 #6 - Kafka/Redis 등 Backing 서비스 Clustering 및 성능관련된 특징에 대한 이해 - Software Architect - KTDS KMS
	JVM 구조 및 동작 특성	02. Software Architecture 핵심 > JVM 구조 및 동작 특성 20. Software Architecture 핵심 > JVM 구조 및 동작 특성	<ul style="list-style-type: none"> Java GC 개념과 Option설정 (Kubernetes에서 설정하는 방법) Java GC 발생 시 성능 개선을 위한 Issue 원인 및 대응 방안 	JVM의 GC 발생 조건 및 동작 방식 JVM 옵션 JDK 버전별 GC 변화	<ul style="list-style-type: none"> 예상 문제 #2 - JVM - Software Architect - KTDS KMS
Software Architect ure 설계/구축 (6문항 22점)	시스템 유형별 참조 SW 아키텍처				
	SW 아키텍처 설계	10. Software Architecture 설계/구축 > SW 아키텍처 설계		개발 환경 및 구성의 트렌드 변화	

대모듈	소모듈	기출문제	공부 주제	참고자료	관련 예상문제
	Framework 적용 (환경, 온라인, 배치, 연계 등)	03. Software Architecture 설계/구축 > Framework 적용 (환경, 온라인, 배치, 연계 등) 25. 주관식 문제	<ul style="list-style-type: none"> Framework의 역할 및 아키텍처 고려 사항 → WAS 기본 성능 개선: DBCP, TCP Connection Pool, JVM Memory Frontend, ECMA(JavaScript) Spec 이해: Browser 내 객체 Scope, HTMLDom 구조 Reactive Programming(비동기) 개념 이해 Springboot Process/ Thread 이해와 Block/ NonBlock 개념 	비동기 Adaptor 작성 02. 제로카피_프론트엔드_트렌드_최종_v1.0.0_박종명.pptx	<ul style="list-style-type: none"> 예상문제 #4 - 수정중 - Software Architect - KTDS KMS
	개발 표준 수립 및 가이드	04. Software Architecture 설계/구축 > 개발 표준 수립 및 가이드	<ul style="list-style-type: none"> Spring 기반의 공통 로직 처리 방법: log, 공통 정보 처리 → 감이 없음 ==> https://www.baeldung.com/mdc-in-log4j-2-logback 	호출 관계에 대한 이해도	

대모듈	소모듈	기출문제	공부 주제	참고자료	관련 예상문제
	개발/배포/모니터링 체계 구축	06. Software Architecture 설계/구축 > 개발/배포/모니터링 체계 구축 07. Software Architecture 설계/구축 > 개발/배포/모니터링 체계 구축		DB 계정정보 암호화 설정	
	솔루션 설치, 구성 및 마이그레이션	05. Software Architecture 설계/구축 > 솔루션 설치, 구성 및 마이그레이션	<ul style="list-style-type: none"> Web 서버 설정: 성능 개선을 위한 설정에 대한 이해, 버퍼 사이즈 등 → nginx, tomcat, apache 	Tomcat JDBC Connection Pool	
	보안(보안 취약점, 권한 관리, 암호화 등)				
	가용성 및 효율성				
Software Architecture 운영/문제해결 (6문항 32점)	모니터링(항목, 지표, 도구 활용)	09. Software Architecture 운영/문제해결 > 모니터링 (항목, 지표, 도구활용)		MSA 기반 환경의 모니터링 도구 활용	

대모듈	소모듈	기출문제	공부 주제	참고자료	관련 예상문제
	성능개선 및 문제해결	<p>11. Software Architecture 운영/문제해결 > 성능 개선 및 문제해결</p> <p>12. Software Architecture 운영/문제해결 > Lighthouse에 대한 이해</p>	<ul style="list-style-type: none"> CDN의 기능과 활용/특징에 대한 이해 (TA) Linux File system에 대한 이해와 Kernel Resource 이해 (성능에 대한 고려를 위한 내용) <ul style="list-style-type: none"> → Linux File Descriptor 개념 OS Ulimit 연관 Linux Resource에 관리 방식에 대한 이해: Kernel/User 모드와 Java I/O 처리에 할당되는 Resource 처리 영역 등 개념적인 성능 개선 고려 사항 : https://learn.microsoft.com/ko-kr/troubleshoot/azure/virtual-machines/linux/troubleshoot-performance-bottlenecks-linux 	<p>Performance Profiling에 대한 이해</p> <p>Lighthouse에 대한 이해</p>	
	로그/덤프 생성, 관리 및 분석	<p>08. Software Architecture 운영/문제해결 > 로그/덤프 생성, 관리 및 분석</p> <p>24. Software Architecture 운영/문제해결 > 로그/덤프 생성, 관리 및 분석</p>		분산 로그 추적	

대모듈	소모듈	기출문제	공부 주제	참고자료	관련 예상문제
Software Architect ure 환경 (4문항 14 점)	프로토콜, 네트 워크 및 방화벽	13. Software Architecture 환경 > 프로토 콜, 네트워크 및 방화벽 14. Software Architecture 환경 > 프로토 콜, 네트워크 및 방화벽	<ul style="list-style-type: none"> HTTP Protocol 에 대 한 이해: HTTP Method, Header 정보의 기능 이해 	TIME_WAIT 상태 HTTP 응답코드 HttpClient 로깅 HTTP Protocol 에 대한 이해: HTTP Method, Header 정보의 기능 이해	
	캐시 서버 및 검 색 엔진	22. Software Architecture 환경 > 캐시 서버 및 검색 엔진		캐싱 메커니즘	
	Software 연관 아키텍처	15. Software Architecture 환경> Software 연 관 아키텍처		트랜잭션 처리와 EAI 호출	

대모듈	소모듈	기출문제	공부 주제	참고자료	관련 예상문제
신기술 (6 문항 21 점)	Cloud Service	16. 신기술 > Cloud Service 17. 신기술> Microservice Architecture 21. 신기술 > Cloud Service 23. 신기술 >Cloud Service	<ul style="list-style-type: none"> 대량파일처리를 위한 WAS, Kubernetes Ingress 설정 방법 Spring Cloud 개별 컴포넌트 역할 이해: Config, Consul, Contract 등 개념과 특징이해 Kubernetes Pod 동작 방식에 대한 이해: QoS 개념, Liveness/Readiness 등 LifeCycle 관리 개념 Ingress 개념 및 확장 기능 이해 (annotation을 통한 세부 기능 제어) Jenkins CI/CD 절차 	Kafka 설계 시 고려 사항 Kafka 병목현상에 대한 조치 컨테이너 프로젝트의 CICD 기본 프로세스 k8s 기본 명령어	
	Microservice Architecture	18. 신기술> Microservice Architecture 19. 신기술> Microservice Architecture	<ul style="list-style-type: none"> MSA, Service Mesh, Cloud Native 개념 이해 SAGA 패턴에 대한 개념 이해와 용어 정의 성능/패턴 관련 outer architecture JWT 개념 이해 	JWT 특징 Liveness, Readiness 점검	
	AI, AIOps, 클라우드 AI Platform				

예상문제 #1

KICE 소프트웨어 아키텍처 카테고리별 샘플 문제

1. I/F 기반의 Data 동기화 문제 (4점, 중급)

대내 시스템 간 실시간 데이터 동기화를 위해 EAI를 통한 트랜잭션 처리 방식을 설계 중이다. 아래 설명 중 옳바르지 않은 것은?

- ① EAI에서 트랜잭션 실패 시 보상 트랜잭션(Compensating Transaction)을 통해 데이터 정합성을 유지할 수 있다.
- ② 실시간 동기화에서는 2PC(Two-Phase Commit)보다 SAGA 패턴이 더 적합하다.
- ③ EAI 허브를 통한 동기화는 각 시스템이 직접 연결되는 방식보다 결합도를 낮춘다.
- ④ 트랜잭션 ID를 통해 분산된 시스템 간의 데이터 일관성을 추적할 수 있다.
- ⑤ EAI에서 동기 방식 호출 시 타임아웃이 발생하면 자동으로 비동기 방식으로 전환된다.

정답: 5 해설: EAI에서 타임아웃 발생 시 자동으로 호출 방식이 변경되지 않으며, 이는 설계 시점에서 결정되어야 한다.

2. Reactive Programming 개념 이해 (3점, 하급)

Spring WebFlux를 사용한 비동기 처리에서 다음 중 올바른 설명은?

- ① Flux는 0-1개의 데이터를 처리하고, Mono는 0-N개의 데이터를 처리한다.
- ② 비동기 처리에서는 스레드 블로킹이 발생하지 않으므로 더 많은 동시 요청을 처리할 수 있다.
- ③ WebFlux는 내부적으로 Servlet Container를 사용한다.
- ④ 비동기 처리 시 예외 처리는 try-catch 블록으로만 가능하다.
- ⑤ CompletableFuture와 Reactor는 동일한 개념이다.

정답: 2 해설: 비동기 논블로킹 방식은 스레드가 블로킹되지 않아 적은 수의 스레드로 더 많은 요청을 처리할 수 있다. Flux는 0-N개, Mono는 0-1개 데이터를 처리한다.

3. MSA Service Mesh 개념 (4점, 중급)

Istio를 활용한 Service Mesh 환경에서 다음 중 잘못된 설명은?

- ① Envoy Proxy는 각 서비스의 사이드카로 배포되어 네트워크 트래픽을 제어한다.
- ② Istio-proxy를 통해 서비스 간 mTLS 암호화가 자동으로 적용된다.
- ③ Circuit Breaker 기능을 통해 장애 전파를 방지할 수 있다.
- ④ Service Mesh를 적용하면 기존 애플리케이션 코드를 대폭 수정해야 한다.
- ⑤ 트래픽 라우팅 규칙을 통해 카나리 배포가 가능하다.

정답: 4 해설: Service Mesh는 애플리케이션 코드 수정 없이 인프라 레벨에서 서비스 간 통신을 제어하는 것이 핵심 장점이다.

