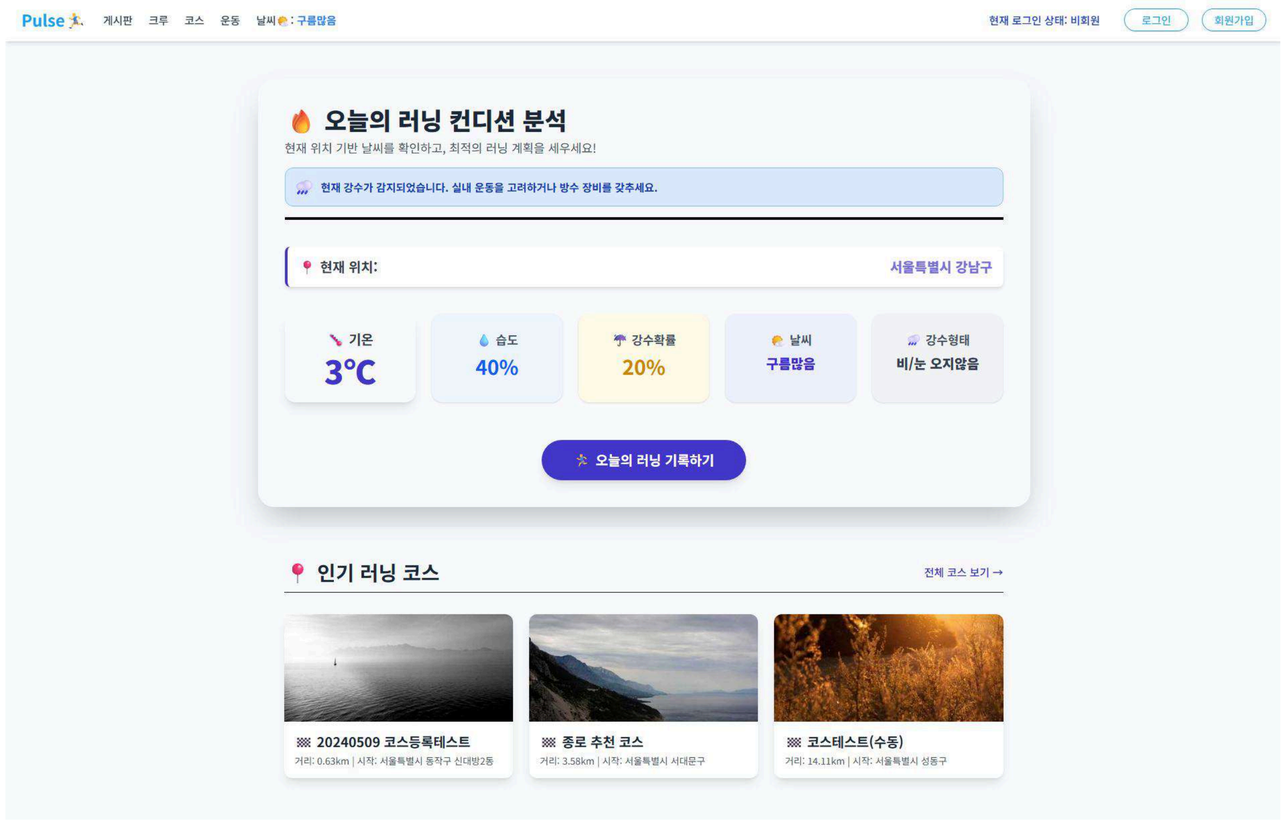
Pulse

Spring Project

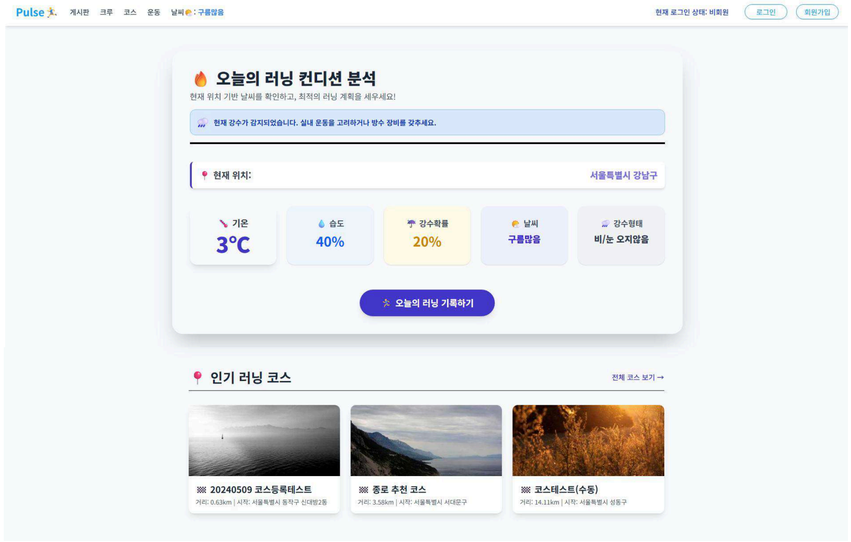
들여다보기 →



Pulse

Spring Project

목차



Pulse 프로젝트

러닝을 더 꾸준히 하게 만드는 크루 기반 러닝 커뮤니티 플랫폼

코스 탐색(지도/날씨/위치) → 크루로 연결 → 커뮤니티로 활동 유지

- 러닝 코스: 등록/추천/조회(지도 기반), 날씨/위치 연동
- 크루: 크루 생성, 가입/탈퇴, 멤버 관리(권한/승인), 크루 전용 소통
- 커뮤니티/게시판: 정보 공유, 댓글/검색/정렬 등

Pulse

개발 환경

Spring Project

프론트엔드 개발

- HTML5
- CSS3 / TailwindCss v4
- JavaScript (ES6)
- jQuery
- AJAX

백엔드 개발

- Tomcat 9.0
- Java
- JSP
- Spring Framework

개발 도구 & 협업 툴

- Eclipse
- Google Drive
- ERD Cloud
- Oracle Cloud
- DataGrip
- intelliJ
- GitHub

