

Movie OTT Playlist

모두의 플리



#대규모 트래픽 대응 글로벌 콘텐츠 평점 및 큐레이션 플랫폼

4팀

정기주, 김재민, 민재영, 정영진

목차

01 프로젝트 개요

02 일정 및 작업관리

03 시스템 전체 구조

04 결과 화면/데모

05 핵심 구현 내용

06 트러블 슈팅

프로젝트 개요

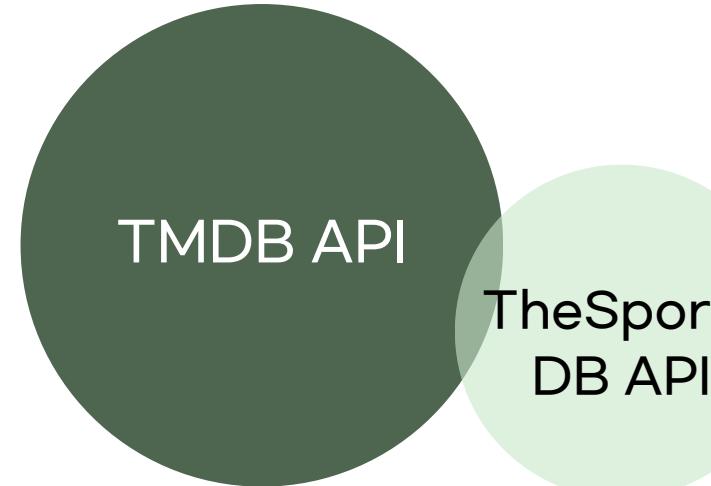
모풀(MOPL)은 영화, 드라마, 스포츠 등 다양한 콘텐츠를

플레이리스트 기반으로 큐레이팅하고 공유하는 실시간 소통 기반 소셜 플랫폼입니다.

주요 기능

- 콘텐츠**  통합 검색 및 자동 완성
자동 수집·동기화
- 플레이리스트**  구독 알림
- 사용자 관리**  소셜 로그인 (OAuth2)
- 실시간 소통**  실시간 같이보기 세션
- 소셜**  팔로우 알림
DM 메시징

OPEN API



기술적 특징

Cursor 페이지네이션과 Redis 캐싱을 통한 대용량 데이터 조회 성능 최적화

WebSocket, SSE, Kafka를 활용한 실시간 같이보기·알림·채팅 시스템 구현

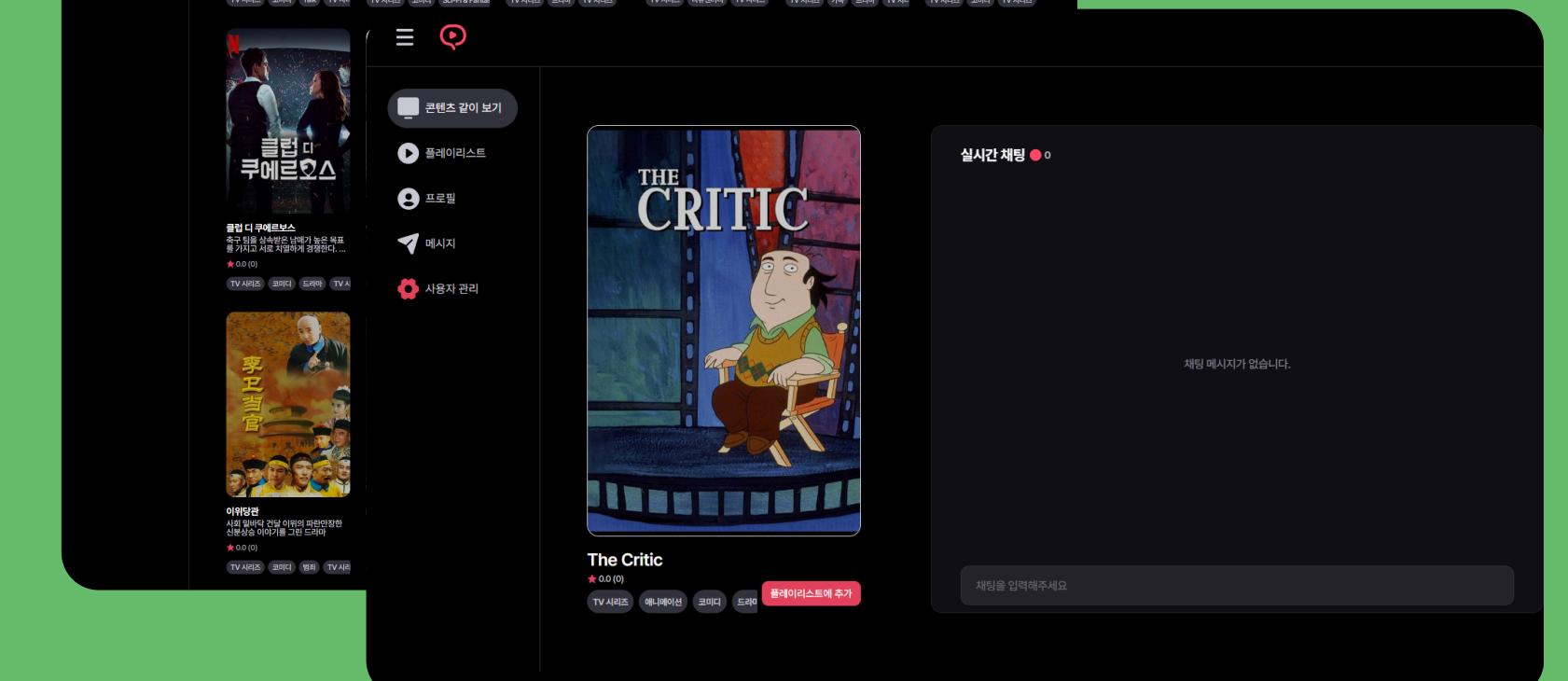
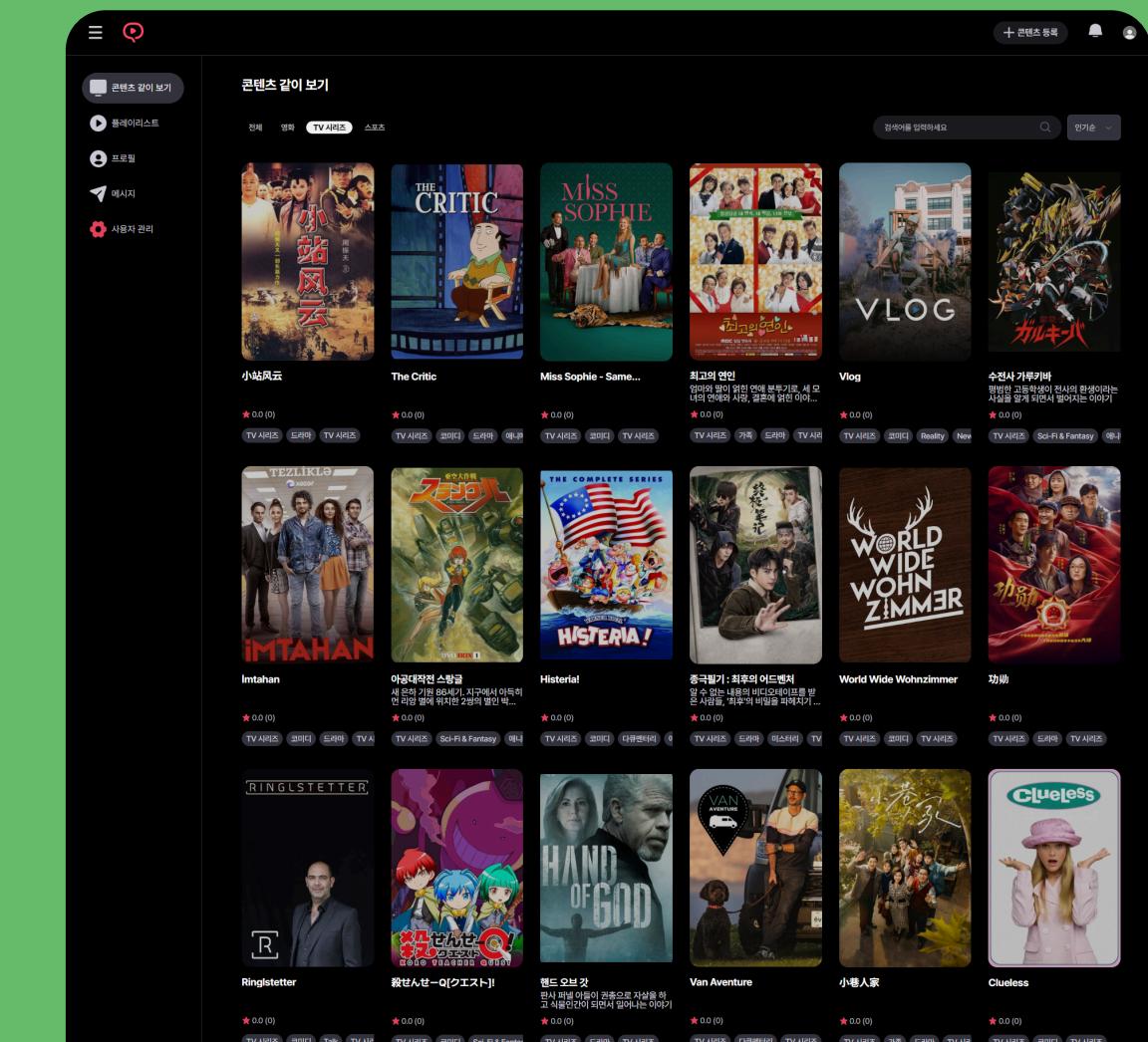
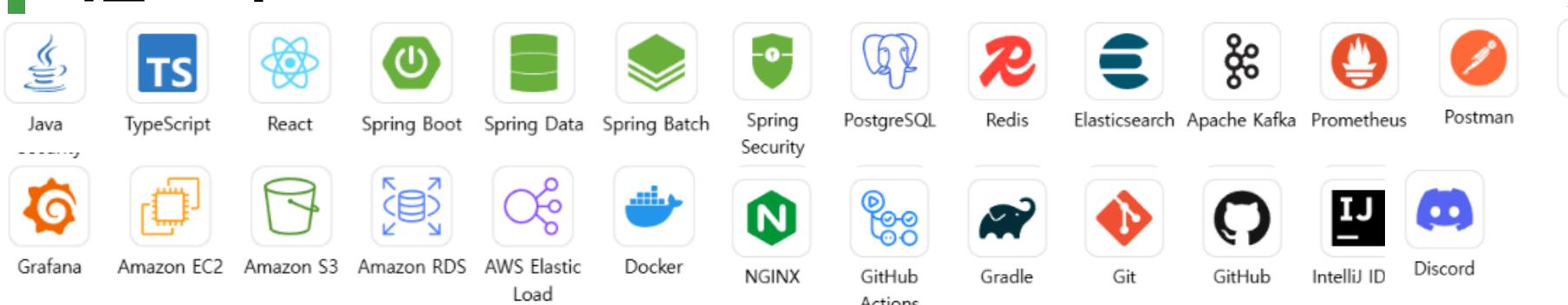
Spring Batch 기반 TMDB·Sports DB API 콘텐츠 자동 수집 및 동기화

