참고 자료

- Software Architecture 설계/구축
    - DB 계정정보 암호화 설정
    - Tomcat JDBC Connection Pool
    - 개발 환경 및 구성의 트렌드 변화
    - 비동기 Adaptor 작성
    - 호출 관계에 대한 이해도
- Software Architecture 핵심
    - Heapdump 참고 자료
    - I/F 솔루션
    - JDK 버전별 GC 변화
    - JVM 옵션
    - JVM의 GC 발생 조건 및 동작 방식
    - Reverse Proxy (14번)
    - 웹 페이지가 로드 되는 과정
- Software Architecture 환경
    - HTTP 응답코드
    - TIME_WAIT 상태
    - 캐싱 메커니즘
    - 트랜잭션 처리와 EAI 호출
- Software Architecure 운영/문제해결
    - HttpClient 로깅
    - HTTP Protocol 에 대한 이해: HTTP Method, Header 정보의 기능 이해
    - Lighthouse에 대한 이해
    - MSA 기반 환경의 모니터링 도구 활용
    - Performance Profiling 에 대한 이해
    - 분산 로그 추적
- SWA-출제 동향(2025)-작업중
- 신기술
    - JWT 특징
    - k8s 기본 명령어
    - Kafka 병목현상에 대한 조치

- Kafka 설계 시 고려사항
- Liveness, Readiness 점검
- MSA 제대로 이해하기
- 컨테이너 프로젝트의 CICD 기본 프로세스

Software Architecture 설계/구축

DB 계정정보 암호화 설정

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

Tomcat JDBC Connection Pool

📚 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

개발 환경 및 구성의 트렌드 변화

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?

비동기 Adaptor 작성

- **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

호출 관계에 대한 이해도

- **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

Software Architecture 핵심

- Heapdump 참고 자료
- I/F 솔루션
- JDK 버전별 GC 변화
- JVM 옵션
- JVM의 GC 발생 조건 및 동작 방식
- Reverse Proxy (14번)
- 웹 페이지가 로드 되는 과정

Heapdump 참고 자료

[JAVA] heap dump 분석 (Out Of Memory 분석)

I/F 솔루션

🔎 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
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

구분	원칙	설명
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
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

JDK 버전별 GC 변화

✓ JDK 버전별 GC 변화 요약

JDK 버전	기본 GC	주요 GC 변화 및 특징
Java 7	Parallel GC	G1GC 등장 (`-XX:+UseG1GC`) – 실험적
Java 8	Parallel GC (서버에서는 G1GC 가능)	G1GC **정식 지원** PermGen 제거 → Metaspace 도입
Java 9	G1GC *(기본 GC 변경)*	기본 GC가 **G1GC로 전환** JEP 248 적용
Java 10	G1GC	**Application Class-Data Sharing** (AppCDS), GC 영향 적음
Java 11	G1GC	**ZGC 최초 도입** *(실험적)* `-XX:+UseZGC` 옵션 추가
Java 12	G1GC	**Shenandoah GC** *(RedHat 기반)* 실험적로 포함
Java 13	G1GC	**ZGC 개선** – Unload class, soft ref 등 지원
Java 14	G1GC	ZGC/ Shenandoah **생산환경 사용 가능** JEP 361: `ZGC` 정식화
Java 15	G1GC	**ZGC가 Linux/macOS/Windows** 전면 지원
Java 17 (LTS)	G1GC	**ZGC, Shenandoah 모두 안정화** GC tuning 옵션 통합 (JEP 318, JEP 376)
Java 21 (LTS)	G1GC	**ZGC와 G1GC 모두 고도화** Generational ZGC 도입 준비 중 (Preview 예정)

✔ 주요 GC 변화 타임라인

Java 7 : G1GC (실험적)
Java 8 : G1GC 정식 도입
Java 9 : 기본 GC → G1GC 전환
Java 11 : ZGC (도입), JEP 333
Java 12 : Shenandoah (RedHat 주도)
Java 14~ : ZGC, Shenandoah 안정화
Java 17+ : G1GC 여전히 기본 / ZGC 안정 운영 가능
Java 21 : Generational ZGC Preview

✔ 실무 선택 가이드 (Java 버전별)

JDK 버전	안정적 선택	초저지연용
Java 8	G1GC	불가
Java 11	G1GC	ZGC (테스트 후 권장)
Java 17	G1GC	ZGC or Shenandoah (안정)
Java 21+	G1GC	**Generational ZGC** *(초기 도입 중)*

✔ 1. 왜 GC가 계속 바뀌었는가?

🔑 주요 이유

- **서비스 규모 확대**: 수백 GB ~ TB급 Heap에서 기존 GC는 멈춤 시간이 너무 길어짐
- **성능 다양성**: GC가 "모든 상황에 완벽"할 수는 없음 → 목적별 선택 필요
- **클라우드/컨테이너 최적화**: 컨테이너 환경에서는 `pause time`과 `예측 가능성`이 더 중요

✔ 2. 주요 GC 별 목적 요약

GC 종류	목적	JVM 적용 버전	특징 요약
Serial GC	가장 단순한 GC	모든 버전	STW(Stop-The-World) 전체 수집
Parallel GC	Throughput 우선	기본 GC (JDK 7까지)	빠르지만 Pause 길음
G1GC	Pause 시간 제어	Java 7u4+, Java 9부터 기본	Region 단위, 세대형 구조 유지
ZGC	초저지연 응답	Java 11+	Pause ≤ 10ms, non-generational
Shenandoah	대용량 병렬 처리	Java 12~ (RedHat 중심)	Pause Time 짧음, Generational 구조 있음
Generational ZGC	ZGC + 세대별 수집	Java 21 (Preview)	ZGC 약점 보완 (Young/Old 분리)

