ELK 환경에서 - RequestId, MDC 히스토리 기반 로그 트래킹 전략

Skill Stack

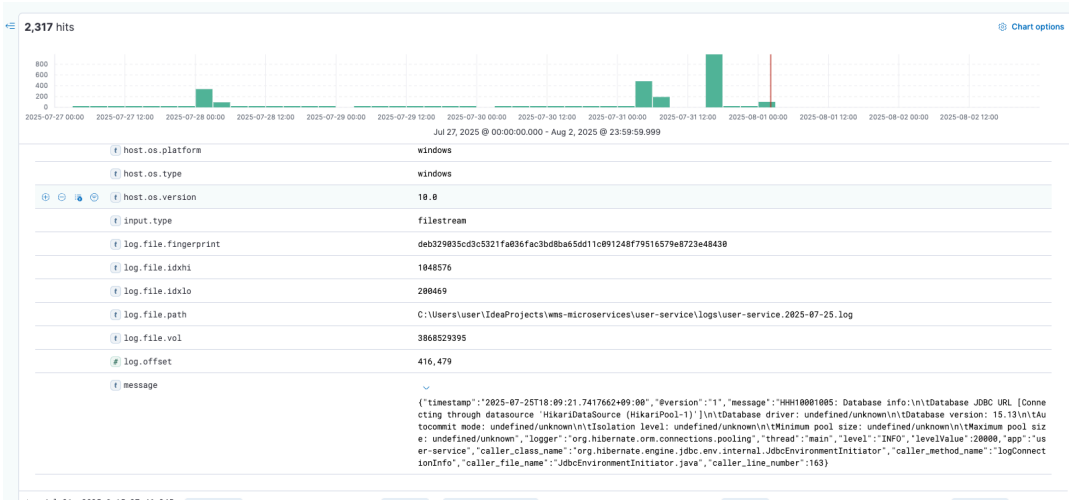
분야	기술 스택
MSA & Gateway	🟦 Eureka, Spring Cloud Gateway, Spring Cloud Netflix
Backend	🟩 Spring Boot 3.x, Spring Web, Spring Security, Spring Data JPA, Spring Retry, Spring Data Redis, Spring Data Elasticsearch, Spring Kafka, Spring Cloud OpenFeign
Database	🟥 MySQL 8.0 (도메인별 DB 분리), PostgreSQL 15
Messaging	🟧 Apache Kafka 2.8.1, Zookeeper 3.8
Cache & Token	🟨 Redis 7, JWT (jjwt 0.12.5)
Logging & Monitoring	🟪 ELK Stack (Elasticsearch 7.17 운영 로그, Elasticsearch 8.13 검색 전용, Logstash, Kibana, Filebeat), Logstash Logback Encoder
Infra & CI/CD	🐳 Docker (docker-compose 기반 MSA 로컬 인프라 구성), GitHub Actions
Libraries & Tools	🛠️ Lombok, Jakarta Validation API, MySQL JDBC Driver

해당글의 소스코드는 다음 깃허브 주소에서 확인 가능합니다!