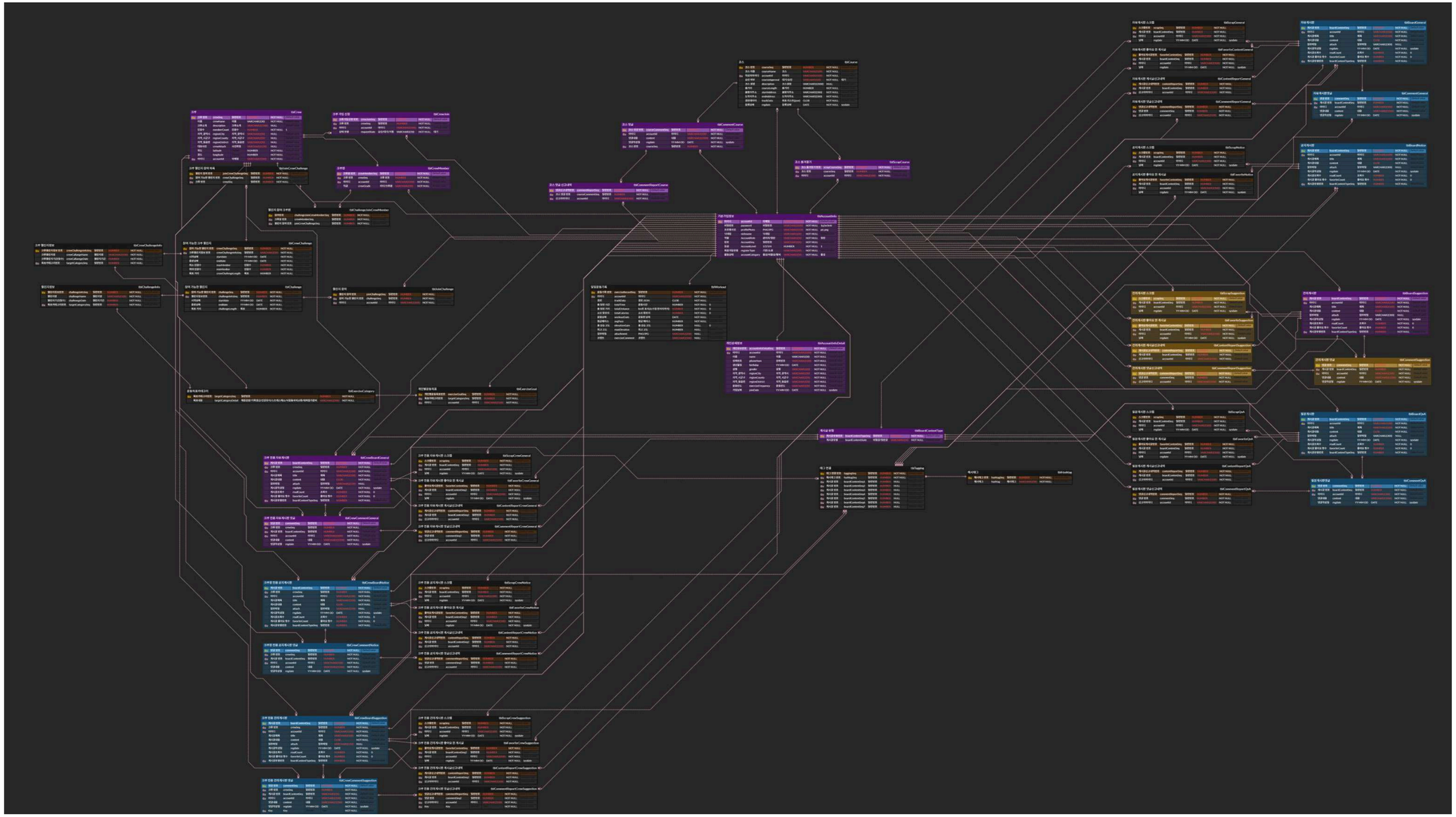
데이터베이스

- Oracle DB

Pulse

Spring Project

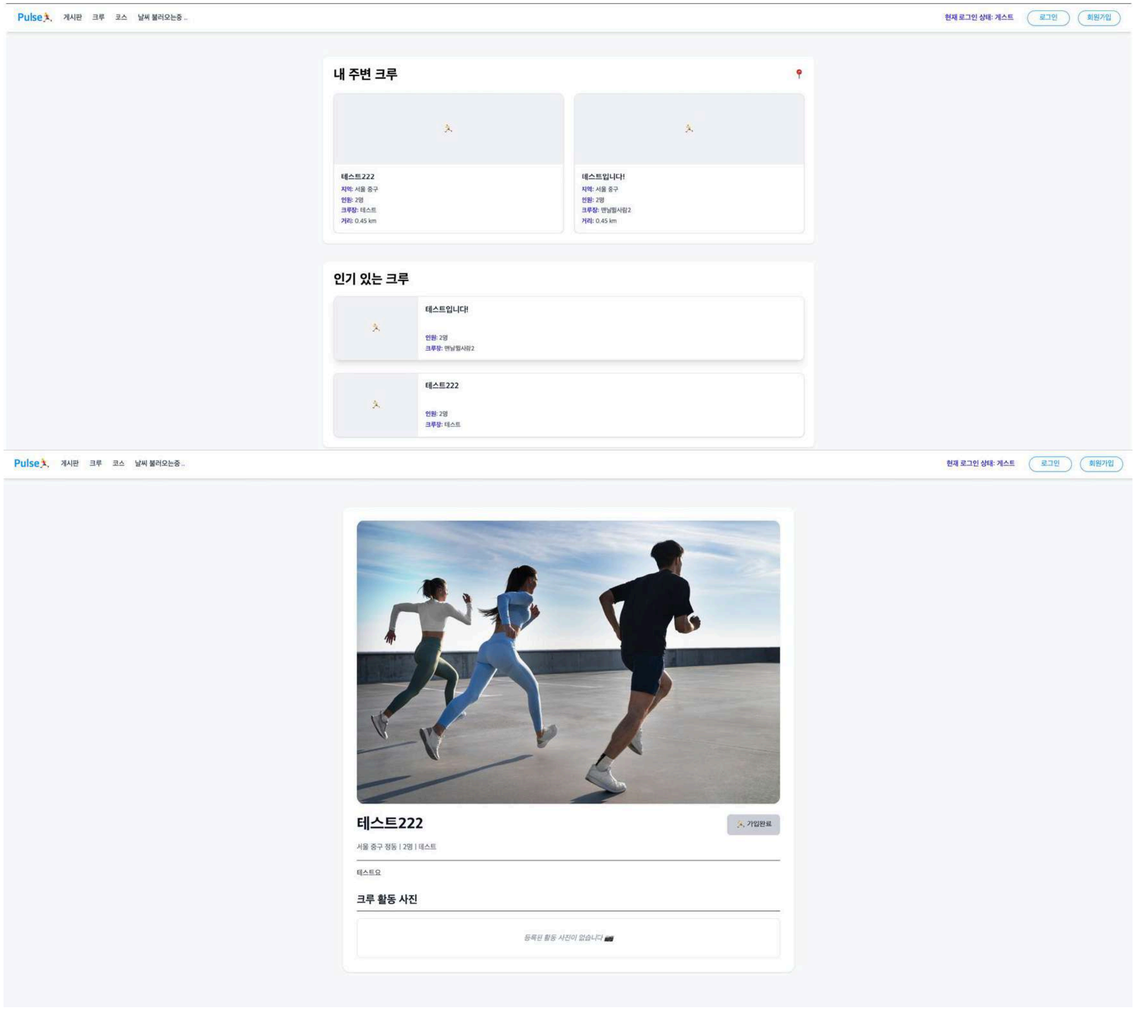
프로젝트 전체 데이터 구조



Pulse

Spring Project

담당 도메인



한눈에 보기

핵심 도메인

