KICE Pilot 기출문제

2025.05.09 SWA Pilot 시험문제풀이

01. Software Architecture 핵심 > 기반 솔루션 주요기능/장단점

문항 1) 차세대 시스템 구축 프로젝트에서 I/F 담당자로 참여 중이다. 대내외 I/F 구축 관련하여 고려해야 할 사항으로 잘못 이해하고 있는 것을 고르시오. [4점]

① 대외/대내 I/F 모두 모니터링이 매우 중요한 요소이다. I/F 실패 발생 시 해당 건을 쉽게 인지하고 담당자에게 inform을 주며 원인 추적이 용이할 수 있도록, 모든 I/F 솔루션들을 아우를 수 있는 모니터링 시스템 구축이 요구되기도 한다.

② I/F 시스템은 Load Balancer 만으로 가용성 확보가 힘든 경우가 많다. 따라서 HA 솔루션을 사용하여 Service IP 및 디스크의 fail-over를 구현해야 할 수 있다. Fail-over 시에는 일부 데이터의 수동 재처리가 필요할 수도 있다.

③ EAI와 같은 I/F 솔루션은 중앙에서 데이터 흐름을 처리하기에 용이하다. 따라서 I/F 처리 과정에 업무 로직을 포함하여 I/F에 대한 유지보수성을 높이는 것이 효율적이다.

④ 대내 I/F는 EAI, ESB, MQ 등 다양한 솔루션을 사용하게 된다. 시스템 간 Peer-to-Peer로 직접 호출하는 방식은 결합도를 높이게 되므로 지양하도록 하며, 프로젝트의 표준화된 프로토콜 및 I/F 방식을 따르도록 한다.

⑤ 기간계 시스템과 재무회계 등 단위 시스템 간의 온라인 I/F를 위해 EAI 솔루션을 도입할 수 있다. 또한 EAI 솔루션을 이용하여 파일 형태의 배치 데이터를 전문 형태로 전송할 수 있다.

(정답)3
(해설)I/F 처리 과정에 로직을 추가하는 것은 적절하지 않다.
(배점)4
(난이도)중

- 문제유형: 선다형

- 출제영역 대분류>소분류: Software Architecture 핵심 > 기반 솔루션 주요기능/장단점

- 문제 제목: I/F솔루션

- 출제 의도: I/F 구축 작업 시 중요 포인트를 잘 알고 있는지 확인

🔎 인터페이스 아키텍처 설계 시 **고려사항**

항목	설명
사용자 또는 호출자 특성	내부 시스템 vs 외부 제3자, 동기 vs 비동기 호출
통신 프로토콜 선택	REST, gRPC, GraphQL, WebSocket, Kafka 등 선택 기준
데이터 포맷	JSON, XML, Protobuf, Avro 등 목적과 사용 환경에 맞게
에러 처리 표준화	HTTP status code + error code + message 조합 방식 등
문서화 자동화	Swagger UI, Postman, Stoplight 등 연동

항목	설명
테스트 전략	계약 테스트 (Contract Testing), Mock 서버 활용
로깅 및 추적	인터페이스 호출 로깅, Tracing(ID propagation 등)
Failover 및 재시도 정책	클라이언트 및 API 게이트웨이 단에서 고려 필요

I/F 아키텍처 설계 원칙

구분	원칙	설명
관심사 분리	Separation of Concerns	I/F 레이어는 데이터 전달만, 업무 로직은 각 시스템에서 처리
느슨한 결합	Loose Coupling	ESB, EAI를 통한 간접 통신으로 시스템 간 독립성 확보
표준화	Standardization	공통 프로토콜, 메시지 포맷, 인터페이스 규약 준수
모니터링	Observability	실시간 상태 감시, 장애 감지, 성능 추적
가용성	High Availability	단일 장애점 제거, Fail-over 메커니즘 구현

I/F 솔루션별 특징

솔루션	장점	단점	적용 시나리오
EAI	- 중앙 집중식 관리 - 데이터 변환 기능 - 모니터링 용이	- 단일 장애점 - 성능 병목 - 높은 복잡도	대용량 배치 처리 레거시 시스템 통합
ESB	- 분산 아키텍처 - 서비스 중심 - 확장성 우수	- 구현 복잡도 - 거버넌스 필요	SOA 환경 실시간 서비스 통합
MQ	- 비동기 처리 - 안정적 전송 - 부하 분산	- 메시지 순서 보장 어려움 - 디버깅 복잡	대용량 트랜잭션 시스템 간 비동기 통신

솔루션	장점	단점	적용 시나리오
API Gateway	- RESTful 인터페이스 - 클라우드 친화적 - 개발 생산성	- 동기식 처리 한계 - 네트워크 의존성	마이크로서비스 외부 시스템 연계

모니터링 체계

레벨	항목	도구 예시	목적
인프라	서버, 네트워크, DB	Zabbix, Nagios	시스템 리소스 상태 감시
애플리케이션	응답시간, TPS, 에러율	APM (New Relic, Dynatrace)	애플리케이션 성능 추적
비즈니스	처리량, 지연건수, SLA	Custom Dashboard	업무 지표 모니터링
로그	에러 로그, 트랜잭션 로그	ELK Stack, Splunk	장애 원인 분석

📚 참고 자료

📖 도서

1. **《Clean Architecture》 – Robert C. Martin (Uncle Bob)**
   - 인터페이스와 구현의 분리, 경계 역할을 강조
   - https://amzn.to/3WZo4ef
2. **《Domain-Driven Design: Tackling Complexity in the Heart of Software》 – Eric Evans**
   - Bounded Context와 인터페이스 설계 관련 지침
   - https://amzn.to/3Rm8aEq
3. **《Designing Web APIs》 – Brenda Jin, Saurabh Sahni, Amir Shevat**
   - API 인터페이스 설계와 문서화, 개발자 경험 개선
   - https://amzn.to/3RlO5Xd
4. **《Building Microservices》 – Sam Newman**
   - 마이크로서비스 간 인터페이스 설계 및 서비스 경계 정의
   - https://amzn.to/4cWoZXQ

🌐 웹사이트 및 아티클

- **Microsoft – API Design Best Practices**
  https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
- **Google API Design Guide**
  https://cloud.google.com/apis/design
- **RESTful API 디자인 가이드 (Naver D2 블로그)**
  https://d2.naver.com/helloworld/7804182
- **OpenAPI Specification (Swagger)**
  https://swagger.io/specification/

IF 시스템이란?

- Interface 시스템
- 서로 다른 시스템들 간에 데이터를 주고받을 수 있도록 연결해주는 시스템
- 구체적인 역할
    - 시스템 A의 데이터를 시스템 B가 이해할 수 있는 형태로 변환
    - 데이터 전송, 수신, 처리
    - 통신 프로토콜 변환
    - 데이터 포맷 변환 (XML ↔ JSON ↔ CSV 등)
- 예시
    - **은행 시스템의 경우:**
        - 인터넷뱅킹 시스템 ← I/F → 계정관리 시스템
        - 계정관리 시스템 ← I/F → 외환거래 시스템
        - ATM 시스템 ← I/F → 중앙 계정 시스템
    - **전자상거래의 경우:**
        - 쇼핑몰 ← I/F → 결제 시스템
        - 쇼핑몰 ← I/F → 재고관리 시스템
        - 쇼핑몰 ← I/F → 배송 시스템
- 특징
    - **처리하는 데이터:**
        - 실시간 거래 데이터
        - 배치 파일 데이터

    - 메시지 큐 데이터
- **중요한 요구사항:**
    - **안정성**: 데이터 손실 없이 전달
    - **성능**: 대용량 데이터 처리
    - **가용성**: 24시간 무중단 서비스

02. Software Architecture 핵심 > JVM 구조 및 동작 특성

문항2) 다음은 jstat을 이용하여 Hotspot JVM의 Heap 메모리 사용량을 모니터링한 결과이다. 아래의 모니터링 결과에 대한 설명 중 올바른 것을 고르시오. [3점]

[jstat 모니터링 결과]

$ jstat -gcutil 16973 1000

S0	S1	E	O	M	CCS	YGC	YGCT	FGC
97.46	0.00	21.86	99.55	98.68	96.82	3548	166.537	3
97.46	0.00	60.18	99.55	98.68	96.82	3548	166.537	3
97.46	0.00	96.35	99.55	98.68	96.82	3548	166.537	3
0.00	98.61	0.60	99.73	98.68	96.82	3549	166.62	3
0.00	98.61	97.84	99.73	98.68	96.82	3549	166.62	3
97.25	0.00	0.00	99.94	98.68	96.82	3550	166.702	3
0.00	0.00	22.28	3.73	98.61	96.69	3550	166.702	4
0.00	0.00	57.87	3.73	98.61	96.69	3550	166.702	4
0.00	0.00	93.27	3.73	98.61	96.69	3550	166.702	4

① Young 영역 중 Survivor 0 영역이 꽉 찬 후 Minor GC가 발생하고 있다.
② Young 영역 중 Survivor 1 영역이 꽉 찬 후 Minor GC가 발생하고 있다.
③ Young 영역 중 Eden 영역이 꽉 찬 후 Minor GC가 발생하고 있다.
④ 어플리케이션에서 호출한 System.gc()코드에 의해 Minor GC가 발생하고 있다.
⑤ Minor GC 반복 과정 중 Eden 영역에서 살아남은 객체가 Old 영역으로 이동하고 있다.

(정답)3
(해설)
1.2. Minor GC는 Eden 영역이 꽉 차게 되면 발생한다.
4. jstat을 사용하여 System.gc()가 호출되는지는 알 수 없고 또한, 이때는 Full GC가 수행된다.
5. Minor GC를 반복하는 과정에서 Survivor 영역에서 살아남은 객체가 Old 영역으로 이동한다.
(배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 핵심 > JVM 구조 및 동작 특성
- 문제 제목: JVM의 GC 발생 조건 및 동작 방식
- 출제 의도: JVM의 GC 발생 조건 및 동작 방식에 대한 이해

📌 정답 해설

❓ 문제 맥락 요약

`jstat -gcutil` 명령은 JVM의 GC(가비지 컬렉션) 관련 힙 메모리 사용률을 실시간으로 보여줍니다.
여기서 GC가 발생하는 트리거 조건과 각 영역의 역할에 대한 이해가 필요합니다.

✔ 보기별 설명

보기	설명	정오
①	**Survivor 0(S0)** 영역이 꽉 차서 Minor GC 발생한다는 설명은 틀림. S0/S1은 Eden에서 GC 후 복사되는 영역일 뿐, GC 트리거 대상 아님.	✖
②	**Survivor 1(S1)**도 위와 동일. S0/S1은 교대로 사용되며 크기도 작기 때문에 꽉 차도 GC를 유발하지 않음.	✖
③	**Eden 영역이 꽉 차면 Minor GC 발생**. Young Generation 중 **Eden**이 가장 먼저 가득 차며, 이때 Minor GC가 발생함.	✔ 정답
④	`System.gc()` 는 **Full GC** 유발 가능성이 큼. Minor GC는 주로 Eden 공간 부족으로 발생하므로 설명 부적절.	✖
⑤	Eden → Survivor → Old 순서로 이동. **Minor GC 시 바로 Old로 가지 않음**, 여러 번 Survivor 거친 후 **Tenuring Threshold**를 넘을 때 Old로 감.	✖

📊 jstat -gcutil 출력 해석

컬럼	설명
S0, S1	Survivor 0, 1 영역 사용률 (%)
E	Eden 영역 사용률 (%)
O	Old 영역 사용률 (%)

컬럼	설명
M	Metaspace 사용률 (%)
CCS	Compressed Class Space 사용률 (%)
YGC	Young GC (Minor GC) 발생 횟수
YGCT	Young GC 누적 시간 (초)
FGC	Full GC 발생 횟수
FGCT	Full GC 누적 시간 (초)
GCT	전체 GC 시간 (YGCT + FGCT)

📌 주요 패턴 확인

🔎 Eden 영역(E) 사용률 패턴

- 60.18% → 96.35% → **0.60%** → 97.84% → **0.00%** → 22.82% → **57.87%** → **93.27%**
- 이 패턴에서 **Eden 영역이 거의 꽉 찼다가 → 0% 또는 아주 낮은 수준으로 떨어짐**을 반복

이는 **Minor GC가 Eden이 꽉 찼을 때 발생**하고, 이후 Eden이 **Clear**되었음을 의미합니다.

🔎 YGC 값 변화

- YGC: 3548 → 3549 → 3550
  → Young GC가 총 2번 발생했음

✓ 위 Eden 영역 변화 시점과 정확히 일치함 → **Eden이 꽉 찼을 때 Minor GC 발생이 맞음**

주요 참고 사이트:

1. **네이버 D2 - Garbage Collection 모니터링 방법**
   - https://d2.naver.com/helloworld/6043
   - 네이버의 기술 블로그로 GC 모니터링에 대한 상세한 설명
2. **삼성SDS - 메모리 모니터링과 원인 분석**

- https://www.samsungsds.com/kr/insights/1232762_4627.html
- jstat을 이용한 실제 모니터링 예제와 분석 방법 제공

3. **jstat을 이용한 JVM 메모리 모니터링 (네이버 블로그)**
   - https://m.blog.naver.com/PostView.naver?blogId=hanajava&logNo=221506209991
   - jstat 사용법과 각 옵션에 대한 상세 설명
4. **자바 메모리 관리 - 가비지 컬렉션**
   - https://yaboong.github.io/java/2018/06/09/java-garbage-collection/
   - Eden 영역이 가득 찰 때 Minor GC 발생에 대한 명확한 설명
5. **개발자 이동욱 - GC 종류 및 내부 원리**
   - https://dongwooklee96.github.io/post/2021/04/04/gcgarbage-collector-%EC%A2%85%EB%A5%98-%EB%B0%8F-%EB%82%B4%EB%B6%80-%EC%9B%90%EB%A6%AC.html
   - Eden 영역과 Minor GC의 동작 원리 상세 설명
6. **가비지 컬렉션 동작 원리 (네이버 블로그)**
   - https://m.blog.naver.com/web-developer/223345931298
   - Minor GC가 언제 발생하는지에 대한 명확한 설명
7. **우아한형제들 기술블로그 - JVM memory leak 이야기**
   - https://techblog.woowahan.com/2628/
   - 실제 운영 환경에서의 메모리 모니터링 경험담
8. **뒤태지존의 끄적거림 - Java Memory Monitoring**
   - https://homoefficio.github.io/2020/04/09/Java-Memory-Monitoring/
   - jstat을 포함한 다양한 Java 메모리 모니터링 도구 소개

추가 기술 블로그:

1. **가비지 컬렉터(GC)에 대하여**
   - https://velog.io/@litien/가비지-컬렉터GC
2. **자바 가비지 컬렉션 설명 및 종류**
   - https://blog.voidmainvoid.net/190

☕ JVM 내부 구조 & 메모리 영역 💯 총정리

JVM Heap 메모리 구조와 GC 동작 원리

JVM의 메모리 관리를 **아파트 관리**에 비유해서 설명해드리겠습니다.

🏢 Heap 메모리 구조 (아파트 건물)

Young Generation (신혼부부 전용 층)

```
┌─────────────────────────────────────────┐
│           Young Generation              │
├─────────────┬─────────────┬─────────────┤
│    Eden     │  Survivor 0 │ Survivor 1  │
│   (신방)    │  (임시방1)  │  (임시방2)  │
│  새 객체    │             │             │
│   생성      │  GC 생존자  │  GC 생존자  │
└─────────────┴─────────────┴─────────────┘
```

Old Generation (기존 거주자 층)

```
┌─────────────────────────────────────────┐
│          Old Generation                 │
│        (장기 거주 구역)                 │
│      오래 살아남은 객체들               │
└─────────────────────────────────────────┘
```

🔄 GC 동작 과정 (아파트 청소)

1단계: 새 객체 생성

```
Eden 영역 (신방)
[새객체1][새객체2][새객체3][새객체4]
```

- 새로운 객체들이 Eden 영역에 계속 생성됩니다
- 마치 신혼부부들이 신방에 입주하는 것과 같습니다

2단계: Eden 영역 가득참 → Minor GC 발생

```
Eden 영역이 꽉 참!
[객체A][객체B][객체C][객체D][객체E]... (가득)

Minor GC 발생!
```

3단계: Minor GC 실행

```
Before Minor GC:
Eden:      [살아있음][죽음][살아있음][죽음][살아있음]
Survivor0: [비어있음]
Survivor1: [비어있음]

After Minor GC:
Eden:      [비어있음] ← 모든 객체 제거
Survivor0: [살아있던객체들] ← 생존자들 이동
Survivor1: [비어있음]
```

4단계: 여러 번의 Minor GC 후

```
객체의 나이(age) 증가:
Survivor0 → Survivor1 → Survivor0 → ... (왔다갔다)

나이가 많아진 객체 (예: 8살)
↓
Old Generation으로 승격 (Promotion)
```

📊 실제 예시로 이해하기

쇼핑몰 시스템 예시:

java

```
// 1. 고객이 상품을 장바구니에 담음
ShoppingCart cart = new ShoppingCart();  // Eden 영역에 생성

// 2. 주문 처리
Order order = new Order(cart);           // Eden 영역에 생성

// 3. 결제 완료 후 장바구니는 더 이상 필요 없음
// → Minor GC 시 장바구니 객체는 제거됨

// 4. 주문 정보는 계속 필요함
// → 여러 Minor GC를 거쳐 Old 영역으로 이동
```

🎯 GC 종류와 발생 조건

Minor GC (젊은 층 청소)

- **발생 조건**: Eden 영역이 가득 참
- **대상**: Young Generation만
- **속도**: 빠름 (몇 ms ~ 수십 ms)
- **빈도**: 자주 발생

Major/Full GC (전체 건물 청소)

- **발생 조건**: Old 영역이 가득 참
- **대상**: 전체 Heap (Young + Old)
- **속도**: 느림 (수백 ms ~ 수 초)
- **빈도**: 드물게 발생

🔍 jstat으로 모니터링하는 이유

```
$ jstat -gcutil 16973 1000
```

모니터링하는 지표들:

- **S0, S1**: Survivor 영역 사용률
- **E**: Eden 영역 사용률
- **O**: Old 영역 사용률
- **YGC**: Minor GC 발생 횟수
- **FGC**: Full GC 발생 횟수

🔔 성능 최적화 포인트

좋은 상황:

- Minor GC가 자주, 빠르게 발생
- Full GC가 드물게 발생
- 대부분의 객체가 Young Generation에서 정리됨

나쁜 상황:

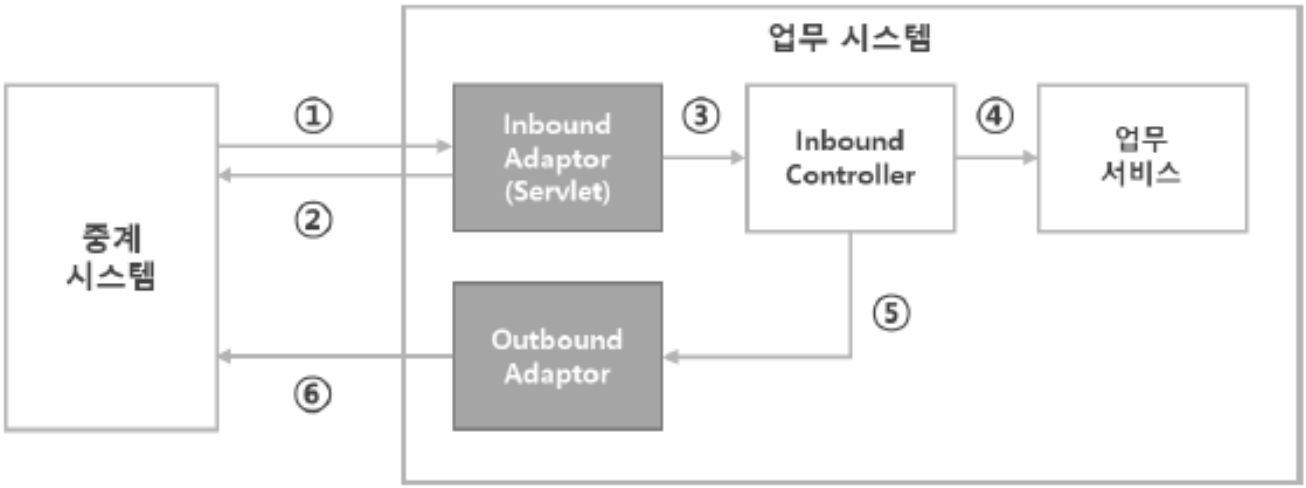
- Full GC가 자주 발생 (Stop-the-World로 애플리케이션 멈춤)
- Old 영역이 계속 증가 (메모리 누수 가능성)

이렇게 JVM은 "**대부분의 객체는 금방 죽는다**"는 가정하에 효율적으로 메모리를 관리합니다. 마치 아파트에서 신혼부부는 금방 이사가고, 오래 사는 사람들만 장기 거주하는 것과 비슷합니다!

03. Software Architecture 설계/구축 > Framework 적용 (환경, 온라인, 배치, 연계 등)

문항3) 아래 그림은 비동기 요청(Inbound) 거래의 흐름도이다. 프레임워크 담당자는 흐름도에 따라 중계 시스템으로부터 HTTP 요청을 처리하는 비동기 Adaptor를 개발하고 있다.

<비동기 거래 흐름 목표 구성>



① 중계 시스템에서 업무시스템으로 HTTP 프로토콜로 거래 요청을 전송
② Framework Inbound Adaptor에서 요청에 대한 정상(200) 응답
③ 정상(200) 응답 후 F/W Inbound Controller에 거래 처리 요청
④ F/W Inbound Controller에서 공통 선처리 후 해당 요청에 대응하는 업무 서비스 호출
⑤ 업무 서비스 호출 후 업무 리턴 된 응답을 F/W Outbound Adaptor에 전달
⑥ F/W Outbound Adaptor에서 중계 시스템으로 HTTP 프로토콜로 응답 전송

<F/W Inbound Adaptor>

```
-- 생략 --

public class FWInboundAsyncAdaptor extends HttpServlet {
-- 생략 --
    @Override
    protected void doPost( HttpServletRequest request, HttpServletResponse response)
throws ServletException {
        // 요청 메시지
        String message = request.getParameter(“msg”);
        -- 생략 --
        // 응답 전송
        response.setStatus(HttpServletResponse.SC_OK);
        response.setContextType(“text/html;charset=utf-8”);

        bos = new BufferedOutputStream(response.getOutputStream() );
        bos.write( “OK”.getBytes() );

        bos.flush();
        bos.close();
```

```
        -- 생략 --
        // FW Inbound Controller 호출
        String result = fwInboundController.execute(message);
        sendMessage(result); // Outbound Adaptor를 통해 메시지 전송
        -- 생략 --
        }
}
```

Adaptor 개발 후 개발 환경에 적용하여 중계 시스템과 I/F 테스트를 하는 도중 중계 시스템 담당자가 2번 응답이 6번 요청보다 늦게 수신된다는 이슈를 제기했다. 해당 현상에 대한 설명 중 옳은 것을 고르시오. [4점]
① bos.flush() 및 bos.close() 코드 라인 시점에 중계 시스템으로 먼저 “OK” 메시지 전송을 시작했으나, 네트워크 속도 등으로 인해 Inbound Controller의 응답 전송이 먼저 완료되는 경우이다.
② Inbound Controller의 업무 처리가 너무 빨리 끝나는 경우 발생하는 경우이며, 따라서 업무 로직 내 일정 기간(ex. 1s) sleep을 수행하면 이 현상은 해소된다.
③ Inbound Controller의 업무 처리 속도와 상관없이 확률적으로 드물게 발생하는 현상이다.
④ HttpServlet의 특성에 기인한 문제로서 Inbound Controller 수행 및 결과 전송을 비동기로 수행하도록 변경하여야 한다.
⑤ bos.close() 가 정상 종료되지 않는 것이 원인이므로, 네트워크 상태를 확인 및 관련 OS 파라미터 설정값을 확인해야한다.

(정답)4
(해설)
1) Servlet 응답은 내부 로직에서 Response 를 전송하는 시점에 응답이 나가지 않고 (즉, bos.flush()나 close() 코드 위치와 무관하게) Servlet 처리가 모두 종료되어야 응답이 전송되게 된다.
2) Controller 업무 처리 속도와 상관없이 항상 Inbound Controller 의 응답이 먼저 전송되게 된다.
3) 2)번 해설 동일

4) 정답
5) 네트워크 문제가 아님
(배점) 4
(난이도) 중
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 설계/구축 > Framework 적용 (환경, 온라인, 배치, 연계 등)
- 문제 제목: 비동기 Adaptor 작성
- 출제 의도: 비동기 Adaptor 작성 시 주의 사항에 대한 이해

📌 해설

🔎 문제 요약

현재 **비동기 HTTP 요청 처리** 흐름에서는:

1. **중계 시스템이 요청 전송**
2. `FWInboundAsyncAdaptor` 가 **"OK"를 응답**

3. 이후 **업무 처리를 진행하고**, 그 결과를 `sendMessage()` 통해 **다시 중계 시스템으로 응답 전송**

🔔 이슈 내용

> *중계 시스템에서 2번 응답(즉시 "OK")이 6번 응답(업무 처리 후 Outbound 응답)보다 늦게 수신됨*

이 말은, **즉각 응답을 주었어야 할 "OK" 응답이 지연되고**, 오히려 **뒤이어 처리된 업무 응답이 먼저 도착했다**는 뜻입니다.

⚠ 원인 분석

🔍 문제 코드 핵심 부분

```
response.setStatus(HttpServletResponse.SC_OK);
response.setContextType("text/html;charset=utf-8");
bos = new BufferedOutputStream(response.getOutputStream());
bos.write("OK".getBytes());
bos.flush();
bos.close();

// 이후 동기적으로 업무 처리
String result = fwInboundController.execute(message);
sendMessage(result);
```

java

`response.setStatus(HttpServletResponse.SC_OK);`
`response.setContextType("text/html;charset=utf-8");`
`bos = new BufferedOutputStream(response.getOutputStream());`
`bos.write("OK".getBytes());`
`bos.flush();`
`bos.close();`

`// 이후 동기적으로 업무 처리`
`String result = fwInboundController.execute(message);`
`sendMessage(result);`

- **문제점**: `doPost()` 메서드 내에서 **업무 처리가 동기적으로 수행**되고 있으며, 서블릿 응답을 **완전히 종료하지 않은 상태**에서 동작이 계속됨.

- 이 경우 **서버가 실제 응답을 클라이언트(중계 시스템)에 커밋하는 시점이 늦어질 수 있음**, 즉 **2번 응답 전송이 지연됨**.

🔧 해결책

- **서블릿의 동작을 비동기 방식으로 전환**해야 함.
- `@WebServlet(asyncSupported = true)` 사용 + `startAsync()` 로 비동기 처리 → 즉시 응답 전송 → 이후 백그라운드에서 처리.

📌 보기 해설

보기	설명	정오
①	네트워크 이슈만으로 응답 순서가 반전되긴 **매우 어려움**. 코드 구조 문제임	✗
②	업무 처리 속도가 빨라져도 문제가 발생하지 않아야 하며, **sleep()은 임시방편**	✗
③	확률적 현상이 아닌, **구조적으로 발생**하는 동기 처리 문제	✗
④	서블릿은 기본적으로 **동기 실행**. 응답 전송 전 `doPost()` 가 완료되어야 하므로, **비동기로 개선해야 해결**됨	✓ 정답
⑤	bos.close() 문제 아님. 실제 응답 전송 타이밍과 무관	✗

보충 설명: 비동기 서블릿 처리 방법

```
@WebServlet(urlPatterns = "/async", asyncSupported = true)
public class FWInboundAsyncAdaptor extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        AsyncContext asyncContext = request.startAsync();

        // 즉시 응답 전송
        response.setStatus(HttpServletResponse.SC_OK);
        response.setContentType("text/plain");
        response.getWriter().write("OK");
        response.getWriter().flush();
```

```
        // 비동기 작업 수행
        asyncContext.start(() -> {
            String message = request.getParameter("msg");
            String result = fwInboundController.execute(message);
            sendMessage(result);
            asyncContext.complete();  // 비동기 작업 완료
        });
    }
}
```

- **JSP/서블릿 예외 처리**
  - https://opentutorials.org/module/3569/21260
  - HttpServlet의 response 처리 방법에 대한 설명
- **자바와 스프링의 비동기 기술**
  - https://jongmin92.github.io/2019/03/31/Java/java-async-1/
  - 서블릿 비동기 처리와 관련된 상세한 설명
- **JSP Servlet Request, Response 객체**
  - https://ip99202.github.io/posts/JSP-Servlet-Request,-Response-객체/
  - HttpServletResponse 객체 사용법과 특성
- **HTTP 프로토콜과 Java 구현**
  - https://velog.io/@godkimchichi/Java-14-HTTP-프로토콜
  - HTTP 응답 처리와 네트워크 전송에 대한 설명
- **HttpServletRequest, HttpServletResponse 객체**
  - https://velog.io/@oliviarla/HttpServletRequest-HttpServletResponse-%EA%B0%9D%EC%B2%B4%EB%9E%80
  - 서블릿 응답 객체의 동작 원리
- **자바스크립트 비동기 처리**
  - https://www.daleseo.com/js-async-callback/
  - 비동기 처리의 일반적인 개념과 원리
- **Request, Response 객체 개념**
  - https://velog.io/@oyeon/Request-Response-객체1
  - 웹 서버에서의 요청/응답 처리 과정
- 서블릿 3.0의 비동기 처리 기능
- https://imprint.tistory.com/230

✓ 추천 블로그 (한글)

1. 자바캔 – Servlet 3.0의 비동기 처리 기능

- `startAsync()` 사용법, `AsyncContext` 예제 코드, 동작 원리 설명
- **주요 내용**: 비동기 요청 시작 → `doGet()` 종료 후 실제 응답은 별도 스레드에서 처리됨
  dzone.com+5javacan.tistory.com+5kamang-it.tistory.com+5dzone.com
  👉 https://javacan.tistory.com/entry/Servlet-3-Async

2. Kamang's IT Blog – JSP/Servlet 비동기 사용하기(AsyncContext)

- `AsyncContext` 설정 단계, 이벤트 리스너, `complete()` 호출법 등
- 실무 적용 시 주의사항까지 상세하게 정리됨 javacan.tistory.com+1fearless-nyang.tistory.com+1
  👉 https://kamang-it.tistory.com/entry/JSPServlet-%EB%B9%84%EB%8F%99%EA%B8%B0%EB%A1%9C-%EC%82%AC%EC%9A%A9%ED%95%98%EA%B8%B0AsyncContext

3. Modern – [Reactive Programming] 자바 & 스프링 비동기 기술

- 서블릿 비동기 기술 소개 및 Servlet 3.x 기반 동작 방식, NIO 커넥터 설명
- 비동기 컨텍스트 흐름을 그린 다이어그램 제공 kamang-it.tistory.com+1frombasics.tistory.com+1imprint.tistory.com+1ch4njun.tistory.com+1
  👉 https://imprint.tistory.com/234

🌐 영어권 블로그

4. JavaNexus – Improving Web App Performance with Asynchronous Servlets

- 최신 관점에서 성능 향상 관점 설명, 스레드 풀 절약 전략 등
- 실전 예제 포함 ch4njun.tistory.com+4imprint.tistory.com+4tonylim.tistory.com+4
  👉 https://javanexus.com/blog/improve-web-app-performance-async-servlets

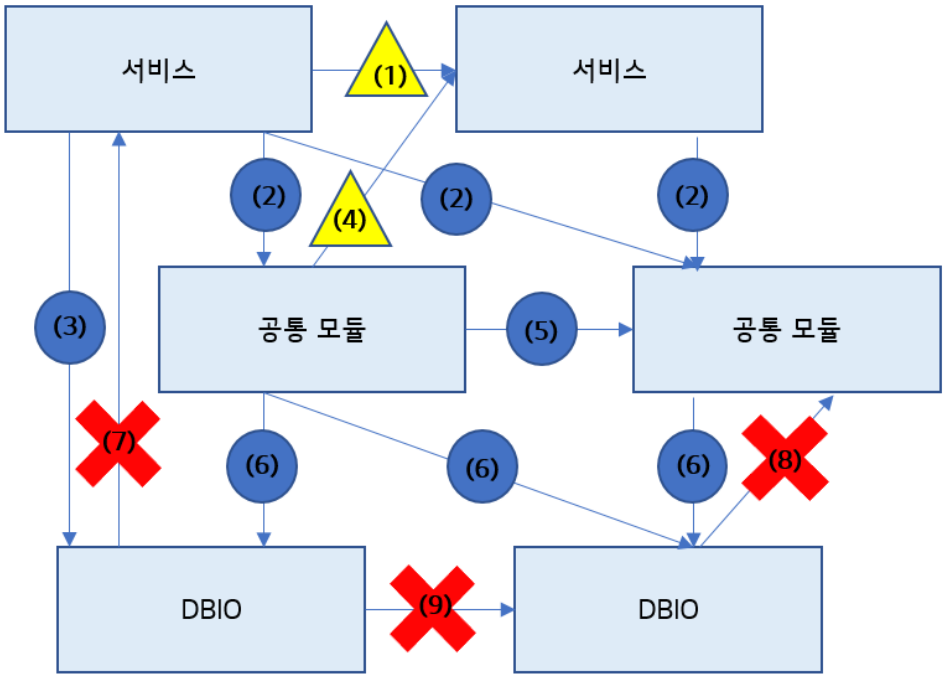
5. Java Code Geeks – Async Servlet Feature of Servlet 3

- `@WebServlet(asyncSupported)`, `AsyncContext`, `AsyncListener` 구조 소개

- `ThreadPoolExecutor`와 `complete()` 사용법까지 상세하게 정리
  wjw465150.github.io+15javacodegeeks.com+15javanexus.com+15
  ☞ https://www.javacodegeeks.com/2013/08/async-servlet-feature-of-servlet-3.html

04. Software Architecture 설계/구축 > 개발 표준 수립 및 가이드

문항4) 다음은 Java 기반 프로젝트의 어플리케이션 호출 관계 표이다.



Caller \ Callee	서비스
서비스	(1) (서비스인
공통 모듈	(4) (서비스인
DBIO	(7)

허용 불가 부분적 허용

*서비스연동 : 서비스 프로그램 직접 호출 방식이 아닌, 연동 파라미터 정보를 참조하여, 연동 모듈을 통해 서비스를 호출함.

아래 프로그램은 호출 관계 중 (4)번의 호출 CASE이다. 아래와 같이 서비스를 호출할 경우, 발생하는 문제점에 대해 적으시오. [4점]

[인터페이스 소스]

```
public interface CommonBeforeInterface{
    public OutputDto process(InputDto in);
}
```

[서비스 소스]

```
public class PS001 implements CommonBeforeInterface{
    private BeforeProcessModule beforeProcessModule;
    …
    //메인 메소드
    public OutputDto process(InputDto inputData){
```

```
        ….
        beforeProcessModule.bizProcess(inputData, outputData, this);
        processBean.process(inputData);
        ….
    }
}
```

[공통 모듈 소스]

```
Public class BeforeProcessModule {
    public OutputDto bizProcess (InputDto inputData, OutputDto outputData,
CommonBeforeInterface svc){
        ….
        outputData = svc.process(inputData);
        .…
    }
}
```

문제점 :

(정답) 순환 참조{|}순한 콜{|}Circular Reference{|}Circular Call 이 발생하여 무한 루프가 발생함.
(해설) 호출 관계도에서 공통 모듈이 서비스 호출 시, 서비스 연동을 해야 하나, 서비스에서 공통 모듈을 호출할 때, 파라미터로 자기 자신(this)을 전달하여, 공통 모듈에서 서비스 메소드를 직접 호출을 함으로써, Circular Reference 를 야기하여 무한 루프가 발생할 수 있는 코드임.
(배점)4
(난이도)중
- 문제유형: 단답형
- 출제영역 대분류>소분류: Software Architecture 설계/구축 > 개발 표준 수립 및 가이드
- 문제 제목: 호출 관계에 대한 이해도
- 출제 의도: 호출 관계 표준에 위배된 프로그램에 대한 원인 분석 능력 검증

✓ 정답 요약

발생하는 문제점:

> `BeforeProcessModule.bizProcess()` *에서* `svc.process(inputData)` *호출 시,*
> *PS001 클래스의* `process()` *메소드가* ***재귀 호출(무한 루프)*** *되면서*
> ***StackOverflowError****가 발생한다.*

📌 호출 관계 분석 (4번 CASE)

🔔 구조 요약

```
PS001.process()
 └── calls → BeforeProcessModule.bizProcess(..., this)
       └── calls → svc.process(inputData)
                └── this.process(inputData) → 즉, PS001.process()
                        → 다시 BeforeProcessModule.bizProcess(...)
                              → 다시 svc.process(...)
                                    → 무한 반복
```

❗ 문제의 핵심

`beforeProcessModule.bizProcess(inputData, outputData, this);`

- 여기서 `this`는 `PS001` 인스턴스이고, `CommonBeforeInterface`를 구현 중.
- 그런데 `bizProcess()` 내부에서 다시 `svc.process()`를 호출하므로,
- 결국 자기 자신의 `process()` 메소드가 재귀적으로 계속 호출됨.