✔ 3. 실무 선택 기준 (Java 17 기준)

상황	추천 GC	이유
일반 Spring Boot / Batch 시스템	G1GC	안정성, 예측 가능한 pause time
실시간 API, 광고 서버, 게임	ZGC	극도로 짧은 응답 지연
대용량 데이터 분석, JavaFX UI	Shenandoah	pause + throughput 둘 다 우수
Kubernetes 환경	G1GC 또는 ZGC	ZGC는 container-friendly (low-pause), G1은 호환성

✔ 4. Kubernetes에서 GC 설정 예시

◆ G1GC

yaml

```
env:
  - name: JAVA_TOOL_OPTIONS
    value: >
      -XX:+UseG1GC
      -XX:MaxGCPauseMillis=200
```

◆ ZGC

yaml

```
env:
  - name: JAVA_TOOL_OPTIONS
    value: >
      -XX:+UseZGC
      -XX:SoftMaxHeapSize=2G
```

> ✓ `SoftMaxHeapSize` *를 지정하면 컨테이너 안에서 메모리를 더 유연하게 GC가 사용할 수 있습니다.*

✔ 5. GC 전환 시 주의사항

항목	주의점
`ZGC`, `Shenandoah`	**Heap 이 크지 않으면 오히려 효율 떨어짐**
G1GC → ZGC	GC pause는 줄지만, **CPU 사용량은 증가** 가능성 있음
GC 로그 분석 방식	GC 로그 형식(GC unified logging)도 버전마다 다름 (`-Xlog:gc*` 사용 권장)

✔ 결론 요약

- **Java 8**: G1GC 안정 운영에 적합
- **Java 11/17**: G1GC은 기본, ZGC는 실시간 시스템에 선택 가능
- **Java 21+**: Generational ZGC로 성능·응답성 균형 기대
- GC 선택은 **Heap 크기, 처리 목적, Pause 민감도**에 따라 달라져야 함

📊 실시간 GC 모니터링

1. **VisualVM 실행** → 실행 중인 JVM 선택
2. **Monitor 탭** → Heap/Perm/Threads 실시간 사용량 확인
3. **VisualGC 탭** (플러그인 필요)
    - **Eden**, **Survivor**, **Old**, **Perm** 영역별 GC 동작 시점 시각화
    - GC 횟수, 시간, promotion 양 등 추적 가능

🔑 분석 포인트

항목	해석 방법
Heap이 자주 Full → Old 영역 GC	객체가 Survivor를 넘어 오래 살아남고 있다는 뜻
Minor GC 증가	Eden 가득 차서 자주 청소 → Eden 사이즈 줄이거나 객체 생존 줄이기
Old 영역 급격히 증가	대량 객체 메모리 릭 의심
GC 시간 증가	GC 튜닝 필요 (`MaxGCPauseMillis` , G1HeapRegionSize 등)

🔔 실무 팁

- `GC activity` 그래프에서 Old 영역이 줄어들지 않으면 Full GC가 비효율적일 수 있음
- VisualGC 플러그인 없이도 `Sampler → Memory` 로 클래스별 객체 수 확인 가능

✔ 2. Eclipse MAT를 활용한 GC Heap 분석

📦 목적

- **Heap Dump 파일**을 분석하여 **누가 메모리를 점유하는지, GC가 수거하지 못하는 이유**를 파악

🔧 준비

1. Heap dump 생성

   bash

   ```
   jmap -dump:live,format=b,file=heap.hprof <PID>
   ```

2. Eclipse MAT 설치
   https://www.eclipse.org/mat/

🔍 분석 흐름

① 힙 덤프 로드

- `.hprof` 파일 열기 → 자동 분석: **Leak Suspects Report** 실행

② Dominator Tree 보기

- `Histogram`에서 메모리 상위 클래스 확인
- `Dominator Tree` : 누가 해당 객체들을 참조하고 있는지 분석

③ GC Root Path 추적

- 특정 객체에서 **"Path to GC Root"** 실행
- 이 참조 체인이 끊기지 않으면 GC가 수거하지 못함 → 메모리 릭 원인

🔎 분석 포인트

항목	설명
`Retained Heap`	해당 객체 제거 시 같이 해제되는 메모리 크기
`Shallow Heap`	해당 객체만의 메모리 사용량
`GC Root`	JVM에서 GC가 참조하는 시작점
`unreachable but not collected`	수거되어야 하지만 누군가 참조 중인 객체 → 릭 가능성

✓ 예시: OOME 발생 원인 분석

1. MAT에서 `Leak Suspect Report` 분석
2. 상위 5개 클래스의 `Retained Heap` 확인
3. `java.util.HashMap`, `byte[]`, `ThreadLocalMap`이 높으면 주의
4. GC Root 경로 확인 후 해당 객체를 참조하는 Bean 또는 캐시 구조 분석

✓ 3. 실전 비교 요약

항목	VisualVM	Eclipse MAT
용도	**실시간 GC/Heap 모니터링**	**정적 힙 구조 분석 및 릭 추적**
GC 행동 확인	가능 (VisualGC 탭)	간접 가능 (Old 객체 증적 추적)
분석 정확도	실행 중 JVM 기준	OOME 순간의 상태 정확히 분석
성능 부하	적음 (low overhead)	없음 (덤프 분석은 JVM 외부)

JVM 옵션

✓ 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)

주요 설정	연관 설정	설명
`-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

JVM의 GC 발생 조건 및 동작 방식

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

```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!

Reverse Proxy (14번)

Reverse Proxy란?

Reverse Proxy는 클라이언트와 서버 사이에서 **서버를 대신해** 요청을 받고 응답을 전달하는 중계 서버입니다.

일반적인 구조