> https://github.com/JENN-YOO/wms-microservices.git

ELK 스택을 사용한 로그 분석

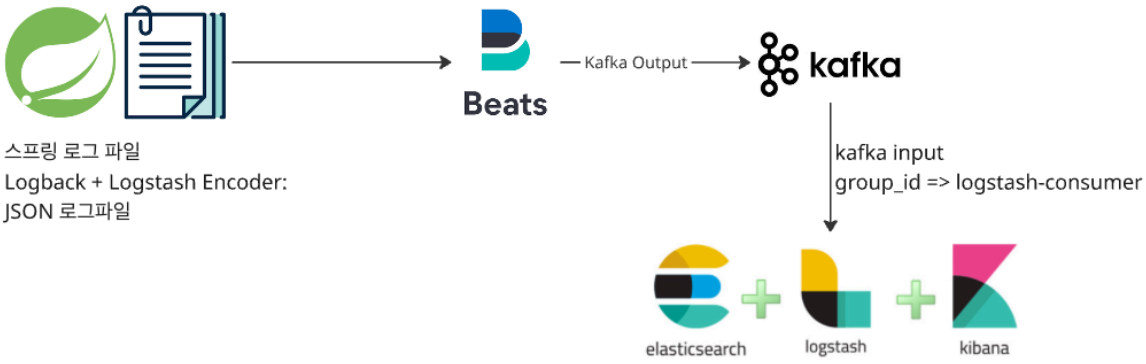
이슈나 에러 발생 시 **원인 분석을 위해 로그 확인은 필수**입니다.
하지만 별도의 분석 환경이 없다면, 파일로 쌓인 로그를 일일이 찾아보며 원인을 추적해야 합니다. **ELK 스택**은 애플리케이션 로그를 **Elasticsearch**에 저장하고, **Kibana 대시보드**를 통해 로그 데이터의 가시성을 제공합니다. 이를 통해 로그를 직접 찾아보는 대신,
Kibana에서 필요한 로그를 즉시 검색·분석하여 보다 **신속한 원인 파악**이 가능합니다.



하지만 위 그림과 같이 **구조화(Structured) 로그 시스템 없이** 사용하면,로그가 단순 텍스트 형태로만 저장되어 Kibana에서 원하는 정보를 **빠르게 검색하기 어렵습니다**.
특정 요청 ID나 에러 유형으로 필터링이 제대로 되지 않아, 결국 필요한 내용을 **직접 눈으로 하나씩 확인**해야 하므로 분석 속도가 느려집니다.
데이터가 많아질수록 장애 흐름 추적이나 통계 집계도 힘들어져,
결국 Kibana를 사용하더라도 **로그 파일을 뒤지는 것과 큰 차이가 없는 상황**이 됩니다.
이러한 로그 분석 환경의 한계를 극복하기 위해, 지금까지 설명한 내용을 토대로
Spring Boot 기반 MSA 환경에서 ELK(Elasticsearch, Logstash, Kibana)와 Filebeat를 활용한 구조화 로그 시스템을 실무 수준으로 구축하였습니다.

구조화(Structured) JSON 로그를 수집하여 **Elasticsearch**에 저장하고, **Kibana** 대시보드로 시각화함으로써 장애 원인 분석, 성능 추적, 트러블슈팅의 **정확도와 속도**를 높였습니다.
이하에서는 구축 전 과정을 단계별로 상세히 설명합니다.

전체 시스템 구성도



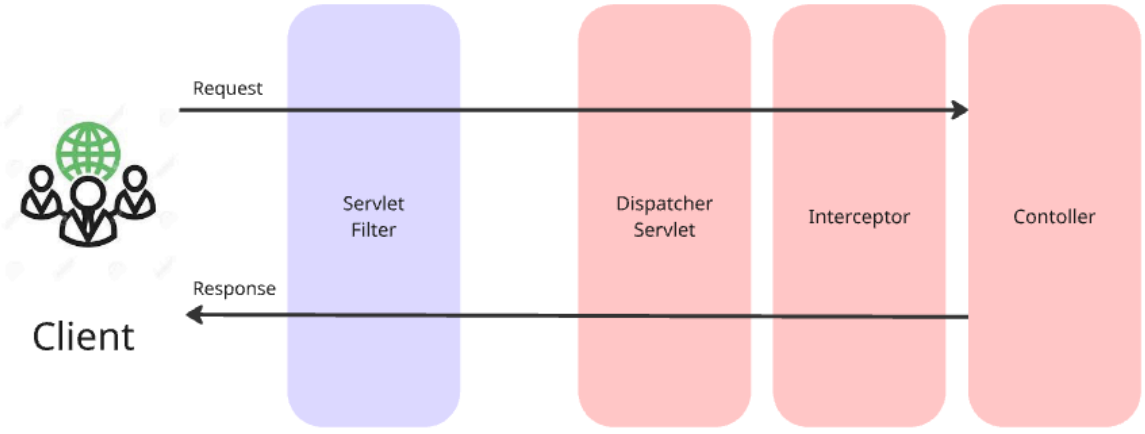
Spring 기반 서비스에서는 **Logback**과 **Logstash Encoder**를 활용해

JSON 형식 로그 파일을 생성합니다.