🔁 결과적으로 다음과 같은 문제가 발생:

🚨 발생 문제

- **무한 재귀 호출**
  → `process()` 안에서 `bizProcess()` 호출
  → 그 안에서 다시 `process()` 호출
  → ... 계속 반복
- **JVM Stack Overflow**
  → `StackOverflowError` 예외 발생

✔ 해결 방향 (보완 아이디어)

- `bizProcess()`에서 `svc.process()`를 호출하는 것이 아니라,
  다른 메소드를 호출하거나 **핵심 로직을 분리**해서 순환 참조를 피해야 함

예시 해결 구조:

```
public class PS001 implements CommonBeforeInterface {
    public OutputDto process(InputDto inputData){
        beforeProcessModule.bizProcess(inputData, outputData, this);
        return outputData;
    }

    // 핵심 로직만 별도로 분리
    public void doMainProcess(InputDto inputData) {
        // 실제 비즈니스 로직 수행
        processBean.process(inputData);
    }
}
```

그리고 `bizProcess()` 내부에서는 다음처럼 변경:

```
public OutputDto bizProcess (InputDto inputData, OutputDto outputData,
CommonBeforeInterface svc){
    // 호출자가  제공한 비즈니스 로직 메서드만 수행
    if (svc instanceof PS001) {
        ((PS001) svc).doMainProcess(inputData);
    }
    return outputData;
}
```

- **재귀와 스택**
  - https://ko.javascript.info/recursion
  - 재귀 호출의 원리와 스택 오버플로우 발생 메커니즘
- **재귀 호출 (TCP School)**
  - https://www.tcpschool.com/java/java_usingMethod_recursive
  - Java 재귀 호출의 기본 개념과 스택 오버플로우 설명
- **JPA 순환 참조 시 StackOverFlow**
  - https://velog.io/@bbbbooo/JPA-순환-참조-시-StackOverFlow-java.lang.StackOverflowError
  - 순환 참조로 인한 StackOverflowError 실제 사례
- **재귀함수의 장점과 단점 그리고 해결책**
  - https://catsbi.oopy.io/dbcc8c79-4600-4655-b2e2-b76eb7309e60
  - 재귀 호출의 문제점과 해결 방법
- **Java Error: StackOverflowError**
  - https://inblog.ai/vlogue/java-error-javalangstackoverflowerror-25127
  - StackOverflowError 발생 원인과 해결책
- **자바 재귀호출 알고리즘**

    - https://velog.io/@ssuh0o0/JavaAlgorithm-재귀호출
    - 재귀 호출 설계 시 고려사항과 주의점
- **자바의 정석 - 재귀함수**
    - https://perfectacle.github.io/2017/02/11/Java-study-009day/
    - 재귀함수의 기본 원리와 스택 오버플로우 예방

05. Software Architecture 설계/구축 > 솔루션 설치, 구성 및 마이그레이션

문항5) Tomcat9 기반으로 프로젝트를 진행 중이다. 최초 설정한 Tomcat의 DB 연동 설정으로는 의도한 만큼 최소 커넥션 수가 유지되지 않아 여러 가지 테스트를 진행하였다. 다음 테스트에 대한 설명 중 틀린 것을 고르시오. [4점]

[테스트케이스 1]

- 테스트 결과 : WAS 기동 후 minIdle 수량으로 커넥션을 생성하지 않으며, DB 재기동 직후에도 minIdle 수량으로 커넥션이 유지되지 않는다.

```
<Resource auth="Container" defaultAutoCommit="false"
    driverClassName="oracle.jdbc.driver.OracleDriver"
    maxTotal="10" minIdle="2" maxIdle="5" maxWaitMillis="3000"
    name="jdbc/test_db"
    validationQuery="select 1 from dual" type="javax.sql.DataSource"
    url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=
    username="HR" password="1234qwer"  />
```

[테스트케이스 2]

- 테스트 결과 : WAS 기동 후 initialSize 수량으로 커넥션을 생성하였으나, DB 재기동 직후 minIdle이나 initialSize 수량으로 커넥션이 유지되지 않는다.

```
<Resource auth="Container" defaultAutoCommit="false"
    driverClassName="oracle.jdbc.driver.OracleDriver"
    maxTotal="10" minIdle="2" maxIdle="5" maxWaitMillis="3000"
    initialSize="5"
    name="jdbc/test_db"
    validationQuery="select 1 from dual" type="javax.sql.DataSource"
    url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HO
    username="HR" password="1234qwer"  />
```

[테스트케이스 3]

- 테스트 결과 : WAS 기동 후 initialSize 수량으로 커넥션을 생성하였고, DB 재기동 직후 minIdle 수량으로 커넥션이 유지되었다.

```
<Resource
    name="jdbc/test_db" auth="Container"
    type="javax.sql.DataSource" driverClassName="oracle.jdbc.driver.OracleDriver"
    url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=or
    username="HR" password="1234qwer"
    maxTotal="10" maxIdle="5" maxWaitMillis="3000" minIdle="2"
    initialSize="2"
    defaultAutoCommit="false" validationQuery="select 1 from dual"
    minEvictableIdleTimeMillis="30000"
    testOnBorrow="true"
    testWhileIdle="true"
    timeBetweenEvictionRunsMillis="30000"
      />
```

① 첫 테스트에도 minIdle이 포함되어 있었으나 최소 커넥션이 보장되지 않았다. 하지만 세번째 테스트 결과를 보면 30초마다 evictor가 동작하여 minIdle 만큼의 최소 커넥션을 유지해 주는 것을 알 수 있다.
② [테스트케이스 3]에서 minEvictableIdleTimeMillis 값이 DB의 idle session timeout이나 방화벽 존재 시 방화벽의 session timeout 보다 클 경우 해당 커넥션은 비정상적인 상태가 된다.
③ [테스트케이스 3]에서 만약 evictor 주기인 30초 간격 사이에 비정상 커넥션이 발생할 경우에 그 비정상 커넥션을 이용하여 SQL을 수행할 경우에는 응용 에러가 발생한다.
④ [테스트케이스 3]에서 evictor는 timeBetweenEvictionRunsMillis에 지정된 30초 주기로 동작하며 testWhileIdle 옵션이 false일 경우에는 동작하지 않는다.
⑤ evictor는 별도의 스레드가 백그라운드로 동작하며 [테스트케이스 3]에서 evictor가 수행한 validationQuery는 APM상에 나타나지 않는다.

(정답)3
(해설)evictor 설정과 별개로 testOnBorrow 옵션이 존재하므로, Tomcat JDBC Connection Pool 이 비정상 커넥션을 먼저 테스트하고 정상인 경우에만 사용자 트랜잭션에 빌려주게 되므로 응용 에러는 발생하지 않는다
(배점)4
(난이도)상
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 설계/구축 > 솔루션 설치, 구성 및 마이그레이션
- 문제 제목: Tomcat JDBC Connection Pool
- 출제 의도: Tomcat JDBC Connection Pool 에 대한 테스트 결과를 해석하고 올바른 결과를 도출할 수 있는지 확인한다.

해설

- **③ “비정상 커넥션을 이용하면 SQL 수행 시 응용 에러가 발생한다”**는 설명이 **틀렸습니다**.
    - 실제로 Tomcat JDBC Connection Pool은 `testOnBorrow=true` 일 경우 **커넥션을 빌려주기 전에 validationQuery 또는 isValid() 호출**을 통해 비정상 커넥션을 걸러냅니다 tomcat.apache.org+14tomcat.apache.org+14tomcat.apache.org+14.
    - 즉, **evictor 주기 사이에 비정상 커넥션이 발생하더라도**, 사용 시점에 **검증을 거쳐 정상 커넥션만 제공**하므로 응용 App에서는 오류가 발생하지 않습니다.

각 선택지 분석

① 첫 테스트에도 minIdle이 포함되어 있었으나 최소 커넥션이 보장되지 않았다. 하지만 세 번째 테스트 결과를 보면 30초마다 evictor가 동작하여 minIdle 만큼의 최소 커넥션을 유지해 주는 것을 알 수 있다.

✓ **올바른 설명**

- 테스트케이스 1: `minIdle="2"` 는 있지만 `timeBetweenEvictionRunsMillis` 설정이 없어서 evictor가 동작하지 않음
- 테스트케이스 3: `timeBetweenEvictionRunsMillis="30000"` 으로 30초마다 evictor가 실행되어 `minIdle="2"` 수량을 자동으로 보장

② [테스트케이스 3]에서 minEvictableIdleTimeMillis 값이 DB의 idle session timeout이나 방화벽 존재 시 방화벽의 session timeout 보다 클 경우 해당 커넥션은 비정상적인 상태가 된다.

✓ **올바른 설명**

- `minEvictableIdleTimeMillis="30000"` (30초)
- 만약 DB의 idle timeout이 20초라면, 커넥션이 30초 동안 유지되려고 하지만 DB에서는 20초 후 세션을 끊어버림
- 결과적으로 "좀비 커넥션" 상태가 되어 실제로는 사용할 수 없는 커넥션이 됨

③ [테스트케이스 3]에서 만약 evictor 주기인 30초 간격 사이에 비정상 커넥션이 발생할 경우에 그 비정상 커넥션을 이용하여 SQL을 수행할 경우에는 응용 에러가 발생한다.

✖ **틀린 설명** (정답)

- `testOnBorrow="true"` 설정으로 인해 **커넥션을 빌려줄 때마다** validation 쿼리가 실행됨
- evictor 주기(30초)와 관계없이 애플리케이션이 커넥션을 요청하는 순간에 즉시 검증
- 비정상 커넥션이면 해당 커넥션을 폐기하고 새로운 정상 커넥션을 생성하여 제공
- 따라서 **응용 에러가 발생하지 않음**

④ [테스트케이스 3]에서 evictor는 timeBetweenEvictionRunsMillis에 지정된 30초 주기로 동작하며 testWhileIdle 옵션이 false일 경우에는 동작하지 않는다.

✓ **올바른 설명**

- `timeBetweenEvictionRunsMillis="30000"` 으로 30초마다 evictor 실행
- `testWhileIdle="false"` 이면 evictor는 실행되지만 유휴 커넥션에 대한 validation은 수행하지 않음
- 하지만 현재 설정에서는 `testWhileIdle="true"` 로 설정되어 있음

⑤ evictor는 별도의 스레드가 백그라운드로 동작하며 [테스트케이스 3]에서 evictor가 수행한 validationQuery는 APM상에 나타나지 않는다.

✓ **올바른 설명**

- Evictor는 별도의 백그라운드 스레드에서 실행
- APM(Application Performance Monitoring) 도구는 일반적으로 애플리케이션 레벨의 SQL만 추적
- 커넥션 풀 내부에서 실행되는 validation 쿼리는 APM에서 모니터링되지 않음

정답: ③번

선택지 ③이 틀렸습니다. `testOnBorrow="true"` 설정으로 인해 커넥션 사용 전에 항상 유효성 검사가 수행되므로, evictor 주기와 관계없이 비정상 커넥션으로 인한 응용 에러는 방지됩니다.

핵심 포인트

testOnBorrow vs evictor의 차이점:

- **testOnBorrow**: 커넥션 요청 시점에 즉시 검증 (실시간 보호)
- **evictor**: 주기적으로 백그라운드에서 검증 (예방적 관리)

테스트케이스 3에서는 두 방식을 모두 사용하여 이중 보호 체계를 구축했습니다.

📌 보충 정리: 선택지별 설명

1. ①: evictor 작동으로 minIdle 수량을 주기적으로 유지하는 설명 – **정상**
2. ②: minEvictableIdleTimeMillis가 DB/방화벽 timeout보다 크면 커넥션이 비정상 상태가 될 수 있다는 설명 – **정상**
3. ③: 비정상 커넥션이 사용되어 응용 에러가 발생한다 – **틀림**
4. ④: evictor는 `testWhileIdle=false` 여도 `timeBetweenEvictionRunsMillis > 0` 등 조건이 충족되면 동작한다는 설명 – **정상**
5. ⑤: Evictor는 백그라운드 스레드로 동작하며 validationQuery는 APM에 안 잡힌다는 설명 – **정상**

✓ [테스트케이스 1]

🔧 설정 요약

항목	값
minIdle	5
initialSize	(미설정 또는 기본값)
timeBetweenEvictionRunsMillis	0 (비활성화됨)
minEvictableIdleTimeMillis	(없음)

🔍 해설

- **evictor 비활성화**: `timeBetweenEvictionRunsMillis=0` 이므로 백그라운드 스레드(evictor)가 동작하지 않음.
- **minIdle 작동 안함**: evictor가 커넥션 풀 상태를 주기적으로 점검하고 minIdle을 유지시켜야 하는데, evictor 자체가 없으니 **minIdle도 무의미**.
- **initialSize 없음**: 애플리케이션 기동 시 초기 커넥션 생성도 안 됨.

> *✓ 결과 요약:*
> *커넥션 풀 시작 시점에도, DB 재기동 후에도 커넥션이 생성되지 않음 → 예상된 동작*

✓ [테스트케이스 2]

🔧 설정 요약

항목	값
initialSize	5
minIdle	5

항목	값
timeBetweenEvictionRunsMillis	0 (비활성화됨)
minEvictableIdleTimeMillis	없음

🔍 해설

- **initialSize 작동**: 애플리케이션 기동 시 5개의 커넥션이 **초기 생성됨**
- **evictor 비활성화**: timeBetweenEvictionRunsMillis = 0 → **minIdle 유지 동작이 아예 수행되지 않음**
- **DB 재기동 시 커넥션 끊어짐**: evictor가 없어 커넥션 회복/재생성 불가 → 풀 비어 있음

> *✓ **결과 요약**:*
> *기동 시는 커넥션 5개 생성되지만, DB 재기동 후에는 **재연결 안됨** → **minIdle도 유지 안 됨***

✓ [테스트케이스 3]

🔧 설정 요약

항목	값
initialSize	5
minIdle	5
timeBetweenEvictionRunsMillis	30000 (30초)
minEvictableIdleTimeMillis	60000 (1분)
testWhileIdle	true
validationQuery	SELECT 1

🔍 해설

- **기동 시 initialSize 작동**: 5개 커넥션 생성됨

- **evictor 활성화**: 30초마다 실행 → idle 커넥션 수 점검 및 생성/제거 수행
- **testWhileIdle=true**: evictor가 커넥션 점검 시 validationQuery 사용
- **minIdle=5 유지**: 비정상 커넥션을 제거하고 다시 5개로 맞춰줌
- **DB 재기동 이후에도 자동 복구됨**

> ✓ ***결과 요약***:
> *기동 시도, DB 재기동 이후도 모두 minIdle 수량(5개)을 **정상 유지**함*

🔁 각 테스트 비교 요약

항목	테스트 1	테스트 2	테스트 3
initialSize	✕ 없음	✓ 있음	✓ 있음
minIdle	✓ 있음	✓ 있음	✓ 있음
evictor 동작	✕ 안 함	✕ 안 함	✓ 30초마다 실행
testWhileIdle	✕ (미설정)	✕ (미설정)	✓ true
minIdle 유지 여부	✕ 실패	✕ 실패	✓ 성공
DB 재기동 시 회복	✕ 안 됨	✕ 안 됨	✓ 자동 회복

📚 실시간 조회 가능한 참고 자료

1. **Tomcat 9 JDBC Connection Pool 공식 문서**
   👉 https://tomcat.apache.org/tomcat-9.0-doc/jdbc-pool.html
2. **DBCP2 설정 옵션 해설 – Velog 블로그**
   👉 https://velog.io/@eastperson/Tomcat-Connection-Pool-설정-정리
3. **TestOnBorrow vs TestWhileIdle – StackOverflow**
   👉 https://stackoverflow.com/questions/41998490/tomcat-jdbc-connection-pool-testonborrow-vs-testwhileidle

1. Apache Tomcat 공식 문서

URL: https://tomcat.apache.org/tomcat-9.0-doc/jdbc-pool.html **내용:** Tomcat JDBC Connection Pool 설정 및 속성 상세 설명

2. Apache Commons DBCP 문서

URL: https://commons.apache.org/proper/commons-dbcp/configuration.html **내용:** DBCP 설정 파라미터 및 동작 원리

3. Tomcat JNDI DataSource 예제

URL: https://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html **내용:** DataSource 설정 예제 및 베스트 프랙티스

4. Oracle 공식 문서 - Connection Pool Tuning

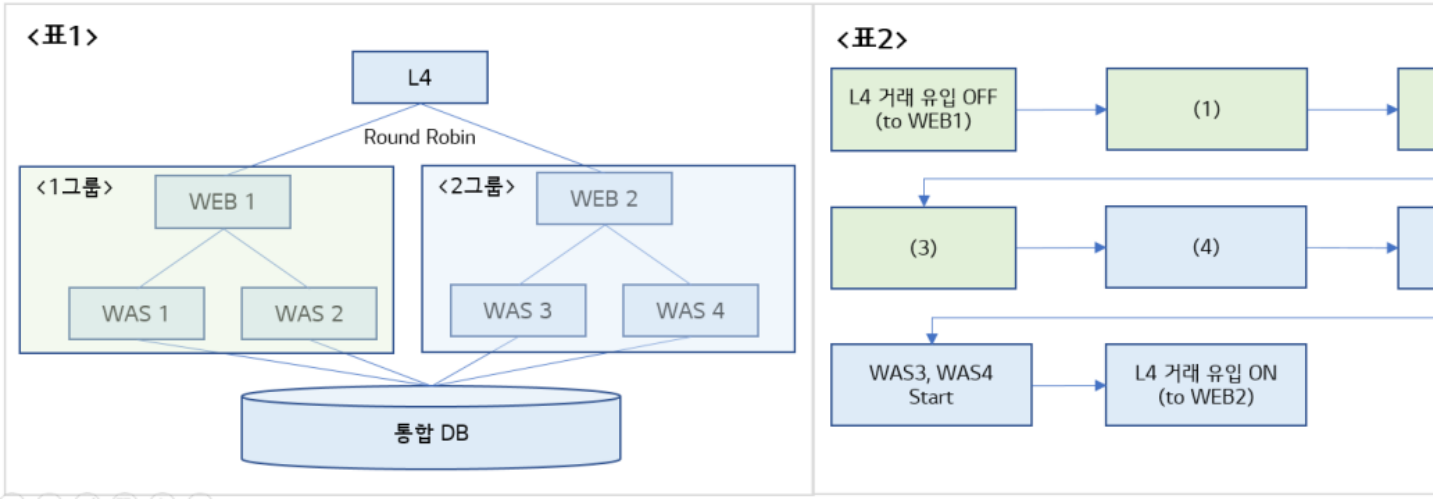
URL: https://docs.oracle.com/cd/E13222_01/wls/docs103/jdbc_admin/connection_pool.html **내용:** 커넥션 풀 튜닝 가이드라인

06. Software Architecture 설계/구축 > 개발/배포/모니터링 체계 구축

문항6) 운영중인 시스템에 신규 서비스 또는 기능을 배포할 경우 서비스 중단 없이 운영 시스템에 반영하고자 한다.

<표2>의 빈칸에 들어갈 TASK 명을 <보기>에서 골라 순서대로 나열한 것을 고르시오. [3점]
[참고 사항]

- 서비스 무중단으로 반영하고자 하는 운영 시스템 구성도는 <표1>과 같이 L4, WEB 서버 2대, 로컬 디스크를 사용하는 WAS 4대로 구성되어 있다.
- <표2>는 서비스 무중단 반영 프로세스이다.



<보기>
가. WAS1, WAS2 Stop
나. L4 거래 유입 ON (to WEB1)
다. WAS3, WAS4 Stop
라. 프로그램 반영 (WAS1, WAS2)
마. 프로그램 반영 (WAS3, WAS4)
바. L4 거래 유입 OFF (to WEB2)

① (가) – (나) – (다) – (마) – (라) – (마)
② (가) – (라) – (나) – (다) – (마) – (바)
③ (가) – (라) – (나) – (바) – (다) – (마)
④ (가) – (라) – (바) – (나) – (다) – (마)
⑤ (가) – (바) – (나) – (라) – (다) – (마)

(정답)3
(해설)WAS 반영 전에 반드시 L4 거래 유입을 차단해야 하고, WAS 반영이 완료된 이후, L4 거래 유입을 해제한다.
(배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 설계/구축 > 개발/배포/모니터링 체계 구축

- 문제 제목: 서비스 무중단 WAS 롤링 배포
- 출제 의도: 서비스 무중단 WAS 롤링 배포 프로세스 이해

---

07. Software Architecture 설계/구축 > 개발/배포/모니터링 체계 구축

문항7) 대다수 프로젝트에서 로컬에서 운영 DB 계정을 사용하여 직접 운영DB로 access하는 것은 허용하지 않는다. 또한 운영 DB 계정 정보가 개발자 등 다수 인원에 노출되지 않도록 관리하고 있다.
아래 내용 중 DataSource 설정 시 계정정보 관리와 관련한 내용 중 옳지 않은 것을 고르시오. [4점]

① WebLogic, Jeus 등의 상용 WAS 제품에는 DB 계정정보 암호화 기능이 포함되어 있기 때문에 JNDI 방식을 사용할 경우 별도의 암호화 처리가 불필요하다.
② Tomcat JDBC Connection Pool에서는 기본 제공하는 계정정보 암호화 기능이 없기 때문에 별도 DataSourceFactory 클래스 구현을 통해 암복호화 기능을 추가할 수 있다.
③ 계정정보 암호화를 거치더라도 설정파일 레벨에서 해당되는 내용이며 메모리 상에는 비밀번호가 복호화 되어 평문으로 저장될 수 있다. 힙덤프를 통해 메모리 내용을 확인해보면 비밀번호가 노출될 수 있다.
④ 로컬 개발환경에는 JNDI 방식보다는 Connection Pool을 어플리케이션에서 직접 사용하는 경우가 많다. 개발/테스트/운영환경 설정파일에 비밀번호가 노출되지 않도록 암복호화 설정을 해주어야 한다.
⑤ 운영 DB 계정정보는 단방향 암호화를 사용하여 비밀번호는 얻어낼 수 없도록 관리한다.

(정답)5
(해설)DB 계정정보를 전달하여 JDBC Driver 객체를 생성해야 하므로, 단방향 암호화가 아닌 양방향 암호화를 통해 복호화 할 수 있어야 한다.
(배점)4
(난이도)중 - 문제유형: 선다형 - 출제영역 대분류>소분류: Software Architecture 설계/구축 > 보안 (보안 취약점, 권한 관리, 암호화 등)
- 문제 제목: DB 계정정보 암호화 설정
- 출제 의도: WAS 설정 시 DB 계정정보 암호화에 대한 이해도를 확인한다.

정답: ⑤

정답 해설: 선택지 ⑤번이 틀렸습니다. DB 연결에 사용되는 계정정보(비밀번호)는 단방향 암호화가 아닌 양방향 암호화를 사용해야 합니다. 단방향 암호화는 복호화가 불가능하므로 DB 연결 시 실제 비밀번호를 사용할 수 없기 때문 SinsiwayKrauser085입니다.

각 선택지 상세 분석

① WebLogic, Jeus 등의 상용 WAS 제품에는 DB 계정정보 암호화 기능이 포함되어 있기 때문에 JNDI 방식을 사용할 경우 별도의 암호화 처리가 불필요하다.

✓ **올바른 설명**

- WebLogic과 JEUS 같은 상용 WAS는 JNDI 데이터소스 설정 시 자체적인 암호화 기능을 제공 NaverTmaxsoft
- 관리 콘솔에서 비밀번호를 입력하면 자동으로 암호화되어 저장됨

② Tomcat JDBC Connection Pool에서는 기본 제공하는 계정정보 암호화 기능이 없기 때문에 별도 DataSourceFactory 클래스 구현을 통해 암복호화 기능을 추가할 수 있다.

✓ **올바른 설명**

- Tomcat은 기본적으로 계정정보 암호화 기능을 제공하지 않음 Apache Tomcat 9 (9.0.106) - JNDI Resources How-To
- 사용자 정의 DataSourceFactory를 구현하여 암복호화 로직을 추가해야 함

③ 계정정보 암호화를 거치더라도 설정파일 레벨에서 해당되는 내용이며 메모리 상에는 비밀번호가 복호화 되어 평문으로 저장될 수 있다. 힙덤프를 통해 메모리 내용을 확인해보면 비밀번호가 노출될 수 있다.

✓ **올바른 설명**

- 암호화는 설정파일 저장 시에만 적용되고, 런타임에는 복호화된 평문이 메모리에 존재
- 힙 덤프나 메모리 분석을 통해 평문 비밀번호가 노출될 가능성 있음

④ 로컬 개발환경에는 JNDI 방식보다는 Connection Pool을 어플리케이션에서 직접 사용하는 경우가 많다. 개발/테스트/운영환경 설정파일에 비밀번호가 노출되지 않도록 암복호화 설정을 해주어야 한다.

✓ **올바른 설명**

- 개발환경에서는 애플리케이션에서 직접 커넥션 풀을 관리하는 경우가 일반적 JNDI 사용하여 DB연결하기.
- 모든 환경에서 설정파일의 비밀번호 보안이 필요함

⑤ 운영 DB 계정정보는 단방향 암호화를 사용하여 비밀번호는 얻어낼 수 없도록 관리한다.

✖ **틀린 설명** (정답)

- 단방향 암호화는 복호화가 불가능한 해시 방식으로, 주로 사용자 비밀번호 저장에 사용 SinsiwayVelog
- DB 연결에는 실제 비밀번호가 필요하므로 **양방향 암호화(복호화 가능)**를 사용해야 함
- 단방향 암호화된 값으로는 DB에 연결할 수 없음 개인정보 암호화에 대해 - Hello, world! I'm Yongdeok An

---

암호화 방식 구분

단방향 암호화 (Hash)

- 복호화 불가능, 주로 사용자 비밀번호 저장용 신시웨이 | 데이터베이스 보안 전문 기업 - PR - 뉴스룸
- 예: 웹사이트 로그인 비밀번호

양방향 암호화 (대칭키/비대칭키)

- 복호화 가능, DB 연결정보 같은 설정값 암호화용 클라우드 DB 암호화란? | 가비아 라이브러리
- 예: DB 계정정보, API 키 등

참고 사이트 (현재 존재하는 페이지들)

암호화 기본 개념

1. **신시웨이 - 암호화 알고리즘 기본**
    - https://www.sinsiway.com/kr/pr/blog/view/412/page/9
    - 단방향/양방향 암호화의 차이점과 활용법 신시웨이 | 데이터베이스 보안 전문 기업 - PR - 뉴스룸
2. **개인정보 암호화 가이드**
    - https://krauser085.github.io/encryption/
    - 개인정보보호법에 따른 암호화 의무사항 개인정보 암호화에 대해 - Hello, world! I'm Yongdeok An

DB 암호화 관련

1. **가비아 - 클라우드 DB 암호화**
    - https://library.gabia.com/contents/infrahosting/8977/
    - DB 암호화 방식과 구현 방법 클라우드 DB 암호화란? | 가비아 라이브러리
2. **펜타시큐리티 - 데이터베이스 암호화**
    - https://www.pentasecurity.co.kr/database-encryption/
    - 엔터프라이즈급 DB 암호화 솔루션 데이터베이스 암호화 (Database Encryption) | 펜타시큐리티

WAS별 JNDI 설정

1. **WebLogic JNDI 공식 문서**
    - https://docs.oracle.com/middleware/1213/wls/WJNDI/wls_jndi.htm
    - WebLogic JNDI 설정과 보안 기능 Understanding WebLogic JNDI
2. **JEUS 데이터소스 설정**
    - https://technet.tmaxsoft.com/upload/download/online/jeus/pver-20140203-000001/server/chapter_datasource.html
    - JEUS DB 연결 설정과 관리 방법 제8장 DB Connection Pool과 JDBC
3. **Tomcat JNDI 공식 문서**
    - https://tomcat.apache.org/tomcat-9.0-doc/jndi-resources-howto.html
    - Tomcat JNDI 리소스 설정 가이드 Apache Tomcat 9 (9.0.106) - JNDI Resources How-To

---

08. Software Architecture 운영/문제해결 > 로그/덤프 생성, 관리 및 분석

문항 8) (문항 수정)MSA 시스템에서는 Trace ID와 Span ID를 사용하여 분산 로그 추적(Distributed Tracing)수행할 수 있다. 이와 관련된 설명 중 옳지 않은 것을 고르시오. [3점]

[설명]

- Trace ID : 하나의 요청(Request)이 여러 서비스를 거치는 전체 흐름을 식별하는 고유한 ID
- Span ID : Trace 내에서 하나의 작업 단위(예: 하나의 서비스 호출, 데이터베이스 쿼리 등)를 식별하는 ID

① Trace ID 및 Span ID는 분산 로그 추적 대상 서비스 간에 누락 없이 전달되어야 한다.
② REST뿐만 아니라 Kafka와 같은 비동기 메시지 기반 호출에서도 Trace ID가 전달되어야 한다.
③ 시각화 도구에서는 Span ID만 있으면 전체 시스템의 서비스 호출 관계를 보여줄 수 있다.
④ 사용자 요청 처리 중 여러 서비스 간의 호출이 발생하더라도 Trace ID는 유지되어야 한다.
⑤ 트레이싱을 모든 요청에 적용할 경우, 성능 저하와 네트워크 부하가 발생할 수 있으므로 샘플링 기반 접근 방식이 필요하다.

(정답)3
(해설)시각화 도구에서는 Trace ID와 Span ID를 함께 사용하여 전체 시스템 호출 흐름을 확인한다.
(배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 운영/문제해결 > 로그/덤프 생성, 관리 및 분석
- 문제 제목: 분산 로그 추적
- 출제 의도: Spring Cloud Sleuth의 특징에 대한 이해

정답: ③

정답 해설: 선택지 ③번이 틀렸습니다. 시각화 도구에서 전체 시스템의 서비스 호출 관계를 보여주려면 Trace ID가 반드시 필요합니다. Span ID만으로는 개별 작업 단위만 식별할 수 있고, 전체 요청 흐름을 연결할 수 없기 때문 New RelicAmazon입니다.

각 선택지 상세 분석

① Trace ID 및 Span ID는 분산 로그 추적 대상 서비스 간에 누락 없이 전달되어야 한다.

✓ 올바른 설명

- Trace ID를 전달하는 프로세스를 메타데이터 전파 또는 분산 컨텍스트 전파라고 부르며, 이는 분산 추적의 핵심 Google CloudAmazon

- HTTP 헤더를 통해 트레이스 ID, 스팬 ID 등의 메타데이터가 서비스 간에 전파됨 spring cloud sleuth - How to send trace ID through kafka - Stack Overflow

② REST뿐만 아니라 Kafka와 같은 비동기 메시지 기반 호출에서도 Trace ID가 전달되어야 한다.

✓ **올바른 설명**

- Kafka에서는 소스 시스템에서 Kafka로, 그리고 Kafka에서 대상 시스템까지 메시지의 엔드 투 엔드 추적을 지원 Kafka와 Exactly Once
- Spring Cloud Sleuth를 사용하여 Kafka를 통한 마이크로서비스 간 Trace ID 전달이 가능

③ 시각화 도구에서는 Span ID만 있으면 전체 시스템의 서비스 호출 관계를 보여줄 수 있다.

✖ **틀린 설명** (정답)

- Trace ID는 분산 추적에서 스팬들을 연결하는 데 도움이 되는 고유한 ID 분산 추적 완벽 가이드 | 뉴렐릭
- Span ID는 개별 작업 단위를 식별하지만, Trace ID가 있어야 전체 요청의 스팬들을 연결할 수 있음 Traces | OpenTelemetry
- 트레이스 내 모든 스팬의 관계를 나타내는 상위/하위 ID를 표시하려면 Trace ID가 필수 분산 추적 시스템 - AWS X-Ray - Amazon Web Services

④ 사용자 요청 처리 중 여러 서비스 간의 호출이 발생하더라도 Trace ID는 유지되어야 한다.

✓ **올바른 설명**

- 각 스팬은 해당 스팬이 속한 원래 요청의 동일한 트레이스 ID를 상속 분산 추적이란? - 분산 추적 설명 - AWS
- 트레이스 ID는 각 스팬에 공통으로 부여되며, 이를 통해 전체 요청 흐름을 추적 9.4. 분산 추적 | Red Hat Product Documentation

⑤ 트레이싱을 모든 요청에 적용할 경우, 성능 저하와 네트워크 부하가 발생할 수 있으므로 샘플링 기반 접근 방식이 필요하다.

✓ **올바른 설명**

- 계측의 세부정보 수준에 따라서는 데이터 추적이 프로젝트 비용에 영향을 줄 수 있어 대부분의 trace 시스템은 여러 양식의 샘플링을 사용 마이크로서비스 애플리케이션의 분산 추적 | Cloud Architecture Center | Google Cloud
- 뉴렐릭의 APM 언어 에이전트는 적응형 샘플링을 사용하여 시스템 활동의 대표적인 샘플을 캡처 기술 분산 추적 세부정보 | New Relic Documentation

---

분산 추적의 핵심 개념

Trace ID vs Span ID

- **Trace ID**: 하나의 요청이 여러 서비스를 거치는 전체 흐름을 식별하는 고유한 ID 분산 추적 완벽 가이드 | 뉴렐릭
- **Span ID**: 특정 서비스 내에서의 개별 작업 또는 요청을 나타내는 ID 9.4. 분산 추적 | Red Hat Product Documentation

분산 추적의 동작 원리

- 요청이 한 서비스에서 다른 서비스로 이동할 때 데이터를 수집하여 여정의 각 세그먼트를 범위로 기록 분산 추적 소개 | New Relic Documentation
- 추적 컨텍스트는 네트워크 또는 메시지 버스를 통해 서비스에서 서비스로 전달 kafka 설정을 사용한 문제해결

참고 사이트 (현재 존재하는 페이지들)

분산 추적 기본 개념

1. **뉴렐릭 분산 추적 완벽 가이드**
    - https://newrelic.com/kr/blog/best-practices/distributed-tracing-guide
    - Trace ID와 Span ID의 역할과 중요성 분산 추적 완벽 가이드 | 뉴렐릭
2. **AWS 분산 추적 설명**
    - https://aws.amazon.com/ko/what-is/distributed-tracing/
    - 분산 추적 시스템의 작동 원리와 구현 방법 분산 추적이란? - 분산 추적 설명 - AWS

기술적 구현 방법

1. **Google Cloud 마이크로서비스 분산 추적**
    - https://cloud.google.com/architecture/microservices-architecture-distributed-tracing?hl=ko
    - OpenTelemetry를 사용한 분산 추적 구현 마이크로서비스 애플리케이션의 분산 추적 | Cloud Architecture Center | Google Cloud
2. **Spring Cloud Sleuth 분산 추적**
    - https://velog.io/@ayoung3052/분산-추적-Spring-Cloud-Sleuth-및-로깅-Zipkin
    - Spring 기반 마이크로서비스에서의 분산 추적 구현 9.4. 분산 추적 | Red Hat Product Documentation

Kafka와 분산 추적

1. **Red Hat Kafka 분산 추적 문서**
    - https://docs.redhat.com/ko/documentation/red_hat_streams_for_apache_kafka/2.6/html/amq_streams_on_openshift_overview/metrics-overview-tracing_str
    - Kafka를 통한 엔드 투 엔드 메시지 추적 Kafka와 Exactly Once

고급 개념

1. **Elastic APM과 OpenTracing**
    - https://www.elastic.co/kr/blog/distributed-tracing-opentracing-and-elastic-apm
    - 표준화된 분산 추적 API와 구현 방법 kafka 설정을 사용한 문제해결
2. **OpenTelemetry 공식 문서**
    - https://opentelemetry.io/docs/concepts/signals/traces/
    - Trace와 Span의 표준 정의와 구조 Traces | OpenTelemetry
3. **뉴렐릭 분산 추적 기술 세부사항**
    - https://docs.newrelic.com/kr/docs/distributed-tracing/concepts/how-new-relic-distributed-tracing-works/
    - W3C 표준과 헤더 전파 메커니즘 spring cloud sleuth - How to send trace ID through kafka - Stack Overflow

09. Software Architecture 운영/문제해결 > 모니터링(항목, 지표, 도구활용)

문항 9) 아래 표는 MSA 기반의 프로젝트에서 선정한 기술요소(Matrix 단어 삭제)표이다. 빈칸에 알맞은 기술 요소가 보기에서 알맞게 짝지어진 것을 고르시오. [3점]

NO.	구분	기술요소	설명
1	External Gateway	Istio Ingress Gateway	API 라우팅, 인증
2	Service Mesh	Istiod	서비스 라우팅
		(1)	Tracing
		Kiali	Topology
3	Runtime Platform	OpenShift Container Platform	컨테이너 관리
4	Backing Service	Redis	캐시
		MySQL	RDBMS
5	Telemetry	(2)	로그 저장소
		(3)	로그 수집
		Kibana	로깅대시보드
		Prometheus	매트릭 수집
		(4)	매트릭 대시보드
6	CI/CD Automation	Jenkins	빌드/배포관리
		GitLab	소스관리
		Nexus	Library 저장소
		Docker-distribution	Private Image Registry

[보기]
Elastic Search, Jaeger, Fluentd, Grafana, Kafka, Jennifer, Subversion
① Jaeger, Elastic Search, Fluentd, Grafana
② Elastic Search, Jaeger, Fluentd, Grafana
③ Jaeger, Fluentd, Kafka, Grafana
④ Elastic Search, Fluentd, Fluentd, Jennifer
⑤ Elastic Search, Subversion, Kafka, Grafana

(정답)1
(해설) MSA 환경의 기본 기술요소
(배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역: 대분류>소분류: Software Architecture 운영/문제해결 > 모니터링(항목, 지표, 도구활용)
- 문제 제목: MSA 기반 환경의 모니터링 도구 활용
- 출제 의도: MSA 기반 환경의 모니터링 도구 이해도

참고 사이트

클라이언트 보안 및 인증

1. **프론트엔드 안전한 로그인 처리**
    - https://velog.io/@yaytomato/프론트에서-안전하게-로그인-처리하기
    - React에서 안전한 인증 방식과 보안 취약점 대응 Vue를 이용할 때 역할 기반 인증방식에 대한 보안처리 · Issue #145 · codingeverybody/codingyahac
2. **React Router 권한 기반 라우팅**
    - https://jeonghwan-kim.github.io/dev/2020/03/20/role-based-react-router.html
    - React에서 권한별 라우팅 제어 구현 방법 [Next.js 2.0] 간단한 React 전용 서버사이드 프레임워크, 기초부터 본격적으로 파보기 | VELOPERT.LOG
3. **Vue 권한 인증 보안 처리**
    - https://github.com/codingeverybody/codingyahac/issues/145
    - 클라이언트에서 권한 체크의 보안 위험성 마이크로 서비스의 컨테이너 오케스트레이션 - Azure Architecture Center | Microsoft Learn

컨테이너 오케스트레이션

1. **IBM 컨테이너 오케스트레이션 가이드**
    - https://www.ibm.com/kr-ko/think/topics/container-orchestration
    - 컨테이너 오케스트레이션의 개념과 필요성 Spring Cloud로 개발하는 마이크로서비스 애플리케이션 (MSA) 강의 | Dowon Lee - 인프런
2. **AWS 컨테이너 오케스트레이션 설명**
    - https://aws.amazon.com/ko/what-is/container-orchestration/
    - 대규모 애플리케이션 배포를 위한 컨테이너 관리 자동화 [Docker] 컨테이너 오케스트레이션
3. **Azure 마이크로서비스 컨테이너 오케스트레이션**
    - https://learn.microsoft.com/ko-kr/azure/architecture/microservices/design/orchestration
    - 마이크로서비스 환경에서의 컨테이너 오케스트레이션 필요성 마이크로 서비스 아키텍처와 개발문화

Spring Boot 및 마이크로서비스

1. **인프런 Spring Cloud MSA 강의**
    - https://www.inflearn.com/course/스프링-클라우드-마이크로서비스

- Spring Cloud를 이용한 마이크로서비스 애플리케이션 개발

2. **마이크로서비스 아키텍처와 개발문화**
    - https://brunch.co.kr/@maengdev/3
    - MSA 도입과 Spring Boot, Kubernetes 환경 구축

Next.js 및 현대적 개발

1. **Next.js React 기본사항**
    - https://wikidocs.net/206500
    - 서버/클라이언트 컴포넌트와 하이브리드 애플리케이션 react | ★★★ Nextjs 인증가이드 - Nextjs 15 + Next Auth V5 + typescript + shadcn 를 oauth 인증, Credential Provider 사용, Next.js + Prisma + Supabase 조합+MongoDB 적용,미들웨어 , 커스텀 백엔드로 토큰 관리하기 | 마카로닉스
2. **Next.js 인증 가이드**
    - https://macaronics.net/index.php/m04/react/view/2378
    - Next.js 15 + Next Auth V5를 이용한 인증 구현 Kubernetes - 컨테이너 오케스트레이션 (Container Orchestration) 이란?

