



실시간 알림 서비스

아이디어 기획 단계지만 모든 아이디어에 동일하게 들어가는 기능인 **실시간 알림 서비스**를 미리 공부하려 한다. *현재 클라이밍 관련 아이디어를 기획 중인 상태라 현재 상황을 고려하여 실시간 알림이 필요한 기능을 간략하게 정리 해 보았다.*

세부 기능

▼ 매칭 관련 알림

- 게임 매칭 요청이 들어왔을 때 알림
- 매칭 성사 알림
- 매칭된 상대방의 준비 상태 알림
- 매칭 취소 알림

▼ 소셜 기능 알림

- 게시물 좋아요 알림

▼ 성취 관련 알림

- 새로운 클라이밍 루트 완료 알림
- 개인 기록 갱신 알림

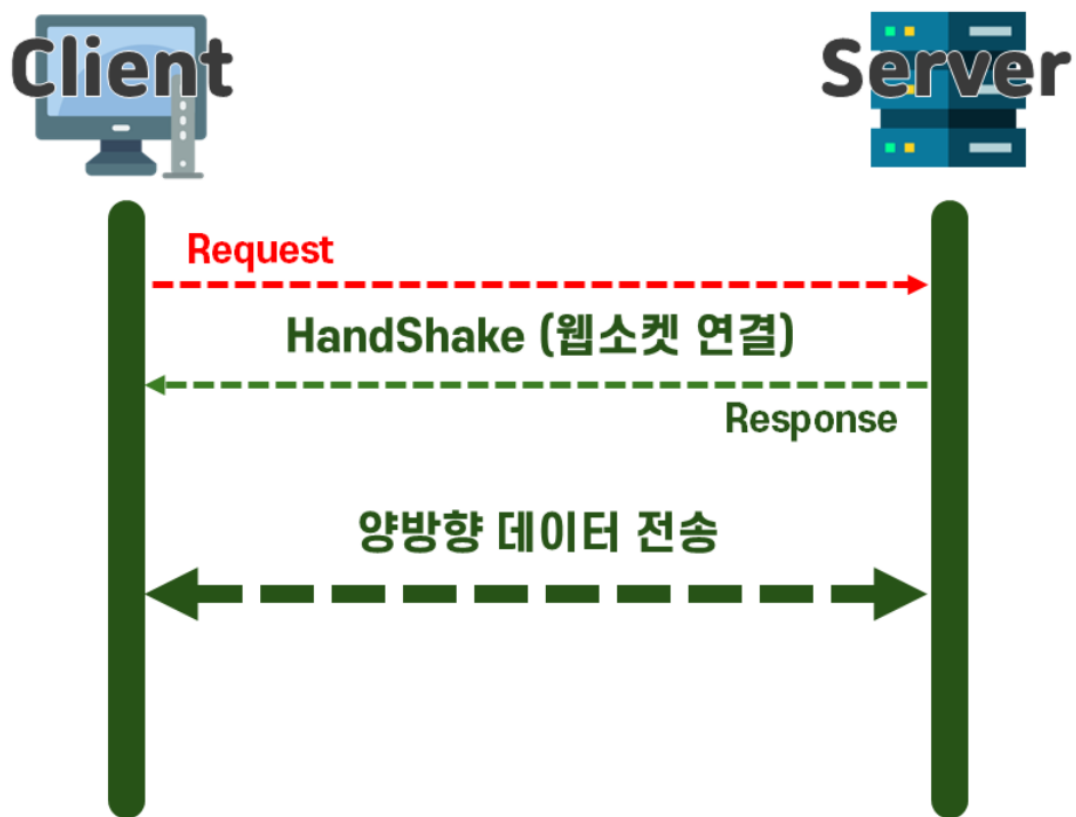
- 랭킹 변동 알림



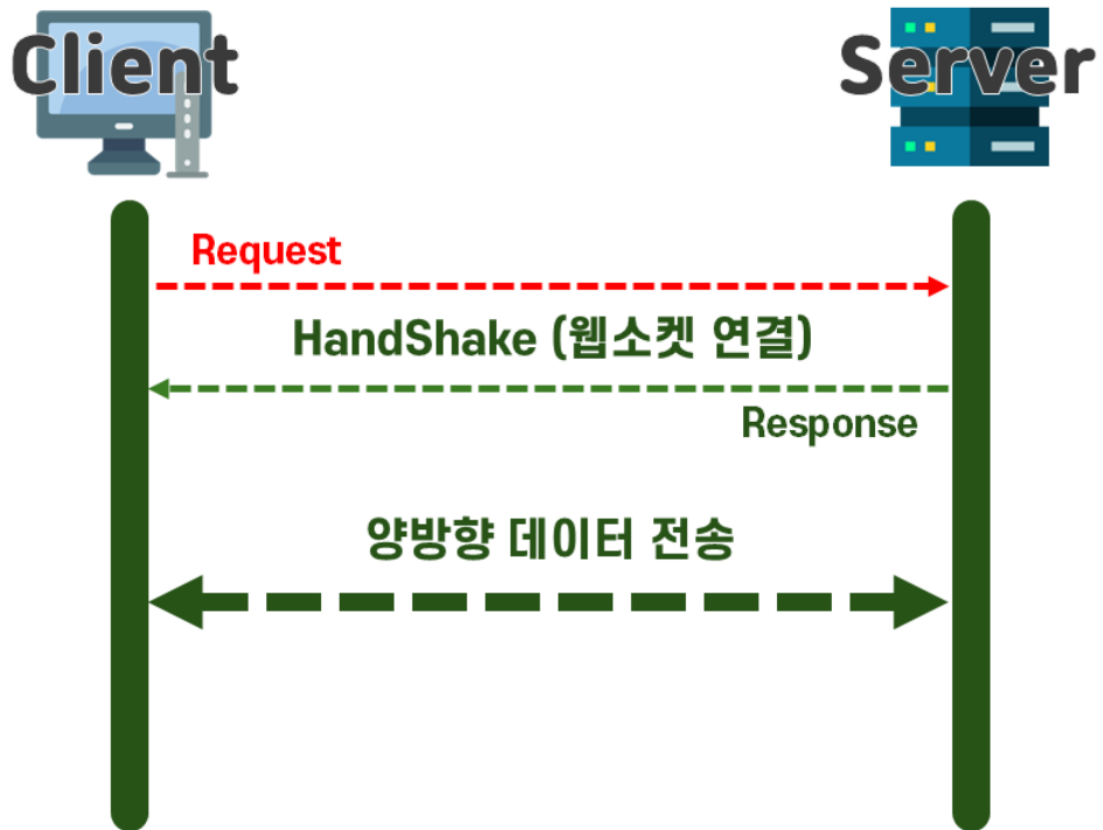
양방향 통신이 필요한 상황 x, 매칭 서비스는 실시간으로 알림을 확인할 수 있어야 함 !