```
클라이언트 → Reverse Proxy(Nginx) → Backend Server(Tomcat WAS)
```

Reverse Proxy의 역할

1. **로드 밸런싱**: 여러 WAS로 요청 분산
2. **SSL 터미네이션**: HTTPS 처리를 Proxy에서 담당
3. **정적 파일 서빙**: 이미지, CSS, JS 등을 직접 처리
4. **캐싱**: 자주 요청되는 내용을 캐시로 빠르게 응답
5. **보안**: 실제 서버를 숨기고 방화벽 역할

문제에서의 Nginx + Tomcat 구조

```
사용자 요청 → Nginx(Web Server) → Tomcat(WAS) → DB
        ←                  ←             ←
```

- **Nginx**: 정적 파일 처리, 리버스 프록시 역할
- **Tomcat**: JSP/Servlet 처리, 비즈니스 로직
- **연동 방식**: Nginx가 동적 요청(JSP)을 Tomcat으로 전달

Forward Proxy vs Reverse Proxy

- **Forward Proxy**: 클라이언트를 대신 (회사 프록시 서버)
- **Reverse Proxy**: 서버를 대신 (Nginx가 Tomcat 앞에서)

2번이 맞는 이유

```
사용자 → Nginx → Tomcat
          ↓
    JSP 요청이 Tomcat으로 안 가
    Nginx에서 처리하려고 시도
          ↓
    Nginx는 JSP 파일을 찾을 수 없음
          ↓
    404 Not Found
```

Nginx의 역할 구분

정적 파일 (Nginx가 직접 처리):

- HTML, CSS, JS, 이미지 파일
- 캐시된 콘텐츠

동적 파일 (Tomcat으로 전달):

- JSP, Servlet
- 비즈니스 로직이 필요한 요청

1번이 틀린 이유: 리다이렉션 URL은 `Location` **헤더**에 담기며, `referrer` 헤더가 아닙니다.

웹 페이지가 로드 되는 과정

웹 페이지가 로드되는 과정을 단계별로 설명

1단계: 서버에 요청하기

- TTFB (Time To First Byte) : 서버에 "페이지 주세요!" 하고 요청했을 때, 서버가 첫 번째 데이터를 보내주기까지 걸리는 시간
  - 마치 음식점에서 주문하고 첫 번째 음식이 나올 때까지 기다리는 시간

2단계: 화면에 뭔가 나타나기 시작

- FP (First Paint) : 화면에 뭔가(색깔이라도) 처음 나타나는 순간
- FCP (First Contentful Paint) : 화면에 텍스트나 이미지 같은 실제 내용이 처음 보이는 순간
- 마치 그림을 그릴 때 첫 번째 붓질을 하는 순간

3단계: 페이지 구조 완성

- DCL (DOMContentLoaded) : 페이지의 기본 골격(HTML)이 완성되는 순간
- 집을 지을 때 뼈대가 완성되는 것과 비슷

4단계: 가장 중요한 내용 완성

- LCP (Largest Contentful Paint) : 페이지에서 가장 큰/중요한 내용(큰 이미지, 제목 등)이 완전히 보이는 순간
- 집에서 거실 가구까지 다 들어온 순간

5단계: 레이아웃 흔들림

- Layout Shift : 페이지가 로드되면서 내용들이 갑자기 위아래로 움직이는 현상
- 마치 책을 읽고 있는데 갑자기 글자들이 위아래로 튀는 것

Software Architecture 환경

HTTP 응답코드

✔ 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
			`404 Not Found`	리소스 없음
			`429 Too Many Requests`	요청 과도 (Rate Limit)

범위	이름	설명	예시 코드	설명 (대표 예시)
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

TIME_WAIT 상태

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

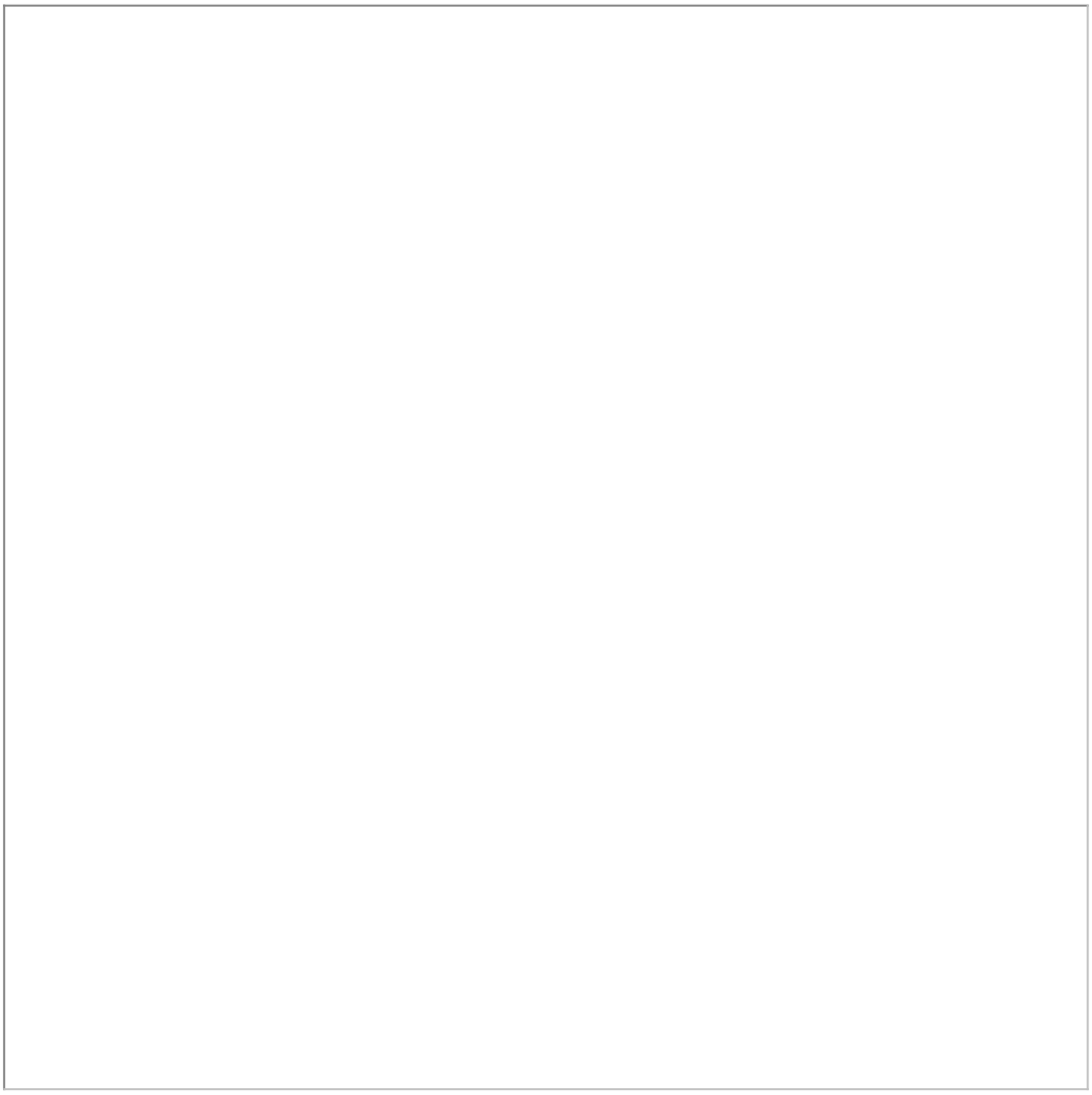
여기서 SYN은 'synchronize sequence numbers', 그리고 ACK는'acknowledgment' 의 약자이다.

