ChatGPT

Kafka와 WebSocket 기반 실시간 웹 알림 시스템 설계

전체 시스템 아키텍처 (이벤트 흐름)

아키텍처 개요: 구매 시스템의 여러 서비스에서 발생하는 이벤트(예: 구매 신청)를 **Apache Kafka** 메시지 브로커를 통해 전달하고, **Spring Boot 3.x** 기반의 알림 서비스(이벤트 수신 서버)가 이를 소비하여 **WebSocket**으로 다수 사용자에게 실시간 전송하는 구조입니다. 전체 흐름은 다음과 같습니다 [1]:

1. **이벤트 발생 및 Kafka 발행:** 구매 시스템의 각 서비스(프로듀서)가 특정 비즈니스 이벤트 발생 시 Kafka **토픽**에 이벤트 메시지를 발행합니다 [1]. Kafka는 분산 메시징 큐로서 다수 생산자의 이벤트를 수용하고 저장합니다 [2]. 예를 들어 "주문 생성" 이벤트가 발생하면 이를 포함한 메시지를 Kafka에 **Producer API**로 전송합니다.
2. **알림 서비스에서 Kafka 이벤트 수신:** Spring Boot로 구현된 **알림 서비스**는 Kafka **컨슈머**로 동작하며, 지정된 토픽을 **구독**(subscribe)합니다. Kafka 브로커는 발행된 이벤트를 해당 토픽의 모든 컨슈머 인스턴스에 전달합니다 [1]. 알림 서비스는 이벤트를 수신하면 필요한 처리를 수행하고 알림으로 변환합니다. 이 과정에서 이벤트 내용을 토대로 알림 메시지(예: 사용자에게 보여줄 제목, 내용 등)를 생성합니다.
3. **알림 로그 저장:** 생성된 알림은 실시간 전송과 함께 **NoSQL 데이터베이스**(예: MongoDB)에 로그로 저장됩니다. MongoDB와 같은 NoSQL을 사용하는 이유는 JSON 형태의 알림 데이터를 유연하게 저장하고, 쓰기 및 조회를 빠르게 처리하기 위함입니다 [3]. 이를 통해 사용자가 나중에 접속하더라도 이전 알림을 조회하거나, 알림의 감사(audit) 기록을 남길 수 있습니다.
4. **WebSocket 실시간 전송:** 알림 서비스는 해당 알림을 즉시 **WebSocket**을 통해 클라이언트들에게 전송합니다. Spring Boot 내에서 **WebSocket Endpoint**를 통해 다수의 클라이언트와 지속적인 연결을 유지하고 있으며, 이벤트 수신 후 **수초 이내**(사실상 실시간으로) 특정 사용자 또는 관련 사용자 그룹에게 알림 메시지를 push 합니다 [4]. WebSocket 연결은 풀 듀플렉스 통신이 가능하여 서버가 클라이언트 요청 없이도 능동적으로 데이터를 보낼 수 있으므로, 폴링(polling) 없이도 실시간성 확보가 가능합니다 [2].
5. **프론트엔드 클라이언트 처리:** 각 최종 사용자는 **Vue.js 프론트엔드 모듈**을 통해 WebSocket에 연결되어 있습니다. 클라이언트는 수신한 알림을 실시간으로 화면에 표시하고, 새로운 알림 도착 시 시각적인 알림(UI 팝업이나 뱃지 표시 등)으로 사용자에게 전달합니다. 만약 사용자가 부재 중이어서 WebSocket을 통한 실시간 수신을 못 한 경우에도, 앞서 저장된 MongoDB의 알림 로그를 기반으로 이후 접속 시 알림 내역을 복원할 수 있습니다.

   구성 요소 요약: 이벤트 프로듀서(구매 시스템 서버) → **Kafka 토픽** → 이벤트 컨슈머(알림 서비스) → **MongoDB 저장** & **WebSocket 브로드캐스트** → 다수 **사용자(Vue.js)**에게 실시간 알림 전달 [1]. 이러한 이벤트 중심의 비동기 흐름은 구성 요소 간 느슨한 결합과 실시간 데이터 흐름을 모두 만족시킵니다.

백엔드 기술 스택 구성 및 주요 라이브러리

1. Spring Boot 3.x 알림 서비스: 백엔드 핵심은 Spring Boot 3.x로 구현된 **알림 서비스**입니다. Spring Boot 3은 최신 Spring Framework 6 기반으로, **Java 17** 이상을 요구하며 향상된 성능과 보안 패치를 포함합니다. 이 서비스는 WebSocket 엔드포인트와 Kafka 소비 로직을 포함한 마이크로서비스로 동작합니다. 주요 의존성은 다음과 같습니다:

- **Spring for Apache Kafka:** Kafka 연동을 위해 `spring-kafka` 라이브러리를 사용합니다. 이는 Spring Boot에서 **@KafkaListener** 등을 활용해 손쉽게 Kafka **Consumer**를 구현하도록 지원합니다. Kafka 클러스터 (예: 3개 이상의 브로커로 구성 [5])와 연동되며, 구성에는 Kafka 브로커 URL, 토픽 이름, 소비자 그룹ID 등을 설정합니다.