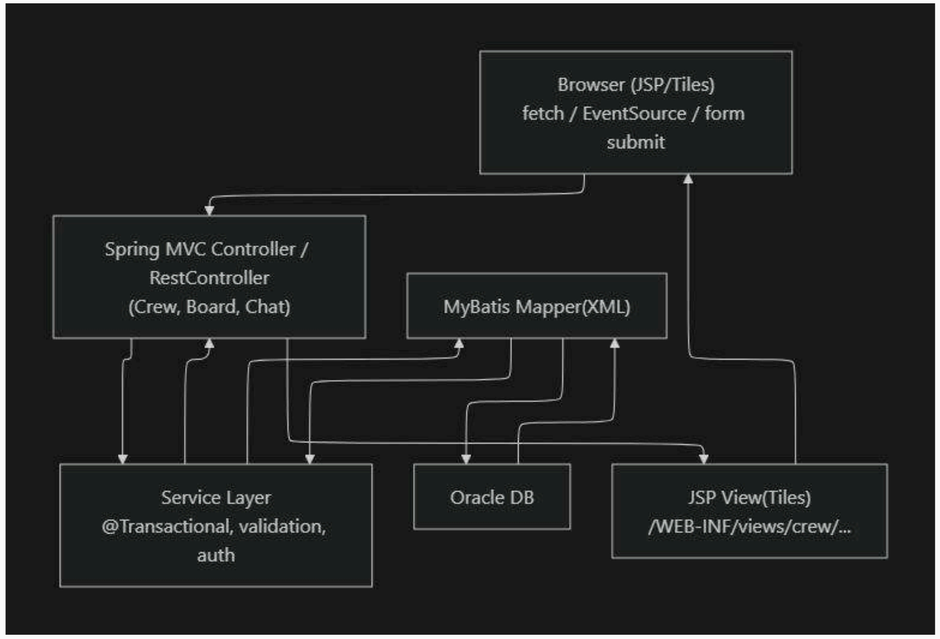
- 크루 생성/가입(승인), 크루 전용 게시판/댓글/좋아요, 대시보드 집계, 실시간 채팅(SSE)

핵심 포인트

- 크루 가입 프로세스에서 요청(대기/승인/거절) ↔ 멤버십(확정) 을 분리해 워크플로우를 명확히 설계
- Tiles 기반 페이지에 채팅을 JSP 컴포넌트처럼 삽입해 “SPA 같은 사용감”을 제공
- SSE(SseEmitter) + REST(POST) 조합으로 구현 복잡도는 낮추고, 실시간성은 확보

Pulse

Spring Project



Crew 도메인 모듈

- 크루 생성/가입 요청/승인: 중복가입 방지, 리더 지정, 멤버수 반영까지 트랜잭션 처리
  CrewRestController.java / CrewService.java / CrewMapper.xml