4. WAS 성능 개선 - DBCP 설정 (4점, 상급)

Tomcat에서 DBCP 설정을 통한 성능 최적화 시 다음 중 잘못된 설정은?

```
<Resource name="jdbc/myDB"
  type="javax.sql.DataSource"
  driverClassName="oracle.jdbc.driver.OracleDriver"
  url="jdbc:oracle:thin:@localhost:1521:XE"
  username="user" password="pass"
  maxTotal="50"
  maxIdle="30"
  minIdle="10"
  initialSize="5"
  validationQuery="SELECT 1 FROM DUAL"
  testOnBorrow="true"
  testWhileIdle="false"
  timeBetweenEvictionRunsMillis="30000"
  minEvictableIdleTimeMillis="60000"
/>
```

- ① validationQuery를 설정하여 연결 유효성을 검증하고 있다.
- ② testWhileIdle를 false로 설정하여 유휴 연결 검증을 비활성화했다.
- ③ maxTotal이 maxIdle보다 큰 값으로 설정되어 있다.
- ④ minEvictableIdleTimeMillis가 timeBetweenEvictionRunsMillis보다 크게 설정되어 있다.
- ⑤ initialSize가 minIdle보다 작게 설정되어 있다.

정답: 2 해설: testWhileIdle를 false로 설정하면 유휴 연결 검증이 비활성화되어 끊어진 연결을 사용할 위험이 있다. true로 설정해야 한다.

5. JVM Memory 튜닝 (4점, 상급)

Kubernetes 환경에서 Java 애플리케이션의 메모리 설정에 대한 설명 중 올바른 것은?

```
resources:
  limits:
    memory: "2Gi"
  requests:
    memory: "2Gi"
env:
- name: JAVA_OPTS
  value: "-XX:MaxRAMPercentage=75.0 -XX:InitialRAMPercentage=50.0"
```

- ① MaxRAMPercentage는 컨테이너 메모리 한계의 75%를 힙 메모리로 할당한다.
- ② InitialRAMPercentage는 애플리케이션 시작 시 힙 메모리 할당량을 결정한다.
- ③ 컨테이너 메모리가 2GB이므로 힙 메모리는 정확히 1.5GB가 할당된다.
- ④ Non-heap 메모리는 별도로 고려할 필요가 없다.
- ⑤ -Xms와 -Xmx 옵션이 더 정확한 메모리 제어를 제공한다.

정답: 1 해설: MaxRAMPercentage는 컨테이너 메모리 한계 대비 힙 메모리 비율을 설정한다. Non-heap 메모리도 함께 고려해야 하므로 75% 설정이 적절하다.

6. Frontend Browser 객체 Scope (3점, 하급)

다음 JavaScript 코드의 실행 결과로 올바른 것은?

```
var x = 1;
function outer() {
  var x = 2;
  function inner() {
    console.log(x);
    var x = 3;
    console.log(x);
  }
  inner();
  console.log(x);
}
outer();
console.log(x);
```

- ① 2, 3, 2, 1
- ② undefined, 3, 2, 1
- ③ 2, 3, 3, 1
- ④ undefined, 3, 3, 1
- ⑤ ReferenceError 발생

정답: 2 해설: 호이스팅으로 인해 inner 함수 내의 var x 선언이 상단으로 올라가지만 할당은 나중에 되므로 첫 번째 console.log는 undefined를 출력한다.

7. Kubernetes Pod QoS 개념 (3점, 하급)

Kubernetes에서 Pod의 QoS 클래스에 대한 설명 중 옳은 것은?

- ① Guaranteed 클래스는 requests와 limits가 동일하지 않아도 된다.
- ② Burstable 클래스는 requests만 설정되고 limits는 설정되지 않은 경우다.
- ③ BestEffort 클래스는 requests와 limits가 모두 설정되지 않은 경우다.
- ④ QoS 클래스는 스케줄링에만 영향을 미치고 리소스 회수에는 영향을 미치지 않는다.
- ⑤ 모든 컨테이너는 동일한 QoS 클래스를 가져야 한다.

정답: 3 해설: BestEffort 클래스는 CPU/메모리에 대한 requests와 limits가 모두 설정되지 않은 경우에 해당한다.

8. Spring Cloud Config 개념 (3점, 하급)

Spring Cloud Config Server에 대한 설명 중 잘못된 것은?

- ① Git Repository를 백엔드 저장소로 사용할 수 있다.
- ② 설정 변경 시 애플리케이션을 재시작하지 않고 @RefreshScope를 통해 반영할 수 있다.
- ③ 암호화된 속성 값을 {cipher}로 표시하여 저장할 수 있다.
- ④ Config Client는 부팅 시점에만 설정을 가져올 수 있다.
- ⑤ 환경별(dev, prod) 설정 파일을 분리하여 관리할 수 있다.

정답: 4 해설: Config Client는 부팅 시점뿐만 아니라 /actuator/refresh 엔드포인트를 통해 런타임에도 설정을 갱신할 수 있다.

9. CDN 기능과 활용 (3점, 하급)

CDN(Content Delivery Network) 활용에 대한 설명 중 올바르지 않은 것은?

- ① 정적 콘텐츠 캐싱을 통해 오리진 서버의 부하를 줄일 수 있다.
- ② 사용자와 지리적으로 가까운 엣지 서버에서 콘텐츠를 제공한다.
- ③ 동적 콘텐츠는 CDN을 통해 캐싱할 수 없다.
- ④ Cache-Control 헤더를 통해 캐시 정책을 제어할 수 있다.
- ⑤ DDoS 공격 완화 기능을 제공할 수 있다.

정답: 3 해설: 최신 CDN은 동적 콘텐츠에 대해서도 엣지 컴퓨팅, API 캐싱 등을 통해 성능 향상을 제공할 수 있다.

10. HTTP Protocol 심화 (4점, 중급)

HTTP/2 프로토콜의 특징에 대한 설명 중 잘못된 것은?

- ① 멀티플렉싱을 통해 하나의 커넥션에서 여러 요청을 동시에 처리할 수 있다.
- ② 서버 푸시 기능을 통해 클라이언트 요청 전에 리소스를 전송할 수 있다.
- ③ HPACK 압축을 통해 헤더 크기를 줄인다.
- ④ 텍스트 기반 프로토콜이므로 디버깅이 용이하다.
- ⑤ Stream 우선순위를 설정하여 중요한 리소스를 먼저 전송할 수 있다.

정답: 4 해설: HTTP/2는 바이너리 프로토콜이므로 HTTP/1.1과 달리 텍스트 기반이 아니다. 이로 인해 파싱 효율성은 높지만 디버깅은 더 어려워진다.

11. SAGA 패턴 개념 (4점, 중급)

MSA 환경에서 분산 트랜잭션 처리를 위한 SAGA 패턴에 대한 설명으로 적절하지 않은 것은?

- ① Choreography 방식에서는 각 서비스가 자율적으로 다음 단계를 결정한다.
- ② Orchestration 방식에서는 중앙 조정자가 트랜잭션 흐름을 제어한다.
- ③ 보상 트랜잭션을 통해 실패한 단계를 롤백한다.
- ④ ACID 특성을 완전히 보장한다.
- ⑤ 이벤트 기반 아키텍처와 잘 결합된다.

정답: 4 해설: SAGA 패턴은 BASE(Basically Available, Soft state, Eventual consistency) 특성을 따르며, ACID의 격리성과 일관성을 완전히 보장하지 않는다.

12. JWT 보안 고려사항 (3점, 하급)

JWT(JSON Web Token) 사용 시 보안 고려사항으로 올바르지 않은 것은?

- ① JWT는 서명되어 있어 위변조가 불가능하므로 민감한 정보를 포함해도 안전하다.
- ② Refresh Token을 별도로 관리하여 Access Token의 유효기간을 짧게 가져간다.

- ③ 서버 측에서 토큰 무효화 목록(blacklist)을 관리할 수 있다.
- ④ HTTPS를 사용하여 토큰 전송 중 탈취를 방지한다.
- ⑤ 토큰에는 사용자 식별 정보만 포함하고 권한 정보는 서버에서 조회한다.

정답: 1 해설: JWT는 서명으로 위변조는 방지되지만 Base64 인코딩된 페이로드는 쉽게 디코딩 가능하므로 민감한 정보를 포함해서는 안 된다.

예상 문제 #2 - JVM

문제 1 [4점, 중]

다음은 운영 중인 Spring Boot 애플리케이션의 JVM 모니터링 결과이다. jstat -gc 명령어 결과를 분석하여 현재 상황에 대한 설명 중 옳지 않은 것을 고르시오.

bash

```
$ jstat -gc 25847 5s
S0C    S1C    S0U    S1U    EC    EU    OC    OU    MC    MU    CCSC
CCSU   YGC   YGCT   FGC   FGCT   GCT
17472.0 17472.0 0.0    16234.1 139648.0 139580.2 349568.0 342156.8 21248.0 20074.3
2560.0 2361.3 1847   12.456 23    8.923 21.379
17472.0 17472.0 0.0    16845.2 139648.0 5234.1 349568.0 348901.2 21248.0 20134.1
2560.0 2387.1 1851   12.478 24    9.456 21.934
17472.0 17472.0 0.0    17234.5 139648.0 45123.8 349568.0 348901.2 21248.0 20198.7
2560.0 2401.9 1855   12.501 24    9.456 21.957
```

- ① Young Generation에서 Minor GC가 정상적으로 동작하고 있으며, Eden 영역이 가득 찰 때마다 GC가 발생하고 있다.
- ② Old Generation 사용률이 약 99%에 달하여 곧 Full GC가 빈번하게 발생할 위험이 있다.
- ③ Survivor 영역 중 S1이 사용되고 있어 최근 Minor GC에서 S0에서 S1으로 객체가 이동했음을 알 수 있다.
- ④ 15초 동안 Minor GC가 8회 발생하여 GC 발생 빈도가 매우 높은 상태이다.
- ⑤ Metaspace 사용률이 약 94%로 높아 Class 로딩 관련 메모리 부족이 우려된다.