10. Software Architecture 설계/구축 > SW 아키텍처 설계

문항 10) (문항 교체)다음은 시스템 개발 환경 및 구성에 대한 트렌드 변화에 대한 설명이다. 다음 설명 중 틀린 것을 고르시오. [3점]

① 기존에는 UI와 비즈니스 로직이 결합된 방식으로 HTML을 생성했지만, 최근에는 UI와 비즈니스 로직을 분리하고, 컴포넌트 기반의 구조를 적용하는 추세이다. React, Next.js, SvelteKit 등 최신 프레임워크를 사용해 UI를 개발하고, 정적 혹은 서버 렌더링 기반으로 빌드된 결과물을 Web 서버나 CDN에 배포하여 빠르고 유연한 사용자 경험을 제공한다
② React, Next.js 등의 프레임워크로 개발하는 경우, 사용자 권한에 따라 UI 화면을 동적으로 구성해야 하므로, 권한 체크 로직은 서버에서 처리하는 대신 클라이언트 UI 애플리케이션으로 이전하여 처리하는 방식이 일반화되고 있다.
③ 최근 서버 프레임워크의 트렌드는 단순한 기능 제공을 넘어, 서비스 간 연동과 메시지 기반의 아우터 아키텍처, 클라우드 네이티브 환경을 효율적으로 구성하는 데까지 확장되고 있다. 특히 Spring Boot는 기본 제공되는 설정과 자동 구성 기능을 통해, REST API, 이벤트 기반 아키텍처, 컨테이너 환경 등 다양한 형태로 손쉽게 적용할 수 있는 유연함이 큰 장점이다.
④ 기존의 WAS는 고정된 수의 인스턴스를 수동으로 운영, 관리하는 방식이 일반적이었지만, 최근에는 Kubernetes와 같은 컨테이너 오케스트레이션 플랫폼이 보편화되면서, 트래픽 변화에 따라 자동으로 확장(Scale Out) 또는 축소(Scale In)되는 동적 WAS 환경을 도입하는 추세이다.

(정답)2
(해설) 오히려 UI와 서버 어플리케이션의 역할이 구분되었기 때문에, UI에서 처리하는 부분도 서버에서 cross check할 필요가 있다. 특히 권한과 같은 경우 client에서 조작이 가능한 점을 고려하여 체크가 필요하다
(배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역: 대분류>소분류: Software Architecture 설계/구축 > SW 아키텍처 설계
- 문제 제목: 개발 환경 및 구성의 트렌드 변화
- 출제 의도: 개발 환경 및 구성의 트렌드 변화를 이해하고 있는지 확인한다.

정답: ②

정답 해설: 선택지 ②번이 틀렸습니다. 클라이언트에서 권한 체크를 하는 것은 보안상 매우 위험하며, 권한 체크는 반드시 서버에서 처리해야 합니다. 클라이언트에서의 권한 체크는 단순히 UI 표시를 위한 것이며, 실제 보안은 서버 API에서 담당해야 함 GitHubVelopert입니다.

각 선택지 상세 분석

① 기존에는 UI와 비즈니스 로직이 결합된 방식으로 HTML을 생성했지만, 최근에는 UI와 비즈니스 로직을 분리하고, 컴포넌트 기반의 구조를 적용하는 추세이다. React, Next.js, SvelteKit 등 최신 프레임워크를 사용해 UI를 개발하고, 정적 혹은 서버 렌더링 기반으로 빌드된 결과물을 Web 서버나 CDN에 배포하여 빠르고 유연한 사용자 경험을 제공한다.

✓ **올바른 설명**

- 전통적인 서버 사이드 렌더링에서 클라이언트-서버 분리 아키텍처로의 전환
- 서버 컴포넌트와 클라이언트 컴포넌트를 사용하여 클라이언트와 서버를 모두 활용하는 하이브리드 애플리케이션 구현 react | ★★★ Nextjs 인증가이드 - Nextjs 15 + Next Auth V5 + typescript + shadcn 를 oauth 인증, Credential Provider 사용, Next.js + Prisma + Supabase 조합+MongoDB 적용,미들웨어 , 커스텀 백엔드로 토큰 관리하기 | 마카로닉스
- 컴포넌트 기반 개발과 정적/서버 렌더링을 통한 성능 최적화가 현재 트렌드

② React, Next.js 등의 프레임워크로 개발하는 경우, 사용자 권한에 따라 UI 화면을 동적으로 구성해야 하므로, 권한 체크 로직은 서버에서 처리하는 대신 클라이언트 UI 애플리케이션으로 이전하여 처리하는 방식이 일반화되고 있다.

✗ **틀린 설명** (정답)

- 클라이언트에서의 인증 처리는 보안상 매우 위험하며, 웹 어플리케이션이 XSS 공격에 취약하다면 어떤 방식을 선택하든 보안이 위험 Vue를 이용할 때 역할 기반 인증방식에 대한 보안처리 · Issue #145 · codingeverybody/codingyahac
- 상용 앱에서는 절대 클라이언트에서 인증 처리를 하면 안 됨 운영 수준의 컨테이너 오케스트레이션
- 클라이언트에서 권한에 따라 버튼을 보이거나 숨기는 것은 하이잭이나 인젝팅 공격에 취약 마이크로 서비스의 컨테이너 오케스트레이션 - Azure Architecture Center | Microsoft Learn
- 올바른 방식: 서버에서 권한 검증, 클라이언트는 UI 표시만 담당

③ 최근 서버 프레임워크의 트렌드는 단순한 기능 제공을 넘어, 서비스 간 연동과 메시지 기반의 아우터 아키텍처, 클라우드 네이티브 환경을 효율적으로 구성하는 데까지 확장되고 있다. 특히 Spring Boot는 기본 제공되는 설정과 자동 구성 기능을 통해, REST API, 이벤트 기반 아키텍처, 컨테이너 환경 등 다양한 형태로 손쉽게 적용할 수 있는 유연함이 큰 장점이다.

✓ **올바른 설명**

- Spring Cloud를 이용한 마이크로서비스 아키텍처 개발과 클라우드 네이티브 환경 구축이 현재 트렌드
- Spring Boot의 자동 설정과 스타터 의존성을 통한 빠른 개발 환경 구축
- 마이크로서비스, 이벤트 기반 아키텍처, 컨테이너 환경 지원

④ 기존의 WAS는 고정된 수의 인스턴스를 수동으로 운영, 관리하는 방식이 일반적이었지만, 최근에는 Kubernetes와 같은 컨테이너 오케스트레이션 플랫폼이 보편화되면서, 트래픽 변화에 따라 자동으로 확장(Scale Out) 또는 축소(Scale In)되는 동적 WAS 환경을 도입하는 추세이다.

✓ **올바른 설명**

- 컨테이너 오케스트레이션을 통해 컨테이너화된 애플리케이션의 배포, 확장, 수명 주기를 자동으로 관리 InflearnVelog
- Kubernetes는 CPU 사용량에 따라 자동으로 애플리케이션의 스케일을 업 또는 다운 컨테이너 오케스트레이션: 컨테이너 관리 및 자동화 방법
- 컨테이너 오케스트레이션은 대규모 및 동적 프로덕션 환경에 매우 중요 마이크로 서비스 아키텍처와 개발문화

첨부 이미지 분석

첨부된 이미지는 전통적인 UI와 Server의 분리된 아키텍처를 보여주고 있습니다:

- **UI App**: UI Framework, Browser
- **Server App**: Server Framework, WAS
- **통신**: REST API 호출

이는 선택지 ①에서 언급한 UI와 비즈니스 로직 분리 트렌드를 시각적으로 보여주는 구조입니다.

참고 사이트

클라이언트 보안 및 인증

1. **프론트엔드 안전한 로그인 처리**
   - https://velog.io/@yaytomato/프론트에서-안전하게-로그인-처리하기
   - React에서 안전한 인증 방식과 보안 취약점 대응 Vue를 이용할 때 역할 기반 인증방식에 대한 보안처리 · Issue #145 · codingeverybody/codingyahac
2. **React Router 권한 기반 라우팅**
   - https://jeonghwan-kim.github.io/dev/2020/03/20/role-based-react-router.html
   - React에서 권한별 라우팅 제어 구현 방법 [Next.js 2.0] 간단한 React 전용 서버사이드 프레임워크, 기초부터 본격적으로 파보기 | VELOPERT.LOG
3. **Vue 권한 인증 보안 처리**
   - https://github.com/codingeverybody/codingyahac/issues/145

- 클라이언트에서 권한 체크의 보안 위험성 마이크로 서비스의 컨테이너 오케스트레이션 - Azure Architecture Center | Microsoft Learn

컨테이너 오케스트레이션

1. **IBM 컨테이너 오케스트레이션 가이드**
   - https://www.ibm.com/kr-ko/think/topics/container-orchestration
   - 컨테이너 오케스트레이션의 개념과 필요성 Spring Cloud로 개발하는 마이크로서비스 애플리케이션 (MSA) 강의 | Dowon Lee - 인프런
2. **AWS 컨테이너 오케스트레이션 설명**
   - https://aws.amazon.com/ko/what-is/container-orchestration/
   - 대규모 애플리케이션 배포를 위한 컨테이너 관리 자동화 [Docker] 컨테이너 오케스트레이션
3. **Azure 마이크로서비스 컨테이너 오케스트레이션**
   - https://learn.microsoft.com/ko-kr/azure/architecture/microservices/design/orchestration
   - 마이크로서비스 환경에서의 컨테이너 오케스트레이션 필요성 마이크로 서비스 아키텍처와 개발문화

Spring Boot 및 마이크로서비스

1. **인프런 Spring Cloud MSA 강의**
   - https://www.inflearn.com/course/스프링-클라우드-마이크로서비스
   - Spring Cloud를 이용한 마이크로서비스 애플리케이션 개발
2. **마이크로서비스 아키텍처와 개발문화**
   - https://brunch.co.kr/@maengdev/3
   - MSA 도입과 Spring Boot, Kubernetes 환경 구축

Next.js 및 현대적 개발

1. **Next.js React 기본사항**
   - https://wikidocs.net/206500
   - 서버/클라이언트 컴포넌트와 하이브리드 애플리케이션 react | ★★★ Nextjs 인증가이드 - Nextjs 15 + Next Auth V5 + typescript + shadcn 를 oauth 인증, Credential Provider 사용, Next.js + Prisma + Supabase 조합+MongoDB 적용,미들웨어 , 커스텀 백엔드로 토큰 관리하기 | 마카로닉스
2. **Next.js 인증 가이드**
   - https://macaronics.net/index.php/m04/react/view/2378
   - Next.js 15 + Next Auth V5를 이용한 인증 구현 Kubernetes - 컨테이너 오케스트레이션 (Container Orchestration) 이란?

11. Software Architecture 운영/문제해결 > 성능 개선 및 문제해결

문항11) 사용자 관점의 페이지 로딩속도 개선을 위해 Chrome 개발자 도구를 활용하여 성능 Profiling을 위해 Reload 하는 시점부터 페이지 로드가 완료된 시점까지를 프로파일링 하였다. 아래 결과를 보고 해석한 내용으로 적절하지 않은 것을 2개 고르시오. [4점]

[Event 범례]

ⓐ FP: First Paint
ⓑ FCP: First Contentful Paint
ⓒ DCL: DOMContentLoaded
ⓓ LCP: Largest Contentful Paint
ⓔ Layout Shift
ⓕ TTFP: Time To First Byte

① Layout Shift가 발생하는 동안 페이지 로딩이 중단되므로, 자주 호출되는 함수들을 모듈화하여 중복 실행을 줄일 수 있는지 검토한다
② 웹 페이지 로딩 과정 중 “Evaluate Script”에 소요되는 시간을 줄여야 하므로 Code Splitting을 통해 로딩에 필요한 코드만 실행되도록 개선한다.
③ LCP가 4초 정도에 관측되므로 렌더링 차단을 방지할 수 있는 JS/CSS를 async나 defer와 같은 Attribute를 활용하여 비동기 방식으로 로드할 수 있는지 검토한다.
④ FCP 및 DOM Content Loaded Event가 2초 안에 발생하므로 화면에 주요 컨텐츠가 로드되는 시간은 적절한 수준이라고 판단할 수 있다.

⑤ 최초 요청 시 TTFB(Time To First Byte)가 1초 정도 소요되므로 CDN 및 캐싱을 고려하여 페이지 로드 시간을 개선할 필요가 있다.

(정답)1{|}4

(해설)
1 – Layout Shift 는 페이지 로딩에 걸린 시간보다 Contents 로드 중 Layout 변경이 일어나는 것과 관련 있다.
4 - DOM Content Loaded Event 발생보다 사용자 경험 개선을 위한 Largest Contentful Paint 이벤트에 집중해서 성능을 개선할 필요가 있다.
(배점)4
(난이도)중
- 문제유형: 선다형
- 출제영역 대분류>소분류 : Software Architecture 운영/문제해결 > 성능개선 및 문제해결
- 문제 제목: Performance Profiling 에 대한 이해
- 출제 의도: Chrome Profiling 도구를 활용하여 성능 개선을 위한 조치 방안을 수립할 수 있다.

주요 개념 정리

용어	설명
FP (First Paint)	첫 픽셀이 그려진 시점
FCP (First Contentful Paint)	텍스트/이미지 등 첫 콘텐츠가 그려진 시점
DCL (DOMContentLoaded)	DOM 파싱 완료 시점
LCP (Largest Contentful Paint)	가장 큰 콘텐츠가 로딩된 시점
TTFB (Time to First Byte)	첫 바이트가 도착하는 시간
Layout Shift	콘텐츠 배치가 갑자기 바뀌는 현상 (CLS 관련)

① **Layout Shift가 발생하는 동안 페이지 로딩이 중단되므로**

- Layout Shift는 페이지 로딩을 중단시키지 않습니다
- Layout Shift는 시각적 안정성 지표(CLS)로, 요소의 위치가 예기치 않게 변경되는 현상입니다

② **Code Splitting을 통해 로딩에 필요한 코드만 실행**

- Evaluate Script 시간 단축을 위해 Code Splitting은 유효한 방법입니다
- 필요한 코드만 초기 로딩하여 스크립트 실행 시간을 줄일 수 있습니다

③ **LCP 4초, async/defer 활용**

- LCP(Largest Contentful Paint) 4초는 개선이 필요한 수준입니다
- JS/CSS를 async/defer로 비동기 로드하면 렌더링 차단을 방지할 수 있습니다

④ **FCP와 DCL이 2초 안에 발생하므로 적절한 수준**

- 웹 성능 최적화에서 **사용자 경험 관점**이 핵심입니다:
- **DOM Content Loaded Event**: 개발자 중심의 기술적 지표
    - HTML 파싱과 DOM 구성이 완료된 시점
    - 사용자가 실제로 보는 것과는 다를 수 있음
- **Largest Contentful Paint (LCP)**: 사용자 경험 중심의 지표
    - 사용자가 실제로 주요 콘텐츠를 보는 시점
    - Core Web Vitals 중 하나로 실제 사용자 경험을 반영

⑤ **TTFB 1초, CDN 및 캐싱 개선 필요**

- TTFB(Time To First Byte) 1초는 개선 여지가 있습니다
- CDN과 캐싱으로 서버 응답 시간을 단축할 수 있습니다

문제의 핵심:

- FCP 2초, DCL 2초가 빠르더라도
- **LCP가 4초**라는 것은 사용자가 실제 주요 콘텐츠를 보기까지 4초가 걸린다는 의미
- 따라서 **사용자 경험상 느린 페이지**입니다

올바른 접근: DOM 완료 시점보다는 사용자가 실제로 의미 있는 콘텐츠를 보는 시점(LCP) 을 우선적으로 개선해야 합니다.

즉, 기술적 지표가 좋다고 해서 사용자 경험이 좋다고 판단하면 안 되고, **실제 사용자가 체감하는 성능 지표**에 집중해야 한다는 것이 4번이 틀린 핵심 이유입니다.

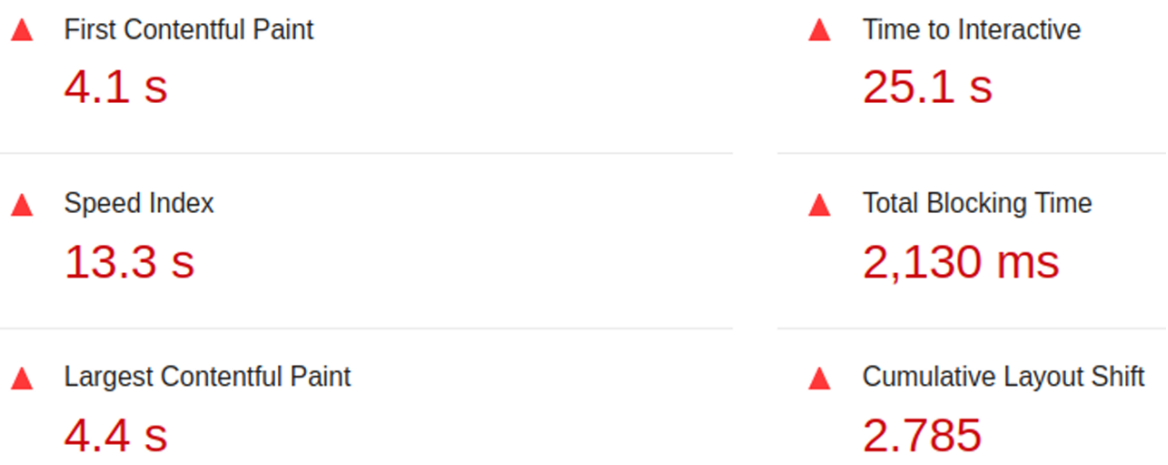
🎯 직접 해보는 실습 가이드

✔ Chrome DevTools로 프로파일링

1. 크롬에서 웹사이트 열기 (예: https://www.naver.com)
2. `F12` → **Performance 탭** → '● Record' 버튼 클릭
3. 페이지 새로고침 (`Ctrl+R`)
4. 로딩 완료되면 '■ Stop'
5. 아래 이벤트 확인
    - FP / FCP / DCL / LCP / Layout Shift / Scripting

12. Software Architecture 운영/문제해결 > Lighthouse에 대한 이해

문항12) 운영중인 B2C 시스템에서 최근 이용자들로부터 성능에 대한 불만이 접수되어 Lighthouse를 통해 분석을 진행해보니 아래와 같이 성능 측정이 되었다. 아래 지표를 개선하기 위한 방안으로 적절하지 않은 것을 고르시오. [4점]



① 이미지 로드에 걸리는 시간을 줄이기 위해 기존 사용하던 png 이미지를 WebP로 변환하여 제공한다.
② 서버 응답 시간 개선을 위해 사용자와 지리적으로 가까운 CDN을 구축하여 네트워크 지연을 감소시킨다.
③ 빌드 도구 및 모듈 번들러를 활용하여 CSS 크기를 줄이고, 초기 렌더링에 필요하지 않은 CSS는 비동기 방식으로 로드하도록 개선한다.
④ 메인 스레드의 과부하를 방지하기 위해, Web Worker가 DOM을 직접 처리하도록 한다.
⑤ Internet Explore는 지원 중단되었으니 IE 지원을 위한 Polyfills을 제거하여 Script Size를 최대한 줄일 수 있는지 검토한다.

(정답) 4
(해설) rel='preload' option 은 초기 로딩속도에 많은 영향을 주기 때문에 반드시 필요한 항목만 식별하여 미리 로드할 수 있도록 한다.
(배점) 4
(난이도) 중

- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 운영/문제해결 > 성능개선 및 문제해결
- 문제 제목: Lighthouse에 대한 이해
- 출제 의도: Lighthouse로 측정된 Metric을 이해하고, 적절한 조치 방안을 수립할 수 있다.

✓ Lighthouse 성능 지표 정리표

항목 (약어)	설명	좋은 기준	사용자 영향	주요 개선 방법
FCP (First Contentful Paint)	사용자 화면에 **처음으로 콘텐츠(텍스트, 이미지)**가 그려진 시점	≤ 1.8초	로딩 시작 인지 (화면이 떴다고 느낌)	HTML 최적화, 렌더링 차단 JS 제거, preload
TTI (Time to Interactive)	페이지가 **완전히 반응 가능**해진 시점 (JS 실행 완료)	≤ 3.8초	클릭 등 인터랙션 반응 시점	JS 최적화, lazy load, code splitting
Speed Index	콘텐츠가 **시각적으로 얼마나 빨리** 로드되었는지를 수치로 표현	≤ 3.4초	콘텐츠가 빠르게 로드됐다고 느끼는 정도	Critical CSS, JS 지연 로딩, SSR
TBT (Total Blocking Time)	페이지가 인터랙션을 **차단한 총 시간** (JS 처리로 인해)	≤ 200ms	클릭/스크롤 먹통 현상	JS 축소, async/defer, Web Worker 사용
LCP (Largest Contentful Paint)	**가장 큰 콘텐츠**(히어로 이미지, 큰 텍스트 등)가 완전히 로딩된 시점	≤ 2.5초	주요 콘텐츠를 본 시점	이미지 최적화, CDN, lazy load 안 쓰는 곳엔 preload
CLS (Cumulative Layout Shift)	로딩 중 콘텐츠의 **위치가 튀는 정도** (시각적 안정성)	≤ 0.1	버튼, 텍스트 등이 갑자기 밀려 사용자 오작동 유발	width/height 명시, 폰트 preload, 광고 위치 고정

✓ Lighthouse 점수 해석 팁

색	의미
(Green)	아주 좋음 (성과 있음)
(Orange)	보통, 개선 여지 있음
◍ (Red)	느리거나 불안정, 최적화 필요

예시 상황 해석

- `FCP` 는 빠른데 `LCP` 가 느리다 → **메인 콘텐츠 늦게 나옴**, 이미지 최적화 필요
- `TBT` , `TTI` 가 길다 → **JS가 너무 무거움**, 사용자 클릭 반응 안 됨
- `CLS` 점수 나쁘다 → **화면 튐 현상**, 버튼 클릭 실수 유발
- `Speed Index` 가 안 좋다 → 화면 구성 전체가 느리게 채워짐

📌 결과

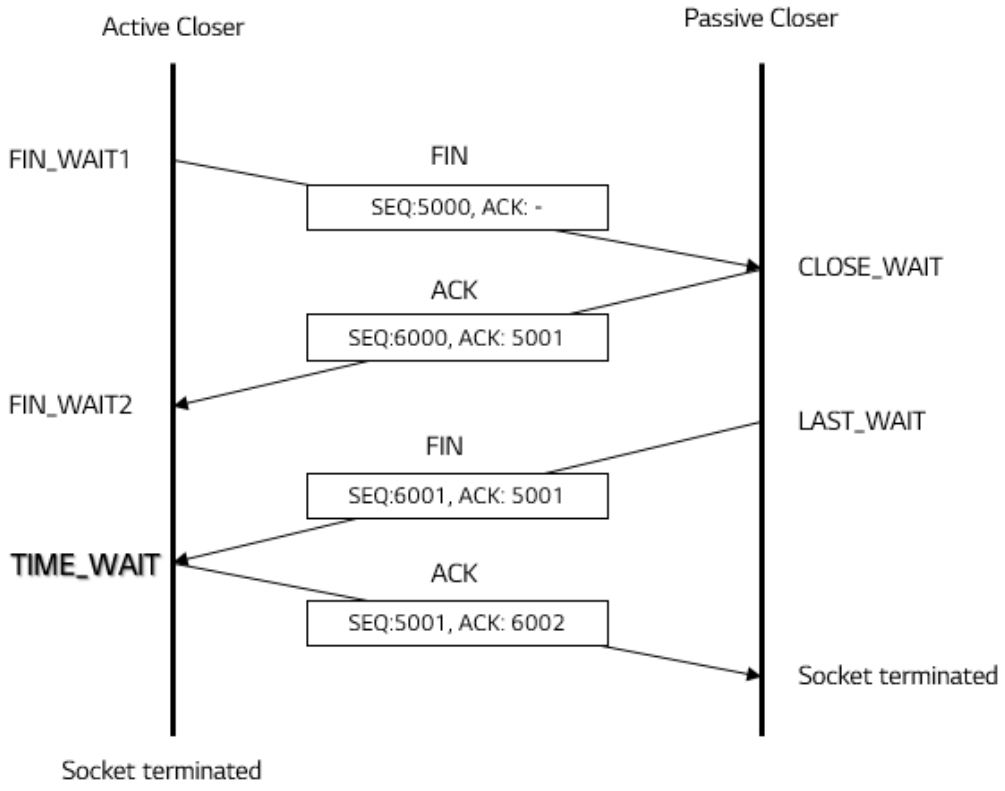
```
FCP: 2.2s
LCP: 4.1s
TTI: 6.3s
TBT: 800ms
CLS: 0.02
Speed Index: 5.0s
```

→ 해석:

- 콘텐츠 시작은 괜찮지만 전체 구성(LCP, Speed Index)이 느림
- JS가 무거워서 반응 늦음 (TTI, TBT)
- 화면 튐(CLS)은 없음

13. Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽

문항 13) 아래 그림은 웹 서버가 설치된 리눅스의 네트워크 상태를 모니터하는 과정에 자주 찾아볼 수 있는 TIME_WAIT 상태를 보여주고 있다. TIME_WAIT은 맺어진 소켓을 끊는 과정에서 나타나는 정상적인 상태이지만 해당 상태의 소켓이 많거나 해당 상태가 오래 지속될 경우에는 가용 포트 부족(또는 FD부족)으로 통신이 어려운 경우가 발생할 수 있다. 다음 중 TIME_WAIT의 설명에 대해 틀린 것을 고르시오. [3점]




```
$ netstat -n -t | grep 'TIME_WAIT'
Proto Recv-Q Send-Q Local Address           Foreign Address         Stat
tcp6       0      0 10.77.57.106:10080      10.77.94.26:37901       TIME
tcp6       0      0 10.77.57.106:10080      10.77.94.26:37900       TIME
tcp6       0      0 10.77.57.106:10080      10.77.94.26:37905       TIME
tcp6       0      0 10.77.57.106:10080      10.77.56.86:41811       TIME
tcp6       0      0 10.77.57.106:10080      10.77.56.82:48168       TIME
```

① 웹 서버에서 HTTP Keepalive 옵션을 사용하지 않은 경우에는 옵션을 적용한 웹서버에 비해 TIME_WAIT 상태를 더 많이 찾아볼 수 있다.
② WAS에서 다른 서버로 HTTP REST API 호출을 할 경우에는 Connection Pool을 적용하여 TIME_WAIT 상태에 의한 로컬 포트 부족에 대비할 수 있다.
③ 로컬 포트 부족으로 통신이 어려운 경우에는 ulimit을 이용하여 FD(File Descriptor)를 충분히 크게 늘려주거나 tcp_tw_reuse 옵션을 사용하여 TIME_WAIT 상태의 소켓을 재사용할 수 있도록 한다.
④ TIME_WAIT 상태는 포트를 잡고 있는 프로세스가 종료되거나 네트워크를 재시작하기 전에는 제거할 방법이 없으므로 반드시 어플리케이션에서 명시적으로 close()를 해주도록 개발되어야 한다.

(정답)4
(해설)4) CLOSE_WAIT에 대한 설명임.
(배점)3
(난이도)하

- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽
- 문제 제목: TIME_WAIT 상태
- 출제 의도: 웹서버나 WAS에서 쉽게 발견할 수 있는 TIME_WAIT에 대한 이해 능력 확인

✓ 정답: ④ TIME_WAIT 상태는 포트를 잡고 있는 프로세스가 종료되거나 네트워크를 재시작하기 전에는 제거할 방법이 없으므로 반드시 어플리케이션에서 명시적으로 close()를 해주도록 개발되어야 한다.

🚫 왜 틀렸는가?

- **TIME_WAIT 상태는 TCP 프로토콜의 정상적인 종료 과정**에서 발생하며, 연결을 먼저 끊는(=Active Closer) 측에서 약 2배의 MSL(Maximum Segment Lifetime) 시간 동안 해당 포트를 유지하게 됩니다.
- 이 상태는 **커널 레벨에서 관리되며**, 명시적으로 `close()` 를 하지 않아도 커널이 소켓 종료 시 처리합니다.
- 어플리케이션에서 명시적으로 close를 하는 건 좋은 습관이지만, **그렇다고 해서 TIME_WAIT이 제거되는 건 아닙니다.**
- 또한 **재부팅이나 프로세스 종료 없이도 커널 설정(tcp_tw_reuse, tcp_tw_recycle 등)을 통해 제어**가 가능합니다.

각 보기 해설

보기	설명	판단
①	HTTP KeepAlive가 꺼진 경우, 매 요청마다 새 연결을 맺고 종료 → TIME_WAIT 증가	✓ 맞음
②	REST API 클라이언트 입장에서 Connection Pool을 통해 재사용하면 포트 고갈 방지 가능	✓ 맞음
③	`ulimit` 은 FD 수 증가에, `tcp_tw_reuse` 는 TIME_WAIT 재사용에 사용됨	✓ 맞음

보기	설명	판단
④	TIME_WAIT은 커널이 관리하며, close()만으로 제어되지 않음. 네트워크 재시작이 필요하지 않음	✕ 틀림

① 정답 - HTTP Keepalive와 TIME_WAIT의 관계

- HTTP Keepalive를 사용하지 않으면 매 요청마다 새로운 TCP 연결을 생성하고 종료
- 연결 종료 시마다 TIME_WAIT 상태가 발생하므로 더 많은 TIME_WAIT 소켓이 생성됨
- Keepalive 사용 시 연결을 재사용하므로 TIME_WAIT 상태가 상대적으로 적게 발생

② 정답 - Connection Pool의 효과

- Connection Pool을 사용하면 연결을 재사용하여 새로운 연결 생성/종료를 최소화
- 따라서 TIME_WAIT 상태로 인한 로컬 포트 부족 문제를 완화할 수 있음

③ 정답 - 해결 방법들

- `ulimit` 으로 FD 한계 증가 가능
- `tcp_tw_reuse` 옵션으로 TIME_WAIT 상태의 소켓 재사용 가능
- 이 외에도 `tcp_tw_recycle` (구버전), `net.ipv4.tcp_fin_timeout` 조정 등의 방법 존재

④ 틀린 설명 - TIME_WAIT 해결 방법

- TIME_WAIT 상태는 시스템 레벨에서 해결 가능한 방법들이 존재
- 커널 파라미터 조정, 소켓 옵션 설정 등으로 해결 가능
- 애플리케이션의 명시적 `close()` 는 기본적인 좋은 관례이지만, TIME_WAIT 문제의 유일한 해결책은 아님

TIME_WAIT 상태 이해

첨부된 다이어그램에서 보듯이:

- Active Closer가 먼저 FIN을 보내고 TIME_WAIT 상태에 진입
- 이는 늦게 도착하는 패킷을 처리하기 위한 정상적인 TCP 상태
- 보통 2MSL(Maximum Segment Lifetime) 시간 동안 유지

TCP 3-Way Handshake와 4-Way Handshake 완벽 가이드: 기초부터 실무 최적화까지 – 기피말고깊이

* TCP 3-way Handshake 란?

TCP는 장치들 사이에 논리적인 접속을 성립(establish)하기 위하여 three-way handshake를 사용한다.

TCP 3 Way Handshake는 TCP/IP프로토콜을 이용해서 통신을 하는 응용프로그램이 데이터를 전송하기 전에 먼저 정확한 전송을 보장하기 위해 상대방 컴퓨터와 사전에 세션을 수립하는 과정을 의미한다..

Client > Server : TCP SYN

Server > Client : TCP SYN ACK

Client > Server : TCP ACK

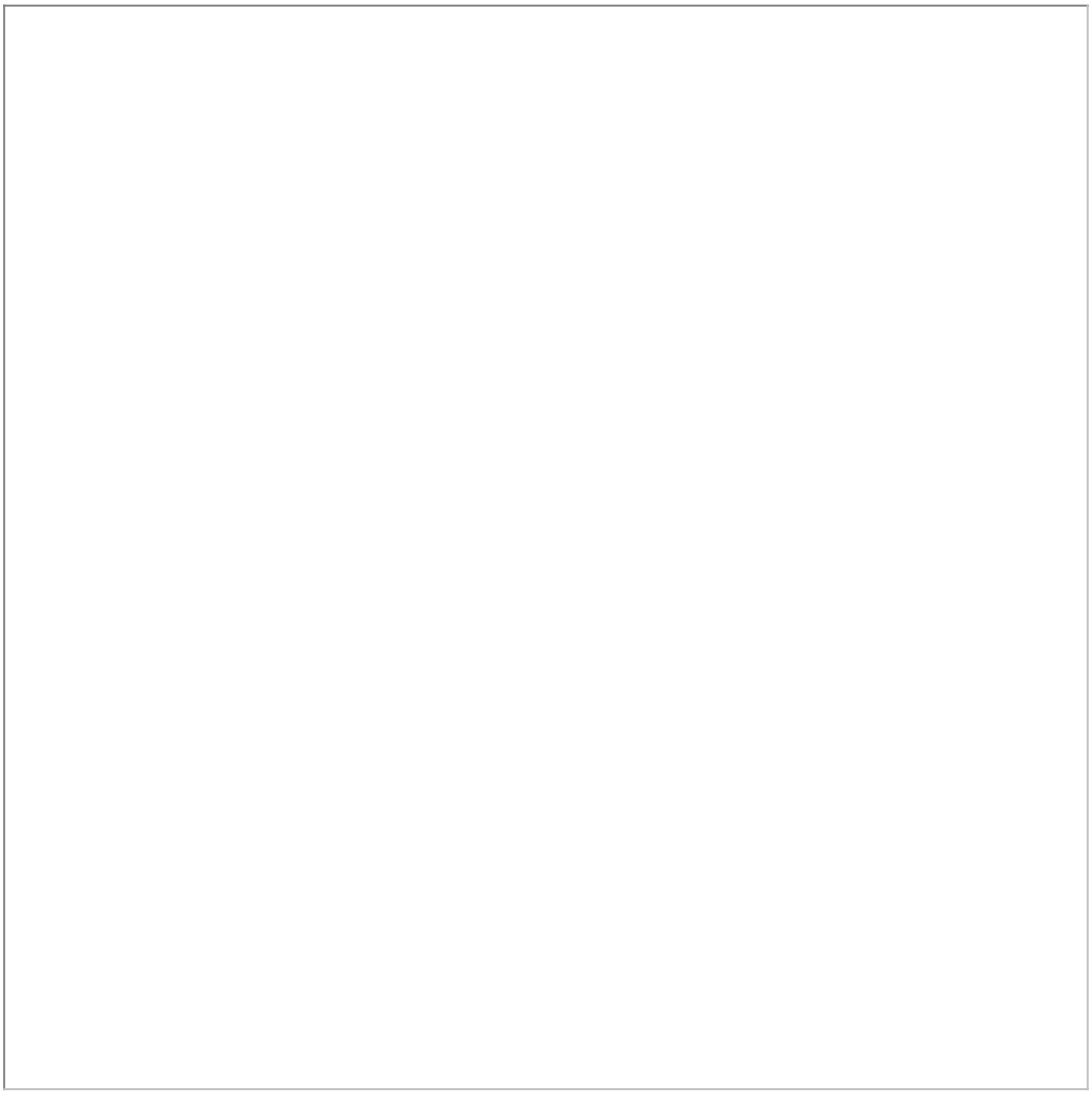
여기서 SYN은 'synchronize sequence numbers', 그리고 ACK는'acknowledgment' 의 약자이다.