- **Spring WebSocket (STOMP 지원):** 실시간 알림 전송을 위해 `spring-websocket`과 `spring-messaging`을 사용하여 **STOMP** 프로토콜 기반의 WebSocket를 구현합니다. WebSocket은 기본적으로 이중통신 채널만 제공하므로, 구독/발행(pub-sub) 및 메시지 라우팅을 위해 STOMP 프로토콜을 상위 레벨에서 사용합니다 [6]. Spring은 STOMP 엔드포인트를 쉽게 구성하고, `@MessageMapping`이나 `SimpMessagingTemplate` 등을 통해 특정 토픽으로 메시지를 브로드캐스트할 수 있습니다. **SockJS**도 함께 설정하여 웹소켓을 지원하지 않는 환경(레거시 브라우저)에서도 폴백(Long Polling)으로 동작하도록 할 수 있습니다.
- **NoSQL 데이터베이스 (MongoDB):** 알림 내역 저장소로 **MongoDB**를 채택합니다. MongoDB는 스키마리스 JSON 문서 저장에 적합하여 알림의 다양한 형태를 유연하게 기록할 수 있습니다 [3]. Spring Data MongoDB를 통해 알림 데이터를 저장/조회하며, 필요에 따라 **TTL Index** 등을 활용해 오래된 알림을 자동 삭제하거나, **읽음 여부** 필드를 두어 사용자가 읽은 알림을 구분하는 등의 기능을 구현합니다. MongoDB는 수평 확장과 Replica Set에 의한 고가용성을 지원하므로, 알림 로그 저장소의 안정성을 높일 수 있습니다.
- **기타:** 로그 관리 및 모니터링을 위해 Spring Boot **Actuator**를 포함하여 애플리케이션 상태, 메시지 처리율 등의 메트릭을 수집합니다. 필요한 경우 Kafka와 연동된 Confluent Schema Registry(avro 등)나, JSON 직렬화를 위한 Jackson 라이브러리, 그리고 보안 강화를 위한 Spring Security(JWT 인증 활용)를 구성할 수 있습니다. 예컨대 WebSocket 연결시 **JWT 토큰**을 전달받아 핸드쉐이크시에 인증하는 방식을 적용하면 알림 수신 권한 제어를 할 수 있습니다.

  Kafka 구성: Kafka 토픽은 알림 유형별로 분리할 수 있으며, 파티션은 트래픽과 소비자 수에 맞게 설계합니다. 예를 들어 단일 토픽으로 모든 알림을 처리하되, 키를 사용자 ID로 지정하면 사용자 단위로 순서를 보장할 수 있습니다. 소비자 **group.id**는 알림 서비스 인스턴스마다 **고유하게 설정**하여 (예: `notification-service-instance-1`, `...-2` 또는 UUID 사용 [7]) **모든 인스턴스가 동일한 이벤트를 각각 수신**하도록 합니다. 이렇게 하면 WebSocket 서버를 여러 대 띄웠을 때도 각 서버가 Kafka로부터 이벤트를 받아 자체 연결된 클라이언트들에게 전달할 수 있습니다 [8]. (동일한 그룹으로 설정하면 Kafka가 소비를 **부하분산**시키므로 한 인스턴스에만 이벤트를 보내게 되므로, 실시간 푸시 시스템에서는 그룹을 다르게 해서 **팬아웃**하는 방식이 유용합니다.)

2. 기술 스택 요약: Spring Boot 3 (Spring WebSocket/STOMP) + Apache Kafka + MongoDB 조합이 핵심입니다. 이러한 스택은 **대규모 실시간 서비스**에서도 활용되는 검증된 조합으로 [9], 확장성과 내결함성을 갖춘 알림 시스템을 구축할 수 있습니다. Kafka는 생산자와 소비자를 분리하여 느슨한 결합의 **이벤트 드리븐 아키텍처**를 만들고, **대용량 메시지 처리와 재시도**에 강점이 있습니다 [10] [9]. Spring Boot는 경량 서버 구현을 돕고 WebSocket 통합을 지원하며, MongoDB는 알림 데이터 영속화에 적합합니다.