문제 2 [4점, 상]

다음은 Kubernetes 환경에서 Spring Boot 애플리케이션의 JVM 설정에 대한 설명이다. 컨테이너 리소스 제한과 JVM 힙 메모리 설정의 조합 중 문제가 될 수 있는 것을 고르시오.

yaml

```
# Pod 리소스 설정
resources:
  limits:
    memory: "2Gi"
    cpu: "1"
  requests:
    memory: "1Gi"
    cpu: "0.5"
```

- ① JVM 옵션: `-Xms1g -Xmx1.5g`
- ② JVM 옵션: `-XX:MaxRAMPercentage=75.0 -XX:InitialRAMPercentage=50.0`
- ③ JVM 옵션: `-Xms512m -Xmx1792m`
- ④ JVM 옵션: `-XX:MaxRAMPercentage=95.0 -XX:MinRAMPercentage=80.0`

⑤ JVM 옵션: `-Xms1g -Xmx1g -XX:+UseContainerSupport`

문제 3 [3점, 하]

웹 애플리케이션에서 OutOfMemoryError가 발생한 후 Heap Dump를 분석한 결과 다음과 같은 정보를 얻었다. 이 상황에서 메모리 누수의 가장 가능성이 높은 원인을 고르시오.

[Eclipse MAT Dominator Tree 분석 결과]

Class Name	Objects	Retained Heap
java.util.concurrent.ConcurrentHashMap	1	156,789,456 bytes
└─ java.util.concurrent.ConcurrentHashMap\$Node[]	1	145,234,567 bytes
└─ com.example.UserSession	45,678	98,765,432 bytes
└─ java.sql.PreparedStatement	45,678	28,456,789 bytes
└─ org.apache.http.impl.client.CloseableHttpClient	12,345	18,567,890 bytes

[GC Root 분석]

Thread "http-nio-8080-exec-1"

└─ Static Variable: SessionManager.activeSessions
 └─ ConcurrentHashMap

- ① PreparedStatement 객체들이 close()되지 않아 누적되고 있다.
- ② HTTP 커넥션이 정상적으로 해제되지 않아 메모리 누수가 발생했다.
- ③ 사용자 세션이 만료되어도 SessionManager에서 제거되지 않고 있다.
- ④ ConcurrentHashMap의 내부 배열이 동적으로 확장되면서 메모리를 과도하게 사용하고 있다.
- ⑤ 스레드 풀의 스레드들이 종료되지 않아 메모리가 해제되지 않고 있다.

문제 4 [4점, 중]

다음은 운영 환경에서 GC 튜닝을 위해 설정한 JVM 옵션들이다. 각 애플리케이션 특성에 맞는 가장 적절한 JVM 옵션 조합을 고르시오.

[애플리케이션 A] - 실시간 트레이딩 시스템, 응답시간 100ms 이내 필수

[애플리케이션 B] - 대용량 배치 처리, 처리량 최대화 필요

[애플리케이션 C] - 일반 웹 서비스, 8GB 힙 메모리 사용

옵션 조합	JVM 설정
①	<code>-XX:+UseG1GC -XX:MaxGCPauseMillis=50</code>
②	<code>-XX:+UseParallelGC -XX:ParallelGCThreads=8</code>

옵션 조합	JVM 설정
③	-XX:+UseZGC -XX:+UnlockExperimentalVMOptions
④	-XX:+UseG1GC -XX:MaxGCPauseMillis=200
⑤	-XX:+UseSerialGC

애플리케이션 A, B, C에 가장 적절한 조합은?

- ① A-①, B-②, C-④
- ② A-③, B-②, C-①
- ③ A-①, B-④, C-②
- ④ A-③, B-②, C-④
- ⑤ A-②, B-③, C-①

문제 5 [4점, 상]

다음은 Spring Boot 애플리케이션에서 메모리 누수를 방지하기 위한 코드이다. 잘못 구현된 부분과 올바른 수정 방안을 찾으시오.

java

```
@RestController
public class UserController {

    private static final Map<String, User> userCache = new ConcurrentHashMap<>();
    private final ThreadLocal<UserContext> userContext = new ThreadLocal<>();

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody LoginRequest request) {
        User user = authenticateUser(request);

        // 사용자 정보 캐싱
        userCache.put(user.getId(), user);

        // 스레드 로컬 설정
        UserContext context = new UserContext(user);
        userContext.set(context);

        return ResponseEntity.ok("Login successful");
    }

    @PostMapping("/logout")
    public ResponseEntity<String> logout(@RequestParam String userId) {
        // 캐시에서 사용자 제거
        userCache.remove(userId);

        return ResponseEntity.ok("Logout successful");
    }
}
```

```

    }

    @GetMapping("/profile")
    public ResponseEntity<User> getProfile() {
        UserContext context = userContext.get();
        if (context != null) {
            return ResponseEntity.ok(context.getUser());
        }
        return ResponseEntity.notFound().build();
    }
}

```

문제점과 해결방안:

- ① ThreadLocal이 static으로 선언되어 메모리 누수 발생 → 인스턴스 변수로 변경
- ② logout 시 ThreadLocal 정리 누락 → `userContext.remove()` 추가
- ③ 사용자 캐시 크기 제한 없음 → LRU 캐시로 변경
- ④ 동시성 제어 부족 → synchronized 블록 추가
- ⑤ 예외 처리 부족 → try-catch 블록 추가

문제 6 [3점, 하]

다음 JVM 메모리 영역에 대한 설명 중 틀린 것을 고르시오.

- ① **Method Area(Metaspace)**는 클래스 메타데이터, 상수 풀, 메소드 바이트코드가 저장되는 영역이다.
- ② **PC Register**는 각 스레드가 현재 실행 중인 JVM 명령어의 주소를 저장한다.
- ③ **Native Method Stack**은 JNI를 통해 호출되는 C/C++ 네이티브 메소드의 스택 정보를 저장한다.
- ④ **Direct Memory**는 Heap 영역에 포함되며 GC의 대상이 된다.
- ⑤ **Stack Frame**은 메소드 호출 시마다 생성되며 지역변수, 피연산자 스택, 메소드 리턴 주소를 포함한다.

정답 및 해설(Drag 하면 보여요!)

-
-
-

```
userContext.remove()
```


예상문제 #3

[CK사 Transactional Outbox Pattern 기반 온라인/배치 Kafka 메시지 현재 처리 흐름]



[처리흐름 설명]

1. 온라인과 배치 프로그램에서 비즈니스 로직에 따라 DB의 업무 데이터 CUD 수행
2. Kafka 전송용 메시지, 대상 Topic 정보 등을 Outbox Table에 Insert 수행 (1번과 2번을 하나의 트랜잭션으로 처리)
3. Kafka 메시지 조회 후,
4. Kafka에 메시지를 Publish 함
5. Kafka에 Publish된 메시지의 전송 여부 Flag 컬럼 Update
6. 온라인 서비스에서 메시지 구독 (Subscribe)후, Biz Logic 처리

다음 질문에 대해 구체적으로 답변하시오.

[과제 1] 제시된 고객 상황에서 온라인 서비스 A의 Kafka 메시지 처리 지연이 발생하는 구간 및 원인을 식별하고 개선 방안을 제시하시오.

- 고객 환경/요구사항 내역을 확인하여 작성하시오.
- 명시하지 않은 부분 중 필요하다고 판단되는 부분은 전제/가정을 기술한 후 작성한다.
- 성능 지연이 발생할 수 있는 모든 구간에 대해 식별하여 원인 및 개선 방안을 상세히 기술한다.

1-1 아키텍처 문제 구간 및 원인 식별, 개선 방안 제시

- 온라인 서비스 A의 Kafka 메시지 처리 지연에 1) 영향을 미치는 지점을 모두 식별 하고 이에 대한 2) 원인을 분석하고 3) 개선 방안 및 4) 고려사항/전제사항을 각각 1가지 이상 구체적으로 제시하시오.

🔍 1. 성능 지연 발생 가능 구간 식별

주어진 처리 흐름을 기준으로, 성능 병목이 발생할 수 있는 구간은 다음과 같습니다:

구간설명성능 지연 발생 가능성

① CUD + Outbox Insert	온라인 서비스 A 또는 배치 프로그램에서 RDBMS에 업무 데이터 및 Outbox 데이터를 동시에 insert	낮음 (일반적인 DB 트랜잭션 성능에 의존)
-----------------------	---	--------------------------

③ Kafka 전송용 CDC 프로그램의 Outbox select	CDC 프로그램이 Outbox 테이블을 주기적으로 조회	중간~높음 (Polling 방식, 부하 집중 시 대기열 지연 발생 가능)
④ Kafka Publish	CDC 프로그램이 조회한 메시지를 Kafka에 Publish	높음 (Kafka Publish 부하 집중 시 레이턴시 증가 가능)
⑥ 온라인 서비스 B의 Subscribe 및 처리	서비스 B가 Kafka 메시지를 처리	중간~높음 (처리량 부족 시 메시지 적체 가능)

2. 주요 지연 원인 분석

✓ 원인 1: Outbox 테이블을 공유한 배치 프로그램의 대량 메시지 삽입

- 배치 프로그램이 대량 메시지를 Outbox 테이블에 Insert → 기존 CDC Polling 방식으로는 한정된 범위 내에서만 메시지 조회 및 처리 가능.
- 메시지 처리 순서에 따라 온라인 서비스 A의 메시지가 뒤로 밀림 → 우선순위 제어 불가

✓ 원인 2: Kafka 전송 CDC 프로그램의 Polling 방식 병목

- 단일 Thread로 Outbox 테이블을 주기적으로 조회 및 Publish → 처리량이 폭주할 경우 Publish 지연.
- Outbox 테이블의 Index 미흡 또는 메시지 상태(전송 여부) 필터링 비효율 → 조회 성능 저하

