



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

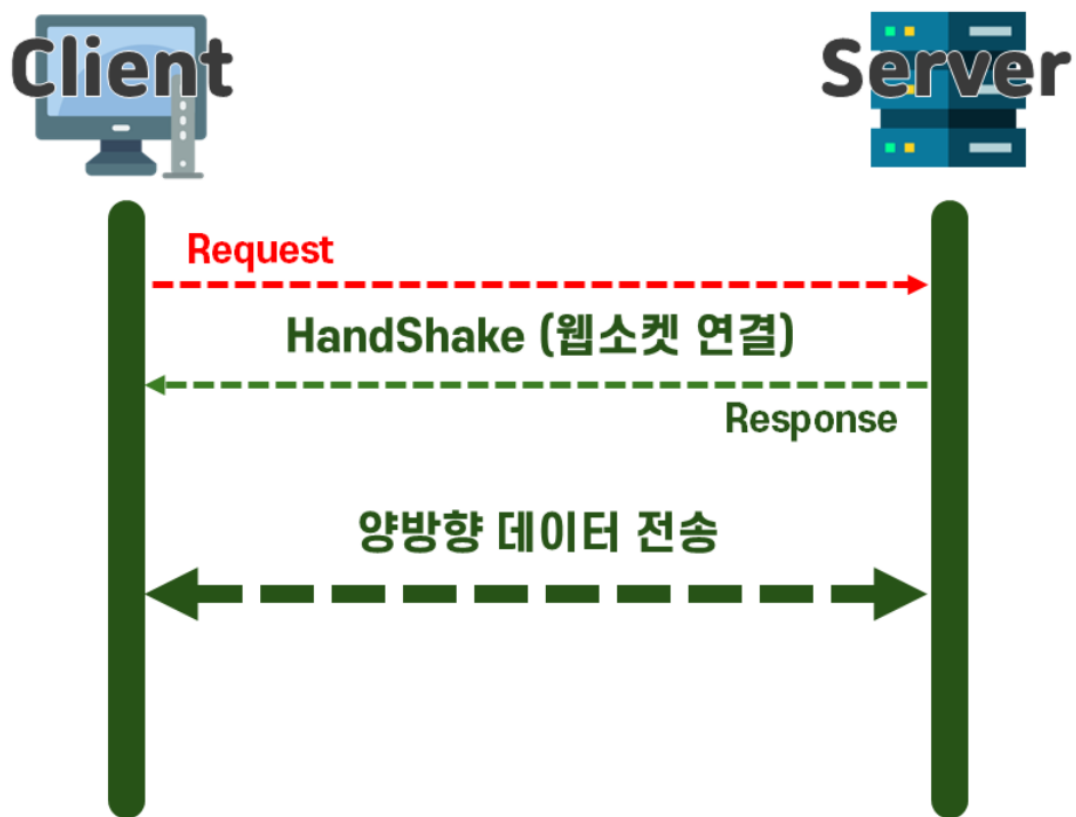
- 랭킹 변동 알림



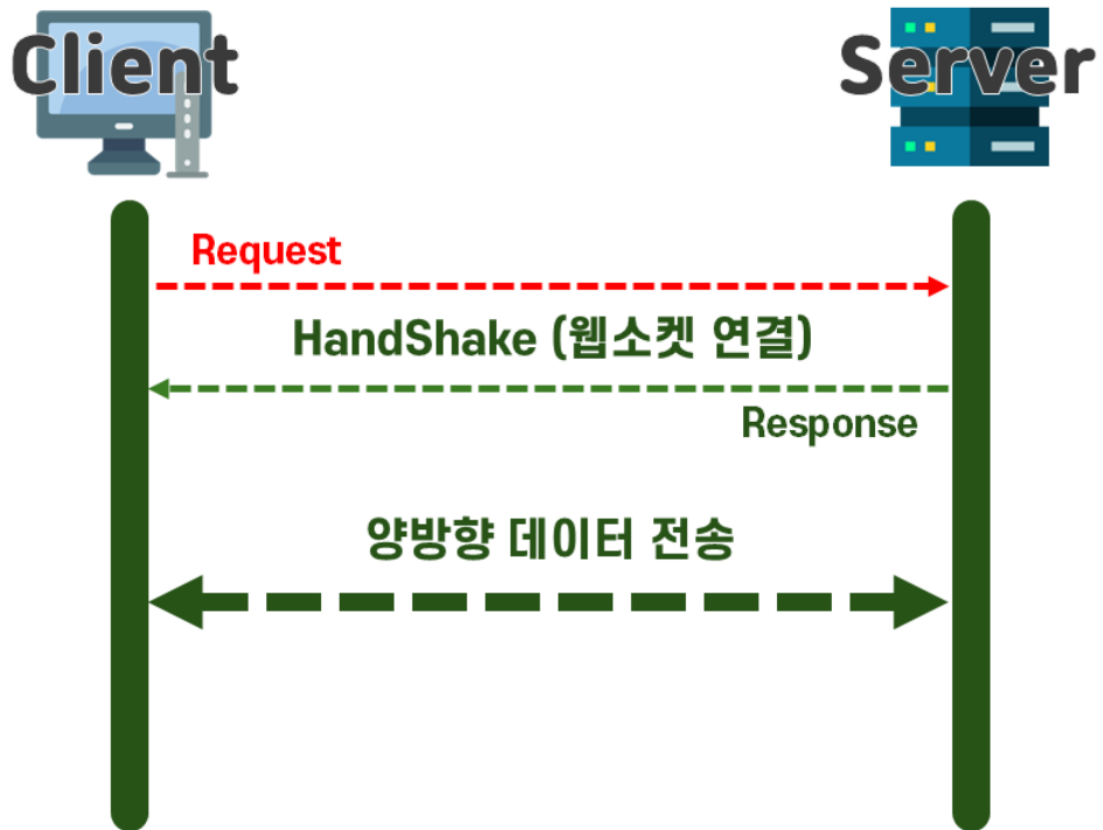
양방향 통신이 필요한 상황 x, 매칭 서비스는 실시간으로 알림을 확인할 수 있어야 함 !