Elasticsearch를 활용한 Full Text Search 및 자동완성 검색 구현

Prometheus + Grafana 기반 모니터링 및 GitHub Actions를 통한 무중단 배포

AWS EC2 다중 인스턴스와 ALB + Nginx를 통한 고가용성 인프라 구축

기술 스택



일정 및 작업 관리

▶ 협업 도구

Notion, GitHub, Discord

▶ 프로젝트 문서, 회의록, 기능 명세

The screenshot shows two Notion pages. On the left is a '회의록' (Meeting Log) page with a table of meeting minutes from December 2025 to January 2026. On the right is an 'API SPECIFICATION' page showing endpoints for notifications (GET, DELETE) and users (GET, POST, PATCH).

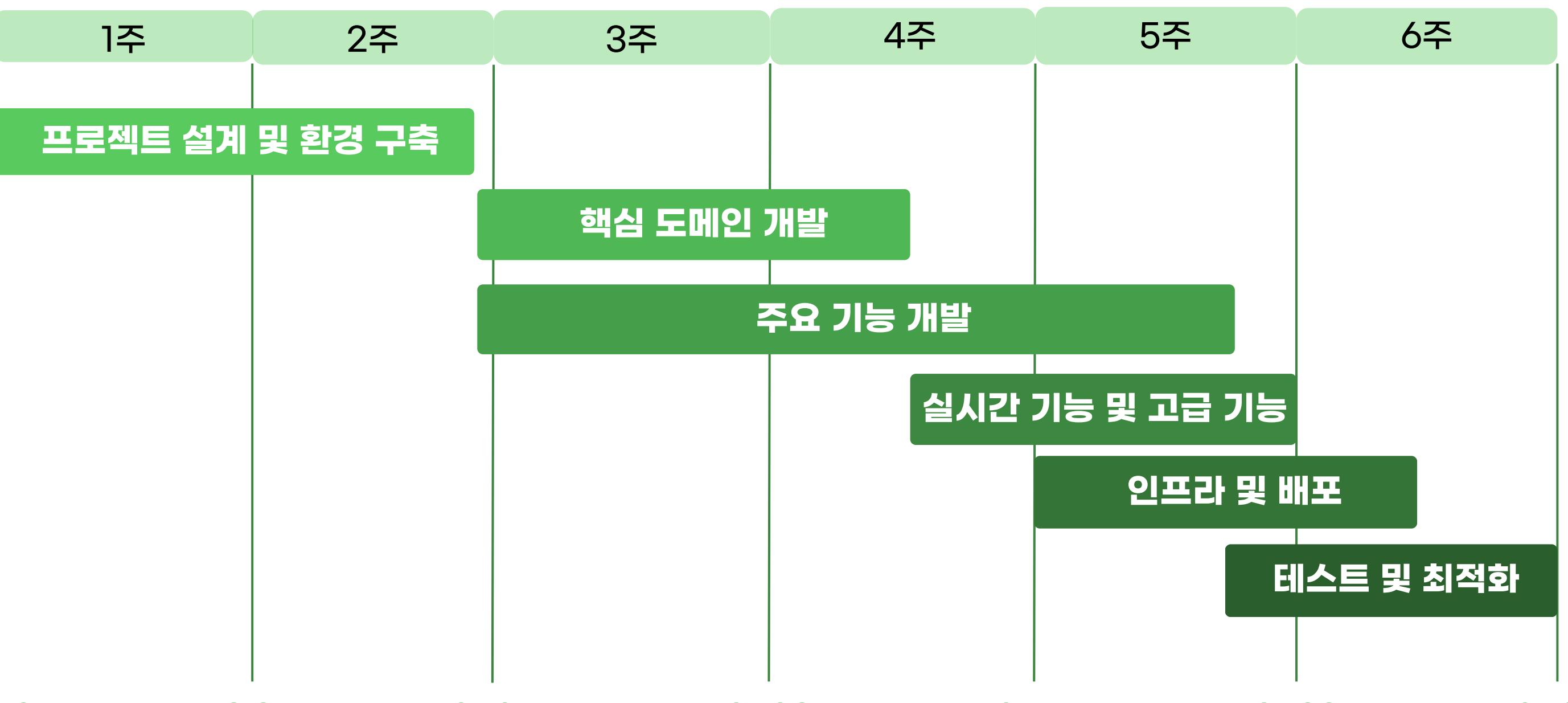
▶ GitHub Issue 기반 작업 관리

The screenshot shows a GitHub Issues board with three columns: 'Todo' (1 item), 'In Progress' (1 item), and 'Done' (10 items). The 'In Progress' item is for monitoring and postgreSQL integration. The 'Done' items include tasks like TMDB API data collection, monitoring metrics, review creation, and review search.

- 12. 18~12. 31 : ERD 설계, API 명세, 개발 환경 구축
- 12. 31~01. 11 : User/Content 도메인, JWT 인증, Spring Security
- 12. 31~01. 20 : Playlist/Review/Follow, Cursor 페이지네이션, Redis 캐싱
- 01. 12~01. 22 : WebSocket, SSE, Kafka, Elasticsearch, OAuth2
- 01. 15~01. 25 : AWS 인프라, Nginx, CI/CD, 모니터링
- 01. 20~01. 29 : 단위/통합 테스트, 성능 최적화, 리팩토링