✓ 원인 3: Kafka Topic 내 메시지 처리 지연

- Kafka는 FIFO로 처리 → 대량 메시지 Publish로 인한 backlog 증가
- 구독 서비스 B의 처리 속도가 느릴 경우 메시지 대기열 누적

✓ 원인 4: 서비스 B의 Auto-scaling 한계

- Pod 수 자동 증설되더라도 초기 warm-up 시간 존재 → 갑작스런 부하에 즉시 대응 어려움

💡 3. 개선 방안 제안

🔥 A. CDC 프로그램 성능 개선

1. **Multi-threaded CDC 프로세스 도입**
 - 단일 Thread → 다중 Thread 또는 파티셔닝 기반 병렬 조회 구조로 개선
 - 메시지 유형, 우선순위 기준 파티셔닝하여 병렬 처리
2. **Outbox 테이블 인덱스 최적화**
 - status, created_at, message_type 등 자주 조회되는 컬럼에 인덱스 적용
 - 조회 쿼리 튜닝 (limit/offset 사용 최소화)
3. **CDC 처리량 모니터링 도구 도입**
 - Grafana + Prometheus 기반 모니터링 구축 → 처리량, 지연 시간 시각화

🔥 B. Outbox 테이블 구조 개선

1. **메시지 우선순위 필드 추가**
 - 배치 메시지: 낮은 우선순위 / 온라인 메시지: 높은 우선순위
 - CDC 프로그램에서 우선순위 기반으로 처리 순서 보장
2. **배치 메시지를 별도 Outbox 테이블로 분리**
 - 온라인 서비스 A용 Outbox와 배치용 Outbox를 분리하여 Publish 순서와 리소스 분리

🔥 C. Kafka Publish 처리 개선

1. **Kafka 전송 대기열 큐 또는 Buffer 처리**
 - Kafka Publish 전에 메모리 기반 큐 도입 → 배치 메시지 Rate-limiting 적용
2. **Kafka Topic 파티셔닝 재구성**
 - 단일 Topic 사용 대신 온라인용/배치용 파티션 분리
 - 서비스 B의 Kafka Consumer 그룹도 파티션별 구독하도록 재구성

🔥 D. 온라인 서비스 B 확장성 개선

1. **Pre-warmed Pod 유지**
 - Auto-scaling 외에 일정 수의 Warm 상태 Pod를 항상 유지하여 초기 응답성 확보
2. **메시지 유형별 Consumer Thread 분리**

- B 서비스 내에서 온라인 메시지 전용 Consumer, 배치 메시지 전용 Consumer 구분

✓ 결론 및 정리

구간병목 원인개선 방안

CDC 조회 및 Kafka Publish (③④)	단일 쓰레드, 비효율적 조회	다중 쓰레드, 파티셔닝, 인덱스 최적화
Outbox 메시지 적체	우선순위 없음, 배치/온라인 혼재	우선순위 필드 도입 또는 테이블 분리
Kafka Topic 처리	순차 처리 지연	파티션 분리, Queue 도입, Rate-limiting
서비스 B 처리	Auto-scaling 한계, Consumer 병목	Pre-warm Pod, Consumer 분리

또 다른 답변

✓ 1. 지연 발생 구간 및 영향 지점 식별

Kafka 메시지 지연의 영향 지점을 다음과 같이 구분할 수 있습니다:

구간영향 지점 설명

(1) Online 서비스 A → RDBMS	Outbox Table에 메시지 저장 트랜잭션
(2) Kafka 전송용 CDC 프로그램 → Outbox Table Select	메시지 조회 병목 가능
(3) Kafka 전송용 CDC 프로그램 → Kafka Publish	메시지 큐 적체 또는 Publish 속도 저하
(4) Kafka Topic 처리	Kafka Topic 처리 순서/지연 가능성
(5) 온라인 서비스 B → 처리 부하 집중	구독 후 처리 지연으로 인한 backpressure 유발 가능

✓ 2. 지연 원인 분석

1. Batch Program에 의한 메시지 폭주

- 월 1회 대량 메시지를 동시에 Insert → Outbox Table에 급격한 쓰기 부하.
- CDC 프로세스가 배치와 온라인 요청을 구분하지 못하고 순차 처리 → 온라인 메시지 지연.

2. CDC 프로세스 단일 쓰레드 병목

- Kafka 전송 CDC 프로그램이 단일 쓰레드로 동작하여 메시지 처리 속도가 급감.
- 배치 트래픽에 의해 온라인 서비스 메시지가 큐 뒤로 밀림.

3. Outbox Table Query 효율성 저하

- 대량 데이터로 인한 select latency 증가.
- 적절한 인덱스나 파티셔닝 부재 시 병목 심화.

4. Kafka Topic 처리 우선순위 미존재

- Kafka Topic에 들어온 메시지 간 우선순위가 없어 순서대로 처리됨.
- 온라인 서비스 메시지가 배치 메시지에 밀려 처리 지연.

✓ 3. 개선 방안 제시

1. Outbox Table에 메시지 우선순위 필드 추가

- priority 필드를 도입하여 CDC 프로그램이 온라인 메시지를 먼저 Publish 하도록 구성
- CDC 프로그램에 우선순위 기반 Query 및 처리 로직 구현

2. CDC 프로그램 병렬 처리 구조 도입

- 단일 쓰레드 → 멀티 쓰레드 or 비동기 큐 기반 구조로 개편
- 배치 메시지와 온라인 메시지를 분리한 큐/쓰레드로 운영

3. Buffer 계층 도입 (개선 후 구조 참조)

- Kafka 전송 전 중간 Buffer 도입으로 우선순위 재정렬 및 일시 저장
- 배치 메시지는 지연 가능성이 허용되므로 비동기 Publish로 전환

4. Outbox Table 파티셔닝

- 날짜, 메시지 유형 또는 우선순위 기반 파티셔닝으로 select 성능 향상
- 대용량 배치 메시지 삽입 시에도 온라인 메시지 조회 효율 보장

✓ 4. 고려사항 / 전제사항

항목내용

전제사항 1	Kafka 인프라 및 파라미터 (Consumer Lag, Broker 설정 등)은 최적화되어 있다고 가정
전제사항 2	배치 메시지는 약간의 전송 지연이 허용되며, 실시간성 요구도가 낮음
고려사항 1	메시지 우선순위 처리 로직 도입 시 CDC 프로그램 및 Outbox 구조 변경 필요
고려사항 2	Buffer 추가 시 메시지 유실 방지를 위한 재처리/Retry 정책 필요
고려사항 3	다중 쓰레드 CDC 운영 시 동시성 제어 및 트랜잭션 정합성 유지 고려 필요

아래는 **CDC 프로그램 병렬 처리 구조**와 **Buffer 기반 우선순위 처리 구조**의 개선 예시를 도식과 함께 설명드립니다.

✓ 1. 개선 구조 ①: CDC 프로그램 병렬 처리 구조 (멀티쓰레드 기반)

🔧 주요 아이디어

- Outbox Table에서 메시지 유형(온라인/배치)이나 우선순위에 따라 분리 처리
- 병렬 쓰레드 또는 Task Queue를 통해 다중 Publish 처리

🔄 흐름 구조

SCSS

복사편집

[Outbox Table] ── Online 메시지 (priority = HIGH) ── Batch 메시지 (priority = LOW) ↓ Select (우선순위 기반) [CDC Dispatcher] ── Thread 1 (Online 전용) → Kafka Topic ── Thread 2 (Batch 전용) → Kafka Topic

✓ 효과

- 온라인 메시지가 배치 메시지보다 우선 처리되어 지연 최소화
- 대량 메시지 병렬 처리로 전체 처리 속도 향상

✓ 2. 개선 구조 ②: Kafka 전 Publish 단계에 Buffer Layer 추가

주요 아이디어

- Outbox → Buffer Queue → Kafka 순서로 처리
- Buffer에서 우선순위 큐를 통해 메시지를 재정렬 후 Kafka Publish

흐름 구조

SCSS

복사편집

[Outbox Table] → [CDC 프로그램] ↓ ↓ Insert Push [Buffer Queue] (Priority 기반 정렬 처리) ↓ [Kafka Publisher] (멀티 쓰레드)

✓ 효과

- Kafka 처리 병목 시에도 온라인 메시지 우선 처리 가능
- Kafka Publish 로직을 Buffer에서 분리하여 CDC 프로그램 부담 감소

선택 가이드

상황추천 구조

기존 CDC 프로세스에 부담이 많음	구조② (Buffer 계층 추가)
단순 메시지 우선순위만 빠르게 적용	구조① (멀티쓰레드 CDC)
CDC 유지보수 인력이 적고 안정성 우선	구조①
대규모 배치 트래픽 분산 필요	구조②

예상문제 #4 - 수정중

- 프롬프트

Framework의 역할 및 아키텍처 고려 사항 → WAS 기본 성능 개선: DBCP, TCP Connection Pool, JVM Memory
 Frontend, ECMA(Javascript) Spec 이해: Browser 내 객체 Scope, HTMLDom 구조
 Reactive Programming(비동기) 개념 이해
 Springboot Process/Thread 이해와 Block/NonBlock 개념 관련해서 예상문제를 내볼래?