이 로그 파일은 **Beats(Filebeat 등)**를 통해 실시간 수집되어 **Kafka**로 전달됩니다.

Kafka는 **중앙 로그 허브** 역할을 하며,
Logstash가 지정된 `group_id` (`logstash-consumer`)로 Kafka에서 로그 메시지를 소비합니다.
이후 Logstash는 수집한 로그를 **Elasticsearch**에 저장하고,
Kibana를 통해 로그를 시각화·분석할 수 있습니다..

Request Logging Filter를 활용한 Request 요청 로깅



Java Servlet의 **Servlet Filter**는 클라이언트 요청과 서버 응답 사이에 위치하여, 요청 또는 응답 데이터를 가로채고 처리할 수 있는 기능을 제공합니다.
Spring 환경에서는 **DispatcherServlet** 앞단에서 동작하며, 클라이언트 요청에 대해 인증, 암호화, 로깅 등 부가적인 작업을 수행할 수 있습니다.
이번 구현에서는 **Servlet Filter**를 활용해, 요청의 앞단에서 Request 정보를 로깅하는 기능을 구현합니다. 이 필터를 `RequestLoggingFilter`로 정의하고 적용하겠습니다.

```
public class RequestLoggingFilter extends OncePerRequestFilter {

    private static final Logger logger = LoggerFactory.getLogger("ELK_LOGGER");

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain
filterChain)
            throws ServletException, IOException {

        String requestId = UUID.randomUUID().toString();
        MDCHelper.init(requestId);

        ContentCachingRequestWrapper wrappedRequest = new ContentCachingRequestWrapper(request);
        ContentCachingResponseWrapper wrappedResponse = new ContentCachingResponseWrapper(resp
onse);

        LocalDateTime requestAt = LocalDateTime.now();
        Exception caughtException = null;

        try {
            filterChain.doFilter(wrappedRequest, wrappedResponse);
        } catch (Exception ex) {
            caughtException = ex;
            throw ex;
        } finally {
            LocalDateTime responseAt = LocalDateTime.now();
            long elapseTime = Duration.between(requestAt, responseAt).toMillis();

            RequestLog requestLog = RequestLog.of(
                    requestId,
                    wrappedRequest,
                    wrappedResponse,
                    requestAt,
                    responseAt,
                    elapseTime,
                    MDCHelper.getMetadata(),
                    (caughtException != null) ? getStackTraceAsString(caughtException) : null
            );

            if (caughtException != null) {
```

```
            logger.error("REQUEST_LOG",
                    StructuredArguments.keyValue("requestId", requestLog.getRequestId()),
                    StructuredArguments.keyValue("request", requestLog.getRequest()),
                    StructuredArguments.keyValue("response", requestLog.getResponse()),
                    StructuredArguments.keyValue("metadata", requestLog.getMetadata()),
                    StructuredArguments.keyValue("exception", requestLog.getException())
            );
        } else {
            logger.info("REQUEST_LOG",
                    StructuredArguments.keyValue("requestId", requestLog.getRequestId()),
                    StructuredArguments.keyValue("request", requestLog.getRequest()),
                    StructuredArguments.keyValue("response", requestLog.getResponse()),
                    StructuredArguments.keyValue("metadata", requestLog.getMetadata()),
                    StructuredArguments.keyValue("exception", requestLog.getException())
            );
        }

        wrappedResponse.copyBodyToResponse();
        MDCHelper.clear();
    }
  }

  private static String getStackTraceAsString(Throwable ex) {
    StringBuilder sb = new StringBuilder();
    sb.append(ex.toString()).append("\n");
    for (StackTraceElement elem : ex.getStackTrace()) {
        sb.append("\tat ").append(elem).append("\n");
    }
    return sb.toString();
  }
}
```

`RequestLoggingFilter`의 역할은 단순합니다.
Request 요청의 비즈니스 흐름을 마무리한 뒤, **요청/응답 정보를 로그로 출력**하는 것입니다.
이때 **ElkLogger**를 사용하여 `RequestLog` 데이터 클래스 형태로 로그를 기록합니다.