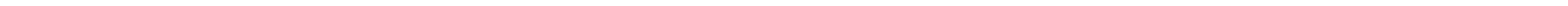
12월

1월

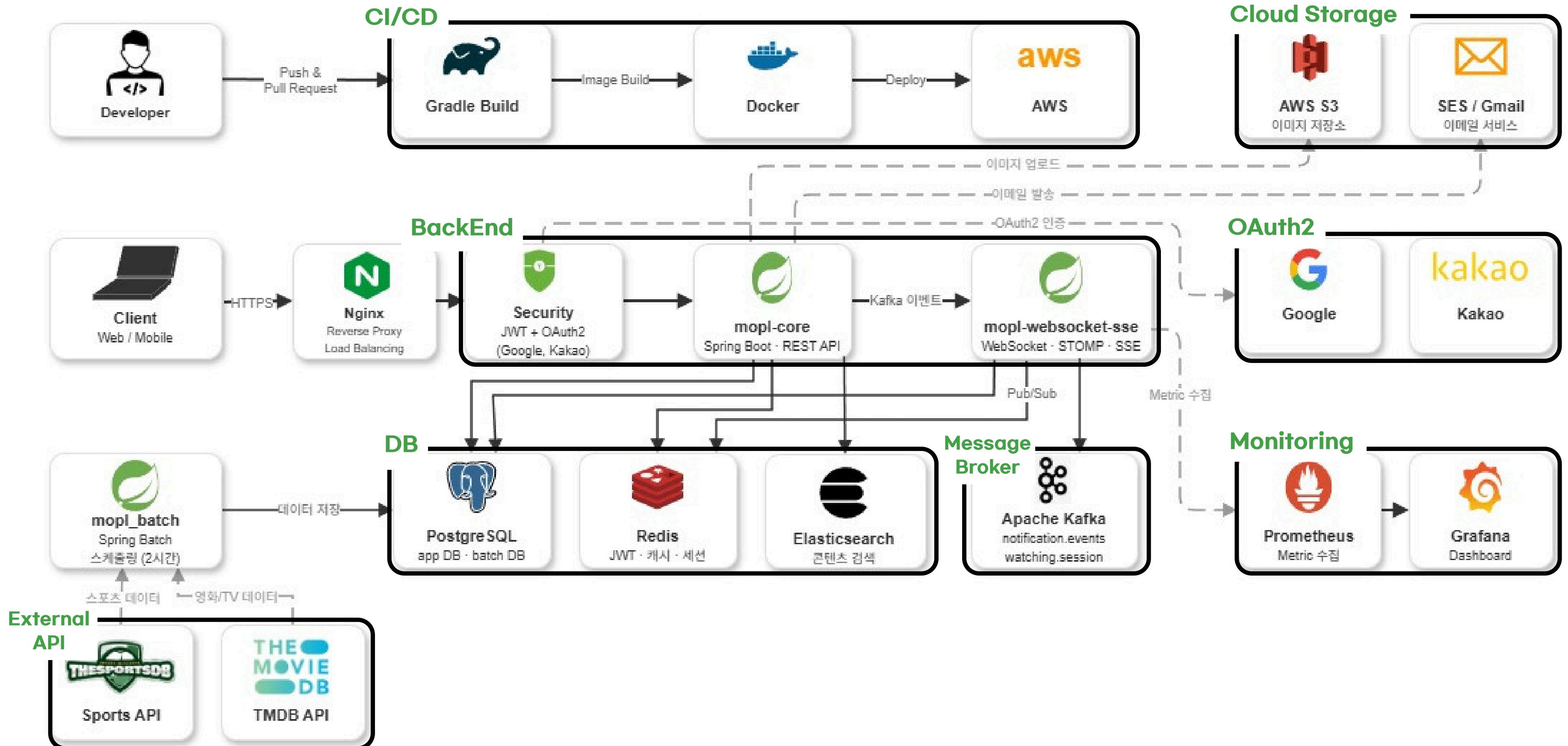


전체 기간 : 2025.12.18 ~ 2026.01.29

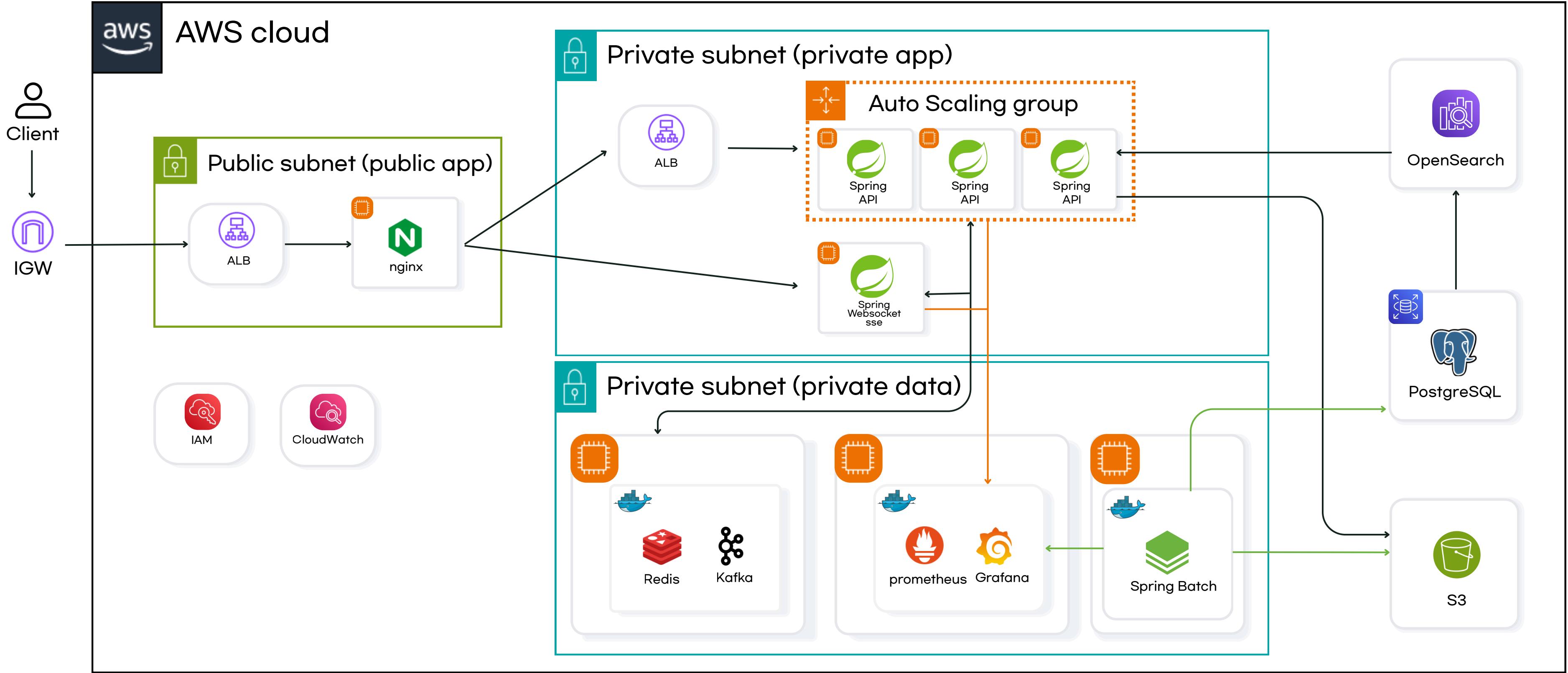
03. 시스템 전체 구조



시스템 아키텍처

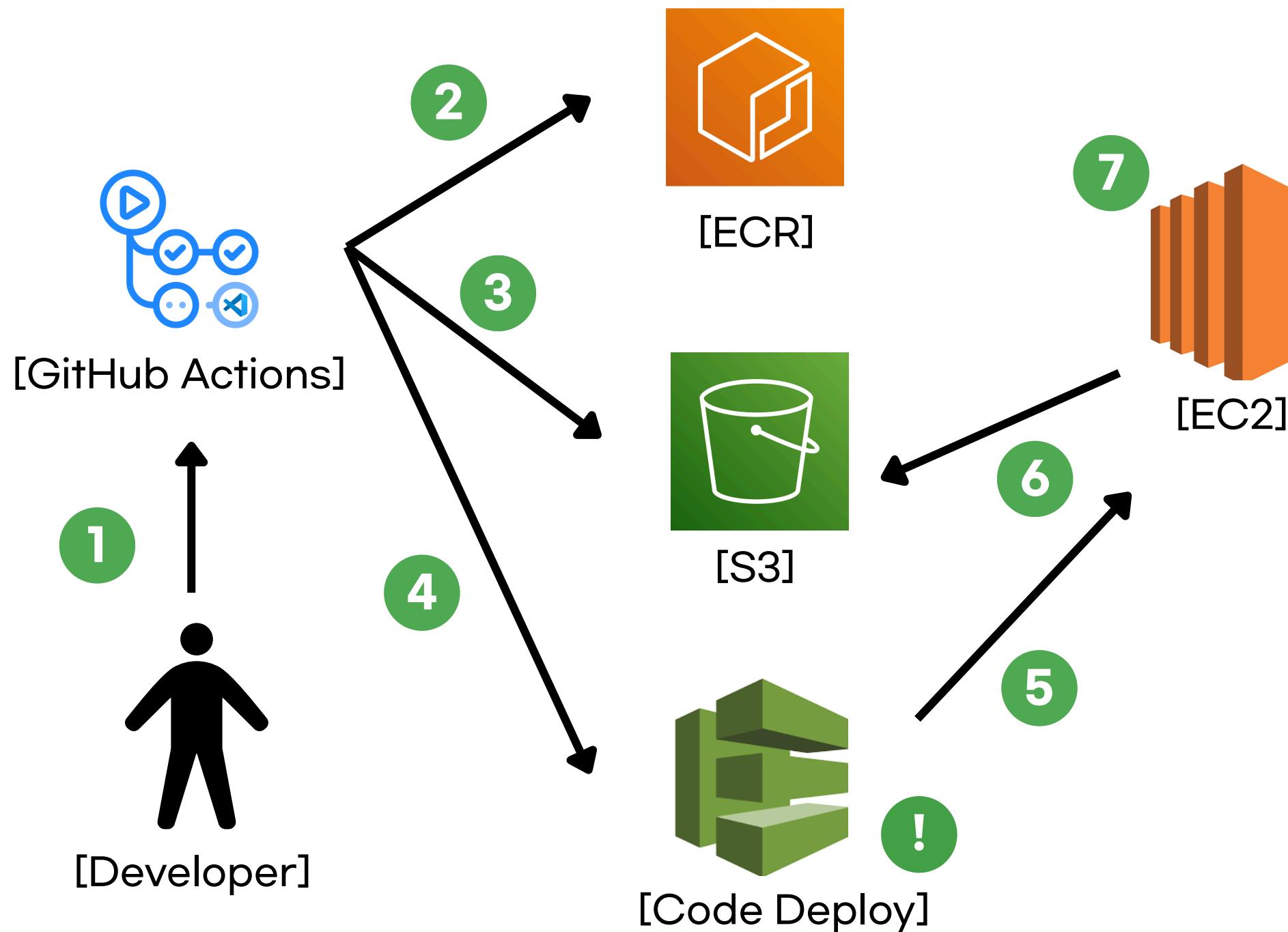


클라우드 아키텍처



CI/CD

Github Action + ECR + Code Deploy + S3 + EC2



1. 코드 Push - 파이프라인 시작

main 브랜치에 코드를 푸시하면 `deploy.yml`의 트리거가 감지

2. 빌드 및 Docker 이미지 생성

Gradle로 빌드 후 Docker 이미지 생성 후 ECR로 Push

3. 배포 산출물 S3 업로드