문제 1. WAS 성능 최적화 - DBCP 설정 [4점]

다음은 Tomcat 9 기반 프로젝트에서 DBCP(Database Connection Pool) 설정을 최적화하는 과정이다. 아래 설정에서 성능 문제를 야기할 수 있는 잘못된 설정을 고르시오.

xml

```
<Resource name="jdbc/testDB"
  auth="Container"
  type="javax.sql.DataSource"
  driverClassName="oracle.jdbc.driver.OracleDriver"
  url="jdbc:oracle:thin:@localhost:1521:XE"
  username="test"
  password="test123"
  maxTotal="50"
  maxIdle="20"
  minIdle="5"
  initialSize="10"
  maxWaitMillis="5000"
  testOnBorrow="false"
  testWhileIdle="true"
  testOnReturn="false"
  validationQuery="SELECT 1 FROM DUAL"
  timeBetweenEvictionRunsMillis="30000"
  minEvictableIdleTimeMillis="60000"
  removeAbandonedOnMaintenance="true"
  removeAbandonedTimeout="300"/>
```

- ① maxTotal이 50으로 설정되어 있어 동시 접속자가 많을 경우 병목이 발생할 수 있다
- ② testOnBorrow가 false로 설정되어 있어 끊어진 커넥션을 사용할 가능성이 있다
- ③ minEvictableIdleTimeMillis가 60초로 설정되어 DB timeout보다 길 경우 문제가 발생할 수 있다
- ④ removeAbandonedTimeout이 300초로 너무 길어 리소스 누수가 발생할 수 있다
- ⑤ maxWaitMillis가 5초로 설정되어 있어 커넥션 대기 시간이 너무 짧다

문제 2. JavaScript 스코프와 비동기 처리 [3점]

다음 JavaScript 코드의 실행 결과를 예측하시오.

javascript

```
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log('A: ' + i);
  }, 100);
}

for (let j = 0; j < 3; j++) {
  setTimeout(function() {
    console.log('B: ' + j);
  }, 100);
}

async function test() {
  for (let k = 0; k < 3; k++) {
    await new Promise(resolve => {
      setTimeout(() => {
        console.log('C: ' + k);
        resolve();
      }, 50);
    });
  }
}

test();
```

출력 순서와 값으로 올바른 것은?

- ① C: 0, C: 1, C: 2, A: 3, A: 3, A: 3, B: 0, B: 1, B: 2
- ② A: 3, A: 3, A: 3, B: 0, B: 1, B: 2, C: 0, C: 1, C: 2
- ③ C: 0, C: 1, C: 2, B: 0, B: 1, B: 2, A: 3, A: 3, A: 3
- ④ A: 0, A: 1, A: 2, B: 0, B: 1, B: 2, C: 0, C: 1, C: 2
- ⑤ 실행 순서가 브라우저마다 다르므로 예측할 수 없다

문제 3. Spring Boot 비동기 처리와 Thread Pool [4점]

다음은 Spring Boot에서 비동기 처리를 구현한 코드이다. 성능상 문제점과 해결방안으로 올바른 것을 고르시오.

java

```
@Service
public class AsyncService {
```

```

@Async
public CompletableFuture<String> processDataAsync(String data) {
    // 데이터 처리 로직 (약 2초 소요)
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return CompletableFuture.completedFuture("Processed: " + data);
}

@RestController
public class DataController {

    @Autowired
    private AsyncService asyncService;

    @GetMapping("/process")
    public String processData() {
        List<CompletableFuture<String>> futures = new ArrayList<>();

        for (int i = 0; i < 100; i++) {
            futures.add(asyncService.processDataAsync("data" + i));
        }

        // 모든 작업 완료 대기
        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();

        return "All tasks completed";
    }
}

```

문제점과 해결방안으로 올바른 것은?

- ① Thread Pool 설정이 없어 기본 SimpleAsyncTaskExecutor가 사용되어 Thread가 무제한 생성될 수 있다
- ② @Async 메서드에서 CompletableFuture를 반환하는 것이 잘못되었다
- ③ 100개의 작업을 동시에 실행하면 메모리 부족이 발생한다
- ④ join() 메서드 사용으로 인해 블로킹이 발생하여 비동기 처리 효과가 없다
- ⑤ @Async 어노테이션은 public 메서드에서만 동작하므로 문제없다

문제 4. TCP Connection Pool과 HTTP Keep-Alive [4점]

다음은 외부 API 호출을 위한 HTTP Client 설정이다. 성능 개선을 위한 설정 중 올바르지 않은 것을 고르시오.

java

```

@Configuration
public class HttpClientConfig {

```

```

@Bean
public RestTemplate restTemplate() {
    HttpComponentsClientHttpRequestFactory factory =
        new HttpComponentsClientHttpRequestFactory();

    // Connection Pool 설정
    PoolingHttpClientConnectionManager connectionManager =
        new PoolingHttpClientConnectionManager();
    connectionManager.setMaxTotal(200);
    connectionManager.setDefaultMaxPerRoute(20);

    // HTTP Client 설정
    CloseableHttpClient httpClient = HttpClients.custom()
        .setConnectionManager(connectionManager)
        .setKeepAliveStrategy((response, context) -> 30000) // 30초
        .setDefaultRequestConfig(RequestConfig.custom()
            .setConnectTimeout(5000)
            .setSocketTimeout(10000)
            .setConnectionRequestTimeout(3000)
            .build())
        .build();

    factory.setHttpClient(httpClient);
    return new RestTemplate(factory);
}

```

- ① setMaxTotal(200)으로 전체 커넥션 풀 크기를 제한했다
- ② setDefaultMaxPerRoute(20)으로 동일 호스트당 최대 커넥션 수를 제한했다
- ③ Keep-Alive 전략을 30초로 설정하여 커넥션 재사용성을 높였다
- ④ ConnectionRequestTimeout을 3초로 설정하여 풀에서 커넥션 대기 시간을 제한했다
- ⑤ SocketTimeout을 10초로 설정하여 응답 대기 시간을 제한했다

문제 5. JVM Memory 최적화와 GC 튜닝 [4점]

다음 JVM 옵션 설정에서 메모리 성능상 문제가 될 수 있는 설정을 고르시오.

bash

```

java -Xms2g -Xmx4g
    -XX:NewRatio=2
    -XX:SurvivorRatio=8
    -XX:MaxMetaspaceSize=256m
    -XX:+UseG1GC
    -XX:MaxGCPauseMillis=100
    -XX:G1HeapRegionSize=32m
    -XX:+PrintGC
    -XX:+PrintGCDetails
    MyApplication

```

- ① Xms와 Xmx가 다르게 설정되어 런타임에 힙 크기 조정으로 인한 오버헤드가 발생한다
- ② NewRatio=2로 설정되어 Young Generation이 전체 힙의 1/3을 차지하게 된다
- ③ G1HeapRegionSize가 32MB로 설정되어 있어 4GB 힙에는 너무 크다
- ④ MaxMetaspaceSize가 256MB로 제한되어 있어 클래스 로딩이 많은 애플리케이션에서 MetaspaceOOM이 발생할 수 있다
- ⑤ MaxGCPauseMillis=100ms는 G1GC의 목표 일시정지 시간으로 적절하다

문제 6. Reactive Programming과 WebFlux [3점]

다음 Spring WebFlux 코드에서 발생할 수 있는 문제점을 고르시오.

java

```
@RestController
public class ReactiveController {

    @Autowired
    private UserRepository userRepository; // Blocking Repository

    @GetMapping("/users")
    public Flux<User> getUsers() {
        return Flux.fromIterable(userRepository.findAll()) // Blocking call
            .map(user -> {
                user.setLastAccess(new Date());
                userRepository.save(user); // Another blocking call
                return user;
            })
            .subscribeOn(Schedulers.boundedElastic());
    }
}
```

- ① Flux.fromIterable() 사용으로 메모리에 모든 데이터를 로드하게 된다
- ② blocking repository를 사용하여 reactive의 이점을 잃는다
- ③ subscribeOn(Schedulers.boundedElastic())이 잘못된 위치에 있다
- ④ map() 내부에서 side effect(save 호출)를 발생시키고 있다
- ⑤ 위의 모든 사항이 문제가 된다

문제 7. DOM 조작과 브라우저 성능 [3점]

다음 JavaScript 코드 중 DOM 조작 성능상 가장 효율적인 방법은?

javascript

```
// 방법 1
for (let i = 0; i < 1000; i++) {
```

```

const div = document.createElement('div');
div.textContent = 'Item ' + i;
document.body.appendChild(div);
}

// 방법 2
const fragment = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
  const div = document.createElement('div');
  div.textContent = 'Item ' + i;
  fragment.appendChild(div);
}
document.body.appendChild(fragment);

// 방법 3
let html = '';
for (let i = 0; i < 1000; i++) {
  html += '<div>Item ' + i + '</div>';
}
document.body.innerHTML += html;

// 방법 4
const divs = [];
for (let i = 0; i < 1000; i++) {
  divs.push('<div>Item ' + i + '</div>');
}
document.body.insertAdjacentHTML('beforeend', divs.join(''));

```

- ① 방법 1 - 직접 DOM 조작이 가장 빠르다
- ② 방법 2 - DocumentFragment 사용으로 reflow 최소화
- ③ 방법 3 - innerHTML 사용이 가장 간단하다
- ④ 방법 4 - insertAdjacentHTML과 join() 조합이 최적
- ⑤ 모든 방법의 성능은 동일하다

예상문제 #5 - I/F 기반 Data 동기화

1. I/F(Interface) 기본 개념

I/F의 정의와 분류

- **대내 I/F**: 동일 조직 내 시스템 간 연동
- **대외 I/F**: 외부 기관과의 시스템 연동
- **동기 I/F**: 실시간 요청-응답 방식
- **비동기 I/F**: 메시지 큐, 배치 방식

주요 I/F 솔루션

- **EAI (Enterprise Application Integration)**: 중앙집중형 연동
- **ESB (Enterprise Service Bus)**: 서비스 지향 연동
- **MQ (Message Queue)**: 비동기 메시지 처리
- **API Gateway**: RESTful API 관리

2. Data 동기화 핵심 원리

동기화 패턴

1. **Push 방식**: 원천 시스템이 변경 시 즉시 전송
2. **Pull 방식**: 대상 시스템이 주기적으로 조회
3. **Event-driven**: 이벤트 발생 시 자동 동기화
4. **Batch 방식**: 정해진 시간에 일괄 처리

데이터 일관성 보장

- **ACID 속성**: 원자성, 일관성, 고립성, 지속성
- **분산 트랜잭션**: 2PC(Two-Phase Commit), Saga 패턴
- **보상 트랜잭션**: 실패 시 롤백 처리