실시간 알림 서비스를 구현하기 위한 3가지 방법

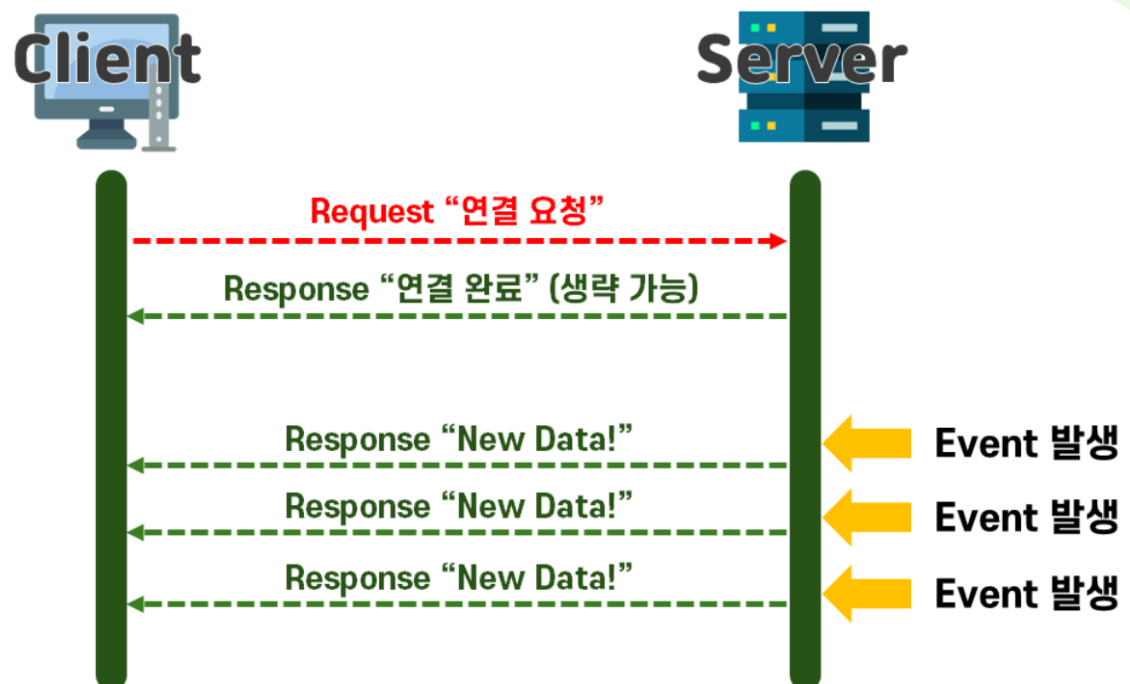
▼ Polling 방식



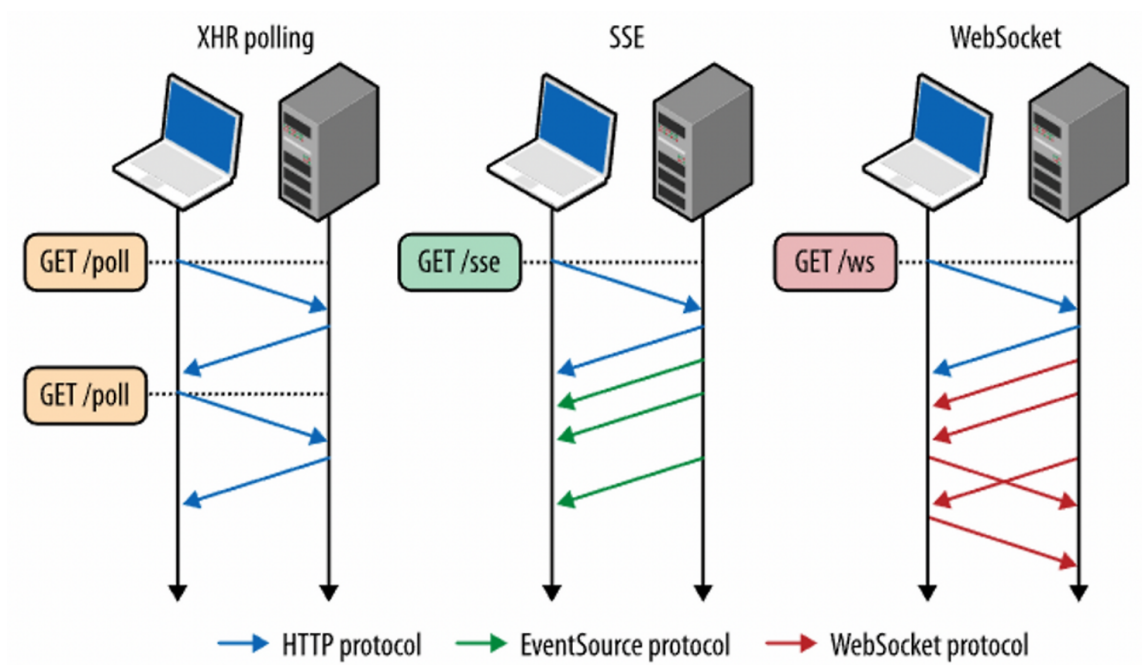
▼ Web-Socket 방식



▼ Server-Send Event 방식



▼ 3가지 방식의 차이점



Server-Send Event 방식으로 구현을 해보려고 한다!