이러한 절차는 TCP 접속을 성공적으로 성립하기 위하여 반드시 필요하다.

* TCP의 3-way Handshaking 역할

• 양쪽 모두 데이타를 전송할 준비가 되었다는 것을 보장하고, 실제로 데이타 전달이 시작하기전에 한쪽이 다른 쪽이 준비되었다는 것을 알수 있도록 한다.

• 양쪽 모두 상대편에 대한 초기 순차일련변호를 얻을 수 있도록 한다.



*** TCP의 3-way**

Handshaking 과정

[STEP 1]

A클라이언트는 B서버에 접속을 요청하는 SYN 패킷을 보낸다. 이때 A클라이언트는 SYN 을 보내고 SYN/ACK 응답을 기다리는SYN_SENT 상태가 되는 것이다.

[STEP 2]

B서버는 SYN요청을 받고 A클라이언트에게 요청을 수락한다는 ACK 와 SYN flag 가 설정된 패킷을 발송하고 A가 다시 ACK으로 응답하기를 기다린다. 이때 B서버는 SYN_RECEIVED 상태가 된다.

[STEP 3]

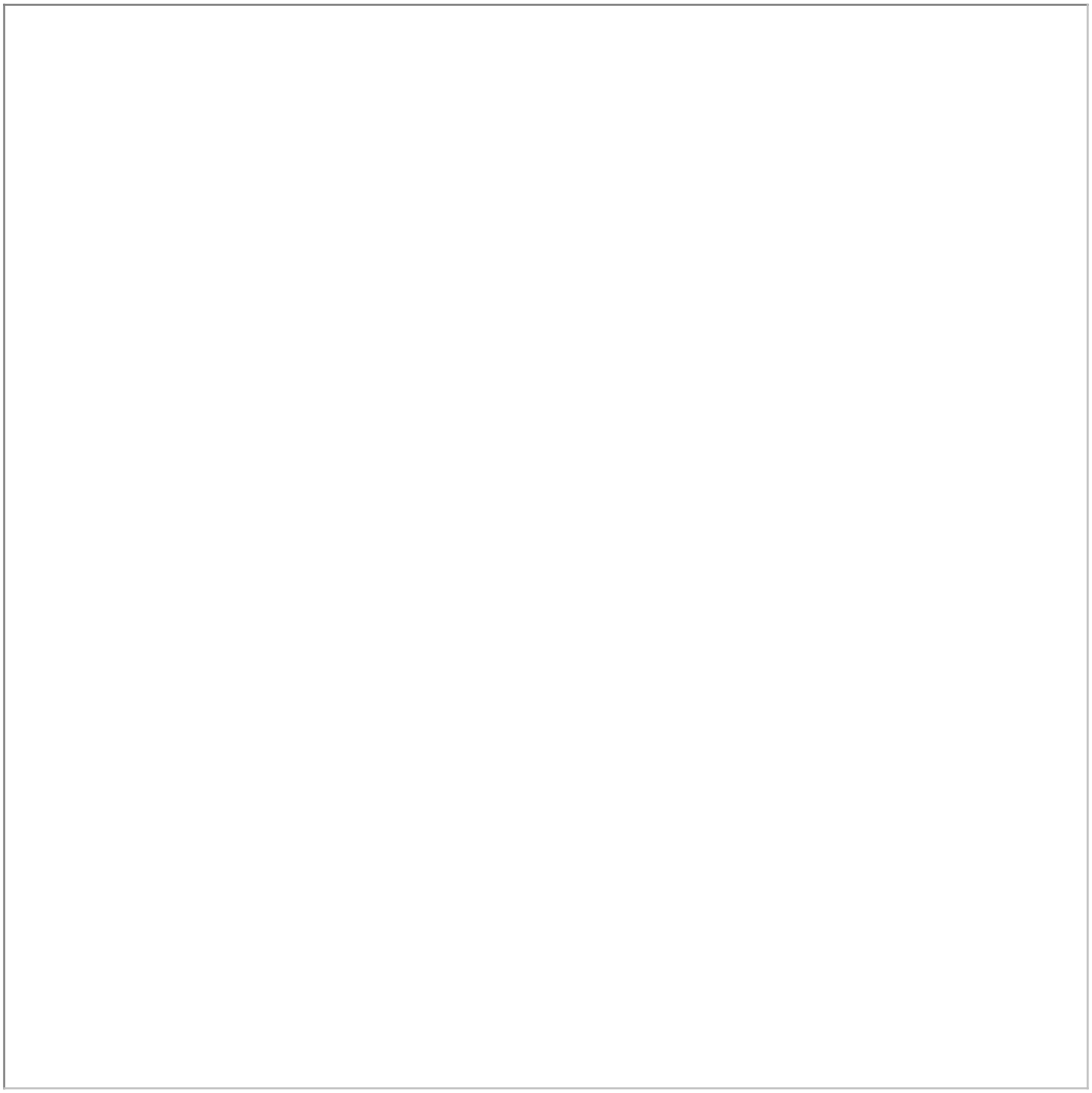
A클라이언트는 B서버에게 ACK을 보내고 이후로부터는 연결이 이루어지고 데이터가 오가게 되는것이다. 이때의 B서버 상태가 ESTABLISHED 이다.

위와 같은 방식으로 통신하는것이 신뢰성 있는 연결을 맺어 준다는 TCP의 3 Way handshake 방식이다.

이번엔 4-way Handshaking 에 대하여 알아보겠습니다.

3-Way handshake는 TCP의 연결을 초기화 할 때 사용한다면, 4-Way handshake는 세션을 종료하기 위해 수행되는 절차입니다.

*** TCP의 4-way Handshaking 과정**



[STEP 1]

클라이언트가 연결을 종료하겠다는 FIN플래그를 전송한다.

[STEP 2]

서버는 일단 확인메시지를 보내고 자신의 통신이 끝날때까지 기다리는데 이 상태가 **TIME_WAIT** 상태다.

[STEP 3]

서버가 통신이 끝났으면 연결이 종료되었다고 클라이언트에게 FIN플래그를 전송한다.

[STEP 4]

클라이언트는 확인했다는 메시지를 보낸다.

그런데 만약 "Server에서 FIN을 전송하기 전에 전송한 패킷이 Routing 지연이나 패킷 유실로 인한 재전송 등으로 인해 FIN패킷보다 늦게 도착하는 상황"이 발생한다면 어떻게 될까요?

Client에서 세션을 종료시킨 후 뒤늦게 도착하는 패킷이 있다면 이 패킷은 Drop되고 데이터는 유실될 것입니다.

이러한 현상에 대비하여 Client는 Server로부터 FIN을 수신하더라도 일정시간(디폴트 240초) 동안 세션을 남겨놓고 잉여 패킷을 기다리는 과정을 거치게 되는데 이 과정을 "TIME_WAIT" 라고 합니다.

출처: https://mindnet.tistory.com/entry/네트워크-쉽게-이해하기-22편-TCP-3-WayHandshake-4-WayHandshake

🔄 TCP 상태 비교

상태	의미 (서버/클라이언트)	언제 발생하는가
LISTEN	서버가 연결 요청을 기다리는 중	`listen()` 호출 후, 클라이언트 연결 요청 직전 community.f5.com+6maxnilz.com+6blog.cloudflare.com+6unix.stackexchange.com
SYN-SENT	클라이언트가 SYN 전송 후 응답 대기 중	`connect()` 호출 후 첫 단계
SYN-RECEIVED	서버가 SYN 받고 SYN-ACK 응답 보냄	서버 측에서 3-웨이 핸드셰이크 중
ESTABLISHED	연결이 성립되어 데이터 송수신 가능 상태	3-웨이 핸드셰이크 완료 후
FIN-WAIT-1	FIN 전송 대기 or ACK 대기 중	연결 종료를 위한 첫 FIN 송신 직후
FIN-WAIT-2	상대 FIN 기다리는 단계	자신의 FIN이 ACK된 후 상대의 FIN 대기 중
CLOSE-WAIT	상대가 FIN 보내고, 애플리케이션이 close() 호출 대기 중	상대 종료 → 응답 후, 프로세스가 `close()` 호출 전까지 지속
CLOSING	FIN 응답 ACK 기다리는 중	양쪽 모두 FIN 교환 중 ACK 대기 상태
LAST-ACK	자신의 FIN 응답에 대한 ACK 기다리는 중	상대 CLOSE_WAIT에서 FIN 후, 마지막 ACK 대기
TIME-WAIT	모든 패킷 소멸까지 2 MSL 대기 중	마지막 ACK 송신 후 짧게(예: 60초) 대기
CLOSED	연결 완전 종료	모든 종료 상태 이후 완전히 해제됨

🛠 상태별 역할과 문제 대응

- **CLOSE_WAIT**:
  - 애플리케이션이 `close()` 호출을 지연하면 무제한으로 유지됨.
  - **대응**: 반드시 소켓 close 호출하고, FD 누수 점검 servicenow.iu.edu+8blog.cloudflare.com+8unix.stackexchange.com+8superuser.com.
- **TIME_WAIT**:
  - 오래된 패킷이 재전송되어 새로운 연결에 영향을 주는 것을 방지하기 위한 장치.
  - **대응**: `tcp_tw_reuse`, `tcp_tw_recycle` 설정 가능; 그러나 기본 유지 시간이 있음 en.wikipedia.org+1web3us.com+1maxnilz.com.
- **FIN_WAIT_2**:
  - 상대의 FIN을 기다리지만 때때로 무기한 지속됨.
  - 커널 `tcp_fin_timeout` 설정 (예: `/proc/sys/net/ipv4/tcp_fin_timeout`)으로 시간 조정 가능 blog.cloudflare.com+2serverfault.com+2unix.stackexchange.com+2.

✓ 주요 커널 튜닝 파라미터

1. `net.ipv4.tcp_tw_reuse`

- **역할**: TIME_WAIT 상태의 소켓을 재사용하도록 활성화 (아웃바운드 연결에서 특히 유용)
- **설정값**: `0` (비활성, 기본), `1` (활성)
- **주의사항**: TCP timestamps 옵션이 활성화되어야 작동 hayz.tistory.com+15sysops.tistory.com+15jirak.net+15
- **설정 예시**:

  bash

  ```
  sysctl -w net.ipv4.tcp_tw_reuse=1
  ```

---

2. `net.ipv4.tcp_fin_timeout`

- **역할**: FIN_WAIT2 상태에서 소켓이 열려있을 최대 시간 (초)
- **기본값**: 60초
- **권장값**: 15–30초 (빠른 자원 회수 유도)

---

3. `net.ipv4.ip_local_port_range`

- **역할**: 클라이언트가 사용할 ephemeral 포트의 범위 지정
- **기본값**: `32768-60999`
- **권장 설정**: `1024-65000` 등으로 넓혀 포트 고갈 방지
  jacking75.github.io+15couplewith.tistory.com+15velog.io+15hayz.tistory.com+9s-core.co.kr+9ibm.com+9

---

4. `ulimit -n` (파일 디스크립터 수)

- **역할**: 프로세스당 열 수 있는 FD(SOCKET 포함) 수 제한
- **기본값**: 약 1024 (시스템마다 다름)
- **권장값**: 65,535 등으로 확장
- **설정 예** (`/etc/security/limits.conf`):

  markdown

  ```
  * soft nofile 65535
  * hard nofile 65535
  ```

🔧 추가 추천 파라미터

5. `net.core.somaxconn`

- **역할**: listen 큐의 최대 backlog (서버 수용 동시 연결 수)
- **기본값**: 128 (버전 >5.4부터 4096)
- **권장값**: 1024–8192 brunch.co.kr+1jirak.net+1brewagebear.github.io+9lifeplan-b.tistory.com+9jirak.net+9

---

6. `net.ipv4.tcp_max_syn_backlog`

- **역할**: SYN_RECEIVED 상태의 큐 크기 (3-way handshake 중)
- **기본값**: 256
- **권장값**: 1024–2048 couplewith.tistory.com+15lifeplan-b.tistory.com+15jirak.net+15

---

7. `net.core.netdev_max_backlog`

- **역할**: NIC 수신 큐의 최대 대기 패킷 수
- **기본값**: 1000
- **권장값**: 2500–5000 meetup.nhncloud.com+6lifeplan-b.tistory.com+6couplewith.tistory.com+6

---

8. `net.ipv4.tcp_keepalive_time`, `tcp_keepalive_intvl`, `tcp_keepalive_probes`

- **역할**: 유휴 연결 감지 및 유지 관리
- **기본값**: 7200 / 75 / 9
- **권장 조정 예**: 300 / 15 / 5 lifeplan-b.tistory.com+10ibm.com+10jirak.net+10

---

9. TCP 버퍼 사이즈 (`tcp_rmem`, `tcp_wmem`, `core.rmem_max`, `core.wmem_max`)

- **역할**: TCP 송수신 버퍼 설정 (네트워크 처리량에 영향)
- **권장 설정 예**: rmem/wmem 기본-최대 4k 12.5M 16M jacking75.github.io+2couplewith.tistory.com+2s-core.co.kr+2

---

10. 기타 유용 설정

- `net.ipv4.tcp_slow_start_after_idle=0` – 유휴 후 slow-start 방지
- `net.ipv4.tcp_max_tw_buckets` – 최대 TIME_WAIT bucket 수 조정 meetup.nhncloud.com+15sysops.tistory.com+15couplewith.tistory.com+15
- `net.ipv4.tcp_timestamps=1` – RFC1323 타임스탬프 활용
- `net.ipv4.tcp_retries1`, `tcp_retries2` – 연결/전송 재시도 횟수 조정
- 필요 시 `net.netfilter.nf_conntrack_max` 등 연결 추적수 확장

설정 적용 방법

1. **즉시 적용**:

bash

```
sysctl -w <파라미터>=<값>
```

2. **영구 적용** (`/etc/sysctl.conf` 또는 `/etc/sysctl.d/99-custom.conf`):

yaml

```
net.ipv4.tcp_tw_reuse = 1
net.core.somaxconn = 4096
……
```

이후 `sysctl -p` 로 반영

📌 정리 요약

파라미터	기본값	권장값	주요 효과
tcp_tw_reuse	0	1	TIME_WAIT 재사용
tcp_fin_timeout	60초	15–30초	FIN_WAIT 타임아웃 단축
ip_local_port_range	32768–60999	1024–65000	포트 고갈 방지
somaxconn	128	1024–8192	동시 연결 수 개선
tcp_max_syn_backlog	256	1024–2048	SYN 공격 방어, 연결 안정성
netdev_max_backlog	1000	2500–5000	패킷 드롭 감소
tcp_keepalive_*	7200/75/9	300/15/5	방화벽 연결 유지
tcp_rmem/wmem	기본값	16M	대용량 데이터 전송 최적화

[김수진][공유] 13. Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽

TIME_WAIT 상태란?

정의

TIME_WAIT은 TCP 연결을 **능동적으로 종료한 쪽**에서 연결이 완전히 종료되기 전까지 일정 시간 동안 유지하는 **정상적인** 상태입니다.

TCP 연결 종료 과정

```
클라이언트(능동종료)          서버(수동종료)
     |                          |
     | ---- FIN ---------->     |
     |                    CLOSE_WAIT
     | <--- ACK ----------      |
FIN_WAIT1                       |
     | <--- FIN ----------      |
FIN_WAIT2                LAST_ACK
     | ---- ACK ---------->     |
  TIME_WAIT                  CLOSED
     |                          |
   (2MSL 대기)
     |-
   CLOSED
```

TIME_WAIT의 목적

1. 지연된 패킷 처리

- 네트워크에서 늦게 도착할 수 있는 패킷들을 처리
- 같은 포트 번호로 새 연결이 생성될 때 이전 연결의 패킷과 혼동 방지

2. 안정적인 연결 종료 보장

- 마지막 ACK가 손실될 경우를 대비

- 상대방이 FIN을 재전송할 수 있도록 대기

TIME_WAIT 지속 시간

- **2MSL**(Maximum Segment Lifetime) 동안 유지
- Linux 기본값: **60초**
- 이 시간 동안 해당 포트는 재사용 불가

실무에서 문제가 되는 경우

1. HTTP Keep-Alive 미사용

bash

```bash
# Keep-Alive 미사용 시
$ netstat -an | grep TIME_WAIT | wc -l
15000  # 많은 TIME_WAIT 발생
```

2. 대량의 아웃바운드 연결

java

```java
// 문제가 되는 코드
for(int i = 0; i < 10000; i++) {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    // 매번 새로운 연결 생성 → TIME_WAIT 누적
}
```

해결 방법

1. Connection Pool 사용

java

```java
// 올바른 방법
CloseableHttpClient httpClient = HttpClients.custom()
    .setMaxConnTotal(200)
    .setMaxConnPerRoute(50)
    .build();
```

2. HTTP Keep-Alive 활성화

apache

```
# Apache 설정
KeepAlive On
```

```
MaxKeepAliveRequests 100
KeepAliveTimeout 5
```

3. 시스템 설정 조정

bash

```bash
# TIME_WAIT 소켓 재사용 허용
echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse

# 로컬 포트 범위 확장
echo "1024 65535" > /proc/sys/net/ipv4/ip_local_port_range
```

TIME_WAIT vs CLOSE_WAIT 비교

구분	TIME_WAIT	CLOSE_WAIT
발생 위치	능동적 연결 종료 쪽	수동적 연결 종료 쪽
원인	정상적인 TCP 종료 과정	애플리케이션이 close() 미호출
해결	시간이 지나면 자동 해결	코드 수정 필요
상태	정상	비정상

모니터링 명령어

bash

```bash
# TIME_WAIT 상태 확인
netstat -an | grep TIME_WAIT

# 상태별 개수 확인
ss -ant | awk '{print $1}' | sort | uniq -c

# 특정 포트의 TIME_WAIT 확인
netstat -an | grep :8080 | grep TIME_WAIT
```

핵심: TIME_WAIT은 TCP의 정상적인 동작이지만, 대량 발생 시 포트 고갈을 일으킬 수 있으므로 Connection Pool과 Keep-Alive를 통해 관리해야 합니다.

TCP 연결 종료 과정 상세 설명

1단계: 클라이언트가 close() 호출

```
클라이언트                    서버
   |                           |
 close() 호출                  |
   |                           |
   | ---- FIN ---------->      | (1) FIN 받음
   |                     CLOSE_WAIT (2) CLOSE_WAIT 상태로 변경
FIN_WAIT1                    | (3) ACK 즉시 전송
   |                           |
```

여기서 중요한 점:

- 클라이언트가 `close()`를 호출하면 **FIN 패킷**을 서버로 전송
- 서버는 FIN을 받자마자 **CLOSE_WAIT 상태**가 됨
- 서버는 **즉시 ACK**를 클라이언트로 전송

2단계: 서버의 ACK 응답

```
클라이언트                    서버
   |                           |
   | <--- ACK ----------       | ACK 전송
FIN_WAIT2                  CLOSE_WAIT
   |                           |
   |                     애플리케이션이
   |                     아직 close() 안함
```

CLOSE_WAIT 상태의 의미:

- "클라이언트가 연결을 끝내려고 하는구나, 알겠어!"
- 하지만 **서버 애플리케이션이 아직 close()를 호출하지 않음**
- 서버는 여전히 데이터를 보낼 수 있는 상태

3단계: 서버 애플리케이션이 close() 호출

```
클라이언트                    서버
   |                           |
   |                     서버도 close() 호출
```

```
   |                          |
   | <--- FIN ----------      | FIN 전송
FIN_WAIT2                LAST_ACK
   |                          |
```

4단계: 최종 ACK와 TIME_WAIT

```
클라이언트        서버
   |                          |
   | ---- ACK --------->      | 최종 ACK
TIME_WAIT                  CLOSED
   |                          |
 (60초 대기)                  |
   |                          |
 CLOSED                       |
```

실제 코드로 보는 예시

정상적인 경우

java

```
// 서버 코드
try (Socket clientSocket = serverSocket.accept();
     BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputSt
ream()))) {

    String inputLine = in.readLine();
    // 처리 완료

} // try-with-resources가 자동으로 close() 호출
```

문제가 되는 경우 (CLOSE_WAIT 누적)

java

```
// 잘못된 서버 코드
Socket clientSocket = serverSocket.accept();
BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

String inputLine = in.readLine();
// 처리 완료

// close()를 호출하지 않음! → CLOSE_WAIT 상태로 남아있음
```

상태 확인하기

bash

```
# CLOSE_WAIT이 많이 쌓인 경우 (문제 상황)
$ netstat -an | grep CLOSE_WAIT
tcp  0  0  192.168.1.100:8080  192.168.1.200:45234  CLOSE_WAIT
tcp  0  0  192.168.1.100:8080  192.168.1.200:45235  CLOSE_WAIT
tcp  0  0  192.168.1.100:8080  192.168.1.200:45236  CLOSE_WAIT
# ... 수백개 더

# TIME_WAIT은 정상적인 경우
$ netstat -an | grep TIME_WAIT
tcp  0  0  192.168.1.200:45234  192.168.1.100:8080  TIME_WAIT
```

핵심 포인트

1. **CLOSE_WAIT**: 상대방이 연결을 끝내고 싶어하지만, **내가 아직 close()를 안한 상태**
2. **TIME_WAIT**: **내가 연결을 끝낸 후** 혹시 모를 지연 패킷을 기다리는 상태

문제 해결

- **CLOSE_WAIT 많음** → 코드에서 close() 누락 확인
- **TIME_WAIT 많음** → Connection Pool이나 Keep-Alive 사용 검토

Connection Pool과 Http Keep-Alive

- Connection Pool은 **미리 생성된 연결들을 재사용**하는 기술입니다.

전통적인 방식 vs Connection Pool

java

```
// ❌ 전통적인 방식 (매번 새 연결)
for(int i = 0; i < 1000; i++) {
    Socket socket = new Socket("server.com", 8080);  // 새 연결 생성
    // 데이터 전송
    socket.close();  // 연결 종료 → TIME_WAIT 발생
}
// 결과: 1000개의 TIME_WAIT 상태 발생!
```

java

```
// ✅ Connection Pool 방식
// 1. Pool 생성 (최초 1회)
CloseableHttpClient httpClient = HttpClients.custom()
    .setMaxConnTotal(20)        // 전체 최대 연결 수
```

```
    .setMaxConnPerRoute(10)      // 호스트당 최대 연결 수
    .build();

// 2. 연결 재사용
for(int i = 0; i < 1000; i++) {
    HttpGet request = new HttpGet("http://server.com/api");
    httpClient.execute(request);  // 기존 연결 재사용
}
// 결과: 최대 20개의 연결만 사용, TIME_WAIT 최소화!
```

- HTTP Keep-Alive는 **하나의 TCP 연결로 여러 HTTP 요청을 처리**하는 기술입니다.

Keep-Alive 없음 vs 있음

```
# ✖ Keep-Alive 없음 (HTTP/1.0 기본)
클라이언트        서버
   |                          |
   | TCP 연결 생성             |
   |=========================>|
   | HTTP 요청 1               |
   |------------------------->|
   | HTTP 응답 1               |
   |<-------------------------|
   | TCP 연결 종료             | ← TIME_WAIT 발생
   |=========================X|
   |                          |
   | TCP 연결 생성 (다시!)      |
   |=========================>|
   | HTTP 요청 2               |
   |------------------------->|

# ✓ Keep-Alive 있음 (HTTP/1.1 기본)
클라이언트        서버
   |                          |
   | TCP 연결 생성             |
   |=========================>|
   | HTTP 요청 1               |
   |------------------------->|
   | HTTP 응답 1               |
   |<-------------------------|
   | HTTP 요청 2 (같은 연결!)   |
   |------------------------->|
```

```
    | HTTP 응답 2                |
    |<-------------------------|
    | 연결 유지...               |
```

HTTP는 TCP 위에서 동작하는 프로토콜입니다

```
애플리케이션 계층     |  HTTP Request/Response
전송 계층           |  TCP Connection
```

중요: HTTP 연결이라는 별도 연결은 없습니다. HTTP는 TCP 연결을 사용하는 **프로토콜**일 뿐입니다.

Connection Pool vs Keep-Alive 정확한 차이

1. Connection Pool (TCP 연결 관리)

java

```
// Connection Pool의 실제 동작
클라이언트                    Connection Pool              서버
    |                            |                         |
    | HTTP 요청 1                |                         |
    |--------------------------->|                         |
    |                            | TCP 연결 #1 사용        |
    |                            |======================>|
    |                            | HTTP/1.1 GET /api       |
    |                            |----------------------->|
    |                            | HTTP 응답               |
    |                            |<-----------------------|
    | 응답 반환                  | TCP 연결 #1을 Pool에 보관 |
    |<---------------------------|                         |
    |                            |                         |
    | HTTP 요청 2                |                         |
    |--------------------------->|                         |
    |                            | 동일한 TCP 연결 #1 재사용 |
    |                            |======================>|
```

Connection Pool의 역할: TCP 연결 자체를 재사용

2. Keep-Alive (HTTP 프로토콜 레벨)

```
# Keep-Alive 없을 때 (HTTP/1.0)
TCP 연결 생성
  → HTTP 요청
  → HTTP 응답
  → TCP 연결 즉시 종료

# Keep-Alive 있을 때 (HTTP/1.1)
TCP 연결 생성
  → HTTP 요청 1
  → HTTP 응답 1
  → HTTP 요청 2 (같은 TCP 연결 사용)
  → HTTP 응답 2
  → ... (연결 유지)
  → 일정 시간 후 TCP 연결 종료
```

실제 패킷 레벨에서 보기

Keep-Alive 없는 경우

http

```
# 첫 번째 요청
GET /api/users HTTP/1.0
Host: server.com
Connection: close          ← 요청 후 연결 종료

HTTP/1.0 200 OK
Connection: close          ← 응답 후 연결 종료
Content-Length: 100

[데이터]
[TCP FIN 패킷] ← 연결 종료
```

Keep-Alive 있는 경우

http

```
# 첫 번째 요청
GET /api/users HTTP/1.1
Host: server.com
Connection: keep-alive      ← 연결 유지 요청

HTTP/1.1 200 OK
Connection: keep-alive      ← 연결 유지 응답
Keep-Alive: timeout=5, max=100
Content-Length: 100

[데이터]

# 두 번째 요청 (같은 TCP 연결 사용!)
GET /api/orders HTTP/1.1
Host: server.com
Connection: keep-alive

HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: timeout=5, max=99
Content-Length: 200

[데이터]
```

조합된 효과

Connection Pool + Keep-Alive 함께 사용

java

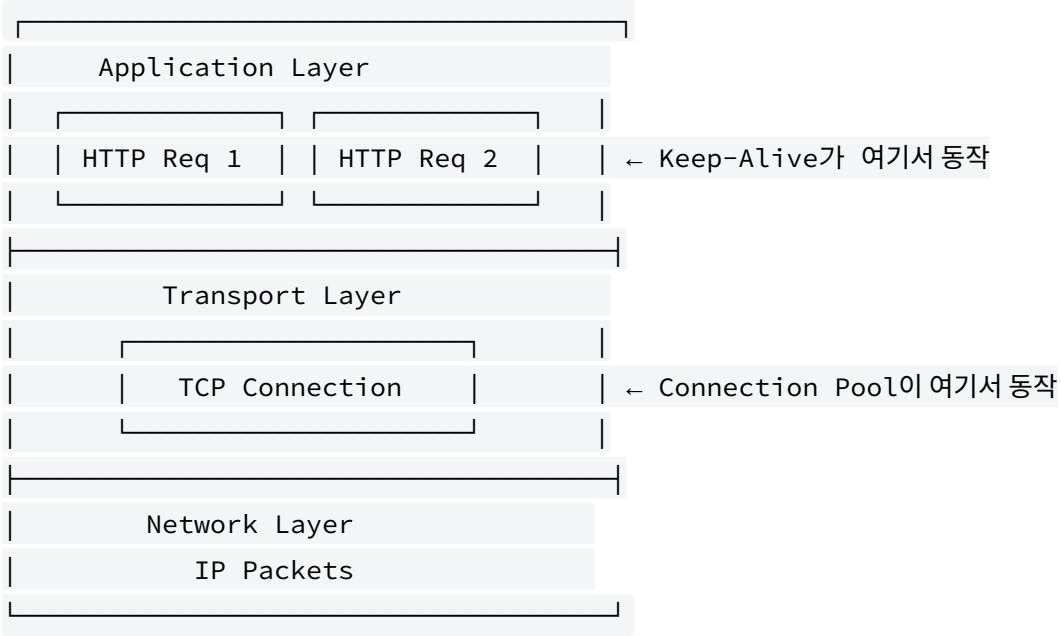
```
// 실제 코드 예시
PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();
cm.setMaxTotal(10);  // 최대 10개 TCP 연결

CloseableHttpClient httpClient = HttpClients.custom()
    .setConnectionManager(cm)
    .setKeepAliveStrategy((response, context) -> 30000) // Keep-Alive 30초
    .build();

// 100번 요청 시나리오
```

```
for(int i = 0; i < 100; i++) {
    HttpGet request = new HttpGet("http://server.com/api");
    httpClient.execute(request);
}
```

결과:

- **TCP 연결**: 최대 10개만 생성 (Connection Pool)
- **HTTP 요청**: 각 연결당 여러 요청 처리 (Keep-Alive)
- **TIME_WAIT**: 최대 10개만 발생 (vs 100개)

네트워크 계층별 정리



실무 설정 예시

java

```
@Configuration
public class HttpConfig {

    @Bean
```

```
    public CloseableHttpClient httpClient() {
        // TCP 연결 관리 (Connection Pool)
        PoolingHttpClientConnectionManager connectionManager =
            new PoolingHttpClientConnectionManager();
        connectionManager.setMaxTotal(200);            // 전체 TCP 연결 수
        connectionManager.setDefaultMaxPerRoute(50);   // 호스트당 TCP 연결 수

        return HttpClients.custom()
            .setConnectionManager(connectionManager)
            // HTTP Keep-Alive 전략
            .setKeepAliveStrategy((response, context) -> {
                // Keep-Alive 헤더에서 timeout 읽기
                HeaderElementIterator it = new BasicHeaderElementIterator(
                    response.headerIterator(HTTP.CONN_KEEP_ALIVE));
                while (it.hasNext()) {
                    HeaderElement he = it.nextElement();
                    String param = he.getName();
                    String value = he.getValue();
                    if (value != null && param.equalsIgnoreCase("timeout")) {
                        return Long.parseLong(value) * 1000;
                    }
                }
                return 30000; // 기본 30초
            })
            .build();
    }
}
```

정리

- **Connection Pool**: TCP 연결을 미리 만들어두고 재사용하는 **애플리케이션 레벨 기술**
- **Keep-Alive**: 하나의 TCP 연결로 여러 HTTP 요청을 처리하는 **HTTP 프로토콜 기능**

둘 다 TCP 연결을 재사용한다는 공통점이 있지만:

- Connection Pool: 애플리케이션이 연결을 관리
- Keep-Alive: HTTP 프로토콜이 연결 유지를 결정

[서정원] 13. Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽

🌐 네트워킹 스터디 정리: TIME_WAIT부터 Keep-Alive까지

🏷 목차

---

1. TIME_WAIT 개념과 원리

TIME_WAIT이 왜 필요할까?

TCP 연결 종료 시 발생하는 2가지 문제를 해결:

1) 마지막 ACK 전달 확인

```
서버: "연결 끊을게!"
클라이언트: "알겠어!" ← 이 메시지가 안 갔을 수도...
```

```
TIME_WAIT = 2분간 대기하여 재전송 요청에 응답 가능
```

2) 늦게 도착하는 패킷 방지

```
기존 연결: 포트 8080 사용 → 종료
새 연결: 바로 포트 8080 재사용
기존 연결의 늦은 패킷 → 새 연결에 섞임
```

```
TIME_WAIT = 충분한 시간을 줘서 모든 패킷 정리
```

📊 TIME_WAIT으로 인한 포트 고갈

```
JMeter TPS 100 테스트:
- 1초에 100개 연결 생성
- 각 연결마다 2분(120초) TIME_WAIT
- 120초 × 100 = 12,000개 포트 점유
- 가용 포트(~28,000개) 중 절반 사용!
```

---

2. 포트와 FD가 필요한 이유

🏢 아파트 주소 시스템으로 이해

포트가 필요한 이유

```
잘못된 주소: "192.168.1.100 → 10.0.0.50"
올바른 주소: "192.168.1.100:12345 → 10.0.0.50:8080"
```

```
왜? 동시에 여러 연결을 구분해야 하니까!
- 크롬: 내PC:12345 → 구글:80
- 카톡: 내PC:23456 → 카카오:443
- 게임: 내PC:34567 → 게임서버:7777
```

FD(File Descriptor)가 필요한 이유

```
Linux/Unix 철학: "모든 것은 파일이다"
```

```
FD 0: 키보드 (stdin)
FD 1: 화면 (stdout)
FD 3: /var/log/app.log
FD 4: 데이터베이스 연결
FD 5: 타 서버 네트워크 연결 ← 우리가 말하는 것!
```

🔍 클라이언트 포트 자동 배정

java

```java
// 내가 B서버 호출할 때
Socket socket = new Socket("192.168.1.50", 8080);

// 운영체제가 자동으로:
// 1. 사용 가능한 포트 스캔 (32768~65535)
```

```
// 2. 빈 포트 발견 (예: 45678)
// 3. 자동 배정: 내PC:45678 → B서버:8080
```

핵심: 서버 포트는 고정, 클라이언트 포트는 자동 배정!

---

3. Keep-Alive의 동작 원리

🏪 편의점 비유

Keep-Alive 없음 (기존 방식)

```
손님1: 문 열고 → 물 사기 → 문 닫기 [TIME_WAIT]
손님2: 문 열고 → 과자 사기 → 문 닫기 [TIME_WAIT]
손님3: 문 열고 → 음료 사기 → 문 닫기 [TIME_WAIT]

결과: TIME_WAIT 3개 발생
```

Keep-Alive 있음 (개선 방식)

```
문 열어놓음 → 손님1,2,3 연속 쇼핑 → 문 닫기

결과: TIME_WAIT 1개만 발생!
```

🌐 웹 요청에서의 실제 차이

HTTP/1.0 (Keep-Alive 없음)

```
브라우저 요청: index.html, style.css, script.js, image1.jpg...

각 파일마다:
연결 → 요청 → 응답 → 연결 종료 → TIME_WAIT

10개 파일 = 10개 TIME_WAIT
```

HTTP/1.1 (Keep-Alive 기본)

```
연결 → index.html → style.css → script.js → ... → 연결 종료

10개 파일 = 1개 TIME_WAIT
```

---

4. 실무 적용: API Gateway 관점

Spring Cloud Gateway의 포트 사용

```
[클라이언트들] → [SCG:8080] → [백엔드 서비스들]
   1000 TPS        1개 포트    여러 서비스 호출
```

SCG도 클라이언트 역할!

```
클라이언트:45678 → SCG:8080 (수신)
```

```
SCG가 백엔드 호출:
SCG:12345 → UserService:8081    ← 포트 필요!
SCG:12346 → OrderService:8082   ← 포트 필요!
SCG:12347 → PaymentService:8083 ← 포트 필요!
```

Fan-out 패턴에서 포트 급증

```
1개 요청 → 4개 백엔드 호출 = 4개 포트 사용
250 TPS × 4 = 1000개 포트/초 소모!
```

⚙ SCG Keep-Alive 설정

yaml

```yaml
spring:
  cloud:
    gateway:
      httpclient:
        pool:
          type: elastic
          max-connections: 500
          max-idle-connections: 100
          max-life-time: 60s
          keep-alive: true
```

---

5. Keep-Alive의 장단점

✓ 장점

1. **포트 절약**: 연결 재사용으로 TIME_WAIT 대폭 감소
2. **성능 향상**: 연결 설정 오버헤드 제거

3. **처리량 증대**: 더 많은 동시 요청 처리 가능

✖ 단점

1) 메모리 사용량 증가

`연결 1개 = ~100KB 메모리`
`1000개 Keep-Alive 연결 = 100MB 메모리 점유`

2) 좀비 연결 문제

`브라우저 강제 종료 → 서버는 연결 살아있다고 착각`
`→ 리소스 낭비 + 새 연결 방해`

3) 로드밸런서 충돌

`Keep-Alive 연결: 특정 서버에 고정`
`로드밸런싱: 요청을 여러 서버에 분산`
`→ 세션 불일치 문제 발생 가능`

4) 네트워크 장비 호환성

`방화벽 타임아웃: 5`
`Keep-Alive 설정: 10`
`→ 중간에 연결 끊어져도 클라이언트는 모름`

---

6. 실무 권장사항

📈 적용 결정 기준

Keep-Alive 적합한 경우

✓ 높은 TPS (수백~수천 요청/)
✓ 같은 서버에 반복 호출
✓ 안정적인 네트워크 환경
✓ 충분한 서버 리소스

Keep-Alive 부적합한 경우

❌ 낮은 TPS (가끔 호출)
❌ 모바일 네트워크 (불안정)
❌ 서버 리소스 부족
❌ 세션 상태 관리 중요

🛠 단계별 도입 전략

1. **문제 확인**: 포트 고갈 현상 발생하는가?
2. **보수적 시작**: 짧은 시간(30초)부터
3. **점진적 조정**: 모니터링하며 시간 늘려가기
4. **환경별 차등**: 개발(10s) → 운영(60s)
5. **롤백 준비**: 문제 시 즉시 비활성화

🔧 핵심 설정 예시

Connection Pool + Keep-Alive

```
@Bean("userServiceClient")
public WebClient userServiceClient() {
    ConnectionProvider provider = ConnectionProvider.builder("user-service")
        .maxConnections(50)           // 최대 50개 연결
        .maxIdleTime(Duration.ofSeconds(30))  // 30초 유지
        .maxLifeTime(Duration.ofMinutes(5))   // 5분 후 갱신
        .build();

    return WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(
            HttpClient.create(provider)))
        .build();
}
```

모니터링

```
# 포트 사용량 확인
ss -tuln | wc -l
```

```
# TIME_WAIT 개수 확인
ss -tan | grep TIME_WAIT | wc -l

# 프로세스별 연결 수
lsof -p <PID> | grep TCP | wc -l
```

핵심 요약

1. **TIME_WAIT은 필요악**: TCP 안정성을 위해 2분간 포트 점유
2. **포트는 연결 구분자**: 동시 연결을 구분하기 위한 필수 요소
3. **Keep-Alive는 양날의 검**: 포트 절약 vs 리소스 점유
4. **API Gateway도 클라이언트**: 백엔드 호출 시 포트/FD 필요
5. **점진적 도입**: 모니터링하며 신중하게 적용

실무 핵심 메시지

> *"높은 TPS 환경에서는 Keep-Alive + Connection Pool 필수!*
> *하지만 적절한 타임아웃과 모니터링 없이는 독이 될 수 있다."*

14. Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽

문항 14) Nginx와 Tomcat을 Reverse Proxy 형태로 연동 구성하였다. 아래의 예시 중 각 상황별로 Web 서버 또는 WAS가 응답하는 HTTP 응답 코드에 대한 설명 중 잘못된 것을 고르시오. [4점]
(단, Web 서버 및 WAS에서는 Exception이나 에러와 관련된 별도의 추가 설정이나 처리를 하지 않는다고 가정한다.)