.env, appspec.yml, scripts/, docker-compose.yml을 tar.gz로 압축 후 S3 버킷에 업로드

4. CodeDeploy 배포 트리거

aws deploy create-deployment 명령으로 배포 트리거

5. EC2 배포 명령 전달

CodeDeploy가 EC2에 배포 명령을 전달

6. S3에서 배포 tar 파일 다운

S3에서 설치 후 /home/ubuntu/spring에 복사

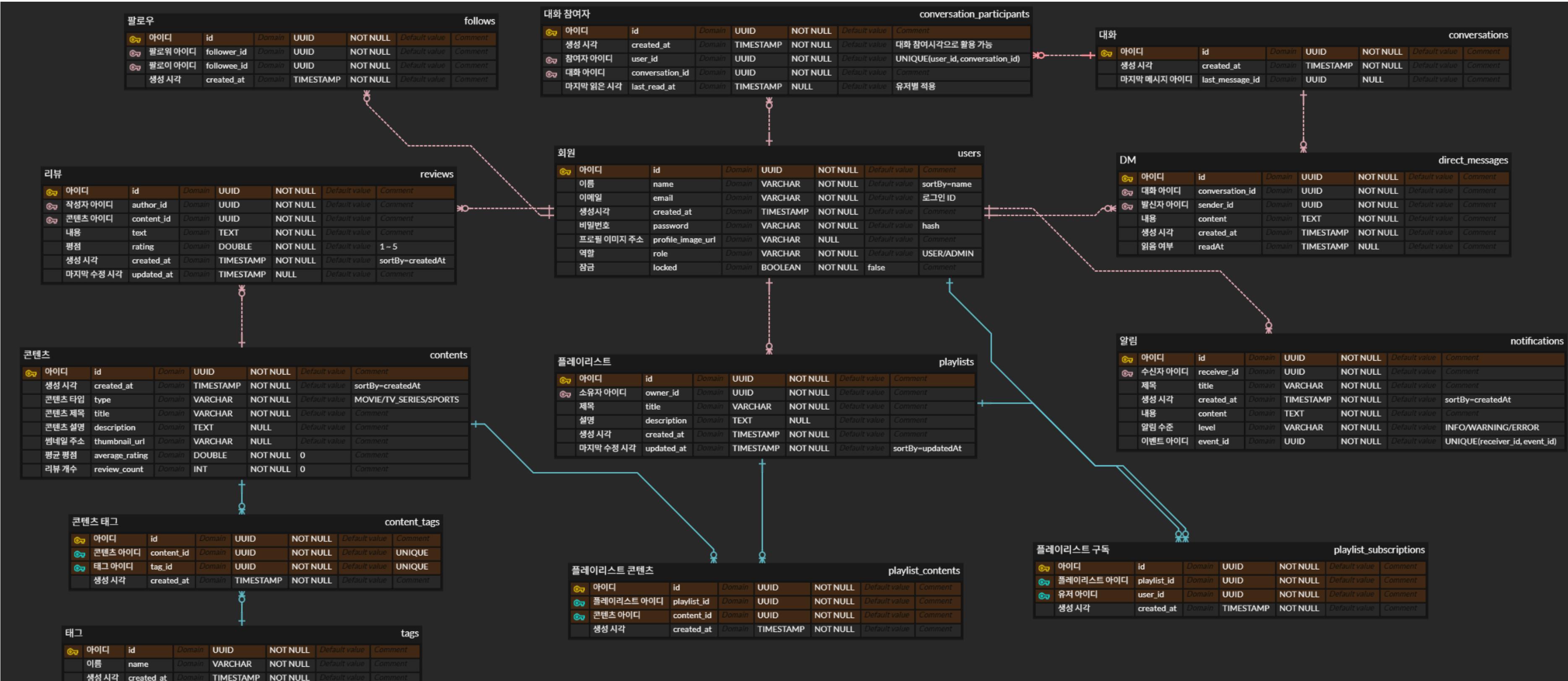
7. ECR에서 Docker 이미지 Pull

starts.sh를 통해 이미지 pull 후
docker compose up -d --build로 컨테이너 실행

! Blue/Green 및 Auto Scale

Code Deploy에서는 자체적으로 Blue/Green을 지원하며,
배포 대상을 Spring 서버의 오토스케일 그룹을 대상으로 지정함

ERD 다이어그램