여기에 **MDC(Mapped Diagnostic Context)**를 추가하면 로그 활용도가 크게 향상됩니다.
MDC는 각 요청마다 고유한 `requestId`(또는 `traceId`)를 생성해 저장하고,
이 값을 모든 로그에 자동 포함시켜줍니다.
덕분에 Kibana 등에서 **requestId를 기준으로 특정 요청의 전체 흐름(모든 로그, 예외, 상태 변화 등)을 한눈에 추적**할 수 있습니다.

또한, 비즈니스 코드 내부에서 `MDCHelper.appendDebug()`와 같이
중요 단계별 상태나 값을 누적 기록하면,해당 요청이 어떻게 처리되었는지에 대한 **debug 히스토리**를 구조화 로그에 함께 남길 수 있습니다.

즉, 단순 요청/응답 기록을 넘어서 **어떤 요청에서, 어떤 흐름으로, 어떤 데이터가 오갔는지**
중간 처리 상태, 에러 발생 지점, 변수 값 변화를 모두 추적할 수 있습니다.

이처럼 **MDC + debug 히스토리**를 결합하면
장애·성능 저하·비정상 요청도 **"요청 단위로 한 번에 추적하고, 빠르게 원인 분석할 수 있는 강력한 구조화 로그 시스템"**이 완성됩니다.

`ContentCaching(Request/Response)Wrapper` 클래스

`RequestLoggingFilter` 코드에서는 `request`와 `response` 객체를 `ContentCaching(Request/Response)Wrapper` 인스턴스로 변환하는 과정을 확인할 수 있습니다. 이는 **HTTP Body Stream이 한 번 읽히면 다시 읽을 수 없기 때문**입니다.

`ContentCaching(Request/Response)Wrapper` 클래스는 내부적으로 읽은 데이터를 **캐싱**하여,
다시 읽더라도 동일한 데이터를 반환할 수 있도록 지원합니다.
따라서 로그 출력을 위해 Body Stream을 읽은 이후에도
동일한 데이터를 반환하여 **요청 정보 유실 없이 정상적인 요청·응답이 가능합니다.**

```
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@JsonInclude(JsonInclude.Include.NON_NULL)
public class RequestLog {
    private String requestId;
    private RequestInfo request;
    private ResponseInfo response;
    private Map<String, Object> metadata;
    private String exception;

    public static RequestLog of(
            String requestId,
            ContentCachingRequestWrapper req,
            ContentCachingResponseWrapper res,
            LocalDateTime reqAt,
            LocalDateTime resAt,
            long elapseTime,
            Map<String, Object> metadata,
            String exception
    ) {
        return RequestLog.builder()
                .requestId(requestId)
                .request(RequestInfo.of(req, reqAt))
                .response(ResponseInfo.of(res, elapseTime, resAt))
                .metadata(metadata)
                .exception(exception)
                .build();
    }
```

```
    @Getter
    @Builder
    public static class RequestInfo {
        private String url;
        private String method;
        private Map<String, String> headers;
        private String body;
        private String requestAt;

        public static RequestInfo of(ContentCachingRequestWrapper req, LocalDateTime reqAt) {
            return RequestInfo.builder()
                    .url(req.getRequestURL().toString())
                    .method(req.getMethod())
                    .headers(getHeaders(req))
                    .body(getBody(req.getContentAsByteArray()))
                    .requestAt(reqAt.toString())
                    .build();
        }
        private static Map<String, String> getHeaders(HttpServletRequest request) {
            Map<String, String> headers = new HashMap<>();
            Enumeration<String> headerNames = request.getHeaderNames();
            while (headerNames.hasMoreElements()) {
                String name = headerNames.nextElement();
                headers.put(name, request.getHeader(name));
            }
            return headers;
        }
        private static String getBody(byte[] buf) {
            return buf.length > 0 ? new String(buf, StandardCharsets.UTF_8) : "";
        }
    }

    @Getter
    @Builder
    public static class ResponseInfo {
        private int status;
        private Map<String, String> headers;
        private String body;
        private int bodySize;
        private long elapseTime;
        private String responseAt;

        public static ResponseInfo of(ContentCachingResponseWrapper res, long elapseTime, LocalDateTim
e resAt) {
            byte[] buf = res.getContentAsByteArray();
            return ResponseInfo.builder()
                    .status(res.getStatus())
                    .headers(getHeaders(res))
```

```
                .body(getBody(buf))
                .bodySize(buf.length)
                .elapseTime(elapseTime)
                .responseAt(resAt.toString())
                .build();
        }
        private static Map<String, String> getHeaders(HttpServletResponse response) {
            Map<String, String> headers = new HashMap<>();
            for (String name : response.getHeaderNames()) {
                headers.put(name, response.getHeader(name));
            }
            return headers;
        }
        private static String getBody(byte[] buf) {
            return buf.length > 0 ? new String(buf, StandardCharsets.UTF_8) : "";
        }
    }
}
```