3. 동기화 구현 방식

CDC (Change Data Capture)

- 데이터베이스 변경 로그 기반 실시간 동기화
- 트리거, 바이너리 로그, 폴링 방식

ETL/ELT 프로세스

- **Extract:** 원천 데이터 추출
- **Transform:** 데이터 변환/정제
- **Load:** 대상 시스템 적재

메시지 기반 동기화

- Kafka, RabbitMQ 등을 통한 비동기 처리
- 이벤트 소싱, CQRS 패턴 적용

4. 동기화 시 고려사항

성능 최적화

- 배치 크기 조정
- 병렬 처리
- 압축 및 델타 동기화

장애 처리

- 재시도 메커니즘
- 데드레터 큐
- 모니터링 및 알림

보안

- 데이터 암호화
- 접근 권한 관리
- 감사 로그

핵심 포인트:

1. **EAI/ESB의 역할:** 단순 데이터 전달에 집중, 업무 로직 포함 금지
2. **Kafka 동기화:** 파티션 키 기반 순서 보장, 컨슈머 그룹 관리
3. **CDC 패턴:** 트리거, 바이너리 로그, 폴링 방식의 특징
4. **동시성 제어:** 분산 환경에서의 데이터 일관성 보장 방법
5. **Saga 패턴:** 분산 트랜잭션과 보상 트랜잭션의 개념

예상문제

문제 1 [4점]

다음은 대용량 거래 시스템에서 실시간 데이터 동기화를 위한 아키텍처 설계안이다. 이 설계에서 **가장 큰 문제점**을 고르시오.

설계안:

- 핵심 거래 시스템에서 거래 발생 시 동기 방식으로 5개 백오피스 시스템에 순차적으로 데이터 전송
- 각 백오피스 시스템 응답 시간: 평균 200ms, 최대 2초
- 네트워크 타임아웃: 5초
- 하나의 시스템이라도 실패 시 전체 거래 롤백

- ① 네트워크 타임아웃이 너무 짧게 설정되어 있다.
- ② 백오피스 시스템 장애 시 핵심 거래가 중단될 위험이 있다.
- ③ 동기화 대상 시스템이 너무 많아 확장성에 문제가 있다.
- ④ 순차 처리로 인해 총 응답시간이 길어진다.
- ⑤ 데이터 일관성을 보장하기 어렵다.

문제 2 [3점]

CDC(Change Data Capture) 구현 시 다음 상황에서 **가장 적합한** 방식을 고르시오.

요구사항:

- 24시간 무중단 서비스
- 초당 10,000건의 트랜잭션 처리
- 실시간 동기화 필요 (지연시간 1초 이내)
- 원천 시스템에 성능 영향 최소화

- ① Database Trigger 방식
- ② Polling 방식 (매초 변경 데이터 조회)
- ③ Binary Log Reading 방식
- ④ Application Level Change Tracking
- ⑤ Snapshot 기반 전체 데이터 비교

문제 3 [4점]

다음은 분산 환경에서 Order 생성 시 발생하는 데이터 동기화 과정이다. Saga 패턴 적용 시 3번 단계에서 실패했을 때의 올바른 보상 트랜잭션 순서를 고르시오.

정상 처리 순서:

1. 주문 생성 (Order Service)
2. 재고 차감 (Inventory Service)
3. 결제 처리 (Payment Service)
4. 배송 요청 (Shipping Service)

3번 단계 실패 시 보상 트랜잭션:

- ① 배송 취소 → 결제 취소 → 재고 복구 → 주문 취소
- ② 재고 복구 → 주문 취소
- ③ 주문 취소 → 재고 복구
- ④ 결제 취소 → 재고 복구 → 주문 취소
- ⑤ 모든 서비스에 대해 동시에 보상 트랜잭션 실행

문제 4 [4점]

Kafka를 이용한 실시간 데이터 동기화에서 다음 문제 상황의 **원인**과 **해결방안**을 서술하시오.

문제 상황:

- 고객 정보 변경 시 10개 마이크로서비스에 동기화
- Topic: customer-events (파티션 3개)
- Consumer Group: customer-sync (컨슈머 10개)
- 현상: 일부 컨슈머는 메시지를 받지 못하고 idle 상태 지속

원인: _____

해결방안: _____

문제 5 [3점]

다음 중 **Eventually Consistent** 모델에 대한 설명으로 틀린 것을 고르시오.

- ① 분산 시스템에서 모든 노드가 동시에 일관된 상태를 유지할 필요는 없다.
- ② 시간이 지나면서 결국 모든 노드가 동일한 상태로 수렴한다.
- ③ 네트워크 파티션 상황에서도 가용성을 보장할 수 있다.
- ④ 읽기 작업 시 항상 최신 데이터를 보장한다.
- ⑤ NoSQL 데이터베이스에서 주로 사용되는 일관성 모델이다.

정답 및 해설

문제 1 정답: ②

해설: 핵심 거래 시스템이 백오피스 시스템 장애에 의존적인 구조는 시스템 가용성을 크게 저해한다. 비동기 방식이나 이벤트 드리븐 아키텍처로 변경 필요.

문제 2 정답: ③

해설: Binary Log Reading 방식은 원천 시스템에 성능 영향 없이 실시간 변경 감지가 가능하며, 대용량 처리에 적합하다.

문제 3 정답: ②

해설: 3번 단계(결제 처리)에서 실패했으므로, 그 이전에 성공한 1, 2번 단계만 보상 트랜잭션이 필요하다. 역순으로 2번(재고 복구) → 1번(주문 취소) 순서로 실행.

문제 4 예시답안:

원인: 파티션 수(3개)보다 컨슈머 수(10개)가 많아서 일부 컨슈머가 파티션 할당을 받지 못함 **해결방안:** 토픽의 파티션 수를 컨슈머 수에 맞게 증가시키거나, 컨슈머 수를 파티션 수에 맞게 조정

문제 5 정답: ④

해설: Eventually Consistent 모델에서는 읽기 작업 시 최신 데이터가 아닌 이전 버전의 데이터를 읽을 수 있다(stale read). 강한 일관성이 필요한 경우에는 적합하지 않다.

핵심 출제 포인트:

1. **동기 vs 비동기 아키텍처:** 핵심 시스템과 부가 시스템 간의 의존성 설계
2. **CDC 구현 방식 선택:** 성능, 실시간성, 시스템 영향도를 고려한 최적 방식
3. **Saga 패턴의 보상 트랜잭션:** 실패 지점에 따른 정확한 롤백 순서
4. **Kafka 파티션과 컨슈머 관계:** 파티션 수와 컨슈머 수의 적정 비율

5. Eventually Consistent 모델: 분산 시스템의 일관성 개념

문제 1 [4점]

다음은 금융시스템에서 계좌잔고 동기화를 위한 트랜잭션 처리 시나리오이다. **문제점 2개**를 고르시오.

시나리오:

1. 고객이 ATM에서 10만원 출금 요청
 2. 계좌 시스템에서 잔고 차감 (200만원 → 190만원)
 3. EAI를 통해 다음 시스템들에 순차적으로 전송:
 - 회계 시스템 (거래내역 기록)
 - CRM 시스템 (고객 거래패턴 분석)
 - 리스크 시스템 (이상거래 탐지)
 4. 3번 과정에서 CRM 시스템 장애 발생
 5. 전체 트랜잭션 롤백하여 계좌잔고 200만원으로 복구
 6. 고객에게는 이미 현금 10만원 지급 완료
- ① CRM 시스템 장애로 인한 전체 트랜잭션 롤백
 - ② 동기식 처리로 인한 응답시간 지연
 - ③ 물리적 현금 지급과 논리적 잔고 차감의 불일치
 - ④ EAI에서 순차 처리로 인한 성능 저하
 - ⑤ 리스크 시스템 연동 순서의 부적절성

문제 2 [3점]

Spring Boot 애플리케이션에서 `**@Transactional(propagation = REQUIRES_NEW)**`를 사용한 메서드가 외부 시스템과 데이터 동기화를 수행할 때 발생할 수 있는 문제로 **가장 적절한** 것은?

java

```
@Service
public class OrderService {

    @Transactional
    public void createOrder(Order order) {
        orderRepository.save(order);
        sendToInventorySystem(order); // 외부 재고시스템 호출
        sendToPaymentSystem(order); // 외부 결제시스템 호출
    }

    @Transactional(propagation = REQUIRES_NEW)
    private void sendToInventorySystem(Order order) {
        // 외부 시스템 호출 로직
        inventoryClient.updateStock(order);
    }
}
```

- ① 외부 시스템 장애 시 주문 데이터가 롤백되지 않는다
- ② 데이터베이스 커넥션 풀이 고갈될 수 있다
- ③ 외부 시스템 응답시간이 길어진다
- ④ 동시성 제어가 불가능하다
- ⑤ 트랜잭션 격리 수준이 변경된다

문제 3 [4점]

다음은 MSA 환경에서 **이벤트 소싱(Event Sourcing)** 패턴을 적용한 주문 시스템이다. 아래 설명 중 **올바른** 것을 고르시오.

이벤트 스토어 내용:

```
Event 1: OrderCreated {orderId: "0001", customerId: "C001", amount: 50000}
Event 2: PaymentProcessed {orderId: "0001", paymentId: "P001", amount: 50000}
Event 3: InventoryReserved {orderId: "0001", productId: "PR001", quantity: 2}
Event 4: PaymentFailed {orderId: "0001", paymentId: "P001", reason: "카드한도초과"}
Event 5: OrderCancelled {orderId: "0001", reason: "결제실패"}
```

- ① 현재 주문 0001의 상태는 "결제완료"이다
- ② Event 4 발생 시 Event 2를 삭제해야 한다
- ③ 주문 상태 조회 시 모든 이벤트를 순차적으로 재생해야 한다
- ④ Event 5로 인해 재고 예약이 자동으로 해제된다
- ⑤ 이벤트는 수정이 불가능하므로 보상 이벤트로 처리해야 한다

문제 4 [4점]

Apache Kafka 환경에서 다음과 같은 **메시지 중복 처리** 문제가 발생했다. 원인과 해결방안을 서술하시오.