- 크루 대시보드 데이터: Top 조회/좋아요, 게시판 집계, 공공 API 기반 마라톤 일정 노출
  dashboard-script.jsp + Mapper 집계 쿼리

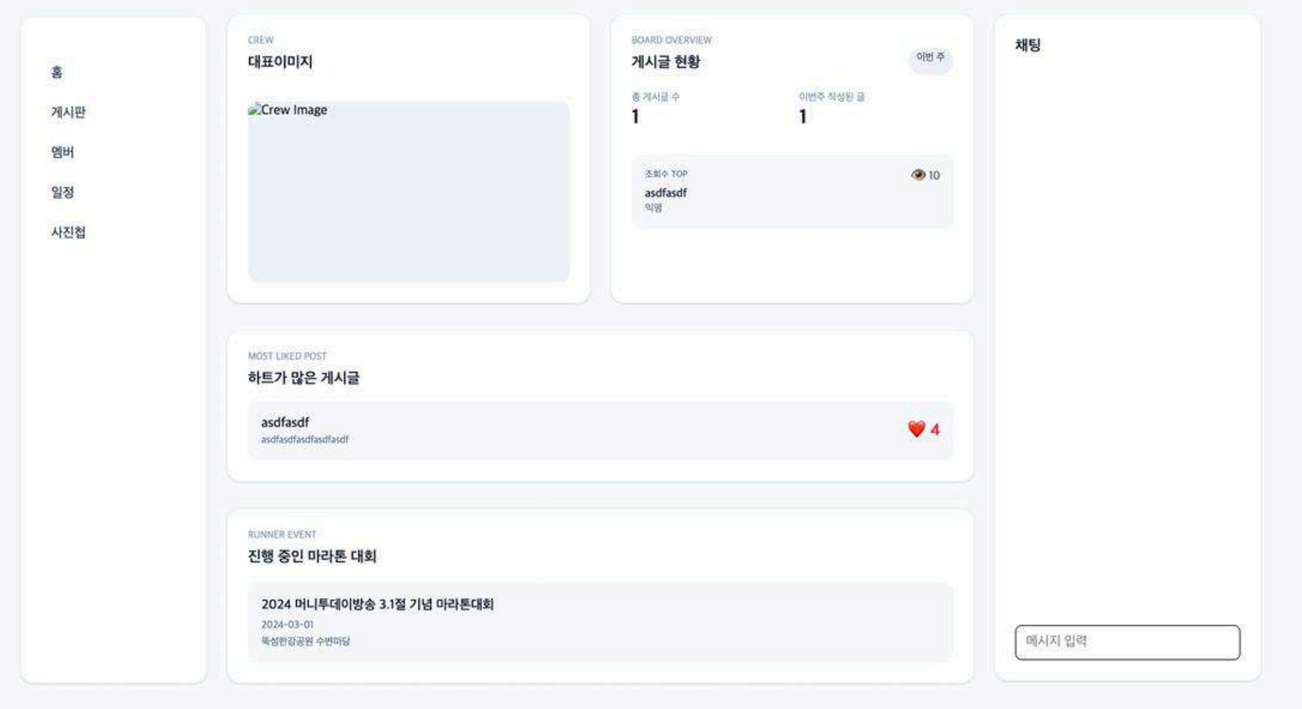
- 크루 게시판 CRUD 마이그레이션:
- JSP/Servlet 기반 로직 → Spring REST + MyBatis로 이관
  CrewBoardRestController.java / CrewBoardService.java / CrewBoardMapper.xml

- 실시간 채팅(SSE): 방(crewSeq) 단위 스트림 구독/브로드캐스트 및 최근 메시지 In-Mem 유지
  CrewChatRestController.java / ChatMemoryStore.java / SseEmitterStore.java

Pulse

SSE Chat.

Spring Project



구현 목표 및 문제점

- 크루 대시보드(JSP/Tiles) 안에 채팅을 위젯처럼 자연스럽게 구현.
- 폴링/롱폴링은 딜레이가 체감되고, 불필요한 요청이 늘어 부하가 커짐.
- WebSocket도 가능하지만, JSP 환경에서 **인증/권한(크루 멤버만) + 연결관리(재연결/정리)**까지 붙으면 구현·테스트 범위가 커짐.
- **채팅 화면을 따로 분리**하면 쉬워지지만, 그럼 대시보드 UX가 SPA 느낌이 사라짐.
- 정리된 목표는 페이지 이동 없이 대시보드 내부에서 실시간 업데이트를 제공.