프론트엔드(Vue.js) 구성 및 알림 수신 구조

프론트엔드는 Vue.js 기반의 독립 모듈로 구현되며, 기존 웹 시스템과 별개로 **플러그인 형태**로 삽입될 수 있습니다. 이는 예를 들어 `<script>`로 번들된 Vue 컴포넌트를 로드하여 간단히 기존 사이트에 알림 기능을 주입하는 방식입니다. 주요 설계 요소는 다음과 같습니다:

- **WebSocket 연결 관리:** Vue.js 모듈은 웹 애플리케이션이 로드될 때 백엔드 알림 서비스의 WebSocket 엔드포인트에 연결을 수립합니다 (예: `ws://notification-server.example.com:8080/notify` 경로). 연결에는 **STOMP 프로토콜**을 사용하여, Vue.js에서 **@stomp/stompjs** 라이브러리나 SockJS 클라이언트를 활용해 subscribe 할 수 있습니다. STOMP를 사용하면 클라이언트가 특정 **토픽(destination)**을 구독하고 메시지를 수신할 수 있어 구현이 단순해집니다 [6]. 예를 들어 모든 사용자가 공통 알림을 받는 경우 Vue 클라이언트는 `/topic/notifications`를 구독하고, 개인별 알림의 경우 `/user/{사용자ID}/queue/notifications`와 같은 엔드포인트를 구독하도록 합니다 (Spring의 user destination 기능 활용).
- **컴포넌트 UI/UX:** 알림 모듈은 화면에 **알림 아이콘**(벨 모양 등)을 표시하고 신규 알림 수를 배지로 보여줄 수 있습니다. 사용자가 아이콘을 클릭하면 최근 알림 목록이 드롭다운으로 표시되는 컴포넌트가 나오며, Vue.js로 목록 렌더링 및 상태관리를 합니다. 실시간으로 들어온 메시지는 Vuex 등 상태관리 스토어를 통해 리스트에 추가되고, 해당 컴포넌트가 반응형으로 UI를 업데이트합니다. 각 알림 항목에는 내용, 시간, (선택적으로) 액션 버튼

등이 포함될 수 있습니다. 알림을 클릭하면 관련 화면으로 이동하거나, 읽음으로 표시되어 배지 카운트가 감소하는 등의 UX를 구현합니다.

- **기존 시스템과의 통합:** 이 Vue.js 모듈은 독립적으로 동작하므로, 호스트 애플리케이션과는 **API 통신**으로 연동될 수 있습니다. 예를 들어 초기 로딩 시 백엔드 알림 서비스의 REST API (`/api/notifications?since=...`)를 호출하여 과거 알림 내역(예: 안 읽은 알림)을 가져와 초기 목록을 구성할 수 있습니다. 또한 사용자 인증 정보를 호스트로부터 전달받아 WebSocket 연결에 사용합니다. 보안을 위해 WebSocket 연결 시 **JWT 토큰**이나 세션ID를 쿼리파라미터 또는 헤더로 보내고, 서버에서 이를 검증하여 해당 사용자의 세션을 식별하도록 합니다. 이렇게 하면 Vue 모듈이 여러 시스템에 이식되더라도, 각 시스템의 인증 맥락을 활용해 **본인에게 해당하는 알림만** 수신할 수 있습니다.
- **에러 처리 및 재연결:** 프론트엔드에서는 연결이 끊어지거나 오류가 발생할 경우를 대비해 **재연결 로직**을 넣습니다. 예를 들어 WebSocket 연결이 오류/종료 이벤트가 발생하면 일정 간격(backoff 전략)으로 재시도를 하고, 성공 시 다시 필요한 토픽을 구독합니다. 또한 알림 수신 중 오류가 있더라도 사용자에게 치명적 영향을 주지 않도록, 실패한 경우 콘솔 로그 및 사용자에게 연결 문제가 있음을 안내하는 등 부가적인 UX 처리를 고려합니다.

요약하면, Vue.js 모듈은 **WebSocket-STOMP 클라이언트**로서 동작하며, 이벤트 수신 시 자동으로 UI를 업데이트하여 사용자에게 **실시간 알림**을 제공합니다. 이 모듈은 독립적이므로 여러 웹 시스템에 손쉽게 포함시킬 수 있고, 공통 알림 서버에 연결해 이벤트를 받아볼 수 있습니다.

실시간성과 확장성 확보를 위한 고려사항

실시간성: 해당 시스템은 **End-to-End**로 실시간 처리를 염두에 두고 설계되었습니다. WebSocket 통신을 통해 서버에서 이벤트 발생 직후 곧바로 클라이언트로 push할 수 있으므로, **수 초 이내**에 알림을 전달한다는 요구사항을 충족할 수 있습니다. Kafka 역시 높은 처리량과 낮은 지연으로 유명하며, **Producer -> Broker -> Consumer** 경로가 수 ms 단위로도 처리될 수 있습니다. 따라서 병목 없이 튜닝된 환경이라면 이벤트 발생부터 사용자 브라우저 전달까지 보통 1초 미만으로도 가능할 것입니다. 추가로 고려할 사항:

- **Kafka 전송 지연 최적화:** Kafka Producer의 `linger.ms`나 batch size 등을 조정해 가능한 한 바로바로 이벤트를 전송하도록 설정할 수 있습니다. Consumer 측도 `max.poll.interval.ms` 등을 조절해 지연을 최소화합니다. 그러나 기본 설정으로도 실시간 알림에는 큰 문제가 없으며, Kafka는 이벤트를 **발생 순서대로 저장 및 전달**하여 순서를 보장합니다 [10]. 덕분에 한 사용자가 연속으로 발생한 알림이라도 뒤바뀌지 않고 순서대로 처리됩니다.
- **WebSocket 전송 최적화:** WebSocket은 OSI Layer 5(세션 계층)에서 지속 커넥션을 유지하므로 연결 설정 이후에는 HTTP보다 오버헤드가 적습니다. 다만, **Ping/Pong** 등 KeepAlive 설정을 통해 중간에 연결이 죽지 않도록 하고, 메시지 크기를 가급적 작게 (필요한 데이터만 포함) 설계하여 네트워크 부하를 최소화합니다. 또한 WebSocket 처리 스레드풀이나 Netty 이벤트루프(만약 Spring WebFlux(Netty) 기반으로 구현한다면) 등의 적절한 튜닝을 통해 동시 다발 메시지 전송에도 성능을 유지합니다.
- **백엔드 부하 분산 및 확장:** 사용자가 300~400명 수준이라 현재 단일 서버로도 충분히 감당 가능하지만, **향후 사용자가 증가하거나 이벤트 발생 빈도가 높아질 경우**를 대비해 확장 구조를 마련합니다. Kafka를 이미 도입했으므로 알림 서비스 인스턴스를 수평 확장(scale-out)하기가 용이합니다 [9]. 추가 인스턴스를 띄우고 각 인스턴스에 고유한 Consumer 그룹명을 주면, **모든 인스턴스가 동일 이벤트를 받아** 자신의 WebSocket 세션들에게 알림을 전송합니다 [8]. 이로써 서버가 여러 대여도 한 이벤트에 **여러 서버의 WebSocket 세션들이 동시에 알림**을 받을 수 있어 일관성이 유지됩니다 [8]. 로드밸런서를 통해 WebSocket 연결을 각 인스턴스로 분산시킬 수 있으며, Spring Session 등을 활용해 Sticky Session이 아니더라도 상관없도록 설계합니다 (Kafka 팬아웃으로 인해 동일한 메시지를 모든 노드가 갖게 되므로 굳이 특정 세션에 집착할 필요가 없습니다).
- **다중 채널/이벤트 분리:** 실시간성을 높이기 위해 한 서버에서 너무 많은 작업을 직렬로 처리하지 않도록 합니다. 알림 종류에 따라 Kafka 토픽이나 소비 쓰레드를 분리하면 특정 무거운 이벤트 처리가 다른 알림에 영향 주는 것을 막을 수 있습니다. 예를 들어 대량의 시스템 알림과 개인 DM알림을 분리된 토픽/Consumer로 처리하면 보다 안정적입니다.
- **SSE 대안:** 참고로, 단방향 실시간 푸시에 한정한다면 **Server-Sent Events (SSE)**도 고려 가능하지만, SSE는 브라우저 표준 지원이 제한적이고, 다중 구독이나 이중 통신이 불가하다는 제약이 있습니다. 현재 요구사항은

WebSocket이 명시되어 있으므로 WebSocket을 사용하지만, **알림 전용 일방향 스트림**에는 SSE도 가벼운 대안이 될 수 있습니다 (Spring MVC의 SseEmitter 등 지원).

확장성: Kafka 도입으로 전체 시스템은 **이벤트 드리븐 확장성**을 확보했습니다. 즉, 생산자와 소비자가 분리되어 있어 각각 독립적인 확장이 가능합니다 [9]. 구매 시스템 측 서비스가 늘어나더라도 (프로듀서 증가) Kafka 토픽에 메시지를 넣기만 하면 되므로 알림 서비스와의 결합도가 낮습니다 [4]. 알림 수신 서비스 측도 앞서 언급한 대로 인스턴스 추가만으로 구독 소비자를 늘릴 수 있어, **수백, 수천 명 이상의 동시 접속자도 수평 확장으로 대응**할 수 있습니다. Kafka 자체도 파티션 증설이나 브로커 증설로 처리량을 높일 수 있고, MongoDB 역시 샤딩이나 Replica Set으로 확장 가능합니다. 또한 WebSocket 연결 300~400개는 현대 서버 스펙에서 큰 부하가 아니지만, 만약 수만 단위로 늘어난다면 한 서버에 너무 많은 소켓을 물리지 않고 여러 대로 나누어 부담을 줄이는 식으로 확장합니다.

마지막으로, **지속적인 성능 모니터링**과 부하 테스트를 통해 병목 지점을 찾아 튜닝하는 것이 중요합니다. 예컨대 Kafka Consumer의 **처리량 대비 MongoDB 쓰기 속도**가 느리다면 비동기 큐를 두거나 Batch Insert를 고려하고, WebSocket 전송이 느리다면 메시지 크기나 네트워크 대역폭을 점검해야 합니다.