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]

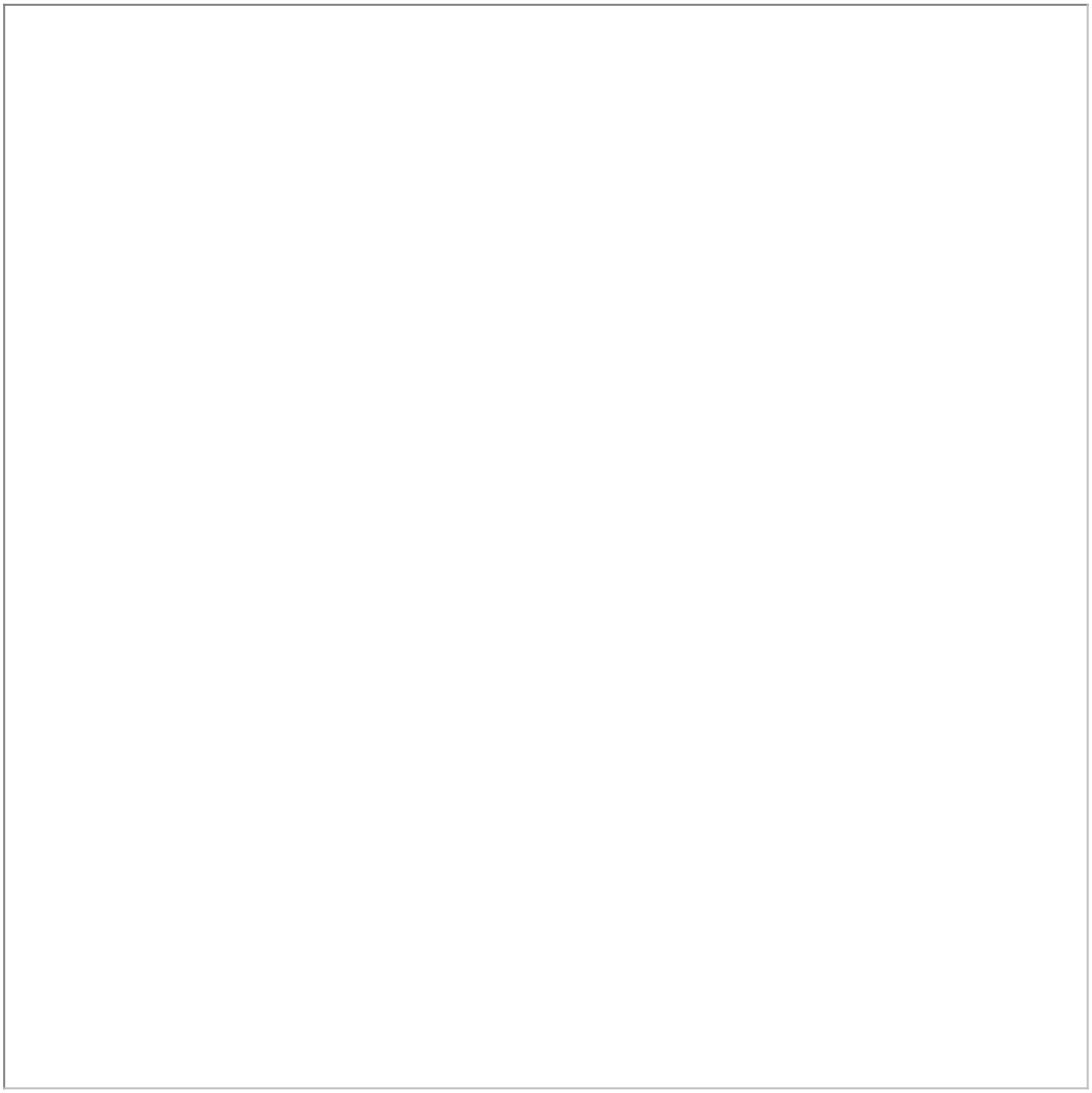
A클라이언트는 B서버에게 ACK을 보내고 이후로부터는 연결이 이루어지고 데이터가 오가게 되는것이다. 이때의 B서버 상태가 ESTABLISHED 이다.

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

캐싱 메커니즘

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

트랜잭션 처리와 EAI 호출

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

Software Architecure 운영/문제해결

HttpClient 로깅

✓ 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);`
`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
`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

HTTP Protocol 에 대한 이해: HTTP Method, Header 정보의 기능 이해

1. HTTP 기본 개념

항목	설명
정의	HyperText Transfer Protocol - 웹에서 클라이언트와 서버 간 데이터 통신 규약
구조	요청(Request) ↔ 응답(Response)
특징	무상태(Stateless), 텍스트 기반 프로토콜
포트	HTTP: 80, HTTPS: 443

HTTP 버전별 특징

버전	주요 특징	연결 방식
HTTP/1.0	단순한 요청/응답	연결당 하나의 요청/응답
HTTP/1.1	지속적 연결, 파이프라이닝	Keep-Alive 지원
HTTP/2	멀티플렉싱, 서버 푸시, 헤더 압축	바이너리 프레이밍
HTTP/3	QUIC 프로토콜 기반	UDP 기반, 더 빠른 연결

2. HTTP Method 상세

Method	목적	특징	멱등성	안전성	Body 사용
GET	데이터 조회	URL 파라미터 사용	✓	✓	✗
POST	새 데이터 생성	폼 데이터, JSON 등	✗	✗	✓

Method	목적	특징	멱등성	안전성	Body 사용
PUT	전체 데이터 교체	리소스 완전 대체	✓	✗	✓
PATCH	부분 데이터 수정	일부 필드만 수정	경우에 따라	✗	✓
DELETE	데이터 삭제	리소스 제거	✓	✗	선택적
HEAD	헤더 정보만 조회	GET과 동일하나 Body 없음	✓	✓	✗
OPTIONS	지원 메소드 확인	CORS 프리플라이트	✓	✓	✗

Method 사용 예시

작업	Method	URL 예시	설명
사용자 목록 조회	GET	`/users?page=1&limit=10`	쿼리 파라미터로 페이징
사용자 생성	POST	`/users`	Body에 사용자 정보
사용자 정보 전체 수정	PUT	`/users/123`	모든 필드 교체
사용자 정보 부분 수정	PATCH	`/users/123`	일부 필드만 수정
사용자 삭제	DELETE	`/users/123`	특정 사용자 제거

3. HTTP Header 분류

3.1 Request Headers (요청 헤더)

일반 헤더

헤더명	설명	예시
Host	요청할 서버 정보	`Host: www.example.com:8080`
User-Agent	클라이언트 정보	`User-Agent: Mozilla/5.0...`
Referer	이전 페이지 URL	`Referer: https://google.com`

콘텐츠 협상 헤더

헤더명	설명	예시
Accept	받을 수 있는 미디어 타입	`Accept: application/json, text/html`
Accept-Language	선호 언어	`Accept-Language: ko-KR,en;q=0.8`
Accept-Encoding	지원 압축 방식	`Accept-Encoding: gzip, deflate`
Accept-Charset	지원 문자 인코딩	`Accept-Charset: utf-8, iso-8859-1`

인증 및 보안 헤더

헤더명	설명	예시
Authorization	인증 정보	`Authorization: Bearer token123`
Cookie	쿠키 정보	`Cookie: sessionId=abc123`

캐시 제어 헤더