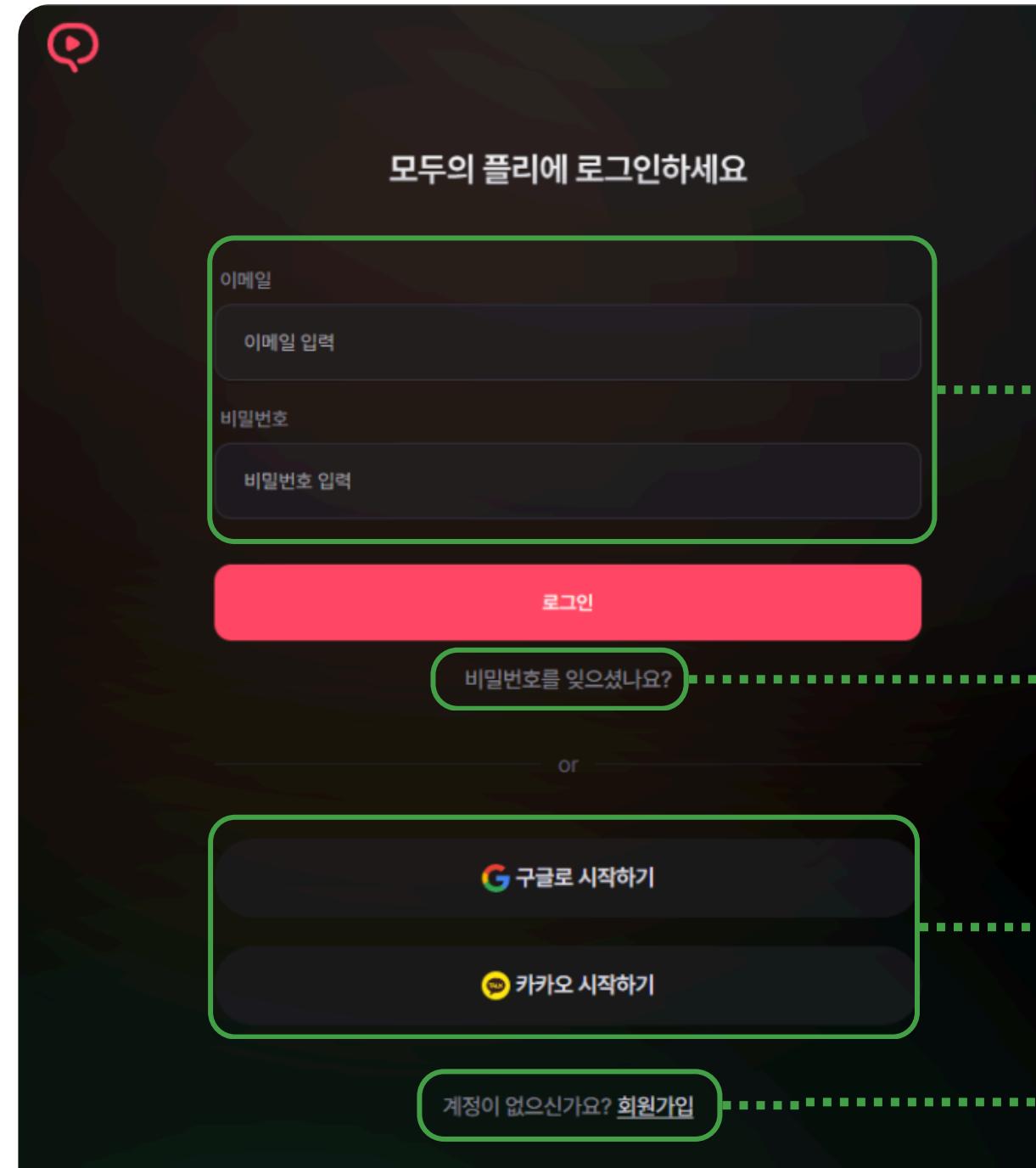


04. 결과 화면/데모



페이지 소개

회원가입 및 로그인



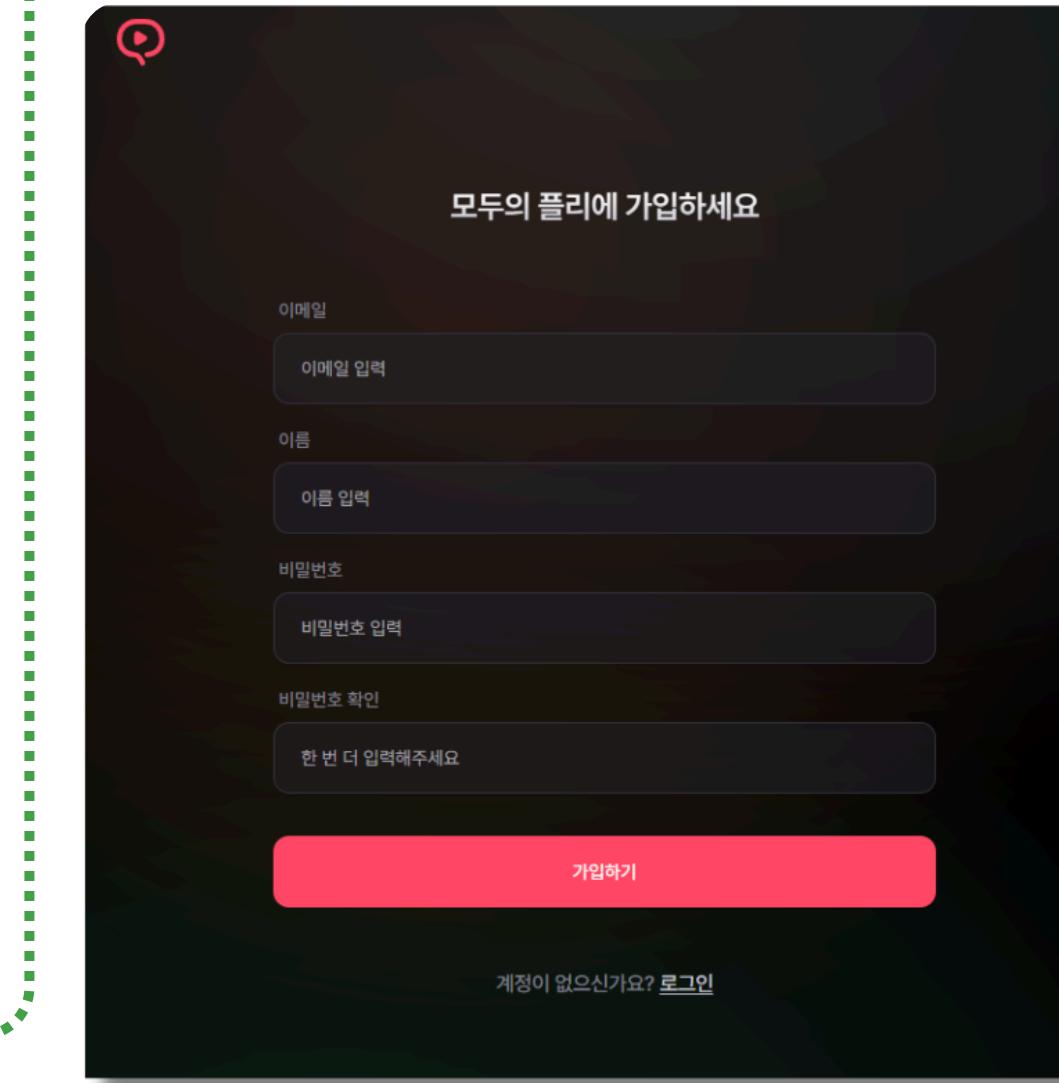
로그인

임시비밀번호

OAuth2 로그인

회원가입

- 이메일 : 이메일 형식
- 비밀번호 : 8자 이상



페이지 소개

임시비밀번호 찾기



가입된 이메일로 임시비밀번호 발송

임시 비밀번호를 받을 이메일을 입력해주세요

이메일 입력

비밀번호 초기화

로그인으로 돌아가기

[Mopl] 임시 비밀번호가 발급되었습니다

us*****@gmail.com으로
전달 받은 임시 비밀번호를 입력해주세요

02:59

임시 비밀번호
c2d84c42

임시 비밀번호 입력

비밀번호 초기화

로그인으로 돌아가기

✓ 임시 비밀번호가 이메일로 전송되었습니다

새로운 비밀번호 변경

새 비밀번호 입력
.....

새 비밀번호 확인
.....