- 연결이 끊어지면 EventSource API가 **자동으로 재연결**을 시도해준다.
- 알림은 **효율적인 단방향 통신이 필요**, Server → Client로 단방향 통신만 지원해도 된다.
- 한번의 연결을 통하여 서버에서 새로운 데이터가 있을 때만 이벤트 스트림을 통해 데이터를 전송

✨ 문자열 데이터 + JSON 데이터 전송

| index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```

<meta charset="UTF-8">
<title>Spring Boot SSE Example</title>
</head>
<body>
<h1>Server-Sent Events</h1>
<div id = "events"></div>

<script>
  const eventSource = new EventSource("/emitter");

  eventSource.onmessage = (event) => {
    const div = document.createElement("div");

    // 문자열 데이터 전송
    div.textContent = `Event received: ${event.data}`;

    // JSON 데이터 전송
    const eventData = JSON.parse(event.data);
    div.textContent = `Message: ${eventData.message}, TimeStamp: ${eventData.timestamp}`;

    document.getElementById("events").appendChild(div);
  }

  eventSource.onerror = (error) => {
    console.error("error", error);
    eventSource.close();
  }
</script>
</body>
</html>

```

EmitterController.java

```

@RestController
@RequiredArgsConstructor
public class EmitterController {

```

```

private final EmitterService emitterService;

@GetMapping(path = "/emitter", produces = MediaType.TEXT_
public SseEmitter sub() {
    SseEmitter emitter = new SseEmitter();
    emitterService.addEmitter(emitter);
    emitterService.sendEvent();
    return emitter;
}
}

```

EmitterService.java

```

public class EmitterService {

    private final List<SseEmitter> emitters = new CopyOnWrite
    private final ObjectMapper objectMapper;

    public void addEmitter(SseEmitter emitter) {
        // 새로운 SSE 연결
        emitters.add(emitter);
        // 완료 콜백, emitters 리스트에서 emitter 제거
        emitter.onCompletion(() -> emitters.remove(emitter));
        // 시간 초과 콜백, emitters 리스트에서 emitter 제거
        emitter.onTimeout(() -> emitters.remove(emitter));
    }

    @Scheduled(fixedRate = 1000)
    public void sendEvent() {
        for (SseEmitter emitter : emitters) {
            try {
                // 문자열 데이터 전송
                emitter.send("연결 완료.");

                // JSON 데이터 전송
            }
        }
    }
}

```

```

        Map<String, Object> eventData = new HashMap<>();
        eventData.put("message", "Hello, world!");
        eventData.put("timestamp", System.currentTimeMillis());
        String json = objectMapper.writeValueAsString(eventData);
        emitter.send(json, MediaType.APPLICATION_JSON);

    } catch (IOException e) {
        emitter.complete();
        emitters.remove(emitter);
    }
}
}
}

```



Scheduled 사용, 1초에 한 번씩 메시지 이벤트를 보내는 것을 확인 !

Server-Sent Events

```

Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:32 GMT+0900 (한국 표준시)
Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:32 GMT+0900 (한국 표준시)
Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:33 GMT+0900 (한국 표준시)
Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:34 GMT+0900 (한국 표준시)
Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:35 GMT+0900 (한국 표준시)
Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:36 GMT+0900 (한국 표준시)
Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:37 GMT+0900 (한국 표준시)
Message: Hello, world!, TimeStamp:Wed Jan 15 2025 23:35:38 GMT+0900 (한국 표준시)

```



SSE를 사용하며 고려해야 할 점이 생겼음!

팀장님 왈: 단일 WAS가 아닌 가용성을 위해 **멀티 WAS 환경**으로 구축된 환경을 사용할 수도 있습니다!

이러한 상황에서는 **SSE만으로는 구현이 불가**

→ **Redis Pub/Sub**을 활용하여 구현하는 방식으로 구현 해보려 한다!

✨ SSE + Redis, 실시간 알림 구현

| RedisConfig.java

```
@Configuration
public class RedisConfig {

    @Value("${spring.data.redis.host}")
    private String host;

    @Value("${spring.data.redis.port}")
    private int port;

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory(host, port);
    }

    @Bean
    public RedisTemplate<?, ?> redisTemplate() {
        RedisTemplate<?, ?> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory());

        GenericJackson2JsonRedisSerializer serializer = new G
        template.setValueSerializer(serializer);
        template.setHashValueSerializer(serializer);
    }
}
```



```

        template.setKeySerializer(new StringRedisSerializer())
        template.setHashKeySerializer(new StringRedisSerializ

        template.afterPropertiesSet();
        return template;
    }
}

```



Redis는 객체 저장시 **직렬화 과정을 거쳐야 함**.

내가 선택한 직렬화 방법은 **GenericJackson2JsonRedisSerializer** 으로 객체의 클래스 지정 없이 **모든 Class Type을 JSON 형태로 저장** 할 수 있는 Serializer이다.

- Class Type에 상관 없이 모든 객체를 직렬화해준다는 장점을 가지고 있다.
- 하지만, 단점으로는 Object의 **class 및 package까지 전부 함께 저장**하게 되어 다른 프로젝트에서 redis에 저장되어 있는 값을 사용하려면 package까지 일치시켜줘야한다.
- 따라서 MSA 구조의 프로젝트 같은 경우 문제가 생길 수 있을 것 같다.

RedisMessagePublisher.java