실시간 알림 서비스를 구현하기 위한 3가지 방법

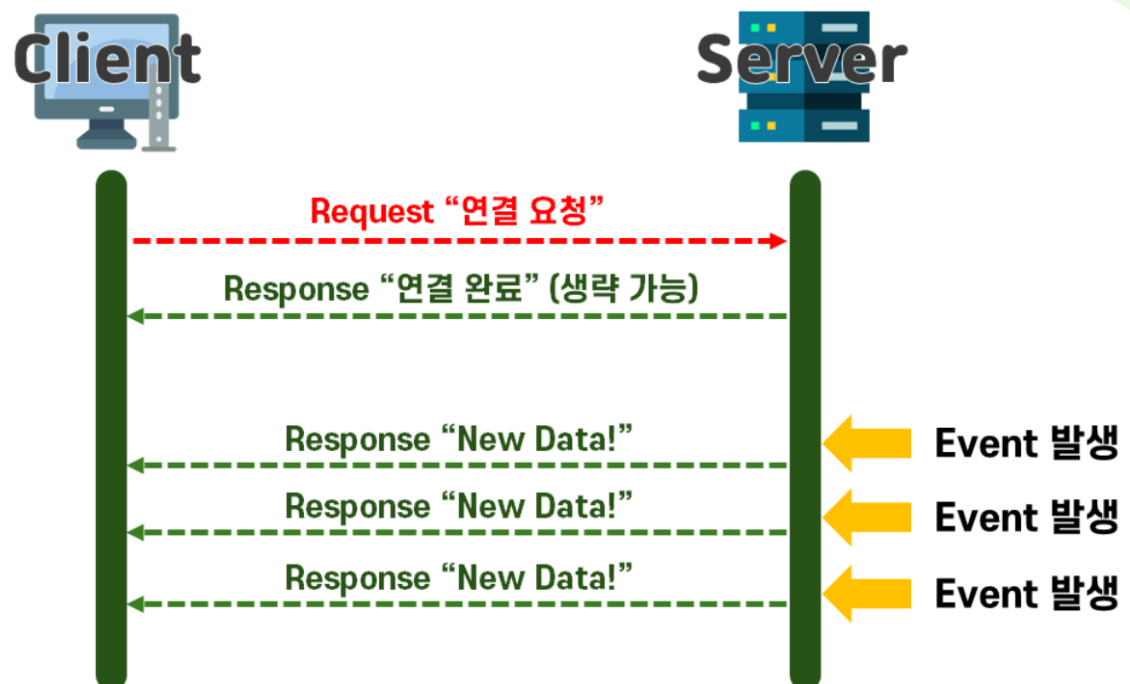
▼ Polling 방식



▼ Web-Socket 방식



▼ Server-Send Event 방식





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}, TimeSta

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 CopyOnWriteArrayList();
    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.serialize());
    }
}
```

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()
    );
}
}

```

NotrificationService.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 ConcurrentHashMap<>();

    // 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 {
                for (Notification notification : unreadNotifications) {
                    messagePublisher.sendMessage(emitter, notification.getMessage());
                }
            } catch (IOException e) {
                // ...
            }
        }
    }
}

```

```

        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);

    // Redis로 실시간 알림 발송
    messagePublisher.publish(new NotificationMessageDto(userId, message));
}

/*
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 이벤트를 그룹화하고 분류하는 논리적 공간, 이벤트는 그 채널
    */
    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);
    }
}

```