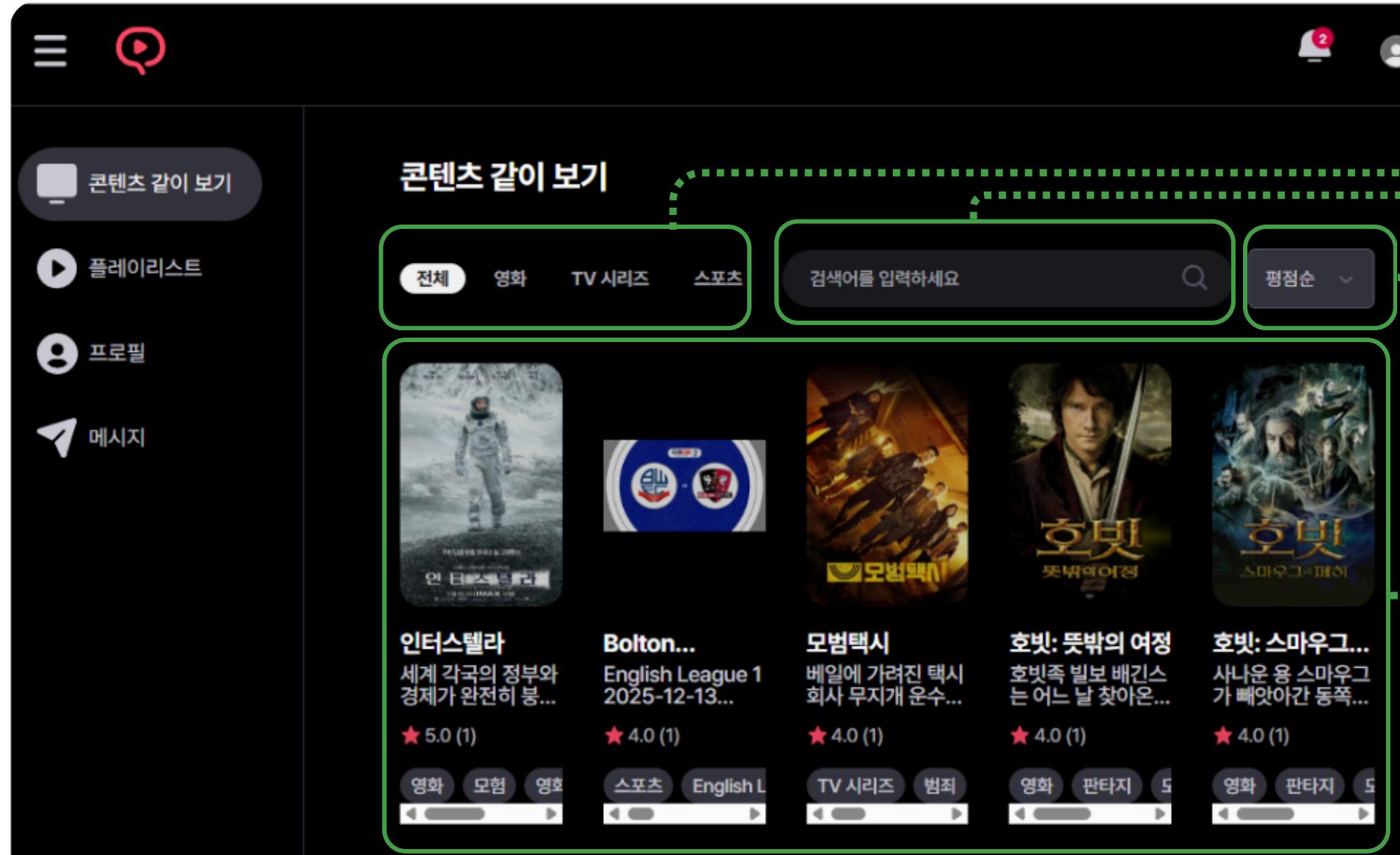
비밀번호 변경

로그인으로 돌아가기

✓ 임시 비밀번호 확인 완료

페이지 소개

콘텐츠 같이 보기 - 사용자



타입별 검색

키워드 검색

정렬 기준

콘텐츠 요약 정보

- 제목 및 설명에 키워드가 포함되면 검색

평점순

인기순

최신순

평점순

페이지 소개

콘텐츠 같이 보기 페이지 - 관리자

The screenshot shows a dark-themed user interface for managing content. On the left, there's a sidebar with icons for 'Content Together' (highlighted), 'Playlist', 'Profile', 'Message', and 'User Management'. The main area is titled 'Content Together' and features a search bar with placeholder 'Search for content' and a dropdown for 'Sort by popularity'. Below the search bar, there are five media items displayed in a grid:

- Brother's War**: Cover art of a war scene, rating 0.0 (0), genres: Drama, English, English.
- Precious Victims**: Cover art of a woman, rating 0.0 (0), genres: Drama, English, English.
- 위아스틸 히어**: Cover art with stylized letters, rating 0.0 (0), genres: Comedy, English, SF.
- ザッツ・ロ...**: Cover art of a group of people, rating 0.0 (0), genres: Drama, English, Documentary.
- Prípad pre...**: Cover art of a man in a suit, rating 0.0 (0), genres: Drama, English.

Each item has its title, rating, and genre tags below it.



- 관리자가 직접 콘텐츠 등록

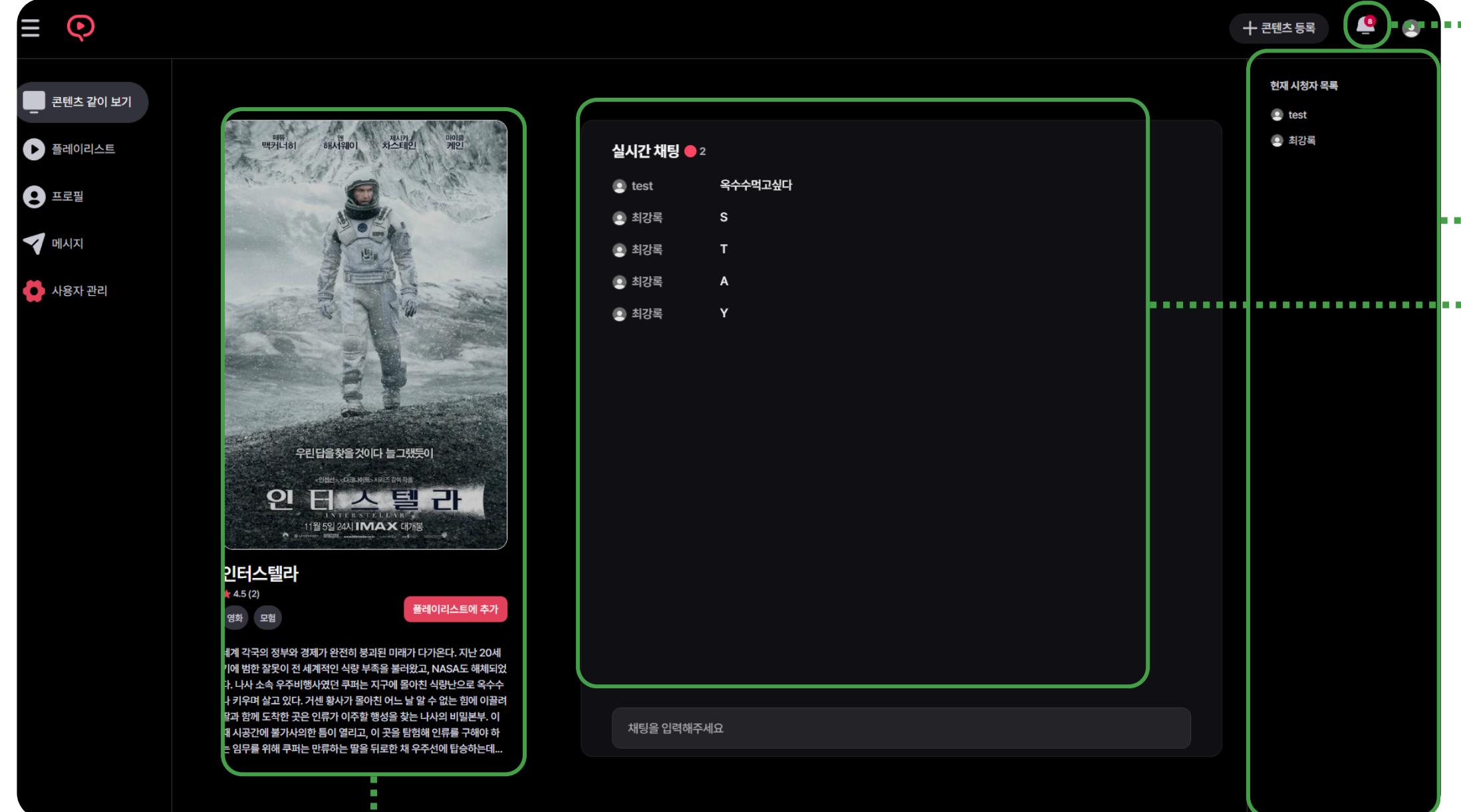
썸네일, 제목, 설명, 콘텐츠 유형 필수값

This is a modal dialog for registering content. It includes fields for 'Thumbnail image' (with a note about selecting a file), 'Title' (input field), 'Description' (input field), and 'Content type' (dropdown menu currently set to 'Movie'). At the bottom are 'Cancel' and 'Register' buttons.



페이지 소개

콘텐츠 상세 보기



콘텐츠 상세 내용

알림

- 팔로우 대상이 시청하는 콘텐츠 알림

실시간 시청자

알림 1

최강록님이 시청을 시작했어요.
방금

[인터스텔라] 을(를) 시청중이에요.

현재 시청자 목록

- test
- 최강록

실시간 채팅

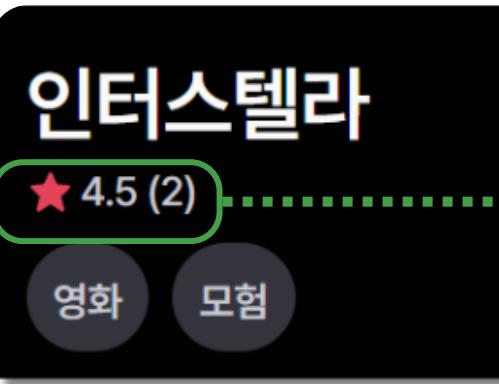
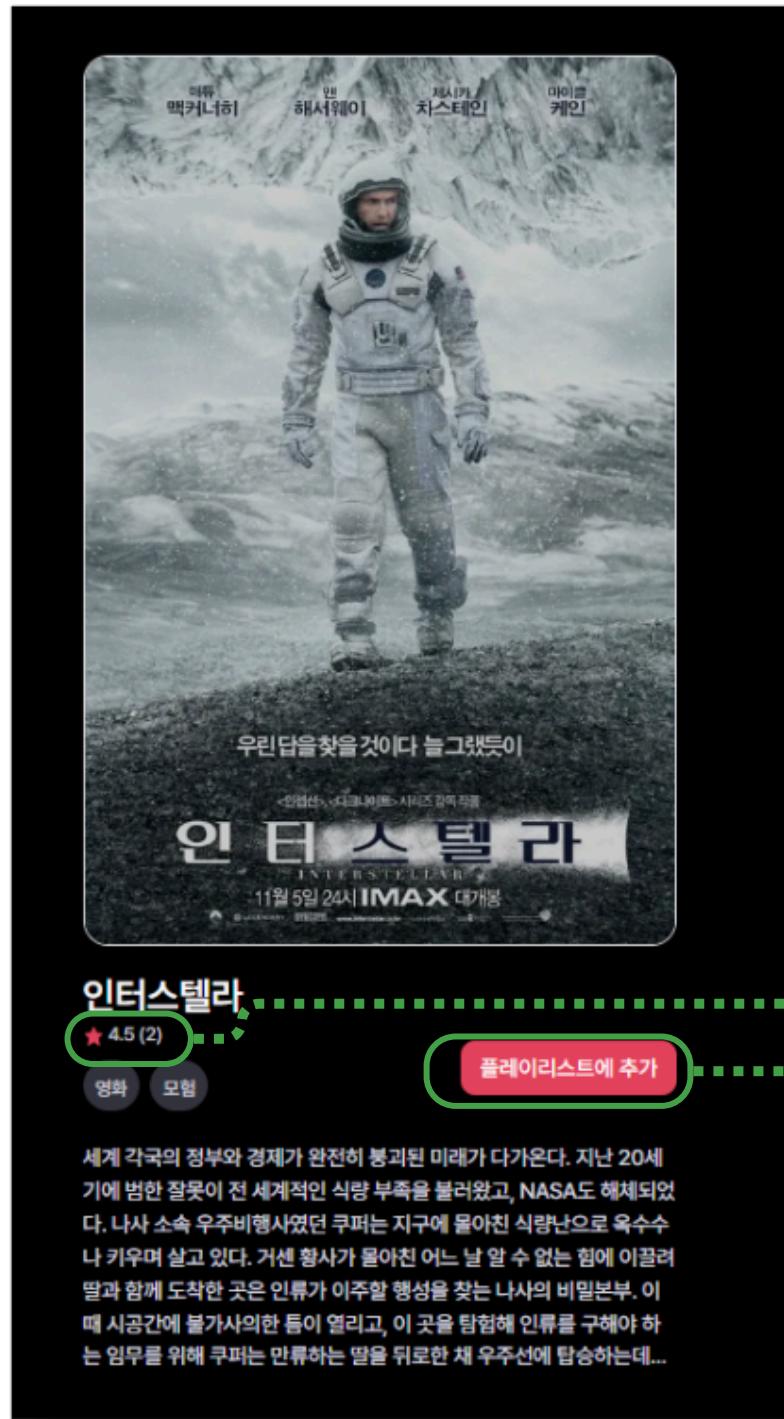
실시간 채팅