① 서버에서 URL 리다이렉션 처리를 하는 경우 301 또는 302 응답코드와 함께 클라이언트가 리다이렉션 할 대상 URL을 referrer 응답 헤더를 통해 전달받게 된다.
② Proxy pass 관련 설정을 잘못 지정하여 JSP 요청이 WAS로 전달되지 않고 Web 서버에서 처리하는 경우 404 응답 코드를 전달받게 된다.
③ DB 처리 과정에서 SQLException이 발생할 때 서블릿 등에서 별도의 예외 처리 로직을 처리하지 않았을 경우에는 500 응답 코드를 전달받게 된다.
④ 과부하 상황에서 WAS의 스레드 개수가 Web 서버의 스레드 개수 보다 작을 경우 JSP를 호출하면 502 또는 503 응답 코드를 전달받게 된다.
⑤ WAS로 유입된 JSP의 처리 시간이 Web 서버와 WAS 사이에 지정된 Timeout 값을 초과하는 경우에는 504 응답 코드를 전달받게 된다.

(정답)1
(해설)
1. 리다이렉션 URL을 담고 있는 헤더는 referer가 아니고 location이다.
(배점)4
(난이도)중
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽
- 문제 제목: HTTP 응답코드
- 출제 의도: 상황 별 발생할 수 있는 HTTP 응답코드에 대한 이해

✓ ① 잘못된 설명 (정답)

> *"서버에서 URL 리다이렉션 처리를 하는 경우 301 또는 302 응답코드와 함께 클라이언트가 리다이렉션 할 대상 URL을* `referrer` *응답 헤더를 통해 전달받게 된다."*

▼ **틀린 이유**:

- 리다이렉션 시 **클라이언트가 리다이렉션할 대상 URL은** `Location` **헤더**를 통해 전달됨.
- `Referrer` (또는 `Referer`)는 **요청 헤더**로, 현재 요청이 어떤 페이지로부터 왔는지를 나타냄. 응답 헤더가 아님.
- ✓ 정답은 `Location` 헤더에 있음. 예:

```
HTTP/1.1 302 Found
Location: https://new.example.com/
```

◆ 따라서, **'referrer 응답 헤더'는 잘못된 설명**이며, 이 보기가 틀렸습니다.

② Proxy pass 오류 시 404 → **맞는 설명**

- JSP 파일이 WAS로 전달되지 않고 Nginx에서 처리 시도 → Nginx는 JSP 파일을 처리할 수 없으므로 404 Not Found 응답
- 예: `location /app/ { proxy_pass http://tomcat/; }` 와 같은 설정이 잘못되면
  Web 서버는 내부로 프록시하지 않고 `/app/foo.jsp` 를 찾으려다 **존재하지 않아 404 발생**

③ SQLException 미처리 시 500 → **맞는 설명**

- 서버 내부 예외 (unchecked) 발생 시, WAS는 특별한 예외 처리가 없으면 **HTTP 500 (Internal Server Error)** 반환
- DB에서 SQLException 발생 후 별도 처리 없을 시, WAS에서 500 Internal Server Error 응답

④ WAS 스레드가 부족한 경우 502 또는 503 → **맞는 설명**

- WAS에서 처리할 스레드가 부족하면 Nginx는 **502 Bad Gateway** 또는 **503 Service Unavailable** 반환
  - `502` : 중간 게이트웨이(예: Nginx)가 WAS로부터 유효한 응답을 받지 못함
  - `503` : WAS가 과부하 상태로 요청 처리 불가

⑤ 처리 시간이 Timeout 초과 시 504 → **맞는 설명**

- 예: `proxy_read_timeout` 값을 초과하면 Nginx는 **504 Gateway Timeout** 응답
  - 클라이언트 → Nginx → (WAS 처리 지연) → Nginx 응답 지연 → 504 발생

✓ 요약

보기	상태	이유
①	✗ 틀림	리다이렉션 대상은 `Location`, `referrer` 는 응답 헤더 아님
②	✓ 맞음	proxy 설정 오류 시 Nginx 자체 404 반환
③	✓ 맞음	예외 미처리 시 WAS가 500 응답

보기	상태	이유
④	✓ 맞음	WAS 과부하 시 502/503 발생 가능
⑤	✓ 맞음	처리 지연 시 Nginx 504 반환

✓ HTTP 상태 코드 유형 정리표

범위	이름	설명	예시 코드	설명 (대표 예시)
1xx	**Informational**	요청을 받았으며 처리 중	`100 Continue`	요청 헤더 수신 후 본문 계속 전송하라는 지시
2xx	**Success**	요청 정상 처리 완료	`200 OK`	일반적인 성공 응답
			`201 Created`	리소스 생성됨 (POST 등)
			`204 No Content`	응답 본문 없음
3xx	**Redirection**	리다이렉션 필요	`301 Moved Permanently`	리소스 영구 이동
			`302 Found`	일시적 이동
			`304 Not Modified`	캐시된 리소스 사용 가능
4xx	**Client Error**	클라이언트 요청 오류	`400 Bad Request`	잘못된 문법의 요청
			`401 Unauthorized`	인증 필요
			`403 Forbidden`	접근 금지

범위	이름	설명	예시 코드	설명 (대표 예시)
			`404 Not Found`	리소스 없음
			`429 Too Many Requests`	요청 과도 (Rate Limit)
5xx	**Server Error**	서버 내부 오류	`500 Internal Server Error`	서버 내부 예외 발생
			`502 Bad Gateway`	게이트웨이 서버가 잘못된 응답 수신
			`503 Service Unavailable`	서비스 일시 중단 (과부하 등)
			`504 Gateway Timeout`	게이트웨이 응답 지연

[서정원] 14. Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽

KICE 소프트웨어 아키텍처 시험 - 문항 14번 스터디 가이드

🏷 문제 개요

문항 14번: Nginx와 Tomcat을 Reverse Proxy 형태로 연동한 환경에서 HTTP 응답 코드에 대한 이해를 묻는 문제

- **배점**: 4점
- **난이도**: 중
- **출제영역**: Software Architecture 환경 > 프로토콜, 네트워크 및 방화벽

🎯 핵심 개념: Reverse Proxy 아키텍처

Nginx + Tomcat 연동 구조

```
[Client] → [Nginx (Web Server)] → [Tomcat (WAS)]
                  ↓
          정적 파일 처리
          동적 요청 전달
          응답 코드 관리
```

처리 흐름:

1. 클라이언트 요청이 Nginx에 도달
2. Nginx가 요청 분석 (정적/동적)
3. 정적 파일: Nginx에서 직접 처리
4. 동적 요청: Tomcat으로 proxy_pass
5. 결과를 클라이언트에게 응답

✖ 정답 분석: ① 리다이렉션 헤더 오류

틀린 설명

> *"리다이렉션 할 대상 URL을 **referrer** 응답 헤더를 통해 전달받게 된다"*

올바른 설명

http

```
HTTP/1.1 302 Found
Location: https://example.com/new-page     ← 올바른 헤더
```

헤더 비교

헤더	용도	방향
`Location`	리다이렉션 대상 URL	응답 헤더
`Referer`	현재 페이지 유입 경로	요청 헤더

📊 HTTP 응답 코드별 상황 분석

1. 404 Not Found

상황: Proxy pass 설정 오류로 JSP가 WAS로 전달되지 않는 경우

nginx

```
# 잘못된 설정 (JSP를 Nginx에서 처리 시도)
location ~ \.jsp$ {
    try_files $uri =404;  # Tomcat으로 전달하지 않음
}

# 올바른 설정
location ~ \.jsp$ {
    proxy_pass http://tomcat;
}
```

2. 500 Internal Server Error

상황: 애플리케이션에서 예외 미처리

java

```
@RequestMapping("/example")
public String example() {
    // SQLException 발생 시 별도 예외처리 없으면 500
    throw new SQLException("Database error");
}
```

3. 502 Bad Gateway vs 503 Service Unavailable

502: WAS가 응답하지 않을 때

nginx

```
# WAS가 다운된 상태
upstream tomcat {
    server 127.0.0.1:8080;  # 응답 불가 상태
}
```

503: WAS 과부하로 요청 처리 불가

- WAS 스레드 풀이 모두 사용 중
- CPU/메모리 과부하 상태

4. 504 Gateway Timeout

상황: WAS 응답 시간이 설정된 타임아웃 초과

nginx

```
location / {
    proxy_pass http://tomcat;
    proxy_read_timeout 30s;  # 30초 초과 시 504
}
```

🔧 실무 설정 예시

Nginx 최적화 설정

nginx

```
upstream tomcat {
    server 127.0.0.1:8080 weight=1 max_fails=2 fail_timeout=30s;
    keepalive 32;
}

server {
    location / {
        proxy_pass http://tomcat;

        # 타임아웃 설정
        proxy_connect_timeout 5s;
        proxy_send_timeout 60s;
        proxy_read_timeout 60s;

        # 헤더 설정
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
```

```
    # 정적 파일 직접 처리
    location ~* \.(css|js|jpg|png|gif)$ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
```

장애 대응 가이드

APM 도구에서 확인할 포인트

응답 코드	확인 사항	대응 방안
502/503	WAS 스레드 풀, CPU/메모리	스케일 아웃, 튜닝
504	느린 쿼리, 외부 API 지연	쿼리 최적화, 타임아웃 조정
500	애플리케이션 예외 로그	예외 처리 로직 추가

모니터링 명령어

bash

```
# Nginx 상태 확인
nginx -t
systemctl status nginx

# Tomcat 스레드 확인
jstack <tomcat-pid>

# 네트워크 연결 상태
netstat -ant | grep :8080
```

추가 학습 키워드

관련 기술 스택

- **Load Balancer**: L4/L7 스위치, HAProxy
- **WAS**: Tomcat, WebLogic, JBoss
- **모니터링**: APM, Grafana, Prometheus
- **로그 분석**: ELK Stack, Fluentd

실무 연관 개념

- **Connection Pool**: HttpClient, JDBC
- **Circuit Breaker**: Hystrix, Resilience4j
- **Service Mesh**: Istio, Envoy
- **Container**: Docker, Kubernetes

📖 참고 자료

공식 문서

- Nginx 공식 문서
- Apache Tomcat 문서
- HTTP 상태 코드 RFC 7231

실무 가이드

- Nginx Reverse Proxy 설정 가이드
- Tomcat 성능 튜닝 가이드
- HTTP 상태 코드 완전 정리

모니터링 & 디버깅

- APM 도구 비교: New Relic vs DataDog vs APM
- Nginx 로그 분석 방법
- JVM 메모리 분석 도구

아키텍처 설계

- 마이크로서비스 아키텍처 패턴
- 웹 애플리케이션 보안 가이드 (OWASP)
- 클라우드 네이티브 아키텍처

15. Software Architecture 환경> Software 연관 아키텍처

문항 15) (문항 교체) Object Storage와 NAS(Network Attached Storage)를 비교 설명한 내용중 틀린 번호를 선택하고, 왜 틀렸는지 이유를 설명하시오. [4점]
1) Object Storage는 브라우저에서 직접 파일을 업로드하거나 다운로드할 수 있어 애플리케이션 서버를 거치지 않고도 처리할 수 있지만, NAS는 파일 입출력 시 반드시 애플리케이션 서버를 통해야 한다
2) Object Storage는 단일 요청 기준에서 웹 서버보다 더 빠른 성능을 제공하기 때문에, 확장성과 운영 효율 면에서도 강점을 가지며, Serverless 아키텍처에서는 Web 서버를 대체해 정적 콘텐츠를 제공하는 데 널리 활용되고 있다.
3) Object Storage는 RESTful HTTP API를 통해 파일을 처리하는 방식으로 동작하며, NAS는 운영체제의 파일 시스템 호출(File I/O 시스템 콜)을 통해 파일에 접근한다는 차이가 있다.
4) Object Storage에서는 파일을 수정하려면 먼저 다운로드한 뒤 수정하고 다시 업로드해야 하지만, NAS는 저장된 파일을 직접 수정할 수 있다.

틀린 번호 : ( )
틀린 이유 : ( )

(정답) 2번, 단일 요청을 처리할 때 latency 관점에서 웹 서버의 단위 성능이 Object Storage보다 우수하다.
(해설)웹서버는 로컬 디스크에서 즉시 파일을 처리하지만 Object Storage는 네트워크 경로, 인증, 내부 분산 처리 등을 거쳐 응답해야 하기 때문에 단위 성능은 떨어진다. 다만, 대용량 처리 시에는 Object Storage의 확장성이 더 우수하다.
(배점)4
(난이도)중
- 문제유형: 단답형
- 출제영역 대분류>소분류: Software Architecture 환경> Software 연관 아키텍처
- 문제 제목: 트랜잭션 처리와 EAI 호출
- 출제 의도: 트랜잭션 처리 과정과 EAI 호출 처리방식에 대한 이해도를 확인한다.

✓ 1) 맞는 설명

- **Object Storage**는 S3 같은 시스템을 통해 브라우저에서 **HTTP API (PUT/GET)** 로 직접 접근 가능
- **NAS**는 NFS, SMB 프로토콜을 통해 **운영체제의 파일 시스템**으로 마운트되어 사용되므로
  파일 입출력은 **애플리케이션 서버를 반드시 거쳐야 함**

✕ 2) 틀린 설명 → 정답

> *"Object Storage는 단일 요청 기준에서 웹 서버보다 더 빠른 성능을 제공한다"*

- **틀린 부분**: Object Storage는 일반적으로 **높은 지연시간 (latency)** 을 가지며,
  단일 요청 성능은 웹 서버(또는 로컬 파일 시스템) 보다 **느린 편**입니다.
- Object Storage의 장점은 **대규모 확장성, 고가용성, 비용 효율성**이지, **낮은 지연 성능이 아님**
- Serverless 환경에서 **정적 콘텐츠 제공에 활용되는 건 맞지만**, 이는 **속도보단 아키텍처 단순화와 비용 절감 때문**

Object Storage는 단일 요청 기준에서 웹 서버보다 더 빠른 성능을 제공한다는 설명이 틀렸습니다.

상세 분석:

1. 성능 비교 - Object Storage vs Web Server:

HTTP 프로토콜은 속도나 효율성을 위해 설계되지 않았으며, SAN과 NAS 프로토콜이 모두 성능에 중점을 둔 반면 Object Storage는 HTTP 프로토콜을 사용합니다 The Battle is On: SAN vs. NAS vs. Object - Quobyte. HTTP 프로토콜은 텍스트 기반이므로 파일 시스템 프로토콜보다 처리 속도가 느리고 비용이 많이 듭니다 File Storage vs. Object Storage: What's the Difference and Why it Matters - Quobyte.

2. Object Storage의 성능 특성:

- Object Storage는 일반적으로 Time to First Byte(TTFB) 측면에서 밀리초 단위의 지연시간을 가집니다 BlogArchitecting IT
- Object Storage의 전통적인 약점은 성능이었습니다 Computer WeeklyTechTarget
- Object Storage의 성능 강점은 수천 개의 연결을 열 수 있는 분석 애플리케이션에서 나타납니다 High-performance object storage: What's driving it? | Computer Weekly

3. 올바른 Object Storage 활용 분야:

Object Storage는 대규모 확장성, 지리적 도달 범위, 대용량 데이터 저장의 경제성 요구사항을 해결하기 위해 개발되었습니다 Object Storage vs NAS | Benefits & Definitions | Scality. 단일 요청 성능보다는 **확장성, 내구성, 비용 효율성**이 주요 강점입니다.

✓ 3) 맞는 설명

- **Object Storage**: REST API 기반 (ex. `PUT /object`, `GET /object`)
- **NAS**: OS가 제공하는 파일 시스템 인터페이스 (`open()`, `read()`, `write()` 등의 시스템 콜)

✓ 4) 맞는 설명

- **Object Storage**는 **immutable(불변성)** 개념을 따르기 때문에
  파일을 **수정하려면 다운로드 → 수정 → 재업로드** 해야 함
- 반면 **NAS**는 로컬 파일 시스템처럼 접근 가능하여 **부분 수정 가능**

정리

보기	설명	판단	비고
1	Object Storage는 직접 접근 가능, NAS는 서버 거쳐야 함	✓	정확

보기	설명	판단	비고
2	Object Storage가 단일 요청 기준으로 더 빠르다	✗	**객관적 사실 아님**
3	접근 방식: REST API vs 시스템콜	✓	정확
4	수정 방식 차이: 재업로드 vs 직접 수정	✓	정확

📚 참고 자료

- AWS S3 vs EFS vs EBS 비교
- Google Cloud - Cloud Storage vs Filestore
- DigitalOcean - What is Object Storage?
- Naver D2 - Object Storage 개념과 활용

- **기술 비교 자료:**
  - Scality의 Object Storage vs NAS 비교 Object Storage vs NAS | Benefits & Definitions | Scality
  - Quobyte의 SAN vs NAS vs Object Storage 가이드 The Battle is On: SAN vs. NAS vs. Object - Quobyte
- **성능 분석 자료:**
  - Computer Weekly의 High-performance Object Storage 분석 High-performance object storage: What's driving it? | Computer Weekly
  - Architecting IT의 Object Storage 성능 가이드 Object Storage Essential Capabilities #4 - Performance - Architecting IT
- **실무 최적화 가이드:**
  - Scaleway의 Object Storage 성능 최적화 가이드 Optimize your Object Storage performance | Scaleway Documentation
  - Microsoft의 Azure Blob Storage 성능 체크리스트 Performance and scalability checklist for Blob storage - Azure Storage | Microsoft Learn

16. 신기술 > Cloud Service

문항 16) 구축중인 프로젝트에서 비동기 처리 아키텍처 구현을 위해 Apache Kafka를 도입하였다. 설계 단계에서 Kafka 설계 기본 원칙을 도출하였고, 아래 내용은 그 중 일부 항목이다. 이 중 아키텍처 설계가 잘못된 항목을 2개 고르시오. [4점]

① 토픽에 발행된 메시지가 저장되는 단위인 파티션은 순서 보장, 병렬 처리, 파일 핸들 수 등을 고려하여 개수를 설계해야 한다. 일반적으로 동일 컨슈머 그룹에 있는 컨슈머 개수와 동일하게 설정하며, 설정한 파티션 개수는 늘릴 수 있어도 줄일 수는 없다.
② 메시지 순서가 보장되어야 하는 토픽은 단일 파티션으로 구성하거나, 여러 개의 파티션으로 구성하는 경우 메시지 offset별 파티셔닝을 적용한다.
③ Kafka 브로커 및 커넥트의 클러스터 관리에 사용되는 주키퍼(Zookeeper)는 가용성 확보를 위해 최소 2대 이상으로 구성한다.
④ 레코드 사이즈 최적화를 위해 compression.type 설정값을 이용해 프로듀서에서 메시지 전송 시 압축을 적용할 수 있다. gzip 방식 적용 시 압축률을 높일 수 있으나 CPU 사용률이 증가하므로 주의한다.
⑤ Kafka는 처리된 메시지가 파일 시스템에 일정 기간 동안 보관하며 해당 특성을 활용하여 배치 처리, 재처리 등에 활용할 수 있으나, 메시지 offset을 조정해야 하므로 운영 시 컨슈머 중단 등 제약이 발생할 수 있다.

(정답)2{|}3
(해설)② 메시지 순서 보장을 위해서는 key별 파티셔닝을 적용해야 한다. Offset은 처리된 메시지 위치 정보이다. ③ 주키퍼는 과반수 선출(majority voting/quorums) 로직에 따라 전체 대수의 절반 이상이 가동 중이어야 동작하므로 반드시 홀수 대수로 구성해야 한다.
(배점)4
(난이도)중
- 문제유형: 선다형
- 출제영역 대분류>소분류: 신기술 > Cloud Service
- 문제 제목: Kafka 설계 시 고려사항
- 출제 의도: Kafka 설계 시 고려사항에 대한 이해

📌 보기별 상세 해설

✔ ① 맞는 설명

- **파티션 개수 설계 기준**:
  - 파티션은 **순서 보장 단위**
  - 병렬 처리 성능 = 파티션 수에 비례
  - 일반적으로 **컨슈머 수 = 파티션 수** 이상이 효율적
  - ❗ **파티션은 늘릴 수는 있어도 줄일 수 없음** → 맞는 설명

✖ ② 잘못된 설명 → 정답

> *"메시지 순서가 보장되어야 하는 토픽은 단일 파티션으로 구성하거나, 여러 개의 파티션으로 구성하는 경우 메시지 offset 별 파티셔닝을 적용한다."*

- **틀린 이유**: Kafka에서 메시지 순서 보장은 **파티션 단위로만 가능**
  - 여러 파티션을 사용하는 경우, **offset이 아니라 key**에 따라 파티셔닝이 적용됨
  - offset은 브로커 내부에서 자동 증가하는 값이지, 파티셔닝 기준으로 쓰이지 않음
- 따라서 **offset별 파티셔닝은 불가능하며 개념 오류**

✓ **수정**: 여러 파티션을 쓸 때도 특정 키(key)를 기준으로 파티셔닝을 해야 **동일한 키의 메시지 순서가 보장**됨

✖ ③ 잘못된 설명 → 정답

> *"Zookeeper는 가용성 확보를 위해 최소 2대 이상으로 구성한다."*

- **틀린 이유**: Zookeeper는 **Quorum(정족수 기반)** 합의 구조를 사용
  - **항상 홀수 개 (3대, 5대 등)**로 구성해야 함
  - 2대로 구성 시 **한 대 장애만 나도 Quorum을 상실** → 클러스터 동작 불가

✓ **정답**: **최소 3대 이상**이 가용성을 위한 올바른 구성

✓ ④ 맞는 설명

- Kafka의 `compression.type` (예: `gzip`, `snappy`, `lz4`)은 **프로듀서에서 메시지 압축** 시 사용
- `gzip`: 높은 압축률, 상대적으로 CPU 사용량 높음 → 실무에서도 자주 쓰는 팁
- ✓ 실무 기준과 일치함

✓ ⑤ 맞는 설명

- Kafka는 **메시지를 디스크에 저장**하므로 재처리/배치가 가능
- 과거 메시지 재처리 시 **offset rewind 필요**
- 이 경우 기존 컨슈머는 **중단하고 offset을 조정**하거나 **다른 컨슈머 그룹**으로 재소비
- ✓ 정확한 운영 특성 설명

정리

번호	설명	판단	이유
①	파티션 수 설계 원칙 설명	✓ 맞음	실무 기준 적합
②	offset 기준 파티셔닝	✗ 틀림	Kafka는 key 기준 파티셔닝
③	Zookeeper 2대 구성	✗ 틀림	최소 3대 이상 홀수 필요
④	압축 및 CPU 영향 설명	✓ 맞음	정확
⑤	배치/재처리 시 offset 이슈	✓ 맞음	정확한 운영 현실 반영

참고 문서

- Apache Kafka 공식 문서 – Partitions and Ordering
- Kafka: Zookeeper Requirements
- Kafka Producer Configuration

- **Apache Kafka 공식 문서:**
  - Apache Kafka 공식 문서 Apache Kafka (https://kafka.apache.org/documentation/)
  - Confluent의 Kafka 파티션 가이드 Intro to Kafka Partitions | Apache Kafka® 101
- **파티셔닝 전략 가이드:**
  - Confluent의 Kafka Partition Key 가이드 Apache Kafka Partition Key: A Comprehensive Guide
  - Redpanda의 Kafka 파티션 전략 가이드 Kafka Partition Strategies: Optimize Your Data Streaming
- **Zookeeper 클러스터 구성:**
  - Apache Zookeeper 관리자 가이드 ZooKeeper Administrator's Guide
  - Red Hat의 Zookeeper 구성 가이드 Chapter 3. Configuring ZooKeeper | Using AMQ Streams on RHEL | Streams for Apache Kafka | 2.1 | Red Hat Documentation
- **기술 블로그 및 튜토리얼:**
  - Medium의 Kafka Topics, Partitions, Offsets 가이드 Apache Kafka Guide #2 Topics, Partitions and Offsets | by Paul Ravvich | Apache Kafka At the Gates of Mastery | Medium
  - Kafka 클러스터 구성 선택 가이드 Running Zookeeper in replicated mode (for Apache Kafka)

[김수진] 16. 신기술 > Cloud Service

문제 해설)

② 메시지 순서 보장 방식이 틀렸습니다

- ✗ "offset별 파티셔닝" → offset은 메시지 위치 정보일 뿐
- ✓ **"key별 파티셔닝"**이 정답
- 같은 key를 가진 메시지는 항상 같은 파티션으로 가서 순서 보장

③ Zookeeper 구성이 틀렸습니다

- ✗ "최소 2대 이상"
- ✓ **"반드시 홀수 대수"**로 구성 (3, 5, 7대...)
- 과반수 선출(majority voting) 방식이므로 홀수 필수

[Kafka 핵심 개념]

기본 구조

```
Producer → Kafka Cluster → Consumer
           (Broker들)
```

주요 컴포넌트

📊 Topic (주제)

- 메시지를 분류하는 카테고리
- 예: `user-orders`, `payment-events`, `log-data`

🔪 Partition (파티션)

- Topic을 물리적으로 나눈 단위
- 병렬 처리와 확장성 제공
- 각 파티션 내에서만 순서 보장

🏢 Broker (브로커)

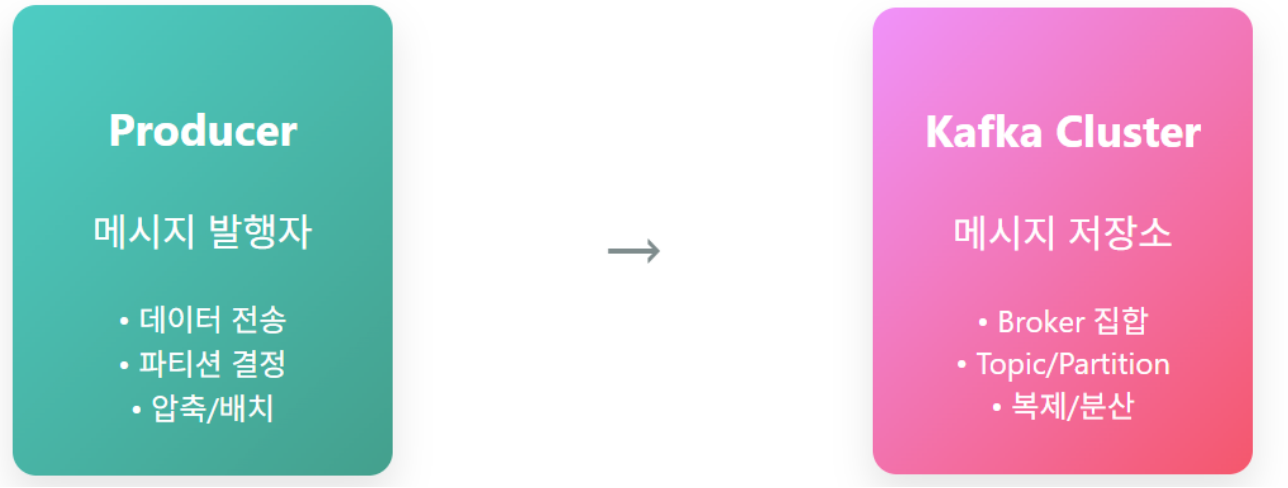
- Kafka 서버 인스턴스
- 데이터 저장 및 클라이언트 요청 처리

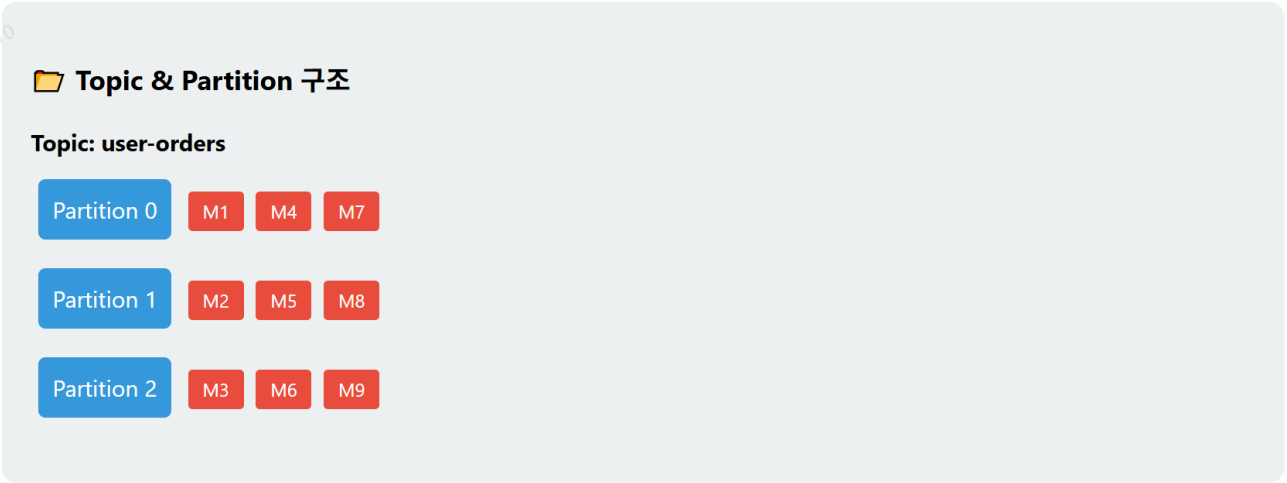
🔢 Offset

- 파티션 내 메시지의 고유 번호

- 0부터 순차 증가

2. 상세 아키텍처





🔄 메시지 처리 흐름

1. Producer가 메시지를 Topic으로 전송
2. Key 기반으로 Partition 결정 (key가 없으면 Round-robin)
3. Broker가 메시지를 Partition에 저장 (Offset 부여)
4. Consumer가 Partition에서 메시지 조회
5. Consumer가 Offset 커밋하여 처리 완료 표시

Broker

- **역할:** 메시지 저장 및 전달
- **클러스터:** 여러 Broker로 구성
- **Leader/Follower:** 파티션별 복제
- **Zookeeper:** 메타데이터 관리

Consumer Group

- **병렬 처리:** 같은 그룹 내 Consumer들
- **파티션 할당:** 1:1 매핑 원칙
- **Rebalancing:** Consumer 추가/제거 시
- **Offset 관리:** 그룹별 독립적

Offset

- **순차 번호:** 0부터 증가
- **파티션별:** 독립적 관리
- **커밋:** 처리 완료 표시
- **재처리:** Offset 조정 가능

[Zookeeper란?]

분산 시스템의 두뇌 역할을 하는 코디네이션 서비스

생각해보세요: 여러 대의 서버가 협력해야 할 때...

- 누가 리더인지 어떻게 알지?
- 설정 정보를 어디서 공유하지?
- 서버가 살아있는지 어떻게 확인하지?

→ 이 모든 걸 Zookeeper가 해결!

분산 시스템의 두뇌 역할

여러 서버가 협력할 때 필요한 모든 코디네이션을 담당하는 중앙집중식 서비스

리더 선출

분산 시스템에서 하나의 리더를 선출하고 관리

- Master/Slave 구조
- 장애 시 자동 리더 교체
- Split-brain 방지

설정 관리

중앙화된 설정 정보 저장 및 배포

- 실시간 설정 변경
- 모든 노드 동기화
- 버전 관리

서비스 디스커버리

서비스 위치 정보 관리

- 동적 서비스 등록
- 헬스 체크
- 자동 failover

분산 락

여러 노드 간 동시성 제어

- 크리티컬 섹션 보호
- 순서 보장
- 데드락 방지

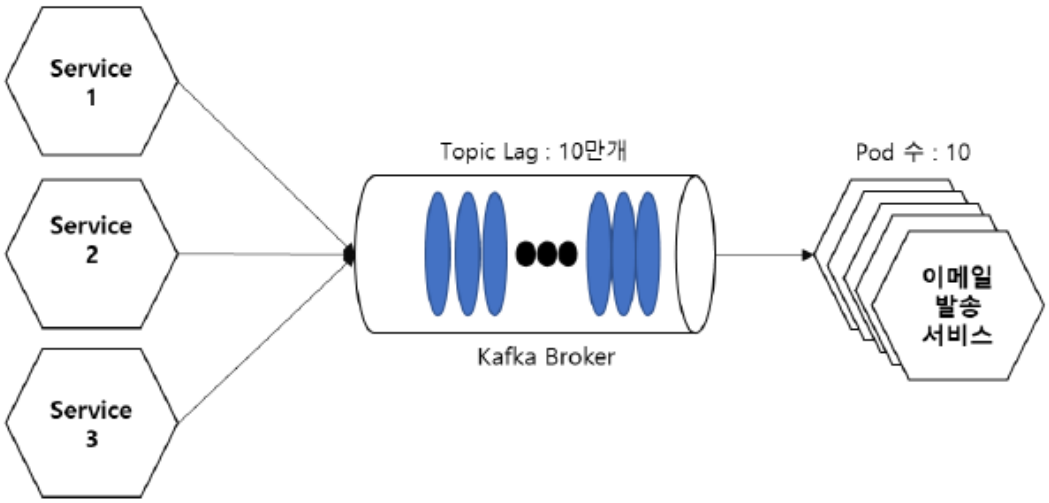
Kafka에서 Zookeeper 역할

1. **Broker 등록:** 각 Kafka Broker가 자신의 정보를 Zookeeper에 등록
2. **파티션 리더 선출:** 각 파티션의 Leader Broker 결정
3. **메타데이터 관리:** Topic, Partition, Consumer Group 정보 저장
4. **장애 감지:** Broker 다운 시 자동 감지하고 리더 재선출
5. **설정 동기화:** 모든 Broker에 설정 변경사항 전파

17. 신기술> Microservice Architecture

문항 17) Spring Boot 기반으로 개발된 이메일 발송 서비스는 다른 서비스에서 발행하는 Kafka 메시지를 구독하여 email을 발송하는 pod 형태의 서비스이다.
발송하는 email은 선입선출이 중요하지 않지만 반드시 발송은 되어야 한다. 토픽의 lag이 다량으로 쌓여 처리가 늦어지는 상황이 발생하였다. 원인 및 해결방안을 간단히 작성하시오. [4점]

[구성도]



[현상]

1. 토픽의 파티션 수는 10개로 설정되어 있다. Consumer도 10개로 정상적으로 연결되어 있다.
2. Service 1,2,3에서 발송한 send_email 토픽의 lag이 10만개이상 누적되어 있다.
3. 이메일 발송 서비스는 pod가 10개로 구동 되었으나 하나의 pod를 제외하고 자원 사용률이 극히 낮다.
4. 서비스 모니터링 도구에서 down되거나 hang 상태의 서비스는 발견되지 않았다.
5. Kafka Broker 서버의 자원 사용률은 높지 않다
6. 네트워크의 대역폭은 여유가 있는 상황이다.