위와 같이 요청과 응답의 주요 정보를 `RequestLog` 객체로 구조화한 뒤,
이를 **MDC(Mapped Diagnostic Context)**와 결합하면 **로그의 가시성과 추적성**을 한층 강화할 수 있습니다.

특히 다음과 같은 방식으로 활용할 수 있습니다.

1. 각 요청마다 고유한 `requestId` (또는 `traceId`)를 생성해 MDC에 저장
2. 비즈니스 로직 처리 과정에서 `MDCHelper.appendDebug()`를 사용해 **중요 상태·값을 단계별 누적 기록**
3. 누적된 정보를 로그에 함께 남겨, Kibana 대시보드에서 **단일 요청 단위로 전체 처리 흐름을 한눈에 추적**

여기서 MDC란?

MDC는 `org.slf4jorg.slf4j` 패키지에서 제공하는 **로그 메타데이터(Key/Value) 저장소**입니다.
MDC는 하나의 **Context(문맥)** 안에서 로그 메타데이터를 공유하며, **Thread-Local** 기반 라이프사이클을 가집니다.
즉, **요청이 처리되는 동안 할당된 Thread 내에 MDC가 존재**하여,

그 Thread에서 발생하는 모든 로그에 동일한 메타데이터를 자동 포함시킬 수 있습니다.

MDC Helper

```
public class MDCHelper {
    public static final String REQUEST_ID = "requestId";
    public static final String DEBUG = "debug";

    public static void init(String requestId) {
        clear();
```

```
        MDC.put(REQUEST_ID, requestId);
    }

    public static void appendDebug(Class<?> clazz, String message) {
        String debugMsg = MDC.get(DEBUG) != null ? MDC.get(DEBUG) : "";
        String time = LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME);
        MDC.put(DEBUG, debugMsg + "\n" + time + " " + clazz.getSimpleName() + " " + message);
    }

    public static String getDebug() {
        return MDC.get(DEBUG);
    }

    public static void clear() {
        MDC.clear();
    }

    public static java.util.Map<String, Object> getMetadata() {
        java.util.Map<String, Object> map = new java.util.HashMap<>();
        map.put(DEBUG, getDebug());
        return map;
    }
}
```

이 클래스는 로그에서 **요청 단위 식별자(`requestId`)**와 **debug 히스토리**를 자동 관리하는 유틸리티입니다.