Pulse

Spring Project

왜 WebSocket 대신 SSE를 선택했는가?

JSP 기반 SSR 프로젝트에서 현실적으로 적용 가능한 선택

JSP + WebSocket 조합에서의 문제점

- 기존 구조가 **전통적인 MVC + JSP**라 WebSocket 도입 시 구조가 크게 바뀜
- STOMP, SockJS 등 추가 셋업 + JS 클라이언트 코드가 복잡해짐
- 동시 접속 수가 크지 않은 프로젝트 규모에서 **오버엔지니어링** 느낌

SSE 선택 이유

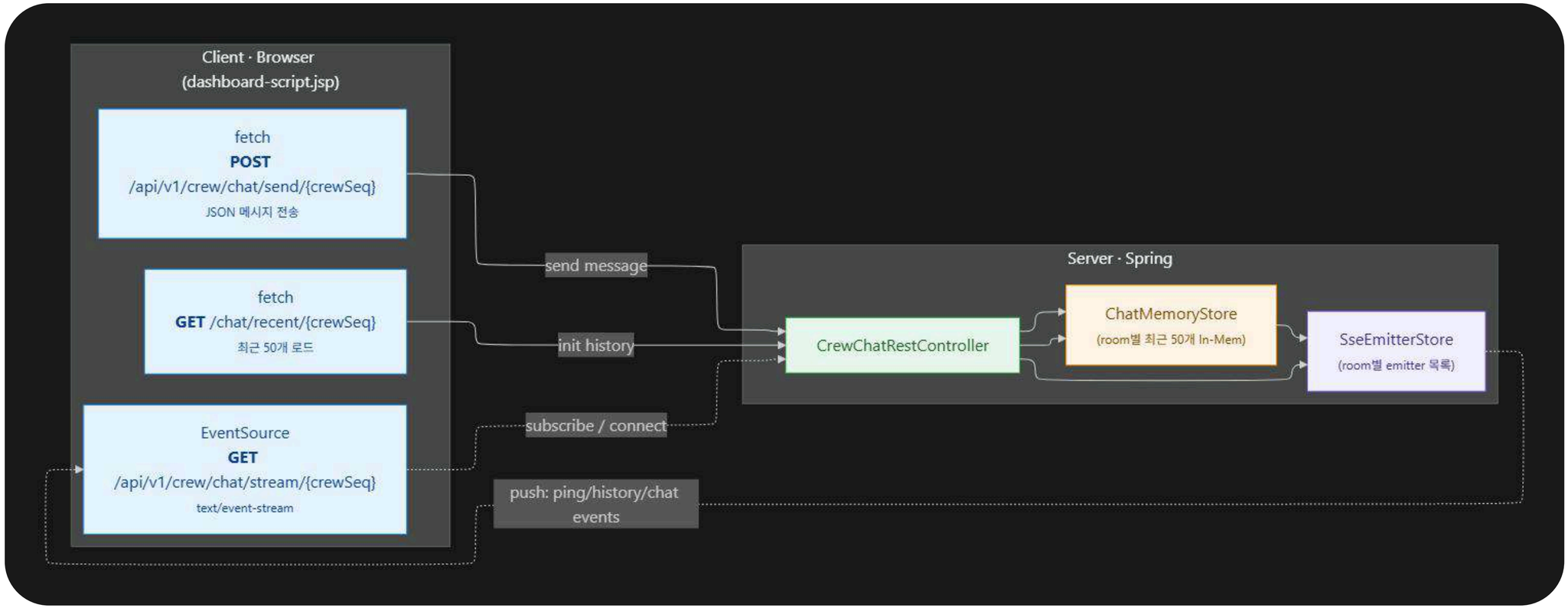
- 브라우저 표준 API EventSource만으로 **실시간 서버 → 클라이언트 푸시** 구현
- HTTP 기반이어서 **기존 Spring MVC 구조와 자연스럽게 연동**
- 단방향(서버 → 클라이언트)이라 SNS 알림 / 채팅 피드용으로 딱 맞음
- 자동 재접속 지원으로 **안정적인 연결 유지**