1)
원인 : _____
2)
해결방안 : _____

(정답)
1. 파티션 키 편중
2. 파티션 키가 10개의 파티션으로 고르게 분산될 수 있도록 파티션 키를 설정
(해설)성능향상을 위해 파티션을 10개로 분할하였으나 파티션 키를 적절히 배분하지 않아 하나의 파티션에만 편중되어 메시지가 전달되고 있는 상황이다. 발송자는 파티션 키가 10개의 파티션으로 분산될 수 있도록 파티션 키를 설정해야 10개의 파티션을 효율적으로 사용할 수 있다.
(배점)4
(난이도)중
- 문제유형: 단답형
- 출제영역: 신기술> Microservice Architecture
- 문제 제목: Kafka 병목현상에 대한 조치
- 출제 의도: Kafka 토픽의 파티션을 성능 향상 목적으로 조정할 수 있고, 파티션 키에 대해 이해하고 있다.

✓ 정답

1) 원인:

이메일 발송 서비스의 Kafka Consumer가 하나의 Pod에서만 메시지를 소비하고 있으며, 나머지 9개의 Pod는 같은 Consumer Group으로 중복 연결되어 있어 실제 메시지 처리를 하지 않기 때문

- Kafka는 동일한 **Consumer Group 내에서는 하나의 파티션을 하나의 Consumer가 독점**
- 현재 구성은 **파티션 수 = 10, Pod 수 = 10**이라 겉보기에는 병렬 소비 가능해 보이나
  실제로는 **하나의 Pod에서만 Consumer가 동작**하고 나머지 9개는 **동일 Consumer ID 또는 비활성화 상태**일 가능성 큼
- 또는 다수 Pod가 **Kafka Consumer Thread를 생성하지 않거나 Blocking 상태**일 수 있음
- 결과적으로 **한 Pod에서만 메시지를 소비 → 처리 병목 → Lag 증가**

2) 해결방안:

- **Kafka Consumer 처리가 각 Pod에서 병렬로 수행되도록 구현 확인 및 수정**
  - Pod마다 고유 Consumer가 파티션을 할당받아 동작하는지 확인
  - `spring.kafka.consumer.group-id`가 모두 동일하고, **Kafka Listener가 각 Pod에서 병렬 Thread로 실행 중인지 점검**
- 각 Pod에서 **독립적으로 메시지를 처리하도록 Kafka Listener 또는 Kafka Streams 구조 점검**
  - 예: `@KafkaListener(concurrency = 1)` 등으로 제한된 설정일 수 있음 → `concurrency` 조정 필요
- 필요 시 **Kafka 파티션 수를 증가**시켜 더 많은 Pod가 병렬 소비할 수 있도록 설계
  - ex) 파티션 수를 20개로 늘리면 최대 20개 Pod가 동시 소비 가능

핵심 요약

항목	내용
📌 원인	대부분의 Pod가 Kafka Consumer 역할을 제대로 수행하지 않고 있음
🛠 해결	Kafka Consumer 병렬성 개선 (`@KafkaListener` , partition 수, group 설정 등 확인)
⚠ 주의	Kafka는 Consumer Group 내 **파티션당 1 Consumer만 소비** 가능함

🔗 참고

- Spring Kafka - Concurrency 설정 가이드
- Kafka Consumer Scaling Guide - Confluent
- Kafka 파티션과 Consumer Group 관계 설명 - Baeldung

✓ Kafka 및 Spring Kafka 병렬 처리 관련 참고 사이트

1. **카프카 기본 개념 및 Consumer 병렬 처리**
   - 우아한형제들 기술 블로그 - Kafka 컨슈머 성능 병목 해결기
     → Pod 여러 개를 띄워도 하나만 소비하는 현상의 원인과 해결법 소개
2. **Spring Kafka 병렬 처리 설정**
   - 토리의 기술 블로그 - Spring Kafka 설정 정리
     → `@KafkaListener`, `concurrency`, `consumer group` 구성 예제 포함
3. **Kafka의 Consumer Group과 파티션 설계**
   - Naver D2 - Kafka의 파티션과 Consumer Group 설명
     → 파티션 수와 병렬 처리 관계, Consumer Group의 동작 방식 설명
4. **Spring 공식 문서 (번역본)**
   - Spring Kafka 한글 번역문 (GitBook)
     → Kafka 설정 및 동작 방식의 한글 요약본

1. 기업 기술 블로그 - 실무 사례

- **사람인HR 기술연구소**: "kafka 설정을 사용한 문제해결" [kafka] The consumer group command timed out while waiting for group to initialize 오류 해결 : 네이버 블로그
  - 메일 시스템에서 Kafka 사용 시 겪은 문제와 해결 방법
  - 메시지 중복 소비 문제, 처리량 개선 방안
  - Spring Cloud Stream Kafka 설정 최적화
  - URL: https://saramin.github.io/2019-09-17-kafka/

2. 개발자 블로그 - Kafka Lag 모니터링

- **Velog**: "[Kafka] Kafka Lag exporter를 이용한 Kafka Consumer Lag 모니터링" Resolving Kafka consumer lag with detailed consumer logs for faster processing
    - Consumer Lag 개념 설명 및 모니터링 방법
    - Kafka Lag Exporter 활용법
    - Prometheus 기반 모니터링 구축
    - URL: https://velog.io/@tedigom/Kafka-Kafka-Lag-exporter%EB%A5%BC-%EC%9D%B4%EC%9A%A9%ED%95%9C-%EB%AA%A8%EB%8B%88%ED%84%B0%EB%A7%81
- **VoidMainVoid Blog**: "아파치 카프카 Lag 모니터링 대시보드 만들기" Monitor Kafka Consumer Lag in Confluent Cloud | Confluent Documentation
    - Burrow, Telegraf, Elasticsearch, Grafana를 활용한 모니터링 대시보드 구축
    - 파티션별 lag 모니터링 방법
    - Slack 알림 설정 방법
    - URL: https://blog.voidmainvoid.net/279

3. 네이버 블로그 - 실제 장애 해결 사례

- **네이버 블로그**: "[kafka] The consumer group command timed out while waiting for group to initialize 오류 해결" [Kafka] Kafka Lag exporter를 이용한 Kafka Consumer Lag 모니터링
    - Kafka 브로커 장애 시 Consumer 그룹 문제 해결
    - `__consumer_offsets` 토픽 복제본 설정 방법
    - 실제 운영 환경에서의 문제 진단 과정
    - URL: https://m.blog.naver.com/PostView.naver?blogId=freepsw&logNo=221058451193

4. 영문 최신 자료 (참고용)

- **DEV Community**: "Kafka Fundamentals: kafka consumer lag" kafka 설정을 사용한 문제해결 (1일 전 게시)
    - 최신 Kafka Consumer Lag 심화 가이드
    - 금융 거래 플랫폼 사례 등 실무 예제

[김수진]17. 신기술> Microservice Architecture

1. 원인: 파티션 키 편중
2. 해결방안: 파티션 키가 10개의 파티션으로 고르게 분산될 수 있도록 파티션 키를 설정

Consumer Group이란?

Consumer Group = 동일한 topic을 함께 처리하는 Consumer들의 논리적 그룹

```
Properties props = new Properties();
props.put("group.id", "email-service-group"); // 이것이 Consumer Group ID
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

핵심 원리: 1 파티션 = 1 Consumer (같은 그룹 내에서)

예시로 이해하기

Topic: send_email (파티션 3개)

```
파티션 0: [msg1, msg4, msg7, ...]
파티션 1: [msg2, msg5, msg8, ...]
파티션 2: [msg3, msg6, msg9, ...]
```

Consumer Group A (email-service-group)

```
Consumer A1 → 파티션 0 담당
Consumer A2 → 파티션 1 담당
Consumer A3 → 파티션 2 담당
```

Consumer Group B (analytics-group)

```
Consumer B1 → 파티션 0,1,2 모두 읽음 (독립적으로)
Consumer B2 → 파티션 0,1,2 모두 읽음 (독립적으로)
```

파티션 할당 시나리오

시나리오 1: Consumer 수 = 파티션 수

```
파티션: [P0, P1, P2]
Consumer: [C1, C2, C3]
결과: C1→P0, C2→P1, C3→P2 (이상적)
```

시나리오 2: Consumer 수 > 파티션 수

```
파티션: [P0, P1, P2]
Consumer: [C1, C2, C3, C4, C5]
결과: C1→P0, C2→P1, C3→P2, C4→유휴, C5→유휴
```

시나리오 3: Consumer 수 < 파티션 수

```
파티션: [P0, P1, P2, P3, P4]
Consumer: [C1, C2]
결과: C1→P0,P1,P2, C2→P3,P4 (부하 불균등)
```

문제 17번 상황 재분석

구성:

- 파티션 10개: [P0, P1, P2, ..., P9]
- Consumer 10개: [C0, C1, C2, ..., C9] (모두 같은 Consumer Group)

정상 할당:

```
C0 → P0, C1 → P1, ..., C9 → P9
```

실제 상황 (파티션 키 편중):

```
P0: 메시지 100,000개 ← C0 과부하
P1: 메시지 0개 ← C1 유휴
P2: 메시지 0개 ← C2 유휴
```

```
...
P9: 메시지 0개 ← C9 유휴
```

Consumer Group의 장점

1. 확장성 (Scale Out)

java

```
// 처리량 증가 시 Consumer만 추가
// 기존: 1개 Consumer → 새로운: 3개 Consumer
// 자동으로 파티션 재할당됨
```

2. 장애 복구 (Fault Tolerance)

```
C1이 죽으면 → C1이 담당하던 파티션을 다른 Consumer가 자동 인수
```

3. 독립적 처리

```
Group A: 실시간 이메일 발송
Group B: 배치 분석
Group C: 로그 저장
→ 같은 메시지를 각각 독립적으로 처리
```

실무 예시

이메일 서비스 아키텍처

java

```
// Producer (주문 서비스)
producer.send(new ProducerRecord<>("order-events",
    orderId.toString(), // 파티션 키 (orderId로 분산)
    orderData));

// Consumer Group 1: 이메일 발송
@KafkaListener(topics = "order-events", groupId = "email-group")
public void sendEmail(OrderEvent event) { ... }

// Consumer Group 2: 재고 관리
@KafkaListener(topics = "order-events", groupId = "inventory-group")
public void updateInventory(OrderEvent event) { ... }
```

결과:

- 같은 주문 이벤트를 이메일 발송과 재고 관리에서 각각 처리
- orderId로 파티셔닝되어 같은 주문의 이벤트는 순서 보장

18. 신기술> Microservice Architecture

문항 18) MSA환경에서는 사용자 인증을 구현하는 방법 중 JWT(JSON Web Token)를 이용한 방식이 있다. 다음 중 JWT의 설명 중 잘못된 것을 고르시오. [3점]

① Signature가 있어 JWT의 위변조를 막을 수 있다.
② Header, Payload는 암호화되어 있지 않기 때문에 정보가 노출될 수 있다.
③ (Private 단어 삭제)Claims에는 시스템에서 사용하는 정보를 추가로 작성할 수 있다.
④ Stateless하기 때문에 별도 세션서버가 필요 없다.
⑤ 공격자에 의해 JWT가 탈취되었을 때 해당 계정을 차단하기 수월하다.

(정답)5
(해설) 기존 세션방식에서는 세션의 해당 계정 정보만 삭제하면 되었지만, JWT는 세션리스한 상태로 JWT만 가지고 인증을 처리하기에 탈취되었을 경우 계정차단을 하기 쉽지 않다. 인증에 대한 정보를 서버측이 아닌 클라이언트측에서 소유하고 있으므로, JWT 무효화에 대해 서버측 구현을 별도로 해야 한다. (배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역: 신기술> Microservice Architecture
- 문제 제목: JWT 특징
- 출제 의도: JWT의 특징 중 세션리스한 방식이 가져오는 효과에 대해 이해한다.

✓ 정답: ⑤

해설 요약

◆ 1~4번은 모두 맞는 설명

- ① Signature는 JWT의 위변조 방지 기능을 제공.
- ② Header/Payload는 단순 Base64 인코딩만 되어 있으므로 **누구나 열람 가능**.
- ③ Claims에는 사용자 ID, 권한, 만료시간 등 커스텀 정보 저장 가능.
- ④ JWT는 상태를 서버에 저장하지 않기 때문에 **세션 서버 불필요 (Stateless 인증)**.

◆ ⑤는 잘못된 설명

- JWT는 서버가 상태를 저장하지 않기 때문에 **토큰 자체가 인증의 유일한 근거**.
- 즉, 탈취된 경우 해당 토큰은 **유효기간까지 계속 유효**하며, 서버는 이를 식별하거나 차단할 수 없음.
- **해결 방안**:
    - 토큰 블랙리스트 관리 (ex. Redis + JWT ID 저장)

- 짧은 만료시간 설정 + Refresh Token 전략
- 토큰에 client 정보(UA, IP 등)를 포함하고 비교

📚 관련 개념 정리

항목	설명
JWT 구조	`Header.Payload.Signature` (Base64 인코딩)
암호화 여부	Payload는 **암호화되지 않음** → 민감 정보 저장 금지
Stateless 인증	서버는 세션 저장 없음. 모든 인증 상태는 클라이언트가 가진 토큰에 있음
위변조 방지	`Signature` 검증으로 위변조 탐지 가능 (HMAC, RSA 등)
토큰 만료 및 재발급	Refresh Token 이용 (보통 Redis 등으로 관리)

✔ JWT (JSON Web Token) 개념 정리

◆ 1. 정의

JWT는 **JSON 기반**의 **토큰 형식**으로, **사용자 인증 및 권한 부여**를 위한 인증 수단이다.
토큰 안에 **사용자 정보, 권한, 만료 시간 등**을 담아 **서버-클라이언트 간 신뢰된 정보 전송**에 사용된다.

◆ 2. 구조

JWT는 3개의 파트로 구성되며, **Base64URL 인코딩**된 문자열 3개를 `.`으로 연결한 형태이다:

<Header>.<Payload>.<Signature>

📌 예시

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. <-- Header
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I... <-- Payload
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c <-- Signature

📦 각 구성요소

구성요소	설명
Header	토큰 타입 (`JWT`), 서명 알고리즘 (`HS256` , `RS256`)
Payload	사용자 정보와 클레임 (Claims)
Signature	Header + Payload 를 secret key로 서명한 값. 위변조 방지

◆ 3. 특징

특징	설명
Stateless	서버가 세션을 저장하지 않고, 클라이언트가 토큰을 소유
Self-contained	사용자 인증 정보와 만료 시간, 권한 등 모든 정보가 토큰 내부에 존재
확장 가능	Payload에 필요한 데이터 필드 자유롭게 추가 가능 (ex. `role` , `email`)
Base64 인코딩	Payload는 암호화되지 않고 인코딩만 됨 → 누구나 내용 열람 가능
서명(Signature)	위변조 방지를 위해 secret key나 공개키로 서명

◆ 4. 사용 흐름

1. **로그인 요청**: 클라이언트가 아이디/비밀번호로 로그인
2. **토큰 발급**: 서버가 JWT 발급 후 응답
3. **토큰 저장**: 클라이언트가 JWT를 로컬스토리지/세션/쿠키 등에 저장
4. **요청 시 사용**: API 호출 시 Authorization 헤더에 토큰 포함
   `Authorization: Bearer <JWT>`
5. **서버 검증**: 서버는 서명을 검증하고 Payload의 정보로 인증 수행

◆ 5. JWT 단점 및 보완

문제점	설명
탈취 시 취소 어려움	서버가 상태를 저장하지 않기 때문에 탈취된 토큰을 실시간으로 무효화하기 어려움
Payload 노출 가능	민감 정보는 Payload에 포함시키면 안 됨 (암호화 아님)
만료 전까지 유효	별도 블랙리스트가 없으면 만료 전까지는 무조건 유효

✓ **보완 방법**:

- Refresh Token 분리
- 토큰 만료 시간 짧게 설정
- 토큰 블랙리스트 (ex. Redis)
- IP/UA 비교 등 추가 검증

✓ JWT vs OAuth2: 개념 및 차이점

항목	JWT	OAuth2
역할	토큰 포맷 (Token Format)	인증 및 권한 부여 프로토콜 (Authorization Protocol)
기능	인증 정보 자체를 담은 자체 완결형 토큰	인증/인가를 위한 표준 절차 (ex. Login, Consent 등)
사용 방식	주로 API 인증용 토큰으로 단독 사용 가능	주로 SSO, 제3자 인증 (ex. Google, Naver Login)에 활용
토큰 형식	JWT (Base64 인코딩된 문자열)	Bearer Token (일반 문자열 또는 JWT 등 사용 가능)
세션 상태	Stateless (서버 세션 불필요)	Access Token은 Stateless, Refresh Token은 상태 저장 가능

항목	JWT	OAuth2
토큰 발급자	직접 발급 가능 (Custom 구현)	Authorization Server 필요 (OAuth 표준 준수 필요)
복잡도	낮음 (간단히 직접 구현 가능)	높음 (권한 코드, 리다이렉션, 인증 서버 등 필요)

- **요약**:
  - **JWT**는 단순한 인증/인가 토큰으로 많이 사용됨.
  - **OAuth2**는 인증 "절차"에 대한 표준이고, 그 결과로 **JWT 같은 토큰을 사용할 수 있음**.

🔧 Spring Boot에서 JWT 구현 방식

◆ 1. 의존성 추가 (`build.gradle` 또는 `pom.xml`)

```
implementation 'io.jsonwebtoken:jjwt:0.9.1'
```

◆ 2. JWT 생성

```
String jwt = Jwts.builder()
   .setSubject(username)
   .setIssuedAt(new Date())
   .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
   .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
   .compact();
```

◆ 3. JWT 검증 및 필터 설정

```
Claims claims = Jwts.parser()
   .setSigningKey(SECRET_KEY)
   .parseClaimsJws(token)
   .getBody();
```

Spring Security를 사용하는 경우 `OncePerRequestFilter` 를 상속받아 커스텀 필터 작성:

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {
   protected void doFilterInternal(...) {
      // JWT 헤더에서 추출 → 인증 정보 설정
      SecurityContextHolder.getContext().setAuthentication(authentication);
```

```
    }
}
```

◆ 4. 인증 및 인가 처리 흐름

1. 로그인 시도 → JWT 발급
2. 클라이언트가 요청 시 JWT를 Authorization 헤더에 포함
3. JWT Filter에서 서명 검증, 클레임 추출
4. SecurityContext에 사용자 정보 저장
5. Spring Security가 권한 체크 처리

◆ 5. Refresh Token 전략

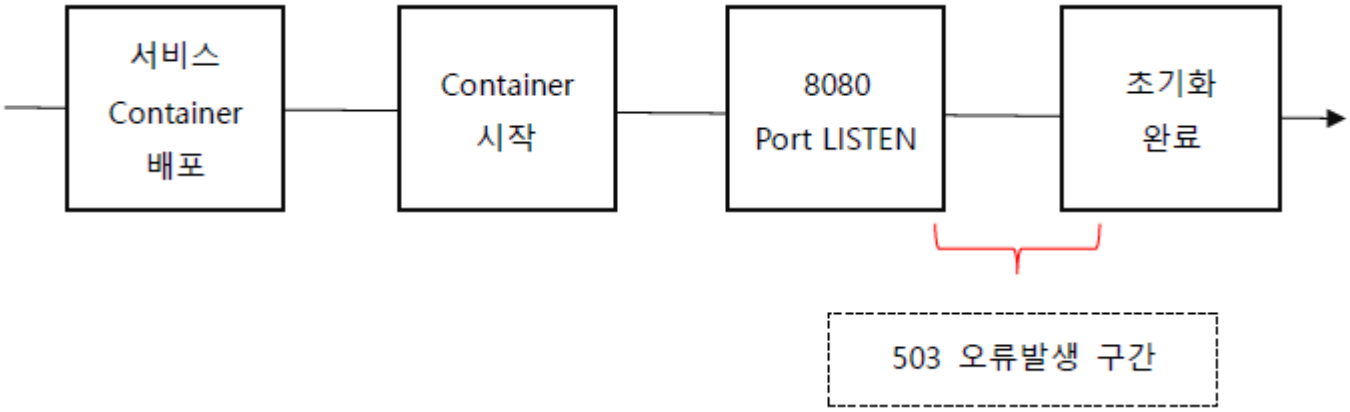
항목	설명
Access Token	유효시간 짧음 (ex. 15분)
Refresh Token	DB나 Redis에 저장, 유효기간 길게 설정 (ex. 2주)
재발급 로직	Access Token 만료 시 Refresh Token으로 새로 발급 요청

19. 신기술> Microservice Architecture

문항19) 다음 Application에서 발생하는 오류를 방지할 수 있는 Kubernetes 상태점검 기능은 무엇인지 보기에서 고르시오. [3점]

[Application 설명]
- spring boot application은 Kubernetes Container로 구동되는 형태로 구성 되어있다.
- application은 8080 포트로 서비스를 한다.
- 서비스가 기동 될 때 일련의 초기화 과정이 필요하다.
- 초기화가 완료되면 아래 API 호출 시 http status 200으로 응답하고, 초기화 전에는 503으로 응답한다.
  상태점검 API : /app/ready

[오류 설명]
Container가 신규로 배포되고 시작할 때마다 순간적으로 503오류가 발생하고 있다. 일정시간이 지난 후에는 서비스가 정상적으로 응답을 한다.
오류가 발생하는 시점을 분석한 결과 아래 그림과 같이 8080 Port가 활성 화 된 직후 일정시간 발생한다.



① Checkup Probe
② Readiness Probe
③ Health Probe
④ Initialize Probe

⑤ Liveness Probe 예시 삭제

(정답)2
(해설) 상태점검에는 컨테이너의 실행 여부 및 컨테이너의 서비스 제공 상태 여부를 체크하며, 각각 Liveness Check 와 Readiness Check 라는 명칭으로 되어 있다. 그 중 서비스가 정상적인 서비스를 제공할 수 있도록 초기화가 완료된 상태를 점검하는 것은 Readiness Probe 이다.
(배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역: 신기술> Microservice Architecture
- 문제 제목: Liveness, Readiness 점검
- 출제 의도: 컨테이너 오케스트레이션 시스템에서 컨테이너 상태점검에 대해 이해한다.

✓ 정답: ② **Readiness Probe**

해설

상태 점검 종류	목적	적합한 시점
Liveness Probe	컨테이너가 **죽었는지** 감지	실행 중이지만 고장난 상태 감지용
Readiness Probe	컨테이너가 **서비스를 받을 준비가 되었는지** 감지	초기화 과정이 끝났는지 판단
Startup Probe	애플리케이션이 **기동이 느릴 때** 사용 (느린 초기화 대응)	Spring Boot처럼 느린 앱
(Checkup, Health, Initialize 등은 Kubernetes 표준 Probe 아님)		

✓ Readiness Probe 설명

- 트래픽 유입 여부를 결정하는 데 사용됨
- 해당 Probe가 성공(`200 OK`)해야만 **Service로부터 트래픽을 받기 시작함**
- 초기화 중인 컨테이너는 `503` 을 반환할 수 있음 → **이때는 트래픽 차단**

📚 YAML 예시

```
readinessProbe:
  httpGet:
    path: /app/ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
  failureThreshold: 3
```

Liveness vs Readiness 차이

항목	Liveness Probe	Readiness Probe
목적	죽었는지 확인	준비되었는지 확인
실패 시	컨테이너 재시작	트래픽 차단 (Service 대상 제외)
적용 대상	장시간 뻗은 서비스 감지용	초기화 지연되거나 외부 의존성 대기 중일 때

✓ Kubernetes Probe 유형 비교

Probe 종류	주요 목적	실패 시 동작	사용 시점	대표 사용 예
Liveness Probe	컨테이너가 **살아있는지** 감지	실패 시 컨테이너 **재시작**	서비스 **운영 중**	무한 루프 등 애플리케이션 hang 상태 감지
Readiness Probe	컨테이너가 **트래픽 받을 준비 완료** 여부 감지	실패 시 **Service 대상에서 제외됨**	서비스 **초기화 중 / 운영 중**	초기화 중 트래픽 차단 등
Startup Probe	**느린 기동** 애플리케이션 초기화 감지	실패 시 컨테이너 **재시작** (기동 실패로 판단)	**기동 시점**	Spring Boot 등 기동이 오래 걸리는 앱

🔎 각 Probe 상세 설명

◆ 1. Liveness Probe

- 목적: 컨테이너가 **정상 실행 중인지** 확인
- 주로 **무한 루프, deadlock**, DB 연결 끊김 등에서 복구용
- 실패 시: `kubectl delete pod` **와 동일한 효과**
  → kubelet이 해당 컨테이너를 재시작

예시:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
```

◆ 2. Readiness Probe

- 목적: 컨테이너가 **서비스 요청을 처리할 준비가 되었는지**
- 실패 시: 해당 컨테이너는 **서비스 트래픽 대상에서 제외**
- 예: Spring Boot 앱에서 `/ready` API가 503 → 200으로 바뀔 때 사용

예시:

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  periodSeconds: 5
  failureThreshold: 3
```

◆ 3. Startup Probe

- 목적: 기동이 느린 애플리케이션이 정상적으로 기동되었는지 확인
- 사용 시: `Liveness Probe` 보다 먼저 실행되고, 기동 완료 시 종료됨
- 기동 완료 전까지 `Liveness Probe` 는 대기함 → **false positive 방지**

예시:

```
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  failureThreshold: 30
  periodSeconds: 10
```

✓ 세 가지 Probe 동작 순서 요약

(Pod 시작)
  ↓
[Startup Probe 실행 중]
  ↓ OK
[Readiness Probe 실행 시작]
  ↓ OK → 트래픽 수신 시작
[Liveness Probe 계속 실행 중]

✓ 최신 Kubernetes Probe 관련 참고 문서

1. 📚 Kubernetes 공식 문서 (한국어)

- **링크**: https://kubernetes.io/ko/docs/concepts/workloads/pods/pod-lifecycle/#%EC%83%81%ED%83%9C-%EC%A0%90%EA%B2%80
- **내용**: Liveness, Readiness, Startup Probe 개념, 동작 방식, YAML 설정 예시 포함
- **특징**: Kubernetes 공식 번역 문서로 신뢰성 높고 업데이트 주기 빠름

---

2. 실무 가이드: Baeldung - Kubernetes Health Checks

- **링크**: https://www.baeldung.com/kubernetes-health-checks
- **내용**: Spring Boot 기반 애플리케이션에 Liveness, Readiness, Startup Probe 설정하는 실무 예제 제공
- **특징**: Spring 기반 시스템과의 연계 설명이 강점

---

3. Medium 실습 예제: Spring Boot Health Check with Probes

- **링크**: https://medium.com/@timokothe/kubernetes-liveness-and-readiness-probes-for-spring-boot-applications-9843a1d22f8
- **내용**: `/actuator/health` API를 활용한 Spring Boot Health Check 설정 실습 예
- **특징**: Spring Boot + Actuator + Kubernetes 통합 예시 중심

---

4. 🇰🇷 블로그: 프로브 개념과 실무 설정 정리

- **링크**: https://junyson.tistory.com/135
- **내용**: 한국어로 정리된 Probe 동작 원리와 YAML 예제

- **특징**: 초보자 관점에서 친절하게 설명됨

---

5. 📊 Spring 공식 문서 (Actuator 연계)

- **링크**: https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/
- **내용**: `/actuator/health`, `/actuator/readiness`, `/actuator/liveness` endpoint 설명
- **특징**: Kubernetes와 Spring Actuator 연계 시 핵심 문서

[서정원] 19. 신기술> Microservice Architecture

Kubernetes Probe 스터디 노트 📚

🏷 문제 상황

KICE 2025 Pilot 문항 19번에서 다음 상황이 제시됨:

- Spring Boot 앱이 Kubernetes Container로 구동
- 컨테이너 시작 시 **초기화 과정 필요**
- 8080 포트 활성화 후 → 초기화 완료 전까지 **503 에러 발생**
- 초기화 완료 후 → 200 정상 응답

🎯 Kubernetes의 3가지 Probe

1️⃣ Liveness Probe

- **목적**: 컨테이너가 살아있는지 확인
- **실패 시**: 컨테이너 재시작
- **사용 예**: 데드락, 무한루프 등으로 응답 불가 상태

2️⃣ Readiness Probe

- **목적**: 트래픽을 받을 준비가 되었는지 확인
- **실패 시**: Service의 엔드포인트에서 제외
- **사용 예**: 의존성 서비스 연결 대기

3️⃣ Startup Probe ☆

- **목적**: 컨테이너가 시작되었는지 확인 (초기화 완료)
- **도입**: Kubernetes 1.16 (2019년)
- **사용 예**: **긴 초기화 시간**이 필요한 애플리케이션

문제 분석

문제에서 제시된 보기:

① Checkup Probe
② Readiness Probe ← 정답
③ Health Probe
④ Initialize Probe
⑤ Liveness Probe

실제 최적 해답:

Startup Probe가 정답이어야 함!

왜 Startup Probe가 정답인가?

yaml

```yaml
# 이 문제 상황에 가장 적합한 설정
startupProbe:
  httpGet:
    path: /app/ready
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
  failureThreshold: 30  # 최대 150초 대기
```

이유:

- **초기화 시간**이 예측 불가능할 때 사용
- Startup Probe 성공 전까지는 Liveness/Readiness Probe 비활성화
- 초기화 완료 후 정상적인 상태 검사로 전환

Probe 선택 가이드

상황	적절한 Probe
애플리케이션 데드락 방지	Liveness
로드밸런서 트래픽 제어	Readiness
긴 초기화 시간 처리	**Startup ☆**

🔍 문제의 한계점

1. **Startup Probe 누락**: 2025년 기준 표준이지만 보기에 없음
2. **차선책 선택**: Readiness Probe를 정답으로 처리
3. **실무 괴리**: 실제로는 Startup Probe 사용이 베스트 프랙티스

💭 스터디 포인트

- **문제 출제 시점**과 **기술 발전** 간의 갭 존재
- **실무에서는** Startup Probe 우선 고려
- **시험에서는** 주어진 보기 내에서 최선 선택

20. Software Architecture 핵심 > JVM 구조 및 동작 특성

문항 20) 다음은 k8s환경에서 pod template을 통해 컨테이너의 Java Heap 메모리를 설정하는 두 가지 방법에 대한 설명이다. 아래의 설명 중 틀린 것을 고르시오. [4점]

예시 1)
...
resources:
limits:
cpu: "1"
memory: "1Gi"
requests:
cpu: "1"
memory: "1Gi"
env:
- name: JAVA_OPTS
value: "-Xms 1g -Xmx 1g"
...

예시 2)
...
resources:
limits:
cpu: "1"
memory: "1Gi"
requests:
cpu: "1"
memory: "1Gi"
env:
- name: JAVA_OPTS
value: "-XX:InitialRAMPercentage= 30.0 -XX:MinRAMPercentage=50.0 -XX:MaxRAMPercentage=80.0"

① 예시 1)의 경우 heap 영역 이외에도 non-heap 영역이 점유하는 메모리로 인해 Pod에 OOMKilled가 발생할 수 있다.
② JVM 기동 시 초기 힙 메모리를 설정하고자 하는 경우에는 '-XX:MinRAMPercentage' 옵션이 아닌 '-XX:InitialRAMPercentage' 옵션을 사용해야 한다.
③ 예시 2)의 경우 JVM 기동 후 초기에 할당된(committed) heap 메모리는 약 300MB이며, 이후에도 최소한 500MB 수준을 유지하게 된다.
④ 초기 기동 시 힙 메모리 확장을 최소화하기 위하여 -Xms와Xmx 또는 InitialRAMPercentage와 MaxRAMPercentage 옵션 값을 동일하게 설정하는 것이 좋다.
⑤ 예시 2)의 경우 pod에 할당된 메모리의 크기를 크게 변경할 경우 힙 메모리의 크기도 자동으로 커지게 되어 GC에 소요되는 시간이 늘어날 수 있다.

(정답)3
(해설)'-XX:MinRAMPercentage' 옵션은 물리 서버(또는 컨테이너)에서 사용 가능한 전체 메모리의 크기가 약 250MB 미만인 경우에 Java 힙 크기의 최대 값을 설정하는데 사용된다. 따라서, 메모리의 크기가 1Gi인 경우에는 무효한 옵션이다.
(배점)4

(난이도)상
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 핵심 > JVM 구조 및 동작 특성
- 문제 제목: JVM 옵션
- 출제 의도: 최신 Java의 힙 메모리 설정 방법에 대한 이해

✓ 정답: ③

해설

✖ ③ 잘못된 설명

- `MinRAMPercentage` 는 컨테이너 메모리가 **작은 경우에만** 적용되며, `1GiB` 와 같이 충분한 메모리 할당 시에는 **효과가 없다**.
- 따라서 "초기 committed heap이 약 300MB, 이후 최소 500MB 유지"는 실제 JVM 동작과 다를 수 있음.