Kafka 기반 이벤트 처리 및 WebSocket 세션 관리 전략

이 부분에서는 백엔드 알림 서비스가 **Kafka 메시지를 어떻게 처리하여 WebSocket으로 전달하는지**, 그리고 **여러 사용자 세션을 어떻게 관리하고 브로드캐스트하는지**를 구체적으로 설명합니다.

1. Kafka 이벤트 처리 흐름: 알림 서비스는 Spring Kafka의 `@KafkaListener`를 통해 특정 토픽의 메시지를 지속적으로 감시합니다. 예를 들어 `@KafkaListener(topics = "purchase-events", groupId = "notify-service-1")`와 같이 리스너를 설정하면, 구매 시스템에서 발행한 이벤트가 해당 토픽에 들어올 때마다 자동으로 메서드가 호출됩니다. 이벤트 메시지는 JSON 직렬화된 도메인 객체 (`NotificationEvent` 등) 형태로 수신하며, 필요시 역직렬화(Deserialization)를 거쳐 Java 객체로 변환됩니다 [11] [12].

- **비즈니스 로직:** 수신한 이벤트 객체를 분석하여 어떤 사용자들에게 어떤 내용의 알림을 보낼지 결정합니다. 예를 들어 구매 승인 이벤트라면 해당 구매를 신청한 사용자에게 "구매 요청이 승인되었습니다"라는 알림을 준비하거나, 여러 관리자들에게 "새로운 구매 요청 발생" 알림을 생성할 수 있습니다. 이 때 알림 제목, 내용, 링크 등의 **Notification DTO**를 구성합니다.
- **데이터 영속화:** 앞서 설명한대로 이 DTO를 MongoDB에 저장하여 영구 보관합니다. MongoDB 저장 시 `_id` (primary key), `userId` (수신 대상 사용자 또는 그룹 식별자), `message`, `timestamp`, `read` 여부 등의 필드를 함께 기록합니다. 이 작업은 보통 매우 빠르게 완료되지만, 만약 저장 지연이 생겨도 WebSocket 푸시에는 큰 영향이 없도록 설계합니다. (필요하다면 Kafka Consumer 쓰레드와 별개로 비동기 작업으로 저장을 수행할 수도 있습니다. 그러나 대부분의 경우 소량 문서 쓰기는 ms 수준으로 빨리 완료됩니다.)
- **오프셋 커밋:** Kafka는 Consumer 그룹별로 메시지 오프셋을 관리합니다. Spring Kafka에서는 리스너 메서드가 예외 없이 완료되면 자동으로 오프셋 커밋(ack)을 처리하거나, 수동 커밋 설정 시 `Acknowledgment.acknowledge()`를 호출합니다. 알림 처리가 성공하면 해당 오프셋을 커밋하여 Kafka에 해당 메시지 처리가 완료됐음을 알립니다. 만약 처리 중 오류가 발생하면 커밋이 보류되고, 재시도로 직이나 DLQ(Dead Letter Queue)로 분기할 수 있습니다.

2. WebSocket 세션 관리 및 메시지 전송: 알림 서비스는 다수의 사용자 WebSocket 세션을 관리합니다. Spring Boot에서는 통상 **STOMP 프로토콜**을 사용하면 세션 관리와 구독 관리가 간편해집니다. STOMP를 사용하지 않는다면 직접 `WebSocketSession` 객체를 리스트로 들고 있고 `session.sendMessage(TextMessage)` 식으로 보내야 하지만, Spring의 경우 **SimpMessagingTemplate**을 이용해 추상화된 방식으로 메시지를 전송합니다. 세션 관리는 다음과 같은 전략을 취합니다:

- **세션 및 구독 관리:** 사용자가 WebSocket 연결을 맺으면 Spring은 해당 세션에 고유 ID와 Principal(사용자 식별 정보)을 부여합니다. STOMP를 쓰는 경우, 클라이언트가 `SUBSCRIBE` frame을 보내 특정 destination을 구독하게 되며, 서버에서는 어떤 세션이 어떤 destination을 구독중인지 매핑을 관리합니다.