```

@Component
@RequiredArgsConstructor
public class RedisMessagePublisher {
    private final RedisTemplate<String, String> redisTemplate
    // 메시지를 발행할 채널명
    private static final String CHANNEL = "notifications";

    public void publish(NotificationMessageDto message) {
        // Redis 채널 선택 후, JSON 문자열로 직렬화
        redisTemplate.convertAndSend(CHANNEL, message.seriali
    }
}

```

RedisMessageSubscriber.java

```
@Component
@RequiredArgsConstructor
public class RedisMessageSubscriber implements MessageListener {
    private final NotificationService notificationService;

    @Override
    public void onMessage(Message message, byte[] pattern) {
        NotificationMessageDto notification = null;
        try {
            // Redis에서 받은 메시지를 DTO로 변환
            notification = NotificationMessageDto.deserialize(
                new String(message.getBody())
            );
        } catch (JsonProcessingException e) {
            throw new RuntimeException(e);
        }
        // 클라이언트로 알림 전송
        notificationService.sendToClient(
            notification.getUserId(),
            notification.getMessage()
        );
    }
}
```

NotificationService.java

```
@Service
@RequiredArgsConstructor
public class NotificationService {
    private final RedisMessagePublisher messagePublisher;
    private final NotificationRepository notificationRepository;
    private final UserRepository userRepository;
    private final Map<Long, SseEmitter> emitters = new Concur
```

```

// 구독 관리
public SseEmitter subscribe(Long userId) {
    SseEmitter emitter = new SseEmitter(60 * 1000L);
    emitters.put(userId, emitter);

    emitter.onCompletion(() -> emitters.remove(userId));
    emitter.onTimeout(() -> emitters.remove(userId));

    // 초기에 읽지 않은 알림 전송
    sendUnreadNotifications(userId, emitter);
    return emitter;
}

private void sendUnreadNotifications(Long userId, SseEmitter emitter) {
    User user = userRepository.findById(userId)
        .orElseThrow(() -> new RuntimeException("User not found"));

    List<Notification> unreadNotifications = notificationRepository.findUnreadNotificationsByUserId(userId);

    if(!unreadNotifications.isEmpty()) {
        try {
            emitter.send(SseEmitter.event().
                name("읽지 않은 메세지입니다.")
                .data(unreadNotifications));
        } catch (IOException e) {
            // 전송 실패시 유저의 emitter 삭제
            emitters.remove(userId);
        }
    }
}

// 알림 발송
public void notify(Long userId, String message) {
    User user = userRepository.findById(userId)
        .orElseThrow(() -> new RuntimeException("User not found"));

    Notification notification = new Notification(user, message);
    notificationRepository.save(notification);
}

```

```

        notificationRepository.save(notification);

        // Redis로 실시간 알림 발송
        messagePublisher.publish(new NotificationMessageDto(u
    }

    /*
        SseEmitter event의 개념

        서버 측
        event: friend_request
        data: {"from": "user123", "message": "친구 요청이 왔습니다"}

        클라이언트 측
        eventSource.addEventListener('friend_request', event :
            console.log('친구 요청:', event.data);
        });

        채널: notifications
        이벤트:{
            "type": "friend-request",
            "from": "user123",
            "to": "user456"
        }

        채널은 이벤트들을 그룹화하고 분류하는 논리적 공간, 이벤트는 그 채널에 속한
    */
    public void sendToClient(Long userId, String message) {
        SseEmitter emitter = emitters.get(userId);

        // 연결이 끊어진 유저는 return null
        if (emitter != null) {
            try {
                emitter.send(SseEmitter.event()
                    .name("notification")
                    .data(message));
            } catch (IOException e) {
                emitters.remove(userId);
            }
        }
    }

```