✓ 다른 선택지 설명

보기	설명
①	Heap 이외의 PermGen, Stack, Metaspace, DirectBuffer 등이 포함되므로 OOM 가능성 있음
②	초기 힙을 설정할 땐 `InitialRAMPercentage` 가 맞고, `MinRAMPercentage` 는 최소 유지 용도
④	`-Xms = -Xmx` 로 설정 시 확장 비용 제거 가능
⑤	Pod memory 제한을 키우면 JVM 힙도 커지고 GC pause도 늘 수 있음

📚 관련 개념 정리

옵션	설명
`-Xms` / `-Xmx`	Heap 초기/최대 크기

옵션	설명
`-XX:InitialRAMPercentage`	전체 메모리 대비 힙 초기 비율
`-XX:MinRAMPercentage`	JVM이 최소 확보하려는 힙 비율 (전체 RAM 기준)
`-XX:MaxRAMPercentage`	전체 메모리 대비 최대 힙 허용치

- Java 10 이상에서는 `-Xmx` 대신 RAM 비율 기반 설정 권장됨

✓ 1. Java Heap 설정 옵션 설명 (OpenJDK 10+ 기준)

옵션	설명
`-Xms<size>`	JVM 초기 Heap 크기 (예: `-Xms512m`)
`-Xmx<size>`	JVM 최대 Heap 크기 (예: `-Xmx1024m`)
`-XX:InitialRAMPercentage=<%>`	전체 메모리의 몇 %를 초기 Heap으로 설정
`-XX:MinRAMPercentage=<%>`	JVM이 확보해야 할 최소 Heap 비율
`-XX:MaxRAMPercentage=<%>`	전체 메모리의 몇 %까지 Heap 확장 허용 기본값: `25%`

- `-Xms`, `-Xmx` 는 **절대값 기준**
- `-XX:XXXPercentage` 는 **컨테이너 제한 메모리 비율 기준**

✓ 2. Kubernetes Pod 메모리 설정 + JVM 튜닝 예제

```
resources:
  requests:
    memory: "1Gi"
```

```
  limits:
    memory: "1Gi"

env:
  - name: JAVA_TOOL_OPTIONS
    value: >
      -XX:InitialRAMPercentage=30.0
      -XX:MinRAMPercentage=40.0
      -XX:MaxRAMPercentage=75.0
      -XX:+UseG1GC
      -XX:+PrintGCDetails
```

설정 항목	설명
`resources.limits.memory`	반드시 명시해야 `RAMPercentage` 옵션이 제대로 동작함
`JAVA_TOOL_OPTIONS`	JVM 힙 설정 및 GC, 로그 설정 일괄 지정 가능

✓ 3. 최적화 가이드

◆ A. Pod 메모리 제한 필수

- `MaxRAMPercentage` 등 비율 옵션은 **컨테이너 제한 메모리(`limits.memory`) 기준**으로 계산
- 설정하지 않으면 전체 노드 메모리 기준으로 잡혀서 OOM 위험

◆ B. GC 튜닝은 G1GC 기반

- `-XX:+UseG1GC` + `MaxRAMPercentage` 조합 권장
- `-XX:MaxGCPauseMillis=200` 같이 설정하면 안정적

◆ C. 초기 힙 크기도 명확히

- `-Xmx` 만 설정하면 JVM은 매우 낮은 기본값(`-Xms`)에서 시작 → GC 급증

- `-Xms=Xmx` 또는 `InitialRAMPercentage` 설정 권장

✓ 4. Kubernetes + Java Heap 관련 **연관 설정 사례**

주요 설정	연관 설정	설명
`-XX:MaxRAMPercentage`	`resources.limits.memory` **필수**	이 옵션은 컨테이너 메모리 상한 기준으로 작동함. limits가 없으면 무한대로 인식됨 (노드 메모리 기준으로 계산됨)
`-Xmx`	`-Xms` 함께 지정	`Xmx` 만 지정 시, 초기 heap 크기 `Xms` 는 기본값(1/64 heap)으로 낮게 시작되어 GC 급증 위험 초기/최대 heap을 같게 설정하면 GC 성능 안정
`-XX:InitialRAMPercentage`	`-XX:MaxRAMPercentage` 함께 사용 권장	`MaxRAMPercentage` 와 비율 차이 너무 크지 않게해야함. 초기 메모리 비율과 최대 비율 간의 차이가 크면 GC가 자주 발생
`-Xms = -Xmx`	GC 튜닝 필요 (`-XX:+UseG1GC`)	초기/최대 heap을 같게 설정 시 메모리 급변 없음. GC 전략도 적절히 조합해야 안정적
`JAVA_TOOL_OPTIONS` 환경변수	GC 옵션, 로깅 옵션과 함께 관리	Spring Boot 등 UTF-8 미설정 시 한글 깨짐 문제 발생 가능 `-Dfile.encoding=UTF-8` , `-Xlog:gc*` 등
`readinessProbe`	`HeapInit` 시간 고려한 `initialDelaySeconds` 조정	JVM 초기 heap 할당이 커지면 초기화 시간이 늘어나 readiness 실패 가능성 있음 힙 크기가 크면 초기화 시간 증가 → 초기 503 가능

✓ 5. 실무 팁

- **메모리 제한 설정**:

```
resources:
  requests:
    memory: "1Gi"
```

```
    limits:
      memory: "1Gi"
```

- **GC 로그 확인용 옵션**:

  ```
  -Xlog:gc*:file=/logs/gc.log:time,level,tags
  ```

- **Spring Boot + Actuator 연동** (Probe에 사용 가능):

  management.endpoint.health.probes.enabled=true
  management.endpoints.web.exposure.include=health

🔍 예시 시나리오 A: 퍼센트 기반 메모리 설정 시

```
resources:
  limits:
    memory: "2Gi"
env:
  - name: JAVA_TOOL_OPTIONS
    value: >
      -XX:InitialRAMPercentage=30.0
      -XX:MaxRAMPercentage=75.0
```

문제 발생 가능성:
`limits.memory`가 **명시되지 않은 경우**, `MaxRAMPercentage`는 전체 노드 메모리를 기준으로 동작 → OOM 위험

✓ **해결**: 반드시 `resources.limits.memory` 지정 필요

🔍 예시 시나리오 B: `-Xmx`만 설정한 경우

```
JAVA_OPTS=-Xmx1024m
```

문제:
초기 heap 크기 `-Xms`는 기본값이 되며, 메모리 부족 시 Full GC 빈번

✓ **해결**:

```
JAVA_OPTS=-Xms1024m -Xmx1024m
```

또는 퍼센트 방식 사용:

```
JAVA_TOOL_OPTIONS=-XX:InitialRAMPercentage=50.0 -XX:MaxRAMPercentage=50.0
```

🔍 예시 시나리오 C: 컨테이너 설정 누락 시 `MaxRAMPercentage` 가 무의미

`JAVA_TOOL_OPTIONS=-XX:MaxRAMPercentage=70.0`

문제:
Pod spec에 `memory.limit` 미설정 → JVM은 시스템 전체 메모리를 기준으로 동작

✓ **해결**:

```
resources:
  limits:
    memory: "512Mi"
```

🧷 GC 옵션 연관 예시

옵션	의미	연계 추천
`-XX:+UseG1GC`	G1 GC 사용	자동 pause 시간 조절 옵션 (`-XX:MaxGCPauseMillis`)
`-XX:+PrintGCDetails`	GC 로그 출력	반드시 stdout 로깅 수집과 함께 조합
`-Xlog:gc*`	Java 9+ GC 로그 표준 방식	G1GC와 함께 GC 타이밍 분석 가능

📚 결론: 연관 설정 시 고려할 체크리스트

- ✓ `MaxRAMPercentage` ⇒ 반드시 `memory.limit` 필요
- ✓ `-Xmx` 만 설정하지 말고 `-Xms` 도 명시
- ✓ 퍼센트 옵션은 Initial / Max / Min 조합으로 구성
- ✓ GC 전략은 heap 설정 및 서비스 특성에 따라 연계 설정
- ✓ Spring Boot 사용 시 `/actuator/health` , `/metrics` 와 probe를 함께 구성

✓ 참고 문서

- 📑 OpenJDK JEP 346: Container-Aware Heap
- 🔧 Red Hat – OpenJDK Container Memory Tuning
- 📊 Datadog – Java on Containers
- 📚 Kubernetes 공식 문서 (리소스 제한)

📑 최신 공식 & 실무 자료

1. ❤ Red Hat (2025-06-26)

"Red Hat build of OpenJDK container awareness for Kubernetes tuning"

- OpenJDK에서 `-XX:MaxRAMPercentage`, 컨테이너 메모리 제한과의 연동 방식 설명
- 매우 최신(2025년 6월) 자료
  reddit.com+3pauldally.medium.com+3community.amperecomputing.com+3stackoverflow.com+10developers.redhat.com+10datadoghq.com+10

---

2. 🌐 Datadog 블로그 (2025-03-xx)

"Java on containers: a guide to efficient deployment"

- JVM 힙 크기 설정, 컨테이너 인식 JVM, GC 튜닝, Startup/Liveness/Readiness probe 등 포괄적 가이드
  reddit.com+2datadoghq.com+2community.amperecomputing.com+2

---

3. 🔧 Medium – Paul Dally (≈2021년)

"Container-aware Java Heap Configuration"

- OpenJDK 기본 힙 설정 규칙 소개 (container limit에 따른 힙 % 자동 계산)
- `-XX:MaxRAMPercentage` 활용 추천
  github.com+14pauldally.medium.com+14developers.redhat.com+14

---

4. 🛠 Red Hat (2023-03-07)

"Overhauling memory tuning in OpenJDK containers updates"

- `-XX:MaxRAMPercentage` 환경 변수(`JAVA_MAX_MEM_RATIO`) 통해 JVM 컨테이너 연계 힙 설정 방식 설명
  reddit.com+4developers.redhat.com+4reddit.com+4

---

5. StackOverflow & Reddit 토론 (최신 사례)

- `-XX:InitialRAMPercentage`, `MaxRAMPercentage`, `MinRAMPercentage` 활용 팁
- 설정 시 `requests = limits`, QoS 보장, 안정적 limit 설정 권장 논의

> *"You can set JVM initial and max ram percentages … introduced in OpenJDK 10"*
> cloudnativenow.com+11reddit.com+11docs.cloudbees.com+11

정리 요약

- **컨테이너 인식 JVM**: JDK 10 이상에서 `-XX:MaxRAMPercentage` 등 자동 조정 기능 기본 제공
- **권장 방식**: `-Xmx` 대신 `-XX:MaxRAMPercentage`, 초기/최소 RAM % 옵션 활용 (`Initial`, `Min`)
- **리미트 전략**: `requests = limits` 설정 → Guaranteed QoS 보장, OOM 방지
- **실무 적용**: Datadog 가이드를 통해 GC와 probe 연동까지 총체적 튜닝 가이드 적용 가능
- **최신 Red Hat 방향**: 별도 스크립트 없이 JVM 자체 기능 사용, 컨테이너 인식 최적화

[서정원] 20. Software Architecture 핵심 > JVM 구조 및 동작 특성

📚 KICE 소프트웨어 아키텍처 문제 20번 스터디 정리

🎯 문제 핵심

k8s 환경에서 Java Heap 메모리 설정 방식의 이해

🔔 1. JVM 메모리 설정 방식 비교

전통적 방식 (예시 1)

yaml

```yaml
env:
- name: JAVA_OPTS
  value: "-Xms 1g -Xmx 1g"
```

- **고정 크기** 방식
- **문제점**: 컨테이너 전체 메모리(1Gi)와 힙(1g)이 거의 동일 → **OOMKilled 위험**

최신 퍼센티지 방식 (예시 2)

yaml

```yaml
env:
- name: JAVA_OPTS
  value: "-XX:InitialRAMPercentage=30.0 -XX:MinRAMPercentage=50.0
-XX:MaxRAMPercentage=80.0"
```

- **동적 퍼센티지** 기반
- **장점**: 컨테이너 메모리에 맞춰 자동 조정

🔍 2. 핵심 포인트: MinRAMPercentage의 함정

MinRAMPercentage 동작 조건

`MinRAMPercentage는 전체 메모리가 250MB 미만일 때만 작동!`

`1Gi = 1024MB > 250MB`
`→ MinRAMPercentage=50.0은 무효!`

실제 동작 (1Gi 메모리 기준)

`InitialRAMPercentage=30%` → 약 300MB (초기 힙)
`MaxRAMPercentage=80%` → 약 800MB (최대 힙)
`MinRAMPercentage=50%` → 무효 (조건 불충족) ✖

정답 ③이 틀린 이유: "최소한 500MB 유지"라고 했지만, MinRAMPercentage가 작동하지 않아 500MB 최소값이 보장되지 않음!

⚡ 3. 실무 적용: 메모리 확장 오버헤드

동적 확장의 문제

bash

```
초기 30% → 사용량 증가 → 50% 확장 → 80% 확장
        ↑오버헤드       ↑오버헤드    ↑오버헤드
```

확장 오버헤드 종류

1. **메모리 할당 지연**
2. **추가 GC 발생**
3. **메모리 단편화**

해결책: 초기값 = 최대값 설정

bash

```
-XX:InitialRAMPercentage=70.0 -XX:MaxRAMPercentage=70.0
```

트레이드오프: 성능 향상 vs 메모리 효율성

🛠 4. 실무 설정 개선 사례

Before (문제 있는 설정)

bash

```
-XX:InitialRAMPercentage=35.0
-XX:InitiatingHeapOccupancyPercent=35   # 동일한 값!
```

문제: 초기 힙(35%) = G1GC 임계값(35%) → **즉시 GC 발생**

After (개선된 설정)

bash

```
-XX:InitialRAMPercentage=50.0
-XX:InitiatingHeapOccupancyPercent=50   # 조화로운 설정
-XX:MaxRAMPercentage=70.0
```

📊 5. Java 버전별 차이점

Java 8 vs Java 17

bash

```
# Java 8 (실험적 기능)
-XX:+UnlockExperimentalVMOptions  # 필요
-XX:+UseStringDeduplication

# Java 17 (안정화된 기능)
-XX:+UseStringDeduplication       # 바로 사용 가능
```

🎯 6. 핵심 Takeaway

1. **MinRAMPercentage는 250MB 미만에서만 작동**
2. **InitialRAMPercentage ≠ InitiatingHeapOccupancyPercent 주의**
3. **컨테이너 환경에서는 70-75% MaxRAMPercentage 권장**
4. **성능 vs 효율성 트레이드오프 고려**
5. **Java 17에서는 UnlockExperimentalVMOptions 불필요**

💻 7. 권장 템플릿

안정성 중심 (운영 권장)

bash

```
ENV JAVA_OPTS="-XX:+UseG1GC \
 -XX:MaxRAMPercentage=70.0 \
 -XX:InitialRAMPercentage=50.0 \
 -XX:MaxGCPauseMillis=200 \
 -XX:InitiatingHeapOccupancyPercent=50 \
 -XX:+UseStringDeduplication \
 -Duser.timezone=Asia/Seoul"
```

성능 중심 (레이턴시 민감)

bash

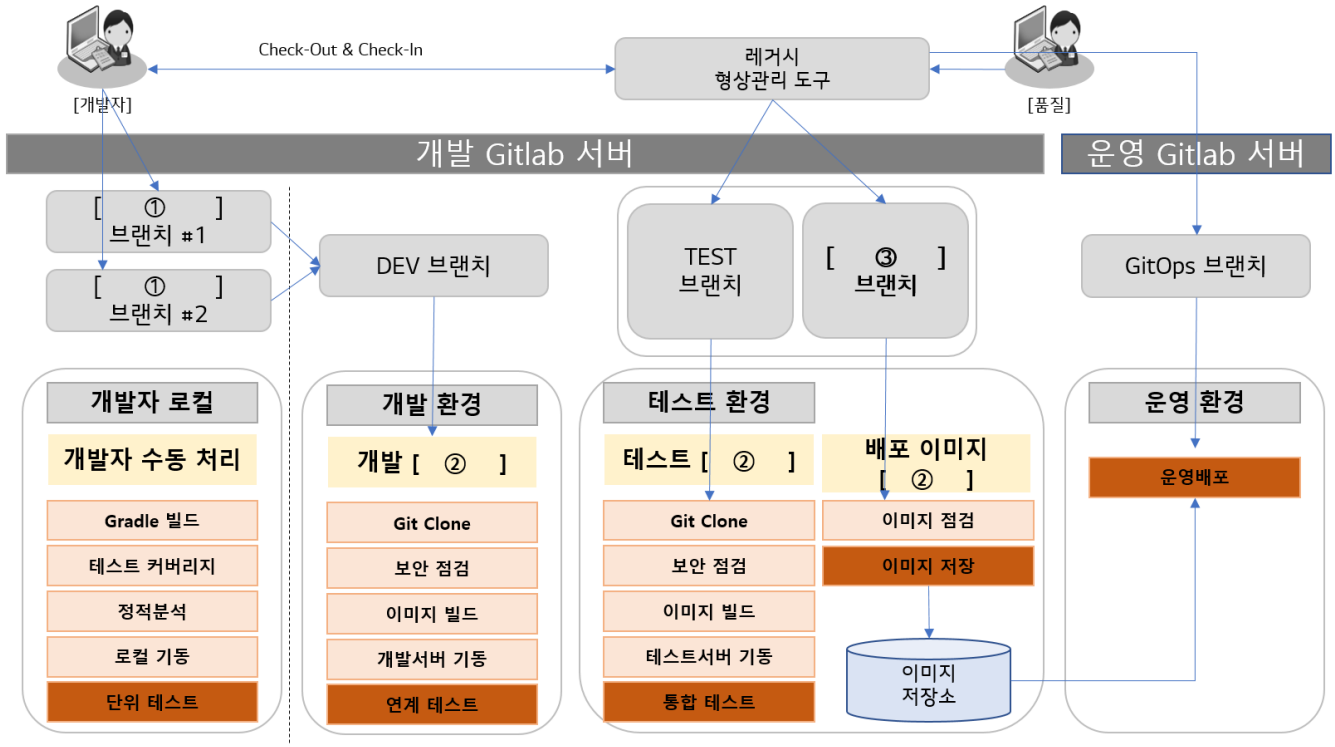
```
ENV JAVA_OPTS="-XX:+UseG1GC \
 -XX:MaxRAMPercentage=70.0 \
 -XX:InitialRAMPercentage=70.0 \   # 동일하게!
```

```
-XX:MaxGCPauseMillis=100 \
-XX:InitiatingHeapOccupancyPercent=45"
```

🔖 이 문제는 최신 Java의 컨테이너 인식 기능과 메모리 옵션의 조건부 동작을 정확히 이해하고 있는지 묻는 고난도 문제였습니다!

21. 신기술 > Cloud Service

문항 21) 아래 그림은 Cloud 환경의 컨테이너 기반 프로젝트를 Legacy 형상관리 도구와 연계하여 CI/CD를 적용한 사례이다. 아래 그림에서 빈칸에 맞는 단어를 보기에서 고르시오. [4점]



[보기]
Release, Master, Feature, 파이프라인, Package Registry, 컴파일, 라이브러리

답안 : (1), (2), (3) (올바른 단어를 순서대로 입력하시오)

(정답) Feature,파이프라인,Master
(해설) 컨테이너 프로젝트의 CICD 기본 프로세스
(배점)부분점수 있음
- 2개 : 2점
- 3개 : 4점
(난이도) 하
- 문제유형: 단답형
- 출제영역: 대분류>소분류 : 신기술 > Cloud Service
- 출제구분: 신규
- 문제 제목: 컨테이너 프로젝트의 CICD 기본 프로세스
- 출제 의도: 컨테이너 프로젝트의 CICD 기본 프로세스 이해도

✓ 정답

Feature, 파이프라인, Master

해설

번호	설명	정답
①	개발자가 브랜치에서 기능 개발 시작 → 통상적으로 기능 개발용 브랜치 명칭	**Feature**
②	Git 저장소에서 코드 변경 → CI/CD 프로세스를 통해 빌드/배포 흐름 구성	**파이프라인**
③	최종 Merge 및 운영 배포 대상으로 관리되는 브랜치	**Master**

🔧 실무 기준 명칭 예시 (Git 기반)

브랜치 종류	용도
`feature/*`	기능 개발 단위 브랜치
`develop`	통합 테스트용
`release/*`	QA, UAT 단계
`master`	운영 반영 대상
`hotfix/*`	운영 이슈 긴급 수정

CI/CD 도구(GitLab CI, Jenkins 등)는 Git hook 또는 merge 이벤트를 통해 파이프라인을 구동하며, Feature → Merge → Master 반영 → 이미지 빌드 및 배포 흐름으로 구성됨.

Git Branch 전략들

1. gitflow

- gitflow에는 5가지 브랜치가 존재

  - **master** : 기준이 되는 브랜치로 제품을 배포하는 브랜치
  - **develop** : 개발 브랜치로 개발자들이 이 브랜치를 기준으로 각자 작업한 기능들을 Merge
  - **feature** : 단위 기능을 개발하는 브랜치로 기능 개발이 완료되면 develop 브랜치에 Merge
  - **release** : 배포를 위해 master 브랜치로 보내기 전에 먼저 QA(품질검사)를 하기위한 브랜치
  - **hotfix** : master 브랜치로 배포를 했는데 버그가 생겼을 때 긴급 수정하는 브랜치

- master와 develop가 중요한 매인 브랜치이고 나머지는 필요에 의해서 운영하는 브랜치이다.
- branch를 merge할 때 항상 -no-ff 옵션을 붙여 branch에 대한 기록이 사라지는 것을 방지하는 것을 원칙으로 한다.

  gitflow 과정
  참고: 우아한형제들 기술블로그 (우린 Git-flow를 사용하고 있어요)

  - master 브랜치에서 develop 브랜치를 분기합니다.
  - 개발자들은 develop 브랜치에 자유롭게 커밋을 합니다.
  - 기능 구현이 있는 경우 develop 브랜치에서 feature-* 브랜치를 분기합니다.
  - 배포를 준비하기 위해 develop 브랜치에서 release-* 브랜치를 분기합니다.
  - 테스트를 진행하면서 발생하는 버그 수정은 release-* 브랜치에 직접 반영합니다.
  - 테스트가 완료되면 release 브랜치를 master와 develop에 merge합니다.

2. github flow

- Git-flow가 Github에서 사용하기에는 복잡하다고 나온 브랜치 전략이다.
- hotfix 브랜치나 feature 브랜치를 구분하지 않는다. 다만 우선순위가 다를 뿐
- 수시로 배포가 일어나며, CI와 배포가 자동화되어있는 프로젝트에 유용

  사용법

  a. master 브랜치는 어떤 때든 배포가 가능하다
  - master 브랜치는 항상 최신 상태며, stable 상태로 product에 배포되는 브랜치
  - 이 브랜치에 대해서는 엄격한 role과 함께 사용한다

- merge하기 전에 충분히 테스트를 해야한다. 테스트는 로컬에서 하는 것이 아니라 브랜치를 push 하고 Jenkins로 테스트 한다

a. master에서 새로운일을 시작하기 위해 브랜치를 만든다면, 이름을 명확히 작성하자

- 브랜치는 항상 master 브랜치에서 만든다
- Git-flow와는 다르게 feature 브랜치나 develop 브랜치가 존재하지 않음
- 새로운 기능을 추가하거나, 버그를 해결하기 위한 브랜치 이름은 자세하게 어떤 일을 하고 있는지에 대해서 작성해주도록 하자
- 커밋메시지를 명확하게 작성하자

a. 원격지 브랜치로 수시로 push 하자

- Git-flow와 상반되는 방식
- 항상 원격지에 자신이 하고 있는 일들을 올려 다른 사람들도 확인할 수 있도록 해준다
- 이는 하드웨어에 문제가 발생해 작업하던 부분이 없어지더라도, 원격지에 있는 소스를 받아서 작업할 수 있도록 해준다

a. 피드백이나 도움이 필요할 때, 그리고 merge 준비가 완료되었을 때는 pull request를 생성한다

- pull request는 코드 리뷰를 도와주는 시스템
- 이것을 이용해 자신의 코드를 공유하고, 리뷰받자
- merge 준비가 완료되었다면 master 브랜치로 반영을 요구하자

a. 기능에 대한 리뷰와 논의가 끝난 후 master로 merge한다

- 곧장 product로 반영이될 기능이므로, 이해관계가 연결된 사람들과 충분한 논의 이후 반영하도록 한다
- 물론 CI도 통과해야한다!

a. master로 merge되고 push 되었을 때는, 즉시 배포되어야한다

- GitHub-flow의 핵심
- master로 merge가 일어나면 자동으로 배포가 되도록 설정해놓는다

3. gitlab flow

- Gitlab에는 Production 브랜치가 있는데, 이는 Gitflow의 Master브랜치역할과 같다.
- Gitlab flow의 Master브랜치는 Production 브랜치로 병합한다.
- production 브랜치는 오직 배포만을 담당한다.

- pre-production 브랜치는 production 브랜치로 결과를 넘기기 전에 테스트를 수행하는 브랜치이다.
- Production브랜치에서 릴리즈된 코드가 항상 프로젝트의 최신버전 상태를 유지해야할 필요가 없다는 장점
- 복잡한 Gitflow와 너무 간단한 Github의 절충안

  > - master 브랜치는 production 브랜치
  > - production브랜치는 master 이상 권한만 push 가능
  > - developer 권한 사용자는 master 브랜치에서 신규 브랜치를 추가
  > - 신규 브랜치에서 소스를 commit 하고 push
  > - merge request를 생성하여 master 브랜치로 merge 요청
  > - master 권한 사용자는 developer 사용자와 함께 리뷰 진행 후 master 브랜치로 merge
  > - 테스트가 필요하다면 master에서 procution 브랜치로 merge하기 전에 pre-production 브랜치에서 테스트

4. Fork와 Pull Request

- 규모가 있는 개발을 할 경우 브랜치 보다는 Fork와 Pull requests를 활용하여 구현을 한다.
- Fork는 브랜치와 비슷하지만 프로젝트를 통째로 외부로 복제해서 개발을 하는 방식이다.
- 개발을 해서 브랜치처럼 Merge를 바로 하는 것이 아니라 Pull requests로 원 프로젝트 관리자에서 머지 요청을 보내면 원 프로젝트 관리자가 Pull requests된 코드를 보고 적절하다 싶으면 그때 그 기능을 붙히는 식으로 개발을 진행한다.

✓ 1. 브랜치 전략 비교 표

구분	GitFlow	GitHub Flow	GitLab Flow
기본 브랜치	`master`, `develop`, `feature/*`, `release/*`, `hotfix/*`	`main`, `feature/*`	`main`, `feature/*`, `staging`, `production`
복잡도	**높음** – 브랜치 종류 다양	**낮음** – 단순한 단일 흐름	**중간** – 배포 환경 반영, 이슈 중심 연계
주요 목적	대규모 릴리즈 중심 개발	빠른 배포 / 지속적 통합	이슈 기반 + 운영 환경 배포를 함께 고려

구분	GitFlow	GitHub Flow	GitLab Flow
릴리즈 방식	`release/*` 브랜치에서 QA 후 `master` 병합	`main`에 머지 후 바로 배포	`main → staging → production` 단계 반영 가능
핫픽스 처리	`hotfix/*` 브랜치 분리하여 `develop`, `master`에 병합	`main`에서 바로 핫픽스	`hotfix/*` 병행 가능, 환경 기반 자동 배포 가능
CI/CD 연계	릴리즈 브랜치 또는 태그 기준 배포	`main` 머지 시 자동 배포	브랜치, 태그, 환경 기준 자유롭게 트리거 구성 가능
장점	릴리즈 관리 철저, QA 선반영 용이	단순함, 빠른 배포	실배포 환경 연계, GitLab 기능(CI/MR) 최적화
단점	브랜치 관리 복잡, 병합 충돌 잦음	테스트 자동화 미흡 시 위험	명확한 가이드 없으면 혼란 가능
적합 대상	대규모 기업형 프로젝트, 정기 릴리즈 시스템	스타트업, 빠른 배포, 단일 환경 서비스	중대형 시스템, 다양한 환경 운영 프로젝트

✓ 2. 전략별 요약 설명

◆ GitFlow (Vincent Driessen, 2010)

- **릴리즈 중심 전략**
- `develop` 브랜치에서 기능 개발 → `release` → `master` 병합
- `hotfix` 브랜치로 운영 이슈 신속 처리
- **특징**: 릴리즈 사이클이 명확한 경우 유리, 브랜치 많음

◆ GitHub Flow

- **경량화된 전략** (CI/CD와 병행 전제)
- `main`에서 분기한 `feature` 브랜치에서 작업 후 Pull Request → Merge → Deploy
- **특징**: 단순, 빠른 배포에 유리. QA는 자동화로 대체

◆ GitLab Flow

- **GitHub Flow + 환경 기반 배포 + 이슈 연동**
- GitHub Flow처럼 기능 브랜치 기반이지만, `production`, `staging` 등의 **배포 환경 브랜치**를 병행 운영
- **이슈 단위 브랜치 → Merge Request → 환경 반영 자동화** 구성
- **특징**: 실 배포 환경을 브랜치로 관리하며 GitLab CI/CD와 연계 용이

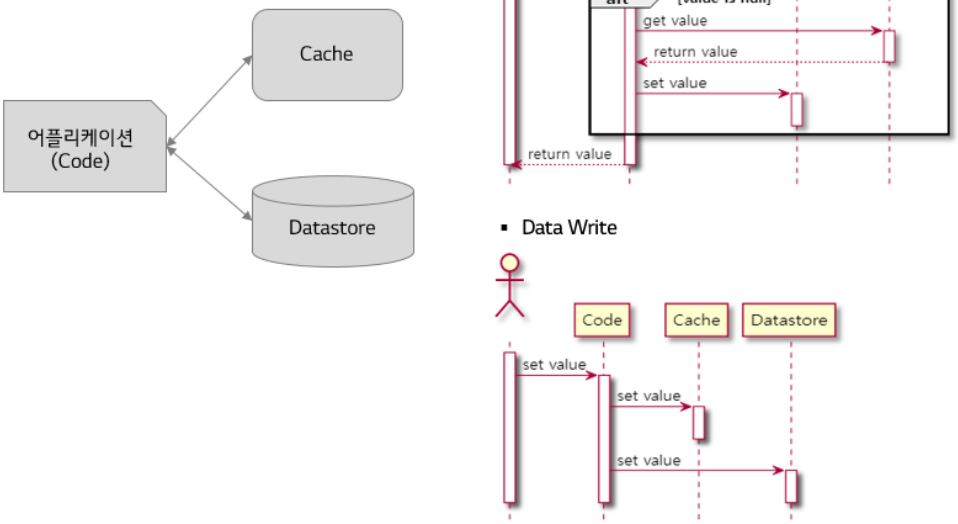
✓ 실무 적용 팁

상황	추천 전략
릴리즈 주기가 명확한 엔터프라이즈 시스템	**GitFlow**
배포가 자주 필요한 스타트업 서비스	**GitHub Flow**
다양한 운영 환경이 존재하거나 GitLab 기반으로 운영	**GitLab Flow**

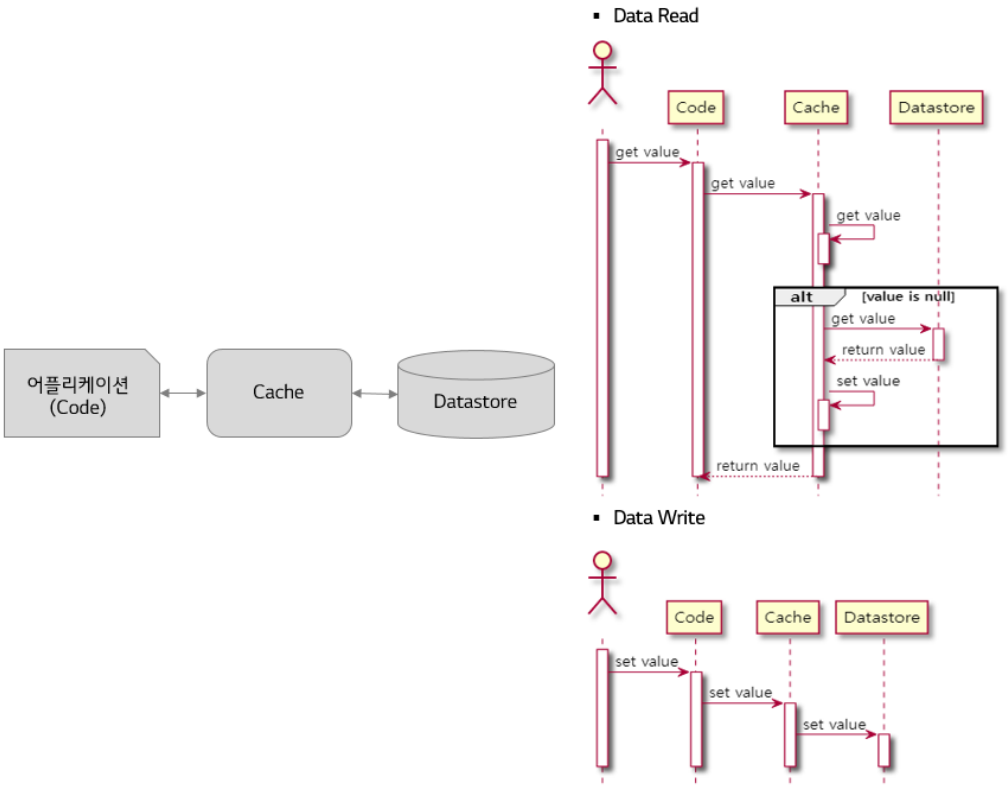
22. Software Architecture 환경 > 캐시 서버 및 검색 엔진

문항 22) 대규모 웹 애플리케이션 구축 시 성능을 향상시키기 위하여 어플리케이션 단에서 데이터 캐시를 사용하고자 한다.
프레임워크 담당자인 A 책임이 아래에 제시된 두 가지 캐싱 디자인 패턴에 대해 이해하고 있는 사항 중 틀린 것을 고르시오. [3점]

[1. Cache-aside pattern]



[2. Inline-cache pattern]



① 읽고자 하는 데이터가 캐시 서버에 없는 경우 1)번 방식은 어플리케이션이 직접 데이터 저장소로부터 데이터를 조회하고 캐시 데이터 업데이트도 수행한다.
② 읽고자 하는 데이터가 캐시 서버에 없는 경우 2)번 방식은 캐시 서버가 데이터 저장소로부터 데이터를 조회한 후 결과를 캐시에 업데이트한 후 그 결과를 어플리케이션에 전달해주는 방식을 사용한다.
③ 데이터 읽기의 경우 CDN은 1)번과 같은 방식으로 동작하며, Redis나 Memcached는 2)번과 같은 방식으로 동작한다.
④ 데이터 쓰기 과정에서 오류가 발생할 경우 1)번 방식의 경우에는 어플리케이션에서 데이터 무결성을 유지하기 위한 예외 처리 로직을 수행해야 한다.
⑤ 1)번 방식과 2)번 방식 모두 지정된 기간 동안 캐싱 데이터에 대한 접근이 발생하지 않으면 데이터를 무효화하고 캐시를 삭제하는 전략이 필요하다.

(정답)3
(해설)데이터 읽기 시 Redis와 Memcached는 1)번 방식과 같이 동작하며, CDN은 2)번 방식과 같이 동작한다.
(배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 환경 > 캐시 서버 및 검색 엔진
- 문제 제목: 캐싱 메커니즘
- 출제 의도: 캐싱 메커니즘에 대한 이해

✓ 정답: ③

해설

보기	설명
①	Cache-Aside 패턴의 전형적 동작. 맞는 설명
②	Inline 캐시는 프록시형 캐시로, DB 조회와 캐시 갱신을 자동 수행. 맞음
③ ✖	실제로는 반대임:

- **Redis/Memcached**: 1번과 같이 **Cache-Aside 방식**
- **CDN**: 프록시형 **Inline Cache 방식** → 캐시가 중개자 역할 수행 |
| ④ | Cache-Aside는 캐시, DB 모두 애플리케이션이 제어하므로 예외 처리 필요 |

⑤	TTL 기반 무효화 전략은 모든 캐시 패턴에서 공통적으로 필요

캐시 패턴 설명

1. Cache-Aside Pattern

- 애플리케이션이 먼저 캐시를 조회
- 캐시에 없으면 DB에서 조회 → 캐시에 업데이트
- **캐시 Miss 대응, 읽기 주도 구조에 적합**

2. Inline Cache Pattern

- 애플리케이션이 캐시를 직접 조회하지 않음
- 캐시 서버가 DB와 연계해 자동으로 캐시 갱신
- **Proxy 또는 CDN처럼 동작**, 응답에 직접 개입

📚 관련 개념 정리

항목	Cache-Aside	Inline Cache
캐시 접근 주체	애플리케이션	캐시 시스템
캐시 Miss 처리	애플리케이션이 DB 조회 → 캐시 업데이트	캐시 시스템이 백엔드로부터 자동 조회
장점	제어 유연성, 복잡한 로직 대응	빠른 응답, 프록시 기반 간편 구조
단점	캐시 일관성, 코드 복잡성	실시간 쓰기/삭제 반영 어려움
적용 예	Redis, Memcached	CDN, API Gateway (Edge Cache)

✓ 1. 읽기 중심 캐시 패턴 (Read Through Caching)

◆ 1-1. Cache-Aside (Lazy Load)

> *애플리케이션이 먼저 캐시를 조회하고, 없으면 DB에서 읽어 캐시에 저장*

`[App] → [Cache] (Miss) → [DB] → [Cache 저장] → 응답`

📌 특징

- **가장 일반적**
- 캐시 조회/업데이트 로직은 **애플리케이션이 직접 수행**
- 예: Redis, Memcached

✓ 장점

- 제어 유연성 높음
- 필요한 데이터만 캐시에 저장 → 메모리 효율적

✖ 단점

- 캐시-DB 불일치 가능성
- 코드 복잡도 증가

◆ 1-2. Inline Cache (Proxy Cache)

> *애플리케이션이 캐시를 직접 조회하지 않고, 캐시가 요청을 중개함*

`[App] → [CDN / Proxy Cache] → [DB 또는 Origin 서버]`

📌 특징

- 주로 **CDN, Reverse Proxy**에서 사용
- 캐시가 미들웨어처럼 동작

✓ 장점

- 구현 단순, 코드 수정 없음
- 응답 속도 매우 빠름 (Static Resource, API 응답)

✖ 단점

- 캐시 무효화/갱신 제어 어려움
- 쓰기 요청에 비효율

✓ 2. 쓰기 중심 캐시 패턴 (Write Caching)

◆ 2-1. Write-Through

> *쓰기 요청 시 **캐시와 DB에 동시에 저장***

`[App] → [Cache + DB]`

✓ 장점

- 캐시 항상 최신 상태
- 읽기 성능 우수

✗ 단점

- 쓰기 성능 저하 가능
- 장애 시 캐시-DB 동시 실패 가능성

◆ 2-2. Write-Behind / Write-Back

> *캐시에 먼저 쓰고, 일정 시간/조건 후 **비동기적으로 DB에 저장***

`[App] → [Cache] → (비동기) → [DB]`

✓ 장점

- 높은 쓰기 처리량, 응답 속도 빠름

✗ 단점

- 캐시 장애 시 **데이터 유실 가능**
- DB 일관성 유지가 어려움

◆ 2-3. Refresh-Ahead

> *TTL 만료 전에 백그라운드로 **미리 캐시 갱신***

```
[App] ← [Cache]
    ↳ (백그라운드) → [DB]
```

✓ 장점

- 캐시 Miss 감소
- 안정적인 응답속도 유지

✖ 단점

- 불필요한 조회 발생 가능

✓ 3. 패턴별 비교표

패턴	조회 주체	쓰기 시점	일관성 유지	성능	코드 복잡도
Cache-Aside	앱	앱이 직접 DB 갱신	낮음	좋음	높음
Inline Cache	캐시	캐시 미제공	낮음	매우 좋음	낮음
Write-Through	앱	동시에 캐시+DB	높음	낮음	중간
Write-Behind	앱	캐시 → 지연 DB 저장	낮음	높음	높음
Refresh-Ahead	캐시	TTL 만료 전 갱신	중간	안정적	중간

✓ 적용 예시

시스템 종류	추천 패턴
대규모 조회 시스템	Cache-Aside + TTL
API Gateway / CDN	Inline Cache
실시간 대기화면/주문현황	Refresh-Ahead
금융권 실시간 통계	Write-Through
쇼핑몰 장바구니/이벤트	Write-Behind

23. 신기술>Cloud Service

문항 23) 컨테이너 관리도구인 쿠버네티스(k8s)의 명령어를 수행한 결과이다. 결과를 바르게 해석한 것을 고르시오. 명령어는 모든 권한이 있는 admin이 수행하였다. [3점]

```
$ kubectl get pods
NAME                      READY     STATUS             RESTARTS     AGE
tctpod-9viejd6fh-khrkv     1/2       CrashLoopBackOff    0            4m
```

① 컨테이너 이름은 tctpod-9viejd6fh-khrkv이다.
② 1개의 컨테이너가 정상적으로 기동 되었다.
③ tctpod-9viejd6fh-khrkv 는 4개월 전에 시작되었다.
④ 현재 namespace의 pod는 2개이다.
⑤ tctpod-9viejd6fh-khrkv는 CrashLoopBackOff가 발생하여 여러 번 재시작 되었다.

(정답)2
(해설)Pod의 정보를 나타내며 READY항목은 POD가 가지고 있는 container에 대한 상태정보이다. 정상기동상태 container수 / 전체 container 수 형식으로 나타낸다. (배점)3
(난이도)하
- 문제유형: 선다형
- 출제영역: 신기술>Cloud Service
- 문제 제목: k8s 기본 명령어
- 출제 의도: k8s의 기본 명령어인 pod 정보 확인 명령어 수행 결과를 보고 분석할 수 있다.