예를 들어 사용자 A가 `/user/A/queue/notifications`를 구독하면, Spring은 세션ID-X가 사용자A임을 알고 해당 세션을 그 destination의 구독 리스트에 등록합니다. 별도로 Redis 같은 스토리지를 쓰지 않더라도, 단일 인스턴스 내에서는 이 세션-구독 정보가 메모리에 유지됩니다. (Spring WebSocket의 SimpleBroker가 이 역할을 수행)

- **브로드캐스팅 전략:** 알림 발생 시 대상이 **전체 사용자라면**, 서버는 `messagingTemplate.convertAndSend("/topic/notifications", msg)` 형태로 topic에 메시지를 발행합니다. 그러면 해당 토픽을 구독한 모든 세션에 알림이 전달됩니다. 대상이 **특정 사용자라면**, `convertAndSendToUser(사용자ID, "/queue/notifications", msg)` 형태로 전송하여 해당 사용자 세션에게만 일대일 전달합니다. Spring은 자동으로 해당 사용자의 세션을 찾아 메시지를 라우팅합니다 (여러 세션이 있을 경우 모두 전송). 예를 들어, Kafka Consumer 코드에서 SimpMessagingTemplate을 사용하여 `template.convertAndSend("/topic/someEvent", payload)` 또는 `template.convertAndSendToUser(userId, "/queue/notifications", payload)`를 호출함으로써 웹소켓 메시지를 push 합니다 [13].
- **다중 인스턴스 시 브로드캐스트:** 만약 알림 서비스가 여러 인스턴스로 구성되어 있을 경우, 앞서 설명한 Kafka 소비 구조에 따라 **각 인스턴스가 동일 이벤트를 수신**하게 됩니다. 그러면 각 인스턴스는 자기한테 연결된 세션 중 해당 토픽/사용자를 구독 중인 세션에만 메시지를 보냅니다. 예컨대 사용자 A는 서버1에 연결, 사용자 B는 서버2에 연결되어 있고 A에게 갈 알림이 발생한 경우, 서버1과 서버2 모두 Kafka에서 이벤트를 받지만, 서버1에만 A의 세션이 있으므로 서버1에서만 전송이 이루어지고 서버2에서는 대상 세션이 없어 전송을 스킵하게 됩니다 [8]. 이런 **팬아웃 + 세션 필터링** 방식으로 서버 간 세션 동기화 문제를 해결하며, Kafka가 일종의 외부 브로커로 동작하여 **모든 서버의 in-memory broker를 동기화**해주는 셈입니다 [8].
- **상태 유지 vs 무상태:** WebSocket 서버는 본질적으로 클라이언트 연결을 **상태적으로** 유지하지만, 위와 같은 아키텍처 덕분에 각 서버 간 세션 정보를 공유하지 않아도 동작이 가능합니다. 필요한 경우를 대비해, 세션 정보(예: 접속중인 사용자 목록)를 Redis와 같은 인메모리 저장소에 저장하여 관리할 수도 있지만, 시스템 복잡도가 증가합니다. 현재 규모에서는 Spring의 기본 세션 관리로 충분하며, 확장 시에도 Kafka 브로드캐스트 방식으로 대응하므로 굳이 세션 클러스터링을 하지 않아도 됩니다.

3. 메시지 포맷 및 프로토콜: 알림 메시지는 일반적으로 JSON 포맷으로 직렬화되어 WebSocket으로 전달됩니다. 예를 들어 `{ "type": "ORDER_APPROVED", "message": "주문이 승인되었습니다", "timestamp": "2025-05-22T15:50:00", "data": {...} }` 같은 구조로 보낼 수 있습니다. 클라이언트에서는 이 JSON을 파싱하여 화면에 적절히 표시합니다. 프로토콜 측면에서, STOMP를 사용하면 프레임 헤더에 destination과 간단한 메타정보가 붙고, 바디에 위 JSON이 실려 전송됩니다. STOMP 프레임은 서버와 클라이언트 양측에서 처리해주므로 개발자는 비즈니스 데이터 생성에 집중하면 됩니다 [6].

4. 에러와 재시도: 만약 WebSocket 전송에 실패하더라도 (예: 해당 세션이 연결이 끊겼거나 네트워크 오류) 서버에서는 큰 영향을 받지 않습니다. Spring의 SimpMessagingTemplate 전송 함수는 기본적으로 best-effort로 동작하며, 실패 시 예외를 남길 뿐입니다. 필요하다면 실패한 메시지를 다시 처리하거나 별도 저장하는 로직을 넣을 수 있지만, 실시간 알림 특성상 실패한 경우를 굳이 재시도하지 않고 **다음 이벤트를 기다리는** 정도로 처리해도 무방합니다. (중요 알림이라면 클라이언트 ACK를 요구하거나, 읽지 않은 알림으로 DB에 남겨두고 사용자가 나중에 확인하도록 유도하는 방법도 있습니다.)

요약하면, **Kafka Consumer -> (비즈니스 처리+저장) -> WebSocket Broadcaster**의 파이프라인으로 동작하며, Spring의 추상화를 적극 활용해 세션과 메시지를 관리합니다. 이러한 전략은 "생산자-이벤트-소비자" 구조를 통해 **확장성과 유연성**, 그리고 STOMP 기반의 **편리한 브로드캐스팅**을 모두 달성합니다 [4]. 또한 Kafka를 매개로 함으로써 **분산 환경에서의 메시지 동기화** 문제를 해결하고, 안정적으로 다수 사용자에게 실시간 알림을 제공할 수 있습니다.