```

        }
    }
}

public void markAsRead(Long userId, Long notificationId) {
    Notification notification = notificationRepository.findById(notificationId)
        .orElseThrow(() -> new RuntimeException("Notification not found"));

    if (!notification.getUser().getId().equals(userId)) {
        throw new RuntimeException("Unauthorized");
    }

    notification.markAsRead();
    notificationRepository.save(notification);
}
}

```



유저가 게시글을 작성하면 모든 유저에게 알림이 가는 시스템으로 코드 작성 테스트를 **도커 컨테이너**에서 진행 해 보고 싶다는 생각이 들었음 !

✨ Dockerfile, docker-compose.file 작성

| Dockerfile

```

FROM openjdk:17-jdk-slim
ARG JAR_FILE=build/libs/*.jar
COPY ${JAR_FILE} app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]

```



- openjdk 17 버전, 필수적인 JDK 파일만 포함하여 이미지 크기 최소화
- 빌드 시점에 사용할 변수 설정, 여기서는 JAR_FILE 이라는 변수로 설정하였음
- 빌드된 JAR 파일을 컨테이너 내부의 app.jar로 복사
- 컨테이너가 8080 포트를 사용하겠다 명시
- 컨테이너가 실행될 때 실행할 명령어, app.jar 파일을 Java로 실행

| docker-compose.file

```
services:
  app:
    build:
      context: .
      dockerfile: ./Dockerfile
    container_name: spring_app
    volumes:
      - ./:/app
    ports:
      - "8080:8080"
    depends_on:
      - redis
      - mysql
    networks:
      - app_network

  redis:
    image: redis:latest
    container_name: redis_container
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    networks:
      - app_network
```

```
mysql:
  image: mysql:8.0
  container_name: mysql_container
  environment:
    MYSQL_ROOT_PASSWORD: 1234
    MYSQL_DATABASE: notification
  ports:
    - "3306:3306"
  volumes:
    - mysql_data:/var/lib/mysql
  networks:
    - app_network

volumes:
  redis_data:
    driver: local
  mysql_data:
    driver: local

networks:
  app_network:
    driver: bridge
```



서비스 구성

app 서비스

- 현재 디렉토리의 Dockerfile 을 사용하여 컨테이너 이름은 spring_app 로 지정 후 빌드
- 8080 포트를 호스트와 컨테이너에 매핑
- redis, mysql 서비스 의존성 설정
- app_network 네트워크에 연결

redis 서비스

- 최신 버전의 redis 이미지 사용, 컨테이너 이름은 redis_container 로 지정
- 6379 포트를 호스트와 컨테이너에 매핑

mysql 서비스

- 8.0 버전 mysql 이미지 사용, 컨테이너 이름은 mysql_container 로 지정
- 환경변수 설정 (비밀번호, 데이터베이스 이름)
- 3306 포트를 호스트와 컨테이너에 매핑

볼륨 설정

- 도커에서 제공하는 로컬 스토리지에 저장

stateless하게 동작하도록 설계

컨테이너가 아닌 외부에 데이터를 저장하고 컨테이너는 그 데이터로 동작하도록 설계하는 것

컨테이너 자체는 상태가 없고 상태를 결정하는 데이터는 외부로부터 제공

컨테이너가 삭제돼도 데이터는 보존

네트워크 설정

- 브릿지 타입의 네트워크

- 모든 서비스는 app_network를 통해 통신



컨테이너 올리기 성공 !!