- `init(String requestId)`
  - MDC 컨텍스트를 초기화하고 새로운 `requestId`를 저장
  - 각 요청마다 고유하게 추적 가능
- `appendDebug(Class<?> clazz, String message)`
  - 비즈니스 로직 내에서 호출
  - 각 단계의 처리 상황이나 상태 값을 **timestamp**와 함께 누적 기록
- `getDebug()` / `getMetadata()`
  - 누적된 debug 히스토리를 문자열 또는 `Map` 형태로 구조화해 로그에 포함 가능
- `clear()`
  - 요청 종료 후 반드시 호출
  - **쓰레드 재사용 환경**에서 MDC 컨텍스트 데이터가 남지 않도록 정리

구조화 로그를 위한 logback-spring.xml 설정

실제 서비스에서는 로그를 JSON 형태로 남기기 위해

logstash-logback-encoder를 활용한 logback-spring.xml을 다음과 같이 설정했습니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <!-- 콘솔 출력용 STDOUT Appender 추가! -->
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="net.logstash.logback.encoder.LogstashEncoder">
      <customFields>{"app":"user-service"}</customFields>
      <fieldNames>
        <timestamp>timestamp</timestamp>
        <message>message</message>
        <logger>logger</logger>
        <thread>thread</thread>
        <level>level</level>
        <levelValue>levelValue</levelValue>
        <version>@version</version>
        <exception>exception</exception>
      </fieldNames>
      <includeMdc>true</includeMdc>
      <includeContext>true</includeContext>
      <includeCallerData>true</includeCallerData>
    </encoder>
  </appender>

  <appender name="USER_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>/Users/hyunjae/Desktop/WMS Project/logs/user-service.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>{저장하고싶은 컴퓨터 경로}/user-service.%d{yyyy-MM-dd}.log</fileNamePattern
>
      <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder class="net.logstash.logback.encoder.LogstashEncoder">
      <customFields>{"app":"user-service"}</customFields>
      <fieldNames>
        <timestamp>timestamp</timestamp>
        <message>message</message>
        <logger>logger</logger>
        <thread>thread</thread>
        <level>level</level>
        <levelValue>levelValue</levelValue>
        <version>@version</version>
        <exception>exception</exception>
      </fieldNames>
      <includeMdc>true</includeMdc>
      <includeContext>true</includeContext>
      <includeCallerData>true</includeCallerData>
    </encoder>
  </appender>
```

```
    <root level="INFO">
        <appender-ref ref="USER_FILE" />
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

이렇게 설정하면, 서비스 동작 중 발생하는 모든 로그가 **콘솔(터미널)과 별도의 로그 파일(user-service.log)에 동시에 기록**됩니다.

각 로그에는 다음 정보가 **JSON 형태**로 포함됩니다.

- 로그가 발생한 서비스명(`app` 필드, 예: `user-service`)
- 발생 시각(`timestamp`)
- 로그 레벨(level)
- 실행 스레드(`thread`)
- 예외 정보(`exception`)

또한 `<includeMdc>true</includeMdc>` 설정을 통해
MDC에 저장된 `requestId`와 debug 히스토리도 모든 로그에 자동 포함됩니다.
이 덕분에 Kibana 같은 대시보드에서 **즉시 검색·필터링·분석**이 가능합니다.