DevOps 고려사항 (모니터링, 장애 대응 등)

모니터링:
- **애플리케이션 레벨:** Spring Boot Actuator를 통해 `/actuator/health`, `/actuator/metrics` 등을 활성화하여 시스템 상태를 추적합니다. 예를 들어 **WebSocket 세션 수**, **평균 메시지 처리량**, **Kafka Consumer Lag**(가공

되지 않은 대기 중인 메시지 개수) 등을 모니터링 지표로 수집합니다. 이러한 메트릭은 Prometheus 등 모니터링 시스템에 연결하고 Grafana 대시보드로 가시화하여 실시간으로 확인할 수 있습니다. 특히 Kafka 소비 지연은 알림 전달의 병목을 나타낼 수 있으므로, **컨슈머 그룹 오프셋** 변화를 추적하여 소비가 늦어지지는 않는지 살펴봅니다. - **Kafka 및 인프라:** Kafka 브로커들의 상태는 **Kafka Manager**, **Confluent Control Center** 또는 JMX exporter 등을 통해 모니터링합니다. Broker의 메시지 처리량, 토픽별 쌓인 메시지, 파티션 리더/팔로워 상태를 점검하여 Kafka가 건강하게 돌아가는지 확인합니다. 또한 MongoDB의 경우 **Cloud Manager**나 프로메트릭스를 통해 쿼리 성능, 커넥션 수, 메모리 사용량을 추적하고, 필요시 인덱스 최적화를 검토합니다. - **로그 및 추적:** 전체 시스템에 **분산 로깅/추적**을 도입해 문제 발생 시 원인을 빠르게 파악합니다. 예를 들어 Kafka 메시지 키나 코릴레이션ID를 로깅하여, 특정 이벤트에 대해 프로듀서 로그부터 컨슈머 로그, WebSocket 전송 로그까지 **연결된 흐름**을 추적할 수 있게 합니다. Elk(Stack)(Elasticsearch+Logstash+Kibana) 나 Grafana Loki 등을 활용해 로그를 중앙집중 관리하면, “특정 사용자에게 알림이 전송되었는가?” 등을 쉽게 검색할 수 있습니다.

장애 대응:
- **Kafka 장애 시:** Kafka는 분산 구조로 **브로커 중 일부 장애시 나머지 브로커가 데이터 유지**하도록 설계됩니다. 토픽 복제 фактор를 2 이상으로 두어 한 브로커 장애에도 데이터가 유실되지 않게 하고, 장애 브로커 복구 시 자동으로 **ISR(In-Sync Replica)** 동기화가 되도록 설정합니다. 만약 Kafka 전체가 다운되면, 알림 서비스는 일시적으로 이벤트를 받지 못하지만, Kafka의 **내구성**으로 인해 이벤트가 사라지지 않고 대기하게 됩니다 [14]. Kafka 복구 후 소비를 재개하면 그동안 밀린 이벤트를 순서대로 처리하여 알림을 보낼 수 있습니다 [14]. 이런 경우 알림 지연이 발생할 수 있으므로, 장애 사실을 운영자가 바로 인지할 수 있게 Alerting을 설정합니다 (예: 소비 지연이 일정 threshold를 넘으면 경고). - **알림 서비스 장애 시:** Spring Boot 알림 서비스 인스턴스에 장애가 발생하면, Kubernetes 등의 오케스트레이션을 통해 자동 재시작하거나 예비 인스턴스를 운영할 수 있습니다. 무중단을 위해 **다중 인스턴스** Active-Active 구성과 로드 밸런싱을 권장합니다. 한 인스턴스가 내려가도 다른 인스턴스가 Kafka에서 이벤트를 계속 소비하므로 전체 서비스 가용성이 유지됩니다. 내려간 인스턴스의 사용자는 WebSocket 연결이 끊어지겠지만, 클라이언트 측 재연결 로직으로 다른 정상 인스턴스로 다시 연결하게 합니다. 즉, 알림 서비스는 Stateless하게 수평 확장 및 장애 대응 구조를 가져가고, Kafka로 인해 이벤트 전달의 연속성도 확보됩니다. - **WebSocket 연결 장애:** 개별 사용자의 WebSocket 연결이 끊어지는 것은 네트워크 불안정이나 브라우저 이슈일 수 있습니다. 이를 보완하기 위해 **Heartbeat(PING/PONG)** 설정으로 유휴 연결 검사를 하고, 끊어진 세션은 서버에서 정리합니다. 클라이언트는 일정 시간 후 자동 재접속을 시도하여 사용자 경험을 유지합니다. 만약 서버 쪽 문제(예: 배포 등)로 전체 연결이 끊긴 경우, 사용자 단에서는 짧은 지연 후 자동으로 재연결되므로 서비스 연속성이 유지됩니다. - **MongoDB 장애 시:** MongoDB에 문제가 생길 경우 알림 로그 저장이 일시 실패할 수 있습니다. 이때도 실시간 전송 자체는 메모리에 있는 메시지를 보내기 때문에 영향이 적지만, 로그 유실을 막기 위해 **로컬 버퍼**나 Kafka 등을 이용한 비동기 저장 전략을 고려할 수 있습니다. 예를 들어 MongoDB가 다운되면 알림 서비스가 Kafka 프로듀서로 별도 로그 토픽에 이벤트를 넣고, MongoDB 복구 후 해당 토픽을 처리하여 재저장하게 하는 방법도 있습니다. 또는 MongoDB 자체의 Replica Set 구성을 통해 장애 조치(failover)를 자동화합니다. - **기타:** 프론트엔드 측에서는 WebSocket 서버의 URL을 **다중**으로 확보해 두고(primary/secondary) 장애 시 스위칭하는 방법도 고려됩니다. 그리고 전체 시스템에 대한 **재해복구(DR)** 계획으로서 주 센터-Kafka와 보조 센터-Kafka 간에 MirrorMaker로 이벤트를 이중화하거나, 정기적인 백업을 통해 알림 로그를 보관합니다.