실시간 시청자 수

- test 옥수수먹고싶다
- 최강록 S
- 최강록 T
- 최강록 A
- 최강록 Y

페이지 소개

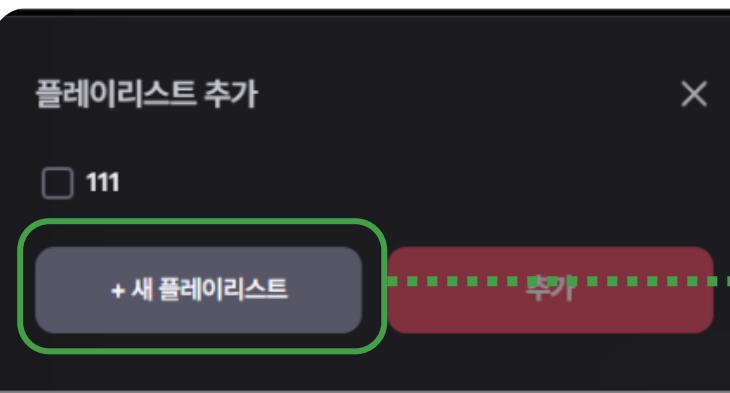
콘텐츠 상세 보기



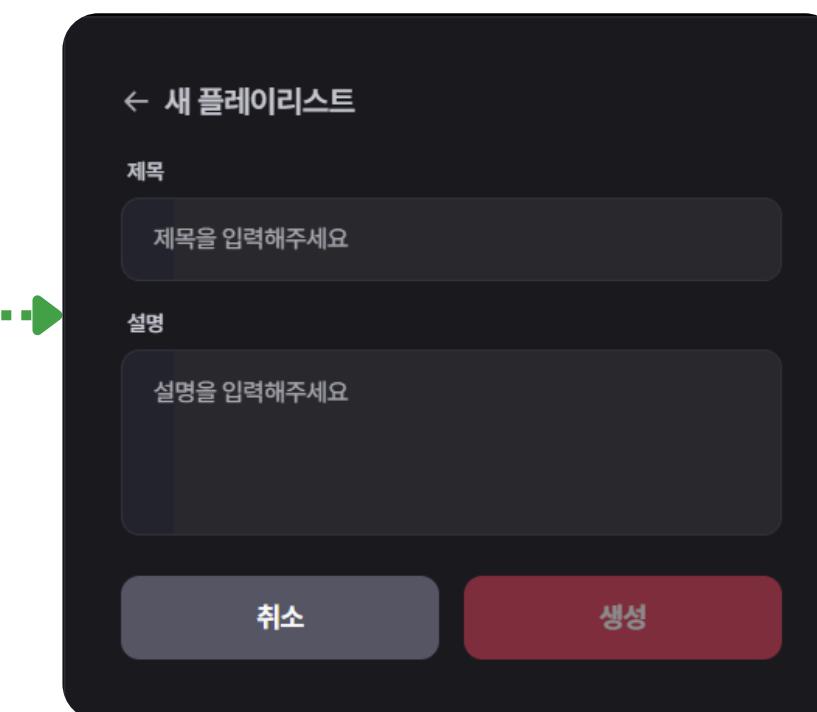
콘텐츠 리뷰 평점

- 클릭 시
리뷰 상세 페이지 팝업

플레이리스에
콘텐츠 추가

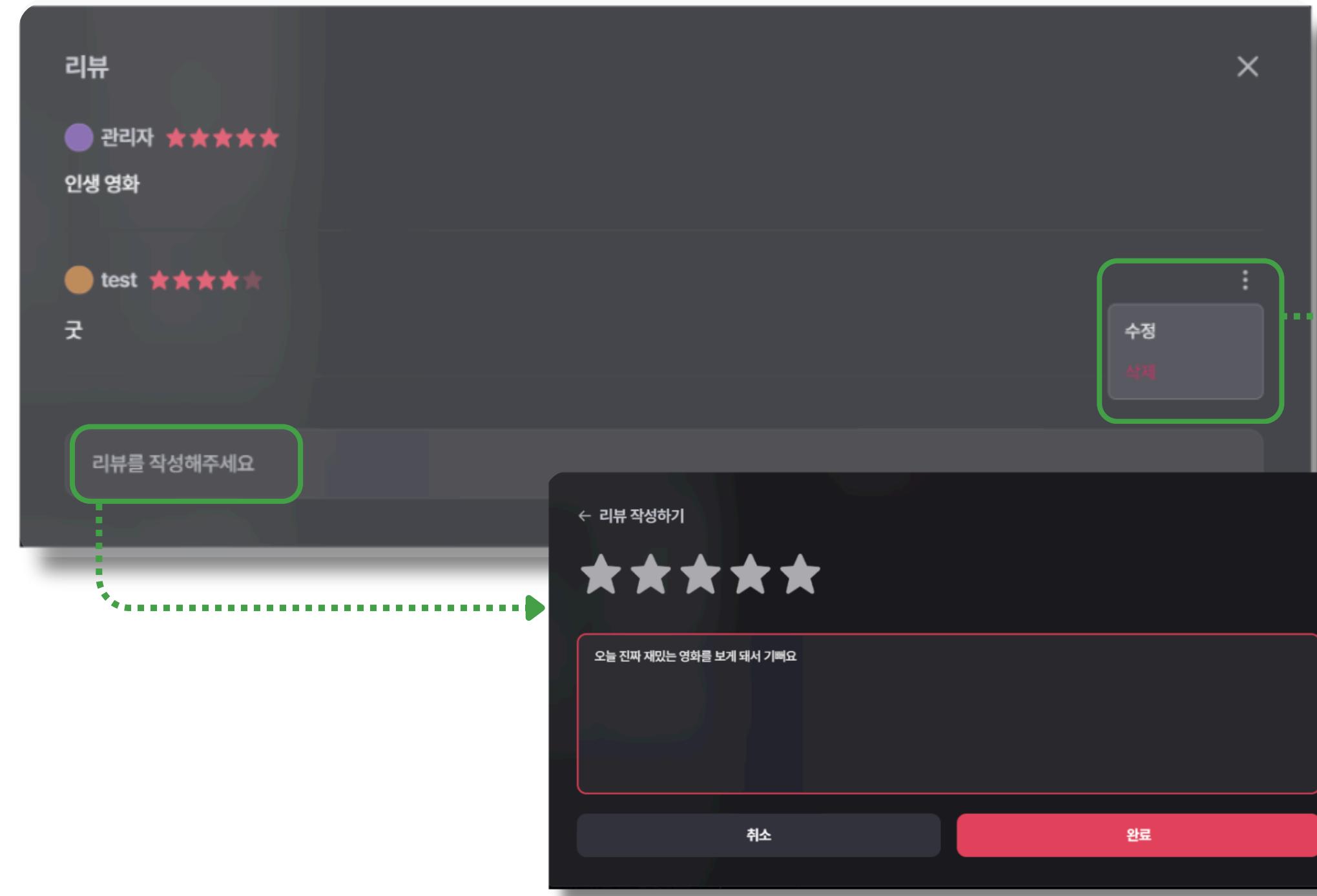


플레이리스트 생성



페이지 소개

리뷰 상세 페이지