헤더명	설명	예시
Cache-Control	캐시 정책	`Cache-Control: no-cache`
If-Modified-Since	조건부 요청	`If-Modified-Since: Wed, 21 Oct 2015 07:28:00 GMT`
If-None-Match	ETag 기반 조건부 요청	`If-None-Match: "33a64df551425fcc"`

콘텐츠 헤더

헤더명	설명	예시
Content-Type	요청 본문 미디어 타입	`Content-Type: application/json; charset=utf-8`
Content-Length	요청 본문 크기	`Content-Length: 1024`
Content-Encoding	요청 본문 압축 방식	`Content-Encoding: gzip`

3.2 Response Headers (응답 헤더)

서버 정보 헤더

헤더명	설명	예시
Server	서버 소프트웨어 정보	`Server: nginx/1.18.0 (Ubuntu)`
Date	응답 생성 시간	`Date: Wed, 21 Oct 2015 07:28:00 GMT`

콘텐츠 헤더

헤더명	설명	예시
Content-Type	응답 본문 미디어 타입	`Content-Type: application/json; charset=utf-8`
Content-Length	응답 본문 크기	`Content-Length: 2048`
Content-Encoding	응답 본문 압축 방식	`Content-Encoding: gzip`
Content-Language	응답 본문 언어	`Content-Language: ko-KR`

캐시 제어 헤더

헤더명	설명	예시
Cache-Control	캐시 정책	`Cache-Control: public, max-age=3600`

헤더명	설명	예시
ETag	리소스 고유 식별자	`ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"`
Last-Modified	마지막 수정 시간	`Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT`
Expires	캐시 만료 시간	`Expires: Thu, 01 Dec 2024 16:00:00 GMT`

보안 및 정책 헤더

헤더명	설명	예시
Set-Cookie	쿠키 설정	`Set-Cookie: sessionId=abc123; HttpOnly; Secure`
Access-Control-Allow-Origin	CORS 정책	`Access-Control-Allow-Origin: https://example.com`
X-Frame-Options	프레임 삽입 방지	`X-Frame-Options: DENY`
X-Content-Type-Options	MIME 스니핑 방지	`X-Content-Type-Options: nosniff`

리다이렉션 헤더

헤더명	설명	예시
Location	리다이렉션 위치 (`302`, `301` 응답 시 사용)	`Location: https://new-url.com/page`

4. HTTP 상태 코드

1xx (정보성 응답)

코드	상태	설명
100	Continue	요청의 일부를 받았으며, 나머지를 계속 보내도 됨
101	Switching Protocols	프로토콜 전환

2xx (성공)

코드	상태	설명	주요 사용
200	OK	요청 성공	GET, PUT, PATCH
201	Created	새 리소스 생성 성공	POST
202	Accepted	요청 접수됨 (처리 중)	비동기 처리
204	No Content	요청 성공, 응답 본문 없음	DELETE

3xx (리다이렉션)

코드	상태	설명	캐시 가능
301	Moved Permanently	영구적으로 이동	✓
302	Found	임시적으로 이동	✗
304	Not Modified	캐시된 버전 사용	-
307	Temporary Redirect	임시 리다이렉트 (메소드 유지)	✗

4xx (클라이언트 오류)

코드	상태	설명	해결 방법
400	Bad Request	잘못된 요청 구문	요청 형식 확인
401	Unauthorized	인증 필요	로그인/토큰 확인
403	Forbidden	접근 권한 없음	권한 확인
404	Not Found	리소스를 찾을 수 없음	URL 확인
405	Method Not Allowed	허용되지 않은 메소드	메소드 변경