방식	장점	단점
1. 폴링 n초마다 계속 요청	• 구현이 **매우 간단** • 그냥 setInterval + AJAX로 끝	• 새 메시지가 없어도 계속 요청 → **서버 부하 증가** • 주기를 길게 하면 **실시간성이 떨어짐**
2. 롱폴링 응답 줄 게 생길 때까지 대기	• 폴링보다 **훨씬 실시간에 가까움** • 새 메시지가 생겼을 때만 응답	• 응답 전까지 커넥션 유지 → 접속자가 많으면 **서버 부담** • 요청/응답 반복 구조 자체는 그대로
3. SSE Server-Sent Events	• **한 번 연결** 후 서버에서 데이터가 생길 때마다 푸시 • EventSource로 **간단하게** 구현 가능 • 자동 재연결 지원 (페이지 재방문시 재접속)	• **단방향** (서버 → 클라이언트)만 지원 • 브라우저 → 서버 전송은 별도의 POST 필요

Pulse

SSE 아키텍쳐

Spring Project



입장(구독): GET /chat/stream/{crewSeq}

- SseEmitterStore.addEmitter(room) 등록 (timeout=0)
- 연결 직후 ping + history 이벤트 전송
- ChatMemoryStore.getRecentMessages(room) → 최근 50개 렌더링

발신: POST /chat/send/{crewSeq}

- 메시지에 timestamp/crewSeq 주입
- ChatMemoryStore.addMessage(room) 적재
- SseEmitterStore.sendToRoom(room) 브로드캐스트(chat 이벤트)
- 연결 정리: completion/timeout/error 시 emitter 제거

프론트 구현 포인트

- connectSSE()로 EventSource 연결 → chat 이벤트 수신 시 버블 UI 렌더링(본인/상대 구분)
- onerror 발생 시 3초 후 재연결로 네트워크 변동 대응
- Enter 전송 → /chat/send/{crewSeq} JSON POST → 입력창 비움

채팅 메시지 저장 전략

DB를 꼭 써야 할까? 메모리 + 최근 50개 전략

1️⃣ DB에 저장하는 경우

- 장점
  - 서버 재시작해도 **기록 유지**
  - 검색, 통계, 관리자 모니터링 등 확장 용이
- 단점
  - 테이블 설계, 인덱스, 조회/페이지네이션 등 작업 필요
  - 이번 프로젝트 규모에서는 살짝 과한 느낌

2️⃣ 메모리에만 저장하는 경우

- 서버 내 **ChatMemoryStore**에만 채팅 저장
- 크루별로 **최근 N개(예: 50개)**만 유지
  - 새 메시지 push 시, 개수가 50개 넘으면 **pop()으로 가장 오래된 것 제거**
- 특징
  - 서버 재시작 시 기록은 사라짐 (로그 목적이 아니라면 괜찮음)
  - 동시 접속자가 많지 않은 Pulse 규모에서는 **메모리 부담 거의 없음**

Pulse

SSE 아키텍쳐

Spring Project

ChatMemoryStore 설계

ConcurrentHashMap으로 안전하게 "채팅방별 최근 50개 메시지" 관리

자료구조 개념

- **ConcurrentHashMap**
  - 멀티 스레드 환경에서 안전한 HashMap
  - 여러 요청이 동시에 읽고/쓰더라도 구조가 깨지지 않음
- 예상 구조
  - Map<String, Deque<ChatMessageDTO>>
  - key: crewSeq (채팅방 ID)
  - value: 해당 방의 메시지 큐 (최근 50개)

```
class ChatMemoryStore {

    // crewSeq 별로 최근 메시지 보관
    public class ChatMemoryStore {

    private final Map> roomMessages = new ConcurrentHashMap<>();

    public void addMessage(String crewSeq, ChatMessageDTO message) {
        roomMessages.putIfAbsent(crewSeq, new ConcurrentLinkedDeque<>());
        Deque messages = roomMessages.get(crewSeq);

        messages.addLast(message);

        if(messages.size() > 50) {
            messages.removeFirst();
        }
    }

    public List getRecentMessages(String crewSeq) {
        Deque messages = roomMessages.get(crewSeq);

        if(messages == null) {
            return Collections.emptyList();
        }

        return new ArrayList<>(messages);
    }
}

}
```

왜 굳이 ConcurrentHashMap인가?