(예시)로그인 서비스에서 흐름 처리

```
public LoginResponse login(LoginRequest loginRequest) {
        MDC.put("loginId", loginRequest.getUsername());
        MDCHelper.appendDebug(LoginService.class, "로그인 처리 시작"); // ← 1. 시작 기록

        try {
            Member member = memberRepository.findByLoginId(loginRequest.getUsername())
                    .orElseThrow(() -> new IllegalArgumentException("가입되지 않은 아이디 입니다."));
            MDCHelper.appendDebug(LoginService.class, "회원 정보 조회 성공: " + member.getLoginId()); // ←
2. 회원 조회 성공 기록

            if(member.getStatusCode() != 1) {
                MDCHelper.appendDebug(LoginService.class, "계정 비활성화 상태. StatusCode: " + member.getS
tatusCode()); // ← 3. 상태 체크 기록
                throw new IllegalArgumentException("활성화되지 않은 계정입니다.");
            }
            MDCHelper.appendDebug(LoginService.class, "계정 활성화 상태 확인 (정상)"); // <-- 3. 상태 체크 기록

            if (!passwordEncoder.matches(loginRequest.getPassword(), member.getPassword())) {
                MDCHelper.appendDebug(LoginService.class, "비밀번호 불일치"); // ← 4. 비밀번호 체크 기록
                throw new IllegalArgumentException("잘못된 비밀번호입니다.");
            }
            MDCHelper.appendDebug(LoginService.class, "비밀번호 일치 확인"); // ← 4. 비밀번호 체크 기록
```

```
            // 실제 Role(권한) 정보 조회
            List<MemberRole> memberRoles = memberRoleRepository.findByMember(member);
            List<String> roleNames = memberRoles.stream()
                    .map(mr -> mr.getRole().getRoleName())
                    .collect(Collectors.toList());
            String roles = String.join(",", roleNames); // ex) "ADMIN,USER"
            MDCHelper.appendDebug(LoginService.class, "사용자 권한 조회 완료: " + roles); // ← 5. 권한 조회 기
록

            String accessToken = jwtTokenProvider.createToken(member.getLoginId(), roles);
            String refreshToken = jwtTokenProvider.createToken(member.getLoginId(), roles);
            MDCHelper.appendDebug(LoginService.class, "Access/Refresh 토큰 생성 완료"); // ← 6. 토큰 생성
기록

            // Redis에 리프레시 토큰 저장 (만료 시간 설정)
            redisTemplate.opsForValue().set(
                    member.getLoginId(),
                    refreshToken,
                    jwtTokenProvider.getRefreshTokenValidityInMilliseconds(),
                    TimeUnit.MILLISECONDS
            );
            MDCHelper.appendDebug(LoginService.class, "Refresh 토큰 Redis 저장 완료"); // ← 7. Redis 저장 기
록

            MDCHelper.appendDebug(LoginService.class, "로그인 처리 성공, 응답 반환"); // ← 8. 최종 성공 기록
            return new LoginResponse(accessToken, refreshToken);
        } catch (IllegalArgumentException e) {
            logger.error("Login failed: {}", e.getMessage());
            MDCHelper.appendDebug(LoginService.class, "로그인 처리 중 오류 발생: " + e.getMessage()); // ←
오류 발생 기록
            throw e; // Re-throw the exception so it can be handled by Spring's exception handling
        } finally {
            MDC.remove("loginId");
        }
    }
```

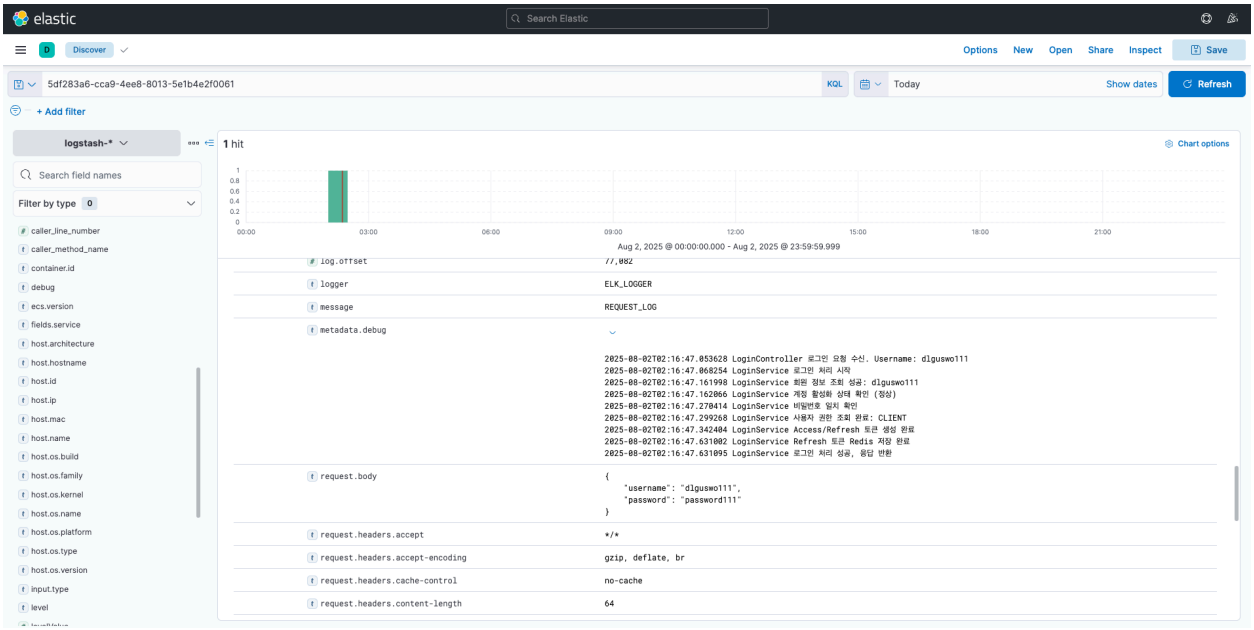
위 예시는 **로그인 서비스**에서 **MDC + debug 히스토리 누적 방식**을 실제 적용한 코드입니다.