상황:

- 주문 생성 시 재고 차감 메시지를 Kafka로 전송
- 재고 서비스에서 동일한 주문에 대해 재고가 2번 차감됨
- 프로듀서 설정: `acks=all`, `retries=3`, `enable.idempotence=false`
- 컨슈머 설정: `enable.auto.commit=true`, `auto.commit.interval.ms=5000`

중복 처리 발생 원인: _____

해결방안 (2가지): _____

문제 5 [3점]

다음 중 ****분산 락(Distributed Lock)****을 사용해야 하는 상황으로 **가장 적절한** 것은?

- ① 단일 데이터베이스에서 트랜잭션 처리
- ② 읽기 전용 캐시 데이터 조회
- ③ 로그 파일 순차 기록
- ④ 여러 인스턴스에서 동일한 스케줄 작업 실행 방지
- ⑤ 메시지 큐의 순서 보장

정답 및 해설

문제 1 정답: ①, ③

해설:

- ①: CRM은 핵심 업무가 아니므로 장애로 인한 전체 롤백은 부적절
- ③: 물리적 현금 지급 후 논리적 롤백은 실제 손실 발생

문제 2 정답: ①

해설: REQUIRES_NEW 는 새로운 트랜잭션을 생성하므로, 외부 시스템 호출이 완료되면 즉시 커밋된다. 이후 메인 트랜잭션이 롤백되어도 재고 업데이트는 롤백되지 않아 데이터 불일치가 발생한다.

문제 3 정답: ⑤

해설: 이벤트 소싱에서는 이벤트가 불변이므로 수정/삭제 대신 보상 이벤트(OrderCancelled)를 추가한다. 현재 주문 상태는 "취소"이며, 상태 조회는 스냅샷을 활용할 수 있다.

문제 4 예시답안:

원인: Producer의 idempotence가 비활성화되어 있고, Consumer의 auto commit으로 인해 메시지 처리 완료 전 offset이 커밋되어 재처리 발생 **해결방안:**

1. Producer에서 `enable.idempotence=true` 설정
2. Consumer에서 수동 커밋 + 멍등성 처리 로직 구현

문제 5 정답: ④

해설: 여러 인스턴스가 동일한 스케줄 작업을 중복 실행하지 않도록 하려면 분산 락이 필요하다. Redis, Zookeeper, 데이터베이스 기반 락 등을 활용할 수 있다.

핵심 포인트:

1. **비즈니스 크리티컬 vs 논크리티컬 시스템:** 금융시스템에서 핵심 업무와 부가 업무의 분리 필요성
2. **Spring 트랜잭션 전파:** REQUIRES_NEW 의 함정과 분산 환경에서의 데이터 일관성 문제
3. **이벤트 소싱 패턴:** 이벤트의 불변성과 보상 이벤트를 통한 상태 변경 방식
4. **Kafka 중복 처리:** Exactly-once 보장을 위한 멍등성 설정과 컨슈머 커밋 전략
5. **분산 락:** 멀티 인스턴스 환경에서의 동시성 제어 필요 상황

문제 1 [4점]

다음은 마이크로서비스 환경에서 **분산 트랜잭션** 처리를 위한 패턴들이다. 각 패턴의 특징에 대한 설명 중 틀린 것을 2개 고르시오.

- ① **2PC(Two-Phase Commit)**: 모든 참여 서비스가 PREPARE 단계에서 준비 완료 응답을 보내야 COMMIT 단계가 실행된다.
- ② **Saga Choreography**: 각 서비스가 로컬 트랜잭션 완료 후 다음 서비스에게 직접 이벤트를 발행하는 방식이다.
- ③ **Saga Orchestration**: 중앙 조정자가 모든 서비스의 트랜잭션을 관리하므로 ACID 속성을 완벽하게 보장한다.
- ④ **TCC(Try-Confirm-Cancel)**: Try 단계에서 리소스 예약, Confirm에서 확정, Cancel에서 해제하는 3단계로 구성된다.
- ⑤ **Outbox Pattern**: 비즈니스 데이터와 이벤트를 같은 트랜잭션 내에서 저장하여 메시지 발행을 보장한다.

문제 2 [3점]

다음 **Redis**를 활용한 분산 락 구현에서 발생할 수 있는 문제 상황으로 가장 적절한 것은?

java

```
@Service
public class DistributedLockService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    public boolean acquireLock(String lockKey, int timeoutSeconds) {
        String lockValue = UUID.randomUUID().toString();
        Boolean result = redisTemplate.opsForValue()
            .setIfAbsent(lockKey, lockValue, Duration.ofSeconds(timeoutSeconds));
        return Boolean.TRUE.equals(result);
    }

    public void releaseLock(String lockKey, String lockValue) {
        String script =
            "if redis.call('get', KEYS[1]) == ARGV[1] then " +
            "return redis.call('del', KEYS[1]) " +
            "else return 0 end";
        redisTemplate.execute(new DefaultRedisScript<>(script, Long.class),
            Arrays.asList(lockKey), lockValue);
    }
}
```

- ① Redis 클러스터 환경에서 마스터-슬레이브 간 복제 지연으로 인한 락 중복 획득
- ② lockValue를 UUID로 생성하여 예측 가능성 문제 발생
- ③ Lua 스크립트 실행 중 메모리 부족으로 인한 성능 저하
- ④ setIfAbsent 연산의 원자성 보장 부족
- ⑤ 락 타임아웃 설정으로 인한 데드락 발생

문제 3 [4점]

이벤트 스토어(Event Store) 기반 시스템에서 다음과 같은 이벤트 시퀀스가 저장되었다. 이에 대한 분석 중 **올바른** 것을 고르시오.

json

이벤트 스토어:

```
[
  { "eventId": "e1", "aggregateId": "order-001", "eventType": "OrderCreated",
    "timestamp": "2025-01-01T10:00:00Z", "version": 1 },
  { "eventId": "e2", "aggregateId": "order-001", "eventType": "PaymentRequested",
    "timestamp": "2025-01-01T10:01:00Z", "version": 2 },
  { "eventId": "e3", "aggregateId": "order-001", "eventType": "InventoryReserved",
    "timestamp": "2025-01-01T10:02:00Z", "version": 3 },
  { "eventId": "e4", "aggregateId": "order-001", "eventType": "PaymentFailed",
    "timestamp": "2025-01-01T10:03:00Z", "version": 4 },
  { "eventId": "e5", "aggregateId": "order-001", "eventType": "InventoryReleased",
    "timestamp": "2025-01-01T10:04:00Z", "version": 5 }
]
```

- ① 현재 order-001의 상태는 "결제 실패"이므로 주문이 취소된 상태이다.
- ② 이벤트 e4 발생 시 e2 이벤트를 삭제하여 일관성을 유지해야 한다.
- ③ version 필드는 동시성 제어를 위한 낙관적 락으로 사용될 수 있다.
- ④ 이벤트 순서를 바꿔서 PaymentFailed를 먼저 처리하면 더 효율적이다.
- ⑤ 스냅샷 없이는 현재 상태 조회가 불가능하다.

문제 4 [4점]

다음 CDC(Change Data Capture) 구현에서 발생한 성능 문제의 원인과 해결방안을 서술하시오.

상황:

- MySQL 바이너리 로그 기반 CDC 구현
- 초당 5,000건의 트랜잭션 처리
- CDC 애플리케이션이 바이너리 로그를 실시간으로 읽어 Kafka로 전송
- 최근 CDC 지연이 30분까지 발생하며 메모리 사용률이 90% 초과

CDC 애플리케이션 구성:

- 단일 스레드로 바이너리 로그 순차 처리
- 메모리에 변경 이벤트를 1,000개씩 배치로 누적 후 Kafka 전송
- Kafka Producer 설정: batch.size=1000, **linger.ms=100**

지연 발생 원인: _____

해결방안 (2가지): _____

문제 5 [3점]

Apache Kafka의 Exactly-Once Semantics 구현에 대한 설명 중 틀린 것을 고르시오.

- ① `enable.idempotence=true` 설정으로 Producer 레벨에서 중복 메시지 전송을 방지할 수 있다.
- ② Transactional Producer를 사용하면 여러 토픽에 대한 메시지 전송을 원자적으로 처리할 수 있다.
- ③ Consumer에서 `isolation.level=read_committed` 설정 시 커밋된 메시지만 읽을 수 있다.
- ④ Kafka Streams의 `processing.guarantee=exactly_once` 설정으로 완벽한 Exactly-Once를 보장한다.
- ⑤ Transactional ID는 Producer 재시작 시에도 동일하게 유지되어 zombie producer 문제를 해결한다.

정답 및 해설

문제 1 정답: ③, ⑤

해설:

- ③: Saga 패턴은 분산 환경에서 최종 일관성(Eventually Consistent)만 보장하며, ACID의 완벽한 보장은 불가능
- ⑤: Outbox Pattern은 메시지 발행을 보장하지만, 중복 처리 가능성은 여전히 존재

문제 2 정답: ①

해설: Redis 마스터-슬레이브 복제는 비동기로 이루어지므로, 마스터에서 락 획득 후 슬레이브로 복제되기 전에 마스터 장애 시 다른 클라이언트가 새로운 마스터에서 동일한 락을 획득할 수 있음.

문제 3 정답: ③

해설: version 필드는 이벤트 순서를 보장하고 동시성 제어를 위한 낙관적 락으로 사용 가능. 이벤트는 불변이므로 삭제하지 않고 보상 이벤트로 처리.