CI/CD 및 운영:
DevOps 측면에서, 지속적인 배포 파이프라인을 구축하여 Kafka 스키마 변경이나 알림 서비스 코드를 배포할 때 **무중단 배포**를 도모합니다. 예컨대 Kubernetes 사용 시 Rolling Update 전략으로 순차 재시작하고, 배포 전에 새 인스턴스를 기동하여 readiness probe 통과 후 오래된 인스턴스를 내려 사용자 영향 없이 업그레이드합니다. 또한 테스트 환경에서 **부하 테스트**를 수행해 400명 동시 접속에서의 CPU, 메모리 사용량을 점검하고, GC 튜닝이나 스레드 설정을 조정합니다. Kafka 토픽의 Retention Period(보관주기)도 운영 정책에 맞게 설정하여, 너무 오래 메시지를 두지 않지만 충분히 이벤트 재처리를 할 수 있을 정도로(예: 1일 또는 몇 시간) 유지합니다.

마지막으로, **알림 메시지 콘텐츠 관리**(예: 중요도에 따른 다른 전달 채널SMS/Email fall-back)나 **사용자 설정**(알림 off 등) 등 추가 요구사항이 있을 경우를 대비해 확장 가능한 구조로 설계합니다. 현재 설계한 Kafka+WebSocket 기반 알림 시스템은 **낮은 지연과 높은 확장성**을 갖추었고, 모니터링/장애대응 원칙을 통해 운영 안정성도 고려하였습니다 [4].

참고 자료: Kafka와 WebSocket을 접목한 실시간 아키텍처에 대한 유사 사례들이 다수 있습니다 [2] [8]. 실제 카카오, 네이버와 같은 대규모 서비스에서도 **Kafka를 이벤트 백본으로** 실시간 알림/댓글 시스템 등을 구현하고 있으며,

Kafka의 **발행-구독 모델, 내구성, 확장성**이 이러한 시스템의 토대가 됩니다 [9]. 이번 설계에도 이러한 업계 모범사례를 적용하여, **효율적이고 신뢰성 있는 웹 알림 플랫폼**을 구현할 수 있을 것으로 판단됩니다.

---

[1] [2] [4] Kafka와 WebSocket을 활용한 실시간 데이터 전송 (1) - 개념 및 설계
https://myrail.tistory.com/88

[3] [5] [최종프로젝트] 실시간 채팅 아키텍처 개선 STOMP+Kafka+MongoDB — Jisung Jung의 기술블로그
https://zzzzseong.tistory.com/92

[6] 단일 메세지 모델 설계 하기(kafka + STOMP + Cassandra)
https://kminu.tistory.com/188

[7] [8] [13] Spring Boot WebSocket with Kafka :: 매일매일 꾸준히
https://junuuu.tistory.com/785

[9] [10] [14] Kafka, Redis, Web Socket, Stomp 를 활용한 채팅 서버 회고
https://velog.io/@comet_strike/Kafka-Redis-Web-Socket-Stomp-%EB%A5%BC-%ED%99%9C%EC%9A%A9%ED%95%9C-%EC%B1%84%ED%8C%85-%EC%84%9C%EB%B2%84-%ED%9A%8C%EA%B3%A0

[11] [12] Kafka 를 써서 실시간 알림 기능 구현하기
https://velog.io/@tigerpoint123/Kafka-%EB%A5%BC-%EC%8D%A8%EC%84%9C-%EC%8B%A4%EC%8B%9C%EA%B0%84-%EC%95%8C%EB%A6%BC-%EA%B8%B0%EB%8A%A5-%EA%B5%AC%ED%98%84%ED%95%98%EA%B8%B0