```
sse_redis - zsh - 151x33
foreign key (user_id)
references user (id)
2025-01-17T17:02:18.682Z DEBUG 1 --- [          main] org.hibernate.SQL           :
alter table post
add constraint FK12njtf8e0jmyb451qfpt6ad89
foreign key (author_id)
references user (id)
Hibernate:
alter table post
add constraint FK12njtf8e0jmyb451qfpt6ad89
foreign key (author_id)
references user (id)
2025-01-17T17:02:18.714Z INFO 1 --- [          main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2025-01-17T17:02:19.250Z INFO 1 --- [          main] o.s.d.j.r.query.QueryEnhancerFactory      : Hibernate is in classpath; If applicable, HQL parser will be used.
2025-01-17T17:02:20.062Z WARN 1 --- [          main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2025-01-17T17:02:20.558Z INFO 1 --- [          main] o.s.b.w.embedded.tomcat.TomcatWebServer   : Tomcat started on port 8080 (http) with context path '/'
2025-01-17T17:02:20.586Z INFO 1 --- [          main] c.example.sse_redis.SseRedisApplication  : Started SseRedisApplication in 6.722 seconds (process running for 7.546)
jiyeon@jiyeon-ui-MacBookAir sse_redis % docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
0f68c9c6c689   sse_redis-app  "java -jar /app.jar"     14 hours ago  Up 14 hours  0.0.0.0:8080->8080/tcp              spring_app