- 웹 서버는 **여러 요청이 동시에 들어오는 멀티 스레드 환경**
- 일반 HashMap 사용 시
  - 삽입 중에 다른 스레드가 끼어들어 구조가 꼬일 수 있음
  - NullPointer, ClassCast, 내부 구조 깨짐 → 난장판
- ConcurrentHashMap은 이런 상황에서도 **안전하게 add/get 가능**

SseEmitterStore 설계

crewSeq 기준으로 연결된 모든 SSEEmitter 관리

역할 정리

- 한 채팅방(crewSeq)에 **여러 유저**가 들어올 수 있음
- 한 유저 = 한 개의 **SseEmitter**
- 새 메시지가 들어오면
  - 해당 crewSeq에 연결된 모든 Emitter에게 **send()**
- 연결 종료/타임아웃 되면 **Emitter 제거**

```
class SseEmitterStore {

@Component
public class SseEmitterStore {

    private final Map> emitters = new ConcurrentHashMap<>();

    public SseEmitter addEmitter(String crewSeq){
        emitters.putIfAbsent(crewSeq, Collections.synchronizedList(new ArrayList<>()));

        SseEmitter emitter = new SseEmitter(0L); // ★ timeout 없음

        emitters.get(crewSeq).add(emitter);

        // 연결 종료 시 정리
        emitter.onCompletion(() -> emitters.get(crewSeq).remove(emitter));
        emitter.onTimeout(() -> emitters.get(crewSeq).remove(emitter));
        emitter.onError((e) -> emitters.get(crewSeq).remove(emitter));

        // 연결 직후 더미 ping 전송 (중요)
        try {
            emitter.send(SseEmitter.event().name("ping").data("connected"));
        } catch (Exception ignored) {}

        return emitter;
    }

    public void sendToRoom(String crewSeq, Object message){
        List list = emitters.get(crewSeq);
        if(list == null) return;

        List dead = new ArrayList<>();

        for (SseEmitter emitter : list) {
            try{
                emitter.send(SseEmitter.event().name("chat").data(message));
            } catch(Exception e){
                dead.add(emitter);
            }
        }

        list.removeAll(dead);
    }
}

}
```

Pulse

프로젝트를
진행하며..

Spring Project

크루 대시보드에 실시간 채팅을 페이지 이동 없이 위젯처럼 구현하는 과정에서,
기능 구현 자체보다 현 개발상황 및 환경을 파악 후 알맞는 스킬을 먼저 고민해야 한다는 걸 크게 느꼈습니다.

SSE는 Spring MVC + JSP 환경에서 구현 난이도가 낮고 UX도 좋았지만, 동시에 “현재 구조가 단일 인스턴스/인메모리 기반”이라는 한계도 명확하게 느껴졌습니다. 하지만 추후 알림기능 등에 잘 적용할 수 있을 것 같습니다.

특히 채팅 메시지를 인메모리방식으로만 유지하면 서버 재시작 시 데이터가 유실되고, 인스턴스가 여러 대로 늘어나는 순간 방 브로드캐스트의 일관성이 깨질 수 있어서 다음 단계로는 메시지 영속화(DB) + Redis Pub/Sub를 붙여,
어떤 인스턴스에서 메시지를 수신하더라도 동일한 room 구독자에게 안정적으로 전파되는 구조로 확장하는 게 필요하다고 판단했습니다.

또한 네트워크가 불안정한 환경에서 재연결이 발생하면, “자동 재연결”만으로는 완전하지 않고 중간에 놓친 메시지 복구가 필요하다는 걸 체감했다. SSE의 Last-Event-ID를 활용해 미수신 메시지를 재동기화하면,실사용 환경에서도 더 신뢰도 높은 채팅 경험을 제공할 수 있을 것 같습니다.

결과적으로 이번 경험을 통해 “실시간 기능”은 연결만 하는 것 뿐만 아니라, 데이터 일관성(확장), 재연결 복구, 권한 정책의 분리까지 고려해야 완성도가 올라간다는 걸 배웠고 구현하는데 즐거웠던 프로젝트 였던 것 같습니다.