문제 4 예시답안:

원인: 단일 스레드 처리로 인한 병목 현상 + 메모리 배치 처리로 인한 메모리 부족 **해결방안:**

1. 멀티 스레드 또는 파티션 기반 병렬 처리 도입
2. 배치 크기 조정 및 스트리밍 방식으로 변경하여 메모리 사용량 최적화

문제 5 정답: ④

해설: Kafka Streams의 exactly-once 보장은 Kafka 내부에서만 적용되며, 외부 시스템과의 상호작용에서는 여전히 at-least-once 특성을 가짐. 완벽한 exactly-once를 위해서는 애플리케이션 레벨에서 멍등성 처리가 필요.

예상문제 #6 - Kafka/Redis 등 Backing 서비스 Clustering 및 성능관련된 특징에 대한 이해

Kafka/Redis 등 Backing 서비스 Clustering 및 성능 관련 개념 설명

1. Kafka Clustering & 성능 특징

파티션 기반 분산 처리

- **파티션**: 토픽 내 메시지 저장 단위로 병렬 처리와 순서 보장을 위해 사용
- **리더/팔로워 구조**: 각 파티션은 하나의 리더와 여러 팔로워로 구성되어 가용성 확보
- **파티션 키**: 동일 키를 가진 메시지는 같은 파티션으로 전송되어 순서 보장

Pub/Sub 모델의 특징

- **프로듀서**: 메시지를 토픽에 발행
- **컨슈머 그룹**: 동일 그룹 내 컨슈머들은 파티션을 나누어 처리 (Load Balancing)
- **오프셋 관리**: 각 컨슈머가 처리한 메시지 위치를 추적하여 중복 처리 방지

순차 거래 처리 방식

- **단일 파티션**: 순서가 중요한 메시지는 하나의 파티션에서 처리
- **키 기반 파티셔닝**: 관련 메시지들을 동일 파티션으로 라우팅하여 순서 보장
- **컨슈머 단일 스레드**: 파티션 내에서는 순차적으로 처리

2. Redis Clustering 및 성능 특징

Redis Cluster 구조

- **해시 슬롯**: 16,384개 슬롯으로 데이터 분산 저장
- **마스터/슬레이브**: 각 마스터 노드는 슬레이브 노드를 가져 복제본 유지
- **자동 페일오버**: 마스터 장애 시 슬레이브가 자동으로 마스터로 승격

비동기 복제의 특성

- **비동기 복제**: 마스터에서 슬레이브로 데이터 복제 시 약간의 지연 발생 가능
- **일관성 vs 성능**: 강한 일관성보다는 성능을 우선시하는 Eventually Consistent 모델

캐시 패턴

- **Cache-aside**: 애플리케이션이 캐시와 DB를 직접 관리
- **Write-through**: 캐시와 DB에 동시 저장
- **Write-behind**: 캐시에 먼저 저장 후 비동기로 DB 업데이트

3. 가용성 처리 방식

Kafka 가용성 전략

- **복제 팩터(Replication Factor)**: 파티션 복제본 수 설정
- **In-Sync Replicas(ISR)**: 리더와 동기화된 팔로워 목록 관리
- **min.insync.replicas**: 최소 동기화 복제본 수 설정으로 데이터 무결성 보장

Redis 가용성 전략

- **Redis Sentinel**: 마스터 모니터링 및 자동 페일오버
- **Redis Cluster**: 분산 환경에서 자동 장애 복구
- **데이터 영속성**: RDB/AOF를 통한 데이터 복구

✓ 핵심 개념 정리

1. Pub/Sub (Publish-Subscribe)

- **의미**: 발행자(Publisher)가 메시지를 보내면, 구독자(Subscriber)는 자신이 구독한 주제(Topic)에 해당하는 메시지를 비동기로 수신.
- **Kafka**: Topic 기반, 메시지는 여러 파티션으로 나뉘며 컨슈머 그룹에 의해 병렬 처리 가능.
- **Redis**: 간단한 Pub/Sub 기능 제공하지만 메시지 지속성 보장 없음 (비영속적).

2. 비동기 처리와 순차성 (순서 보장)

- Kafka는 파티션 단위로만 메시지 순서 보장. 파티션이 다르면 순서는 보장되지 않음.
- Redis Pub/Sub은 순서 보장보다 실시간 알림에 적합.
- 순차 거래가 중요한 경우, **Kafka**는 단일 파티션을 사용하거나 키 기반 파티셔닝(key-based partitioning)을 통해 특정 메시지를 같은 파티션에 보내야 함.

3. Clustering 및고가용성 처리 방식

Kafka

- **Broker**: 메시지를 저장하고 클러스터의 일부를 구성.
- **Zookeeper/ KRaft**: 클러스터 관리, 리더 선출 등 담당.
- **Replication**: Topic의 각 파티션은 여러 브로커에 복제 가능 → 장애 시 failover 가능.
- **Consumer group**: 병렬 처리 + 재처리 유연하게 가능.

Redis

- **Redis Sentinel**: 마스터 장애 감지 및 자동 failover.
- **Redis Cluster**: 데이터 분산 저장(샤딩), 노드 수평 확장 가능.
- **Replication**: 마스터-슬레이브 복제 구조, 가용성 높임.

4. 성능 이슈 및 고려사항

- Kafka의 경우 파티션 수, 프로듀서/컨슈머 수, 파티셔닝 키 설정이 성능에 결정적 영향.
- Redis는 I/O 성능, 메모리 크기, eviction 정책이 주요 요소.
- Kafka는 디스크 기반 저장, Redis는 메모리 기반이므로 **지연보다는 처리량**에 차이가 있음.

예상 문제 출제

문제 1

Kafka 클러스터에서 주문 처리 시스템을 구축할 때, 고객별 주문 순서를 보장해야 한다. 다음 중 올바른 설계 방안은? [4점]

- ① 모든 주문을 단일 파티션으로 처리하여 전체 순서를 보장한다
- ② 주문 ID를 파티션 키로 사용하여 분산 처리한다
- ③ 고객 ID를 파티션 키로 사용하여 고객별 순서를 보장한다
- ④ 타임스탬프를 파티션 키로 사용하여 시간 순서를 보장한다
- ⑤ 파티션 키 없이 라운드 로빈 방식으로 분산 처리한다

문제 2

Redis Cluster 환경에서 데이터 일관성과 관련된 설명 중 틀린 것은? [3점]

- ① Redis Cluster는 비동기 복제 방식을 사용하여 마스터와 슬레이브 간 데이터 지연이 발생할 수 있다
- ② 마스터 노드 장애 시 슬레이브가 자동으로 마스터로 승격되어 가용성을 보장한다
- ③ 강한 일관성을 위해 모든 쓰기 작업은 모든 슬레이브에서 확인 후 완료된다
- ④ 해시 슬롯을 통해 데이터를 여러 마스터 노드에 분산 저장한다
- ⑤ 네트워크 분할 상황에서 Split-brain 문제를 방지하기 위한 쿼럼 메커니즘이 있다

문제 3

MSA 환경에서 비동기 메시징을 통한 이벤트 처리 시 고려사항으로 적절하지 않은 것은? [3점]

- ① 메시지 중복 처리를 방지하기 위한 멍등성(Idempotency) 설계
- ② 메시지 처리 실패 시 재시도 및 Dead Letter Queue 활용
- ③ 모든 이벤트를 동기적으로 처리하여 데이터 일관성 보장
- ④ 이벤트 순서가 중요한 경우 파티셔닝 전략 수립
- ⑤ 서비스 간 결합도를 낮추기 위한 이벤트 스키마 설계

문제 4

Kafka Producer에서 메시지 전송 시 성능과 신뢰성을 고려한 설정 중 올바른 것은? [4점]

- ① acks=0으로 설정하여 최고 성능을 확보한다
- ② batch.size를 크게 설정하여 처리량을 높인다
- ③ linger.ms=0으로 설정하여 즉시 전송한다
- ④ compression.type=none으로 설정하여 CPU 사용률을 낮춘다
- ⑤ retries=0으로 설정하여 중복 메시지를 방지한다

문제 5

다음 Redis 캐시 시나리오에서 Cache-aside 패턴과 Write-through 패턴의 차이점을 설명하고, 각각의 장단점을 서술 하시오. [4점]

시나리오: 사용자 프로필 정보를 캐시하는 시스템에서 사용자가 프로필을 수정하는 경우

정답 및 해설

문제 1: 정답 ③

해설: 고객별 주문 순서를 보장하려면 동일 고객의 모든 주문이 같은 파티션으로 가야 한다. 고객 ID를 파티션 키로 사용하면 같은 고객의 주문들이 동일 파티션에서 순차적으로 처리되어 순서가 보장된다.

문제 2: 정답 ③

해설: Redis Cluster는 성능을 위해 비동기 복제를 사용한다. 모든 슬레이브 확인을 기다리지 않고 마스터에서 쓰기가 완료되면 클라이언트에 응답한다. 따라서 강한 일관성보다는 최종 일관성(Eventually Consistent)을 제공한다.

문제 3: 정답 ③

해설: 비동기 메시징의 핵심은 서비스 간 느슨한 결합을 통한 확장성 확보이다. 모든 이벤트를 동기적으로 처리하면 비동기 메시징의 장점을 상실하고 시스템 복잡성만 증가한다.

문제 4: 정답 ②

해설: batch.size를 적절히 크게 설정하면 여러 메시지를 묶어서 전송하여 네트워크 효율성과 처리량을 높일 수 있다. 다른 옵션들은 성능이나 신뢰성을 크게 저하시킨다.

문제 5: 정답

Cache-aside 패턴:

- 애플리케이션이 캐시와 DB를 직접 관리
- 프로필 수정 시: DB 업데이트 → 캐시 무효화 또는 업데이트
- 장점: 캐시 장애가 서비스에 미치는 영향 최소화
- 단점: 캐시 관리 로직의 복잡성

Write-through 패턴:

- 캐시와 DB에 동시 저장
- 프로필 수정 시: 캐시와 DB 동시 업데이트
- 장점: 데이터 일관성 보장, 캐시 히트율 높음
- 단점: 쓰기 지연 발생, 캐시 장애 시 서비스 영향