✓ 정답: ②

해설

🔍 출력 의미 분석

항목	값	설명
`READY`	`1/2`	총 2개의 컨테이너 중 **1개만 정상 기동**, 1개는 비정상
`STATUS`	`CrashLoopBackOff`	1개 컨테이너가 기동 직후 실패 반복 중

항목	값	설명
`RESTARTS`	`0`	아직 재시작 횟수 기록 없음 (설정에 따라 초기화 가능)
`AGE`	`4m`	생성된 지 4분 (4개월 아님)

✖ 오답 해설

보기	설명
①	**Pod 이름**이지, 컨테이너 이름은 아님
③	`4m` 은 **4분(min)**. `4mo` 또는 `120d` 가 4개월
④	명령어 결과상 **Pod 수는 1개** (다중 컨테이너 구성일 뿐)
⑤	`RESTARTS=0` 이므로 **CrashLoopBackOff가 발생 중이지만 아직 재시작은 기록 안 됨**

📚 관련 지식 정리

◆ CrashLoopBackOff

- 컨테이너가 **기동되자마자 예외 발생/종료**하고 반복적으로 재시작되는 상태
- 보통 **애플리케이션 예외, 환경변수 누락, 포트 충돌** 등 원인

◆ Pod READY 항목

- 형식: `<ready container 수>/<전체 container 수>`
- 예: `2/2` = 정상, `0/2` 또는 `1/2` = 일부 비정상

✓ 1. Kubernetes 리소스 조회 명령어 (kubectl get)

명령어	설명
`kubectl get pods`	현재 namespace의 모든 Pod 목록 조회
`kubectl get pods -n <namespace>`	특정 네임스페이스의 Pod 조회
`kubectl get svc`	서비스(Service) 목록 조회
`kubectl get deployments`	Deployment 목록 조회
`kubectl get nodes`	클러스터의 Node 목록 확인
`kubectl get events`	최근 이벤트 로그 확인 (Crash, Restart 등 추적)
`kubectl get all`	Pod, Service, Deployment 등 주요 리소스 전체 조회

✓ 2. 상세 정보 확인 명령어

명령어	설명
`kubectl describe pod <pod명>`	Pod의 상태, 이벤트, 스케줄 정보 등 상세 확인
`kubectl describe node <node명>`	노드의 상태, 자원, 스케줄된 Pod 정보 확인
`kubectl logs <pod명>`	기본 컨테이너의 로그 출력
`kubectl logs <pod명> -c <컨테이너명>`	멀티 컨테이너 Pod의 특정 컨테이너 로그
`kubectl exec -it <pod명> -- /bin/sh`	Pod 안으로 접속 (쉘 진입)

명령어	설명
`kubectl top pod`	Pod의 실시간 CPU/Memory 사용량
`kubectl top node`	Node의 자원 사용률 모니터링

✓ 3. Pod 상태 확인 주요 지표

필드	의미
`READY`	`x/y` 형식 → y개 중 x개 컨테이너가 정상 실행 중
`STATUS`	`Running`, `Pending`, `CrashLoopBackOff`, `Error`, `Completed` 등
`RESTARTS`	해당 컨테이너의 재시작 횟수
`AGE`	리소스 생성된 시간 (분, 시간, 일 단위)

주요 STATUS 설명

상태	의미 및 원인
`Running`	정상 실행 중
`Pending`	스케줄링은 되었으나 자원 부족 또는 이미지 Pull 대기 중
`CrashLoopBackOff`	애플리케이션이 반복적으로 Crash (환경 변수 누락, 포트 충돌 등)
`Completed`	Job 등 일회성 작업이 성공적으로 종료됨
`ImagePullBackOff`	컨테이너 이미지를 못 불러오는 상태 (이미지명, 권한 오류 등)

상태	의미 및 원인
`OOMKilled`	메모리 초과로 종료됨 (Out Of Memory)

✓ 4. 자주 사용하는 실전 진단 명령어

1. Pod 상태 확인
kubectl get pods -o wide

2. 이벤트 로그 확인
kubectl describe pod <pod명> | less

3. 로그 확인
kubectl logs <pod명> --tail=100

4. Pod에 직접 접속해서 진단
kubectl exec -it <pod명> -- /bin/sh

5. 실시간 리소스 확인
kubectl top pod

6. 네임스페이스별로 구분
kubectl get pods -n default
kubectl get pods -n kube-system

✓ 5. YAML 기반 리소스 확인

kubectl get pod <pod명> -o yaml
kubectl get deployment <이름> -o json

- 리소스 구성, 환경 변수, 볼륨 마운트, 포트, 주석 등 모두 확인 가능

✓ 1. Pod 상태 확인 명령어

📌 명령어

`kubectl get pods`

📄 예시 출력

```
NAME                     READY   STATUS             RESTARTS   AGE
myapp-76f8c5f9c7-q1vzs   1/1     Running            0          2d
nginx-7cbbdbb85c-mt6hp   1/1     CrashLoopBackOff   5          10m
job-batch-231231-abc123  0/1     Completed          0          5h
```

필드 해석

필드	설명
`NAME`	Pod 이름
`READY`	`x/y` → 총 y개의 컨테이너 중 x개가 정상 작동 중
`STATUS`	Pod의 전체 상태
`RESTARTS`	재시작 횟수 (Crash 시 누적 증가)
`AGE`	생성된 시간 (분, 시간, 일 단위)

✓ 2. Pod 상세 정보 확인

📌 명령어

```
kubectl describe pod <pod명>
```

📄 출력 예시 일부

```
Name:             myapp-76f8c5f9c7-q1vzs
Namespace:        default
Status:           Running
IP:               10.42.1.4
Containers:
```

```
  app:
    State:          Running
    Ready:          True
    Restart Count:  0
Events:
  Type     Reason     Message
  ----     ------     -------
  Normal   Scheduled  Successfully assigned default/myapp to node-1
  Normal   Pulled     Successfully pulled image
```

🔍 확인 항목

- `State`, `Ready`, `Restart Count` : 컨테이너 상태
- `Events` : Crash, OOMKilled, ImagePullBackOff 원인 확인 가능

✓ 3. 컨테이너 로그 확인

📌 명령어

kubectl logs <pod명>
kubectl logs <pod명> -c <컨테이너명>

📄 출력 예시

Started application on port 8080
Connected to MySQL
User login request received
Exception: DB timeout

🔍 활용

- 애플리케이션 오류, 연결 실패 등 로그 기반 분석 가능

✓ 4. Pod 내부 접속 (쉘 접속)

📌 명령어

```
kubectl exec -it <pod명> -- /bin/sh
```

📄 예시

inside pod
/ # ls
app.jar lib/ tmp/ etc/ proc/

✓ 5. 리소스 사용량 확인

📌 명령어

kubectl top pod
kubectl top node

📄 출력 예시

```
NAME                          CPU(cores)   MEMORY(bytes)
myapp-76f8c5f9c7-q1vzs        120m         220Mi
nginx-7cbbdbb85c-mt6hp        10m          50Mi
```

🔍 활용

- OOMKilled, CPU 부하 원인 분석
- JVM Heap 설정에 따른 메모리 사용 확인

✓ 6. 상태별 STATUS 설명

STATUS	의미	원인 및 조치
`Running`	정상 작동 중	컨테이너가 준비 완료됨
`Pending`	스케줄은 되었으나 실행되지 않음	이미지 Pull 지연, 노드 리소스 부족
`CrashLoopBackOff`	컨테이너가 기동 직후 Crash 반복	환경변수 누락, 포트 충돌, 애플리케이션 예외
`ImagePullBackOff`	이미지 다운로드 실패	이미지 경로 오타, 권한 오류
`Completed`	Job이 정상 종료됨	일회성 Job 성공
`Error`	컨테이너 종료 시 오류 발생	종료 코드 ≠ 0

✓ 7. 전체 리소스 확인

📌 명령어

```
kubectl get all
```

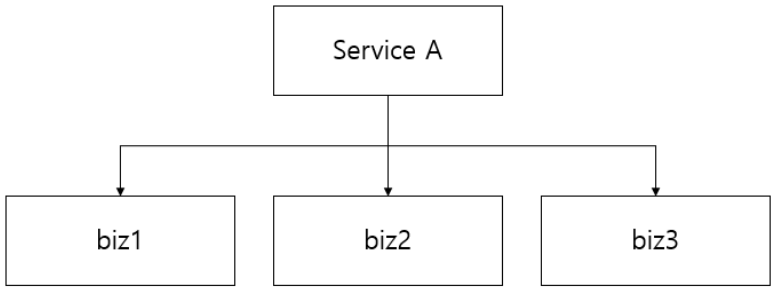
📄 예시 출력

```
NAME                    READY   STATUS    RESTARTS   AGE
pod/myapp-76f8c5f9c7-q1vzs     1/1     Running   0          2d
svc/myapp-service              ClusterIP   10.96.0.1
deployment.apps/myapp          1/1     1         2d
```

24. Software Architecture 운영/문제해결 > 로그/덤프 생성, 관리 및 분석

문항 24) 성능 시험 진행 중 단일 업무 테스트케이스에서 목표 성능을 도달하지 못하여 원인을 분석하고 있다.

[구성도]



[설명]

- Service A, biz1, biz2, biz3은 모두 서비스 단위이다.
- Service A는 업무 요건에 따라 biz1, biz2, biz3 서비스를 Apache HttpComponents를 이용하여 선택적으로 호출하는 구조이다.
- Service A의 응답시간에는 biz1, biz2, biz3내에서 처리하는 시간이 포함되어 있다.

[현상 및 APM 분석 결과]

- Service A에서 간헐적으로 응답시간 지연이 발생하고 있다.
- Service A의 응답시간 지연 구간은 biz1 서비스(http://biz1:9191)를 요청/응답하는 과정에서 발생하고 있다.
- 피호출자인 biz1 서비스 자체의 응답시간은 지연없이 빠르게 처리되고 있다.

위 분석 결과를 토대로 Service A의 간헐적 응답시간 지연 원인을 Http Connection Pool 병목으로 추정하고 아래 그림과 같이 HttpComponents 라이브러리의 로그 레벨을 DEBUG로 조정하여 디버깅 로그를 통해 Connection Pool의 상태를 확인하였다.

[HttpComponents 로그 레벨]
logging.level.org.apache.http.impl.conn=DEBUG

[HttpComponents 관련 Log4j 로그]

```
2021-04-25 23:12:11.632 DEBUG 4952 --- [io-9090-exec-58] h.i.c.PoolingHttpClientConnectionManager : C
route allocated: 10 of 10; total allocated: 10 of 20]
2021-04-25 23:12:11.632 DEBUG 4952 --- [io-9090-exec-58] h.i.c.DefaultManagedHttpClientConnection : h
2021-04-25 23:12:11.811 DEBUG 4952 --- [io-9090-exec-35] h.i.c.PoolingHttpClientConnectionManager : C
seconds
2021-04-25 23:12:11.811 DEBUG 4952 --- [io-9090-exec-35] h.i.c.DefaultManagedHttpClientConnection : h
2021-04-25 23:12:11.811 DEBUG 4952 --- [io-9090-exec-35] h.i.c.PoolingHttpClientConnectionManager : C
route allocated: 10 of 10; total allocated: 10 of 20]
```

58번 스레드와 35번 스레드가 커넥션을 획득 또는 반납하는 과정에서의 Connection Pool의 상태를 토대로 간헐적인 응답시간 지연이 발생하는 원인으로 추정되는 항목을 고르시오. [3점]

① biz1에 할당된 connection 수가 biz2, biz3 보다 작기 때문이다.
② biz1의 Socket Timeout이 biz2, biz3 보다 짧기 때문이다.
③ biz1에 할당된 connection 수가 부족하기 때문이다.
④ http-outgoing Thread 수가 적어서 Socket을 생성할 Thread가 부족하였기 때문이다.
⑤ biz2, biz3 보다 biz1로 거래가 더 많이 유입되었기 때문이다.

(정답) 3
(해설) biz1에는 10개의 max connection이 할당되었고, available 개수가 0인 로그가 있는 것으로 볼 때 connection 부족으로 인한 지연이 발생되었다고 추정할 수 있다. 나머지 예시들은 로그의 내용과는 무관하며 진위 여부를 판단할 수 없다.
(배점) 3
(난이도) 하
- 문제유형: 선다형
- 출제영역 대분류>소분류: Software Architecture 운영/문제해결 > 로그/덤프 생성, 관리 및 분석
- 문제 제목: HttpClient 로깅
- 출제 의도: HttpClient 커넥션풀 로깅을 위해서 설정하는 방법과 그 결과를 해석할 수 있는지 확인한다

✓ 정답: ③

해설

🔎 핵심 원인

- **Connection Pool 로그에** `available=0` **상태**가 발생
- 즉, **모든 커넥션이 점유 중이라 새 연결을 대기해야 하는 상황**

🔍 왜 ③이 맞는가?

- `biz1` 에 충분한 커넥션 수가 **미리 확보되지 않아 병목** 발생
- 커넥션 수 부족 → 커넥션 대기 → 응답 지연

✖ 오답 설명

보기	설명
①	모든 서비스에 동일 수 할당되어도, 요청량 편차로 병목 발생 가능

보기	설명
②	Timeout은 지연 원인이 아님 (짧을수록 더 빨리 실패함)
④	스레드 부족은 JVM 레벨에서 확인되어야 함, 본 로그와 무관
⑤	원인일 수는 있지만, **명확한 Pool 부족 증거**가 있음 → 직접 원인 아님

📚 관련 개념 정리

◆ HttpClient Connection Pool

- **DefaultMaxPerRoute**: 특정 주소(biz1 등)에 허용할 최대 커넥션 수
- **MaxTotal**: 전체 허용 커넥션 수
- 커넥션 풀 고갈 시, `lease connection` 대기 상태 → 응답 지연

🔧 예시 설정 (Spring 기반)

```
PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();
cm.setMaxTotal(200);
cm.setDefaultMaxPerRoute(50);
```

✓ 1. Apache HttpClient Connection Pool 관련 주요 옵션 설명

◆ 핵심 구성 클래스: `PoolingHttpClientConnectionManager`

`PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();`

설정 메서드	설명	예시
`setMaxTotal(int)`	전체 커넥션 수 제한	`cm.setMaxTotal(200);`

설정 메서드	설명	예시
`setDefaultMaxPerRoute(int)`	대상 호스트(도메인)별 커넥션 최대 수	`cm.setDefaultMaxPerRoute(50);`
`setMaxPerRoute(HttpRoute, int)`	특정 라우트에 대한 제한 설정	`cm.setMaxPerRoute(route, 100);`
`closeIdleConnections(long, TimeUnit)`	유휴 커넥션 정리 시간 설정	`cm.closeIdleConnections(30, TimeUnit.SECONDS);`
`closeExpiredConnections()`	만료된 커넥션 정리	주기적 호출 필요

✓ 2. HttpClient 연결 설정 옵션 (Builder 기준)

```
RequestConfig config = RequestConfig.custom()
    .setConnectTimeout(3000)       // TCP 연결 수립 타임아웃 (ms)
    .setSocketTimeout(5000)        // 데이터 수신 대기 타임아웃 (ms)
    .setConnectionRequestTimeout(2000) // Pool에서 커넥션 얻기 대기 시간
    .build();
```

옵션 명	설명
`ConnectTimeout`	TCP 연결 시도 제한 시간
`SocketTimeout`	소켓 읽기 제한 시간
`ConnectionRequestTimeout`	커넥션 풀에서 커넥션을 얻기까지 기다리는 시간

> ⚠ ***ConnectionRequestTimeout이 초과되면*** `Timeout waiting for connection from pool` *예외 발생*

✓ 3. Log4j 설정으로 커넥션 풀 상태 확인 (HttpClient 4.x 기준)

📌 A. Log4j 설정 예시 (log4j2.xml or log4j.properties)

log4j.properties

```
log4j.logger.org.apache.http=INFO
log4j.logger.org.apache.http.wire=ERROR
log4j.logger.org.apache.http.impl.conn=DEBUG
log4j.logger.org.apache.http.impl.client=DEBUG
```

- `impl.conn` : 커넥션 풀 관련 로깅
- `impl.client` : HttpClient 내부 상태

B. 출력 로그 예시

```
DEBUG o.a.h.i.conn.PoolingHttpClientConnectionManager - Connection request: [route: {}->http://biz1, state: null]
DEBUG o.a.h.i.conn.PoolingHttpClientConnectionManager - Connection leased: [id: 12][route: {}->http://biz1][total kept alive: 0; route allocated: 20 of 20; total allocated: 100 of 100]
DEBUG o.a.h.i.conn.PoolingHttpClientConnectionManager - Connection released: [id: 12][route: {}->http://biz1][total kept alive: 1; route allocated: 19 of 20; total allocated: 99 of 100]
```

✓ 4. 로그 항목 해석

로그 항목	의미
`Connection request:`	커넥션 풀에서 커넥션 요청 시작
`Connection leased:`	커넥션 풀에서 커넥션 할당 완료
`total allocated`	현재 사용 중인 커넥션 수 (전체)
`route allocated`	특정 대상 서버(biz1 등)에 할당된 커넥션 수

로그 항목	의미
`kept alive`	커넥션 유지된 수 (재사용 가능 커넥션 수)
`Connection released:`	응답 후 커넥션 반납 (KeepAlive 적용 시 유지됨)

✓ 5. 실전 팁

상황	조치
`route allocated: 20 of 20`, `Pending > 0`	→ **해당 호스트 커넥션 풀 고갈,** `setMaxPerRoute` **증가 필요**
`total allocated: 100 of 100`	→ **전체 풀 고갈,** `setMaxTotal` **증가 고려**
`ConnectionRequestTimeout` 예외 빈발	→ **풀 크기 증가 또는 커넥션 재사용 여부 확인 (KeepAliveHeader)**
유휴 커넥션 과다	`closeIdleConnections()` 주기적 호출 필요

25. 주관식 문제

서술형 1번 (10점)

서술형 문제는 장애상황을 주고 해당 장애가 왜 발생 했는가, 그에 대한 처리 방법, 처리에 따르는 Trade-off를 서술하시오.

1. A(클라이언트) -> B(중계서버) -> C(업무시스템) 호출간 B 시스템은 성능을 위해 병렬처리 중이다.
프로모션 기간 대비를 위해 병렬 쓰레드를 늘리려고 하나, OOME 이 발생하였고, 분석 결과 이는 기존에도 나던 오류였다.
OOME가 나는 이유와 이를 해소 할수 있는 방안 해소 시 발생되는 Trade-off를 서술 하시오

taskexecutor threadpool 7개로 설정되어 있으나 oom이 발생함

→ 정답 : 1)힙 메모리 부족 원인

✓ 채점 기준 (모범 답안 요약)

1. OOME 발생 원인

- 병렬로 생성된 **Thread마다 JVM Stack 메모리와 힙 객체를 소비**함
- thread pool이 작아도, 처리 지연으로 인해 큐에 작업(Task)이 **과도하게 누적**
- 결국 **Heap 메모리 부족으로 OOME 발생**

---

2. 기술적 방안 (예시)

- ✓ **ThreadPoolExecutor 설정 개선**
  - `corePoolSize`, `maxPoolSize`, `queueCapacity`를 적절히 조정하여 메모리 사용 제어
- ✓ **작업 단위 축소 및 객체 재사용**
  - 비동기 작업 내 불필요한 객체 생성을 줄이고, 외부 의존성을 정리
- ✓ **Back Pressure 적용 / Queue 제한**
  - 작업 큐를 제한하고, 초과 시 예외 처리 또는 CallerRunsPolicy 적용
- ✓ **메시지 큐 기반 처리 전환**
  - Kafka, RabbitMQ 등으로 메시지를 비동기 전송하여 메모리 부담 분산

---

3. Trade-off (예시)

방안	Trade-off
큐 제한	요청 유실/지연 가능성 증가
스레드 수 제한	Throughput 감소, 응답 속도 저하
메시지 큐 도입	시스템 복잡성 증가, 트랜잭션 관리 어려움
GC 최적화	일시적 응답 지연 또는 pause 시간 증가

서술형 2번 (5점)

아래 소스는 병렬 처리를 위해 CompleteableFuture API를 사용해서 비동기로 구현했는데 성능 개선 효과가 없었다. 원인과 조치 방안에 대해서 서술하시오

정답 :

- 원인 : get()으로 개발되어 성능 저하
- 조치 방안 : get() -> join() 사용으로 nonblocking 방식으로 변경

✓ 문제에 포함된 예시 코드 (추정 예시)

```
CompletableFuture<String> task1 = asyncService.callServiceA();
CompletableFuture<String> task2 = asyncService.callServiceB();

String result1 = task1.get();   // blocking 발생
String result2 = task2.get();   // blocking 발생

return result1 + result2;
```

해설 및 채점 기준

✓ 1. 동작하지 않는 이유

- `CompletableFuture` 는 **비동기적으로 처리되도록 설계**된 API지만,
  `get()` 호출은 내부적으로 **blocking 메서드**로서, 해당 결과가 완료될 때까지 **현재 스레드를 정지**시킨다.

- 따라서 `task1.get()` 호출 시 `task1` 의 완료를 **동기적으로 기다리게 되어**, `task2` 는 그 이후에야 실행되므로 **병렬 실행이 되지 않음**
- 이로 인해 병렬 처리의 성능 향상 효과가 **무력화됨**

> ⚠ `get()` = ***blocking***, *현재 스레드가 결과를 기다림 → 병렬 구조 무효화*

✔ 2. 조치 방안

✔ 방법 1: `join()` 으로 변경

```
String result1 = task1.join();  // non-blocking
String result2 = task2.join();
```

- `join()` 은 내부적으로 예외 처리를 감싼 **non-blocking 스타일의 API**로, 병렬 구조를 유지한 채 결과를 받음
- 특히 `.get()` 은 예외를 throws 하므로 try-catch 필요 → `join()` 은 `CompletionException` 으로 wrapping

✔ 방법 2: `CompletableFuture.allOf()` 활용

```
CompletableFuture<String> task1 = asyncService.callServiceA();
CompletableFuture<String> task2 = asyncService.callServiceB();

CompletableFuture<Void> all = CompletableFuture.allOf(task1, task2);

all.join();  // 모든 작업 완료될 때까지 대기 (non-blocking)

String result1 = task1.join();
String result2 = task2.join();
```

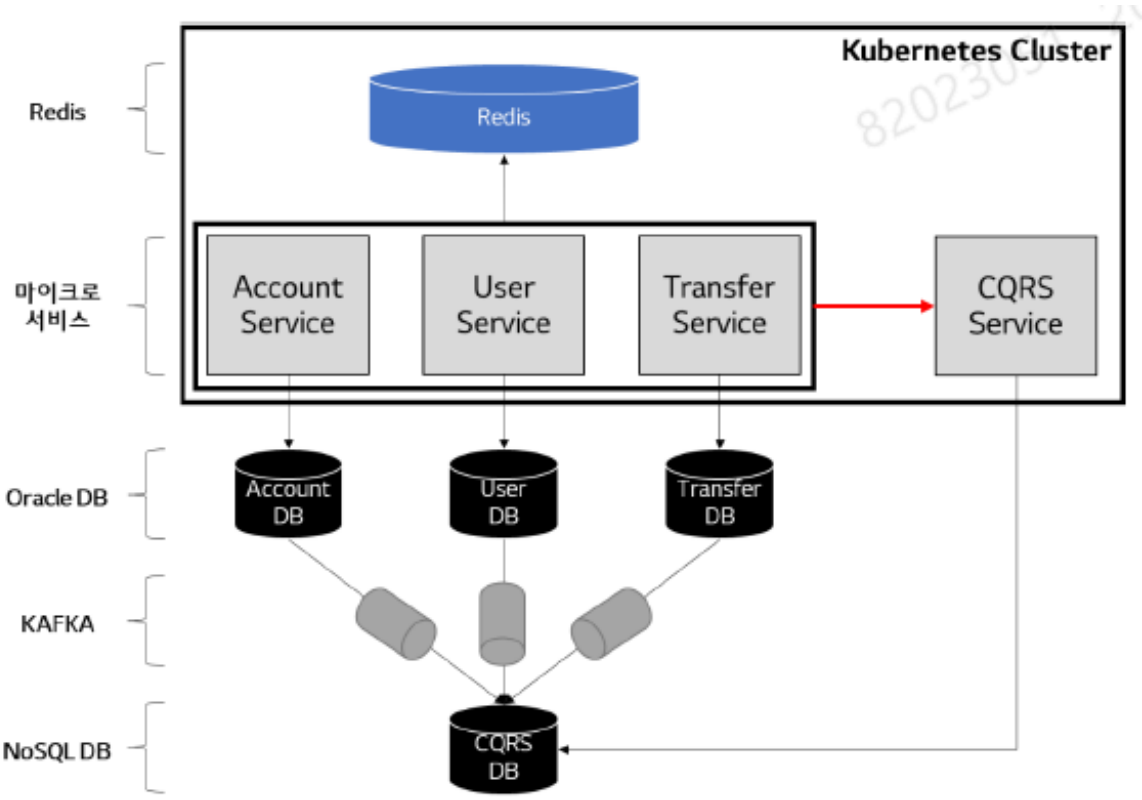
- 여러 작업의 **동시 완료를 대기**할 수 있으며, **성능 최적화 및 흐름 제어**에 유리함

📌 모범답안 요약

> *병렬 처리를 위해 `CompletableFuture` 를 사용했으나, 중간에 `get()` 을 사용하여 결과를 동기적으로 기다리게 되어 실제로는 **순차 실행**됨.*
> *이는 병렬 처리 구조를 무력화하며 성능 저하를 유발함.*
> *이를 해결하기 위해 `get()` 대신 `join()` 을 사용하여 **non-blocking 방식으로 결과를 대기**하게 하면 병렬 구조를 유지하며 성능을 향상시킬 수 있다.*
> *또한 `CompletableFuture.allOf()` 를 활용하여 전체 작업 완료 후 결과를 수집하는 방식도 적절하다.*

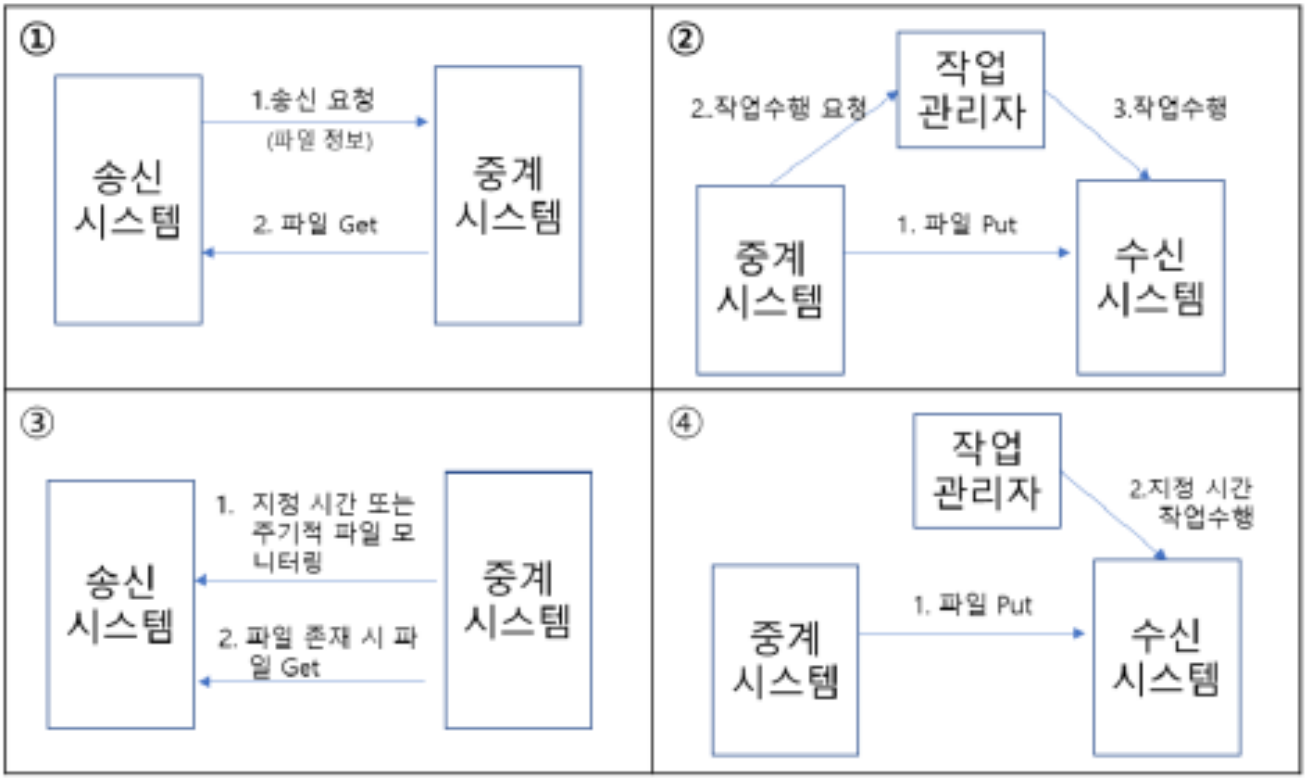
99. 22년도 기출문제 샘플

문항 1) MSA 프로젝트에서 CQRS DB로 복제하는 기준으로 적합한 것을 보기에서 모두 고르시오. [4점]
(CQRS DB는 MongoDB를 사용하며, 마이크로서비스 분산DB는 모두 Oracle DB를 사용한다.)



<보기>
ㄱ. Account Service의 트랜잭션 처리비율이 높은 대량조회서비스를 대상으로 CQRS DB에 복제하여 성능을 높인다.
ㄴ. User Service에서 Transfer Service의 지연 없는 실시간 데이터를 조회하기 위해 CQRS Service를 호출한다.
ㄷ. Account DB 테이블과 User DB의 테이블 간의 대량 데이터 조인 처리를 위해 CQRS DB에 복제하여 처리한다.
ㄹ. Transfer Service에서 Account DB내 데이터를 포함하여 주기적인 배치 처리가 필요한 경우 배치 대상 테이블을 CQRS DB에 복제한다.
ㅁ. 복제 대상 선정은 CUD 빈도가 잦은 테이블을 대상으로 하는 것이 유리하다.
① ㄱ, ㄴ, ㄷ
② ㄱ, ㄴ, ㅁ
③ ㄱ, ㄷ, ㄹ
④ ㄱ, ㄴ, ㄹ, ㅁ
⑤ ㄴ, ㄷ, ㄹ, ㅁ
(정답)3
(난이도)하

문항 6) 프로젝트에서 중계 시스템(EAI등)을 이용한 파일 송수신 패턴에 대한 그림이다. 프레임워크 담당자가 고려해야 할 사안 중 잘못 설명하고 있는 것을 모두 고르시오. [4점]



<전제 사항>

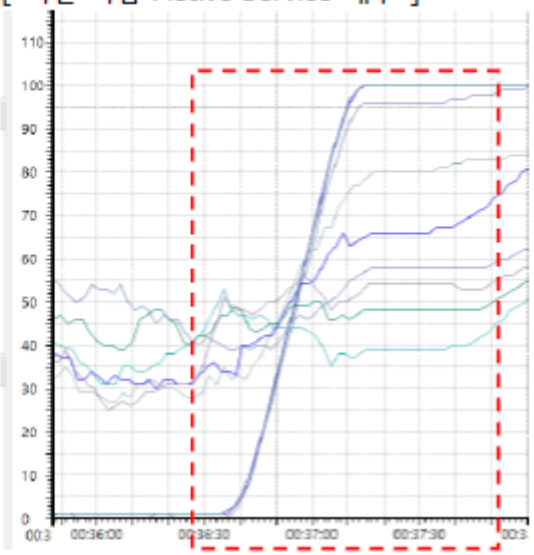
- 시스템간 파일 생성 완료 여부 및 파일의 정합성 검증을 위해 송신 파일 외 체크 파일이 필요할 수 있다.
- 체크 파일은 송신 파일에 대한 검증 용도로 사용되며 내부에 Line수 또는 전체 Byte 수를 포함하고 있다.

<고려 사항>
ㄱ. 파일 연계 패턴 중 체크 파일의 필요성이 높은 순서로 나열하면 3 > 4 > 1 > 2 순이다.
ㄴ. 3번 패턴에서 송신 시스템이 파일을 생성한 후 연계 디렉토리로 복사하는 경우라면, 체크 파일이 송신 파일 보다 먼저 복사되어야 한다.
ㄷ. 4번 패턴에서 중계 시스템으로부터 수신한 파일을 작업 관리자가 시작한 배치 프로그램이 체크 파일을 확인한 후 처리하도록 한다.
ㄹ. 1, 3번 송신 패턴에서 체크 파일은 배치 재수행을 고려하여 송신 파일 생성 전에 삭제하도록 한다.
ㅁ. 체크 파일은 송신 파일 생성 후 만들어져야 하며, 체크 파일 생성 시 동일한 체크 파일이 존재하는 경우라면 오류를 발생시켜야 한다.
① ㄱ, ㄴ
② ㄱ, ㄷ
③ ㄴ, ㄷ
④ ㄷ, ㄹ
⑤ ㄹ, ㅁ
(정답)1
(난이도)상

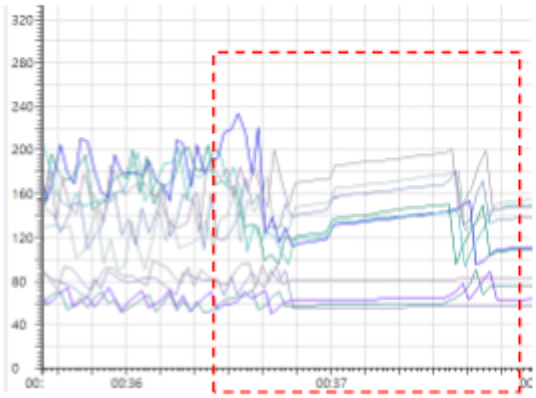
문항7) A 시스템에서 지연이 발생하여 수 분간 정상적인 서비스를 할 수 없었다. Access 로그 분석을 통해 지연 시작 전후로 특별히 더 많은 트랜잭션이 유입된 것은 아닌 것으로 확인되었다.
아래 제공된 여러 지표를 분석하여 성능 개선을 위해 우선적으로 확인 및 조치해야 할 부분을 [A시스템 성능개선 항목 보기]에서 2개 고르시오. [5점]
(※ 빨간 점선으로 표시된 부분은 지연이 발생했던 동일 시점을 표시한 것이다.)

[지연 시점 X-View]

[지연 시점 Active Service 개수]



[지연 시점 메모리 사용량]



[지연 시점 XLog List]

XLog - Tomcat

Object	Elapsed	Service	StartTime	EndTime	Cpu	SQL ...	SQL ...	API ...	API Time	KBytes	IP
/jpetstore_5/jpetstore-595bbb8b79-sqkcn	60,146	orderviewOrder	00:36:34.181	00:37:34.327	0	0	0	1	60,093	0	19
/jpetstore_2/jpetstore-595bbb8b79-nmlxn	60,080	[hystrix]order#order	00:36:34.317	00:37:34.397	0	0	0	1	60,079	0	0.0
/jpetstore_2/jpetstore-595bbb8b79-nmlxn	60,169	ordernewOrder	00:36:34.248	00:37:34.417	0	0	0	0	0	0	19
/jpetstore_5/jpetstore-595bbb8b79-sqkcn	60,179	ordernewOrder	00:36:34.464	00:37:34.643	0	0	0	1	60,114	0	19
/jpetstore_0/jpetstore-595bbb8b79-mzsld	60,341	[hystrix]order#order	00:36:34.320	00:37:34.661	0	0	0	1	60,340	0	0.0
/jpetstore_0/jpetstore-595bbb8b79-mzsld	60,426	ordernewOrder	00:36:34.247	00:37:34.673	0	0	0	0	0	0	19
/jpetstore_1/jpetstore-595bbb8b79-drwr5	65,380	ordernewOrder	00:36:29.464	00:37:34.844	0	0	0	1	60,103	0	19
/jpetstore_5/jpetstore-595bbb8b79-sqkcn	60,129	orderviewOrder	00:36:34.750	00:37:34.879	0	0	0	1	60,122	0	19
/jpetstore_1/jpetstore-595bbb8b79-drwr5	66,316	ordernewOrder	00:36:28.611	00:37:34.927	0	0	0	1	60,101	0	19
/jpetstore_6/jpetstore-595bbb8b79-58hx9	60,230	ordernewOrder	00:36:34.847	00:37:35.077	0	0	0	1	60,109	0	19
/jpetstore_0/jpetstore-595bbb8b79-mzsld	60,395	[hystrix]order#order	00:36:34.787	00:37:35.182	0	0	0	1	60,394	0	0.0
/jpetstore_0/jpetstore-595bbb8b79-mzsld	60,407	orderviewOrder	00:36:34.776	00:37:35.183	0	0	0	0	0	0	19
/jpetstore_5/jpetstore-595bbb8b79-sqkcn	60,111	ordernewOrder	00:36:35.168	00:37:35.279	0	0	0	1	60,065	0	19
/jpetstore_5/jpetstore-595bbb8b79-sqkcn	60,243	ordernewOrder	00:36:35.594	00:37:35.837	0	0	0	1	60,169	0	19
/jpetstore_6/jpetstore-595bbb8b79-58hx9	60,109	orderviewOrder	00:36:35.772	00:37:35.881	0	0	0	1	60,073	0	19
/jpetstore_1/jpetstore-595bbb8b79-drwr5	60,260	ordernewOrder	00:36:35.731	00:37:35.991	0	0	0	1	60,109	0	19
/jpetstore_0/jpetstore-595bbb8b79-mzsld	60,149	[hystrix]order#order	00:36:36.188	00:37:36.337	0	0	0	1	60,148	0	0.0

[A시스템 성능개선 항목 보기]
ㄱ. 느린 쿼리 식별 및 튜닝
ㄴ. 지연 응용 로직 개선
ㄷ. DB Connection Pool 부족 확인 및 상향 조정
ㄹ. WAS Thread Pool 부족 확인 및 상향 조정

ㅁ. HttpClient Connection Pool 부족 확인 및 상향 조정
ㅂ. HeapDump 발생 확인 및 크기 조정
ㅅ. 외부 시스템 지연 확인 요청
(정답)ㅁ{|}ㅅ
(난이도)중