409	Conflict	리소스 충돌	충돌 해결
429	Too Many Requests	요청 횟수 초과	요청 빈도 조절

5xx (서버 오류)

코드	상태	설명	대응 방법
500	Internal Server Error	서버 내부 오류	서버 로그 확인
502	Bad Gateway	게이트웨이 오류	업스트림 서버 확인
503	Service Unavailable	서비스 이용 불가	서버 상태 확인
504	Gateway Timeout	게이트웨이 타임아웃	네트워크/서버 확인

5. 주요 Content-Type

Content-Type	설명	사용 예시
`application/json`	JSON 데이터	REST API

Content-Type	설명	사용 예시
`application/xml`	XML 데이터	SOAP API
`application/x-www-form-urlencoded`	HTML 폼 데이터	기본 폼 전송
`multipart/form-data`	파일 업로드 포함 폼	파일 업로드
`text/html`	HTML 문서	웹 페이지
`text/plain`	일반 텍스트	로그 파일
`image/jpeg`, `image/png`	이미지 파일	이미지 응답

6. 캐시 제어 옵션

Cache-Control 옵션	설명	예시
`no-cache`	캐시 사용 전 서버 검증 필요	`Cache-Control: no-cache`
`no-store`	캐시 저장 금지	`Cache-Control: no-store`
`max-age`	캐시 유효 시간 (초)	`Cache-Control: max-age=3600`
`public`	모든 캐시에서 저장 가능	`Cache-Control: public`
`private`	브라우저 캐시에만 저장	`Cache-Control: private`
`must-revalidate`	만료 시 반드시 재검증	`Cache-Control: must-revalidate`

7. 실습 명령어

cURL 명령어 예시

작업	명령어
GET 요청	`curl -X GET "https://api.example.com/users" -H "Accept: application/json"`
POST 요청	`curl -X POST "https://api.example.com/users" -H "Content-Type: application/json" -d '{"name":"홍길동"}'`
PUT 요청	`curl -X PUT "https://api.example.com/users/123" -H "Content-Type: application/json" -d '{"name":"김철수"}'`
DELETE 요청	`curl -X DELETE "https://api.example.com/users/123"`
헤더 정보만 확인	`curl -I "https://api.example.com/users"`
상세 정보 출력	`curl -v "https://api.example.com/users"`

HTTP Protocol 학습 가이드 - Method & Header 완전정복

1. HTTP Method 완전 분석

1.1 주요 HTTP Method 특성

Method	안전성(Safe)	멱등성 (Idempotent)	캐시 가능	Request Body	Response Body
GET	✓	✓	✓	✗	✓
POST	✗	✗	✗	✓	✓
PUT	✗	✓	✗	✓	✓
DELETE	✗	✓	✗	✗	✓
PATCH	✗	✗	✗	✓	✓
HEAD	✓	✓	✓	✗	✗
OPTIONS	✓	✓	✗	✗	✓

1.2 Method별 상세 설명

GET

- **용도**: 리소스 조회
- **특징**: 서버 상태 변경 없음 (Safe)
- **멱등성**: 동일 요청 반복 시 동일 결과
- **실무 포인트**:
    - 캐시 전략 수립 시 중요
    - URL 길이 제한 고려 (2048자)
    - 민감 정보 URL 노출 주의

POST

- **용도**: 리소스 생성, 데이터 처리
- **특징**: 서버 상태 변경 (Not Safe)
- **멱등성**: 없음 (반복 요청 시 다른 결과)
- **실무 포인트**:
    - 중복 요청 방지 로직 필요
    - CSRF 공격 대비책 필요
    - Request Body 크기 제한 고려

PUT vs PATCH