1. **로그인 요청 수신 시**
   - `MDC.put("loginId", ...)`로 현재 로그인 시도 중인 사용자의 ID를 로그 컨텍스트에 저장
   - 각 단계에서 `MDCHelper.appendDebug()`를 호출하여 비즈니스 흐름의 주요 포인트를 **debug 히스토리에 순차 기록**
   - 예시 :
     - 회원 정보 조회 성공
     - 계정 상태 체크

- 비밀번호 일치 여부 확인
- 사용자 권한 조회
- 토큰 발급 및 Redis 저장

2. **오류 발생 시**
   - `MDCHelper.appendDebug()`로 **오류 메시지와 발생 지점**을 기록
   - `logger.error(...)`로 에러 로그 출력 → 장애 상황의 상세 추적 가능
3. **finally 블록**
   - 사용이 끝난 MDC 필드를 `MDC.remove("loginId")`로 제거
   - 쓰레드 재사용 환경에서 **컨텍스트 데이터 꼬임 방지**

그리고 다음과 같이 Kibana에서 `metadata.debug` 필드와 값을 확인할 수 있습니다.



이슈 분석에 필요한 정보가 있다면 언제든지 `appendDebug()` 또는 `alsoAppendDebug()` 함수를 호출해, 그 순간의 상태나 값을 그대로 기록할 수 있습니다.

이렇게 누적된 **debug 히스토리는 Kibana에서** `metadata.debug` **필드로 한눈에 확인 가능**하며, 단순히 요청/응답 정보뿐만 아니라, 코드에서 "여기가 중요하다!"고 **표시한 모든 지점을 그대로 로그에 남기고 실제 분석에 활용할 수 있습니다.**

덕분에 예전처럼 로그를 뒤적이며 *"대체 어디서 꼬였지?"* 고민하는 시간이 줄었고,
문제 발생 흐름과 데이터 변화를 한 번에 확인할 수 있어 이슈 분석과 트러블슈팅이 훨씬 수월해졌습니다.

마치며...

처음에는 텍스트 로그를 하나하나 뒤지는 것이 당연하다고 생각했습니다.
하지만 **구조화된 로그 시스템**을 구축한 뒤로는, 예전 방식으로는 돌아갈 수 없을 만큼 편리함을 느끼고 있습니다.

실제로 장애가 발생했을 때 몇 시간이 걸리던 원인 분석이 **단 몇 분**으로 줄었고,로그 데이터를 기반으로 더 **객관적인 분석**과 **신속한 대응**이 가능해졌습니다.

무엇보다 중요한 점은, 이 시스템을 구축하는 과정에서 단순히 도구를 다루는 법을 배우는 것을 넘어 **문제를 바라보는 시각**과 **대응 능력**까지 한 단계 성장할 수 있었다는 것입니다.
아직 보완할 부분은 남아 있지만, 제가 겪은 경험과 고민이 여러분께도 작은 도움이 되었으면 합니다.

읽어주셔서 감사합니다.