27ced08840ef   redis:latest   "docker-entrypoint.s..." 14 hours ago  Up 14 hours  0.0.0.0:6379->6379/tcp              redis_container
c98dc691c466   mysql:8.0      "docker-entrypoint.s..." 14 hours ago  Up 14 hours  0.0.0.0:3306->3306/tcp, 33060/tcp  mysql_container
```

🌟 문제점 발생



현재 진행하는 프로젝트에서 메인 기능인 게임(클라이밍 대결) 매칭 서비스에서 SSE 방식으로 해결하지 못하는 상황 발생

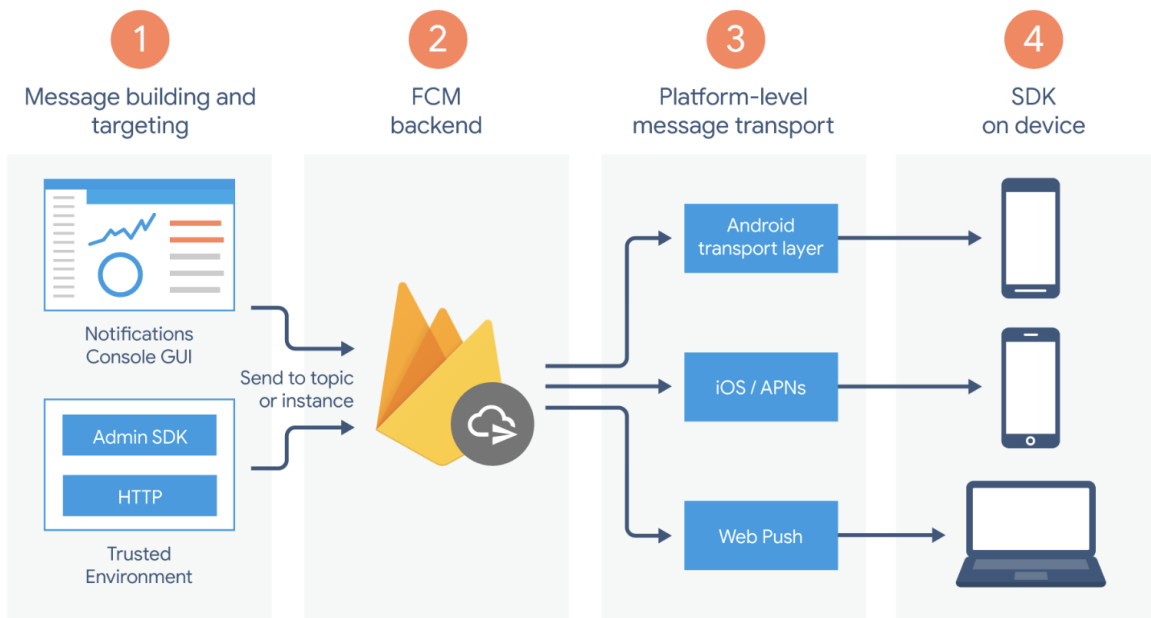
- 게임 매칭되었을 때 **백그라운드 환경에서도 알림이 푸쉬 가능해야 함.**

→ FCM으로 알림 구현하는 방식으로 가기로 함 !

🌟 FCM(Firebase-Cloud-Messaging) 아키텍처 이해하기

교차 플랫폼 메시징 솔루션

- 플랫폼에 종속되지 않고 push 메시지를 보낼 수 있다는 점, push 메시지를 보내기 위해 기존에 각 플랫폼 환경별로 개발해야 하는 불편함을 해결하는 대안이 될 수 있음



1. 메세지 만드는 곳 (송신자)

- 알림을 작성하는 곳, Firebase용 Cloud Functions, App Engine 또는 자체 앱 서버

2.FCM 백엔드 (1번의 메시지의 이상 유무에 따라 적절한 응답)

- topic, channel을 통해 메세지 출력, 메시지 ID와 같은 메시지 메타데이터를 생성

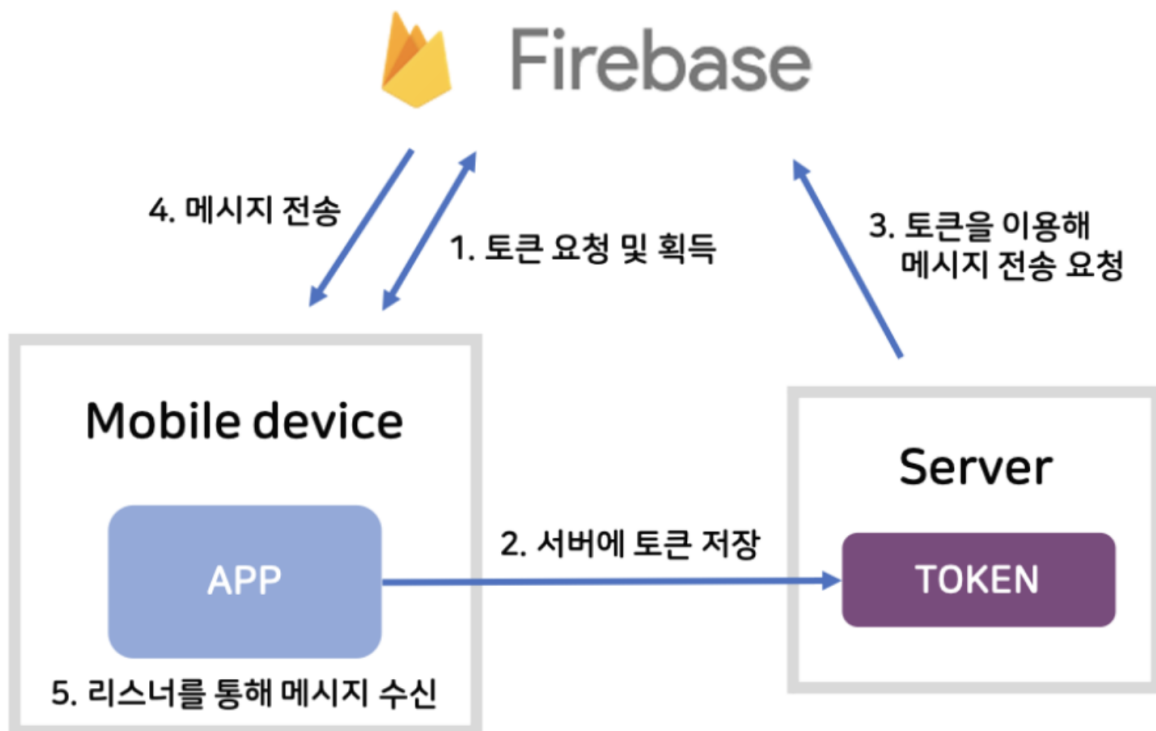
3.플랫폼 전송 레이어

- 기기로 타겟팅된 메시지를 라우팅하고, 메시지 전송을 처리

4.사용자 기기 (수신자)

- 알림이 표시되거나 앱의 포그라운드/백그라운드 상태 및 관련 애플리케이션 로직에 따라 메시지가 처리

✨메세지 처리 흐름



1. FCM 토큰 요청 및 획득

- 프론트엔드는 Firebase에 FCM 토큰을 요청하고, 성공 시 사용자별 고유 FCM 토큰(디바이스 개별 토큰)을 발급

2. 서버에 FCM 토큰 저장

- 백엔드는 FCM 토큰을 저장

3. FCM 토큰으로 메시지 전송요청

- 백엔드는 푸시 메시지가 필요할 때 저장된 FCM 토큰을 사용해 Firebase에 메시지 발급을 요청
- 유효한 토큰에 한해 메시지가 발급되며, 그렇지 않은 경우 에러 코드가 반환

4. 메시지 전송

- 푸시 메시지 발급이 완료되면 Firebase는 FCM 토큰을 기반으로 해당 사용자의 Service Worker에게 메시지를 전송
- Service Worker가 백그라운드에서 실행 중이면 메시지를 수신

5. 리스너를 통해 메시지 수신

- Service Worker는 수신한 푸시 메시지를 사용자에게 표시