```
PUT /users/123
{
  "name": "김철수",
  "email": "kim@example.com",
  "age": 30
}
→ 전체 리소스 교체

PATCH /users/123
{
  "age": 31
}
→ 부분 업데이트
```

1.3 HTTP Method 선택 기준

```
flowchart TD
    A[API 설계] --> B{데이터 조회?}
    B -->|Yes| C[GET]
    B -->|No| D{리소스 생성?}
    D -->|Yes| E[POST]
    D -->|No| F{전체 업데이트?}
    F -->|Yes| G[PUT]
    F -->|No| H{부분 업데이트?}
    H -->|Yes| I[PATCH]
    H -->|No| J{리소스 삭제?}
    J -->|Yes| K[DELETE]
```

2. HTTP Header 완전 분석

2.1 Request Header 주요 항목

인증 관련

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Authorization: Basic dXNlcjpwYXNzd29yZA==
Cookie: sessionId=abc123; userId=kim
```

콘텐츠 관련

```
Content-Type: application/json; charset=utf-8
Content-Length: 1024
Content-Encoding: gzip
Accept: application/json, text/html;q=0.9
Accept-Language: ko-KR,ko;q=0.9,en;q=0.8
Accept-Encoding: gzip, deflate, br
```

캐시 관련

```
Cache-Control: no-cache
Cache-Control: max-age=3600
If-Modified-Since: Wed, 21 Oct 2024 07:28:00 GMT
If-None-Match: "686897696a7c876b7e"
```

보안 관련

```
Origin: https://example.com
Referer: https://example.com/page
X-Requested-With: XMLHttpRequest
X-CSRF-Token: abc123def456
```

2.2 Response Header 주요 항목

상태 및 메타데이터

```
Content-Type: application/json; charset=utf-8
Content-Length: 2048
Server: nginx/1.18.0
Date: Wed, 20 Jul 2025 10:30:00 GMT
```

캐시 제어

```
Cache-Control: public, max-age=3600
ETag: "686897696a7c876b7e"
```

```
Last-Modified: Wed, 20 Jul 2025 09:30:00 GMT
Expires: Wed, 20 Jul 2025 11:30:00 GMT
```

보안 헤더

```
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Strict-Transport-Security: max-age=31536000; includeSubDomains
Content-Security-Policy: default-src 'self'
```

리다이렉션

```
Location: https://example.com/new-url
```

2.3 Header 실무 활용 패턴

API 버전 관리

```
# URL 버전
GET /api/v1/users

# Header 버전
Accept: application/vnd.api+json;version=1
```

콘텐츠 협상

```
# 클라이언트 요청
Accept: application/json, application/xml;q=0.5

# 서버 응답
Content-Type: application/json
Vary: Accept
```

CORS 처리

```
# Preflight Request
OPTIONS /api/users
Origin: https://frontend.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-Type

# Preflight Response
Access-Control-Allow-Origin: https://frontend.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 86400
```

3. HTTP 상태 코드별 Header 패턴

3.1 성공 응답 (2xx)

```
# 200 OK
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=300

# 201 Created
HTTP/1.1 201 Created
Location: /api/users/123
Content-Type: application/json

# 204 No Content
HTTP/1.1 204 No Content
```

3.2 리다이렉션 (3xx)

```
# 301 Moved Permanently
HTTP/1.1 301 Moved Permanently
Location: https://new.example.com/page

# 304 Not Modified
HTTP/1.1 304 Not Modified
ETag: "686897696a7c876b7e"
Cache-Control: max-age=3600
```

3.3 클라이언트 오류 (4xx)

```
# 400 Bad Request
HTTP/1.1 400 Bad Request
Content-Type: application/json
{
  "error": "Invalid request format"
}

# 401 Unauthorized
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="API"

# 404 Not Found
HTTP/1.1 404 Not Found
Content-Type: application/json

# 429 Too Many Requests
HTTP/1.1 429 Too Many Requests
Retry-After: 60
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 0
```

```
X-RateLimit-Reset: 1658310000
```

3.4 서버 오류 (5xx)

```
# 500 Internal Server Error
HTTP/1.1 500 Internal Server Error
Content-Type: application/json

# 502 Bad Gateway
HTTP/1.1 502 Bad Gateway
Server: nginx/1.18.0

# 503 Service Unavailable
HTTP/1.1 503 Service Unavailable
Retry-After: 120

# 504 Gateway Timeout
HTTP/1.1 504 Gateway Timeout
```

4. 실무 적용 시나리오

4.1 웹 애플리케이션 시나리오

```
# 로그인 요청
POST /api/auth/login
Content-Type: application/json
X-CSRF-Token: abc123

{
  "username": "user",
  "password": "pass"
}

# 로그인 응답
HTTP/1.1 200 OK
Set-Cookie: sessionId=xyz789; HttpOnly; Secure; SameSite=Strict
Content-Type: application/json

{
  "token": "jwt_token_here"
}
```

4.2 API Gateway 시나리오

```
# 클라이언트 → API Gateway
GET /api/users/123
Authorization: Bearer token
X-API-Key: client_api_key

# API Gateway → Backend Service
```

```
GET /users/123
Authorization: Bearer internal_token
X-Forwarded-For: 192.168.1.100
X-Request-ID: req-12345
```

4.3 마이크로서비스 간 통신

```
# Service A → Service B
POST /internal/orders
Content-Type: application/json
X-Service-Name: order-service
X-Correlation-ID: corr-67890
X-User-ID: user123

{
  "productId": "prod456",
  "quantity": 2
}
```

5. Header 보안 Best Practices

5.1 필수 보안 헤더

```
# XSS 방지
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

# HTTPS 강제
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload

# CSP 설정
Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline'
```

5.2 민감 정보 처리

```
# 민감한 응답의 캐시 방지
Cache-Control: no-store, no-cache, must-revalidate, private
Pragma: no-cache
Expires: 0

# 서버 정보 숨김
Server: WebServer
```

6. 디버깅 및 모니터링

6.1 요청 추적을 위한 헤더

```
X-Request-ID: req-12345
X-Correlation-ID: corr-67890
X-Trace-ID: trace-abcdef
X-Span-ID: span-123456
```

6.2 성능 모니터링 헤더

```
X-Response-Time: 150ms
X-Cache-Status: HIT
X-Upstream-Response-Time: 0.045
```

7. 학습 점검 포인트

기본 개념

- [ ] HTTP Method별 특성 (Safe, Idempotent) 이해
- [ ] 상태 코드별 적절한 사용 시기 파악
- [ ] Request/Response Header 구조 이해

실무 응용

- [ ] API 설계 시 적절한 Method 선택
- [ ] 캐시 전략에 따른 Header 설정
- [ ] 보안 요구사항에 맞는 Header 적용
- [ ] CORS 정책 설정 및 처리

문제 해결

- [ ] HTTP 통신 문제 디버깅 능력
- [ ] 성능 최적화를 위한 Header 활용
- [ ] 보안 취약점 방지를 위한 Header 설정

8. 참고 자료

공식 문서

- RFC 7231 (HTTP/1.1 Semantics and Content)
- RFC 7232 (HTTP/1.1 Conditional Requests)
- RFC 7233 (HTTP/1.1 Range Requests)
- RFC 7234 (HTTP/1.1 Caching)
- RFC 7235 (HTTP/1.1 Authentication)

실습 도구

- curl, httpie (CLI)
- Postman, Insomnia (GUI)
- Browser DevTools
- Wireshark (패킷 분석)

이 자료를 바탕으로 체계적으로 학습하시면 HTTP Protocol에 대한 깊이 있는 이해를 얻을 수 있습니다.

✓ 참고 자료

- MDN Web Docs - HTTP 개요

Lighthouse에 대한 이해

✓ 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

MSA 기반 환경의 모니터링 도구 활용

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?

Performance Profiling 에 대한 이해

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

분산 로그 추적

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

신기술

JWT 특징

✔ 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

특징	설명
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
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)

항목	설명
재발급 로직	Access Token 만료 시 Refresh Token으로 새로 발급 요청

k8s 기본 명령어

✓ 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

명령어	설명
`kubectl exec -it <pod명> -- /bin/sh`	Pod 안으로 접속 (쉘 진입)
`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

상태	의미 및 원인
`ImagePullBackOff`	컨테이너 이미지를 못 불러오는 상태 (이미지명, 권한 오류 등)
`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

STATUS	의미	원인 및 조치
`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
```

Kafka 병목현상에 대한 조치

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

Kafka 설계 시 고려사항

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)

Liveness, Readiness 점검

✔ 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. K R 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

MSA 제대로 이해하기

MSA 제대로 이해하기

컨테이너 프로젝트의 CICD 기본 프로세스

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

구분	GitFlow	GitHub Flow	GitLab Flow
주요 목적	대규모 릴리즈 중심 개발	빠른 배포 / 지속적 통합	이슈 기반 + 운영 환경 배포를 함께 고려
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**

SWA-출제 동향(2025)-작업중

- I/F 기반의 Data동기화 문제
- Reactive Programming(비동기) 개념 이해
- MSA, Service Mesh, Cloud Native 개념 이해
- Framework의 역할 및 아키텍처 고려 사항
- WAS 기본 성능 개선: DBCP, TCP Connection Pool, JVM Memory
- Frontend, ECMA(Javascript) Spec 이해: Browser 내 객체 Scope, HTMLDom 구조
- 대량파일처리를 위한 WAS, Kubernetes Ingress 설정 방법
- Spring Cloud 개별 컴포넌트 역할 이해: Config, Consul, Contract, Circuitbreak 등 개념과 특징 이해
- CDN의 기능과 활용/특징에 대한 이해
- (TA) Linux File system 에 대한 이해와 Kerenel Resource 이해(성능에 대한 고려를 위한 내용)
- HTTP Protocol 에 대한 이해: HTTP Method, Header 정보의 기능 이해
- Web 서버 설정: 성능 개선을 위한 설정에 대한 이해, 버퍼 싸이즈 등
- Kubernetes Pod 동작 방식에 대한 이해: QoS 개념, Liveness/Readiness 등 LifeCycle 관리 개념
- Ingress 개념 및 확장 기능 이해(annotation을 통한 세부 기능 제어)
- SAGA 패턴에 대한 개념 이해와 용어 정의
- Springboot Process/Thread 이해와 Block/NonBlock 개념
- JWT 개념 이해
- Java GC 개념과 Option설정(Kubernetes에서 설정하는 방법)
- Java GC 발생시 성능 개선을 위한 Isseu 원인 및 대응 방안
- Spring 기반의 공통 로직 처리 방법: log,공통정보 처리 ==> https://www.baeldung.com/mdc-in-log4j-2-logback
- Linux Resource에 관리 방식에 대한 이해: Kerneal/User 모드와 Java I/O 처리에 할당되는 Resource 처리 영역 등
- 개념적인 성능 개선 고려 사항: https://learn.microsoft.com/ko-kr/troubleshoot/azure/virtual-machines/linux/troubleshoot-performance-bottlenecks-linux
- Kafka/Redis 등 Backing 서비스 Clustering 및 성능관련된 특징에 대한 이해: Pub/Sub, 비동기에 따른 순차거래 특징, 가용성 처리 방식
- CDN의 기능과 용도 이해