리뷰 수정 및 삭제

- 자신이 쓴 리뷰는 수정 및 삭제 가능

리뷰 작성

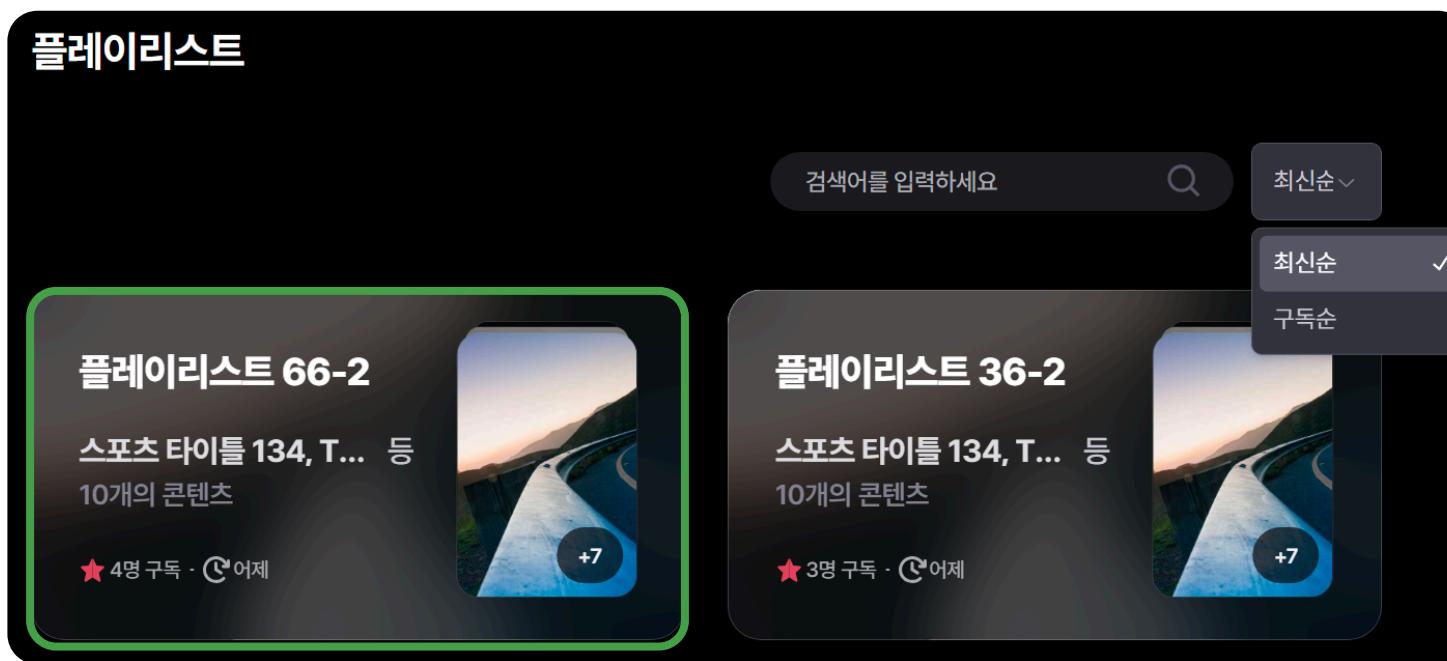
- 평점은 1~5 점까지
- 리뷰 내용 필수
- 한 콘텐츠에 유저당 하나의 리뷰만 작성 가능

이미 해당 콘텐츠에 리뷰를 작성하셨습니다

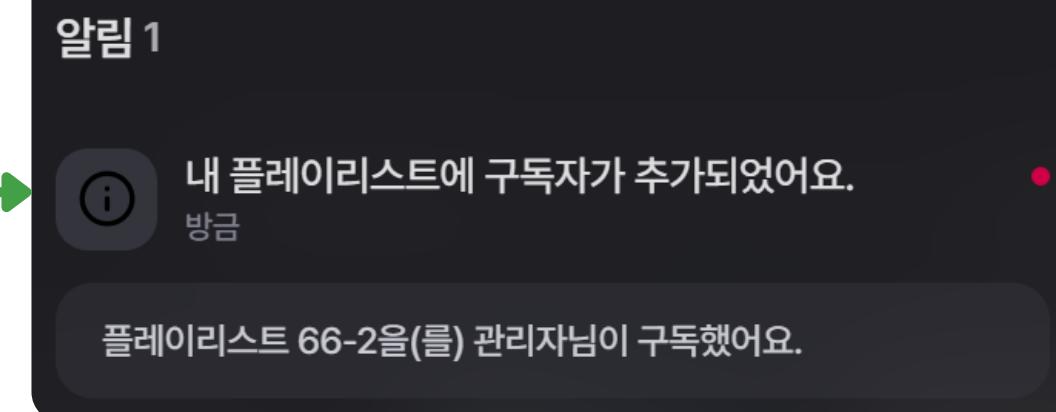
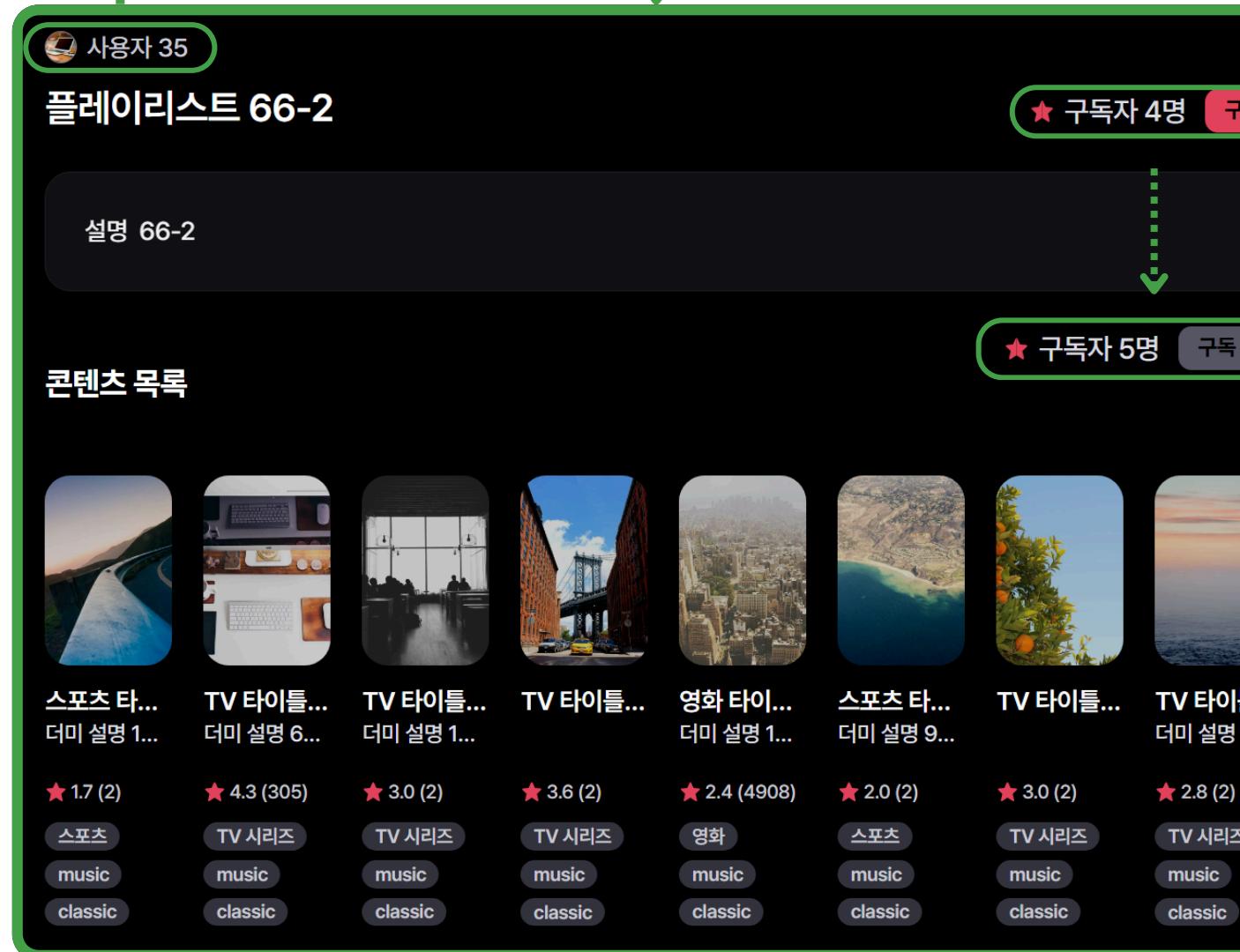
페이지 소개

플레이리스트 - 전체 조회 및 구독

플레이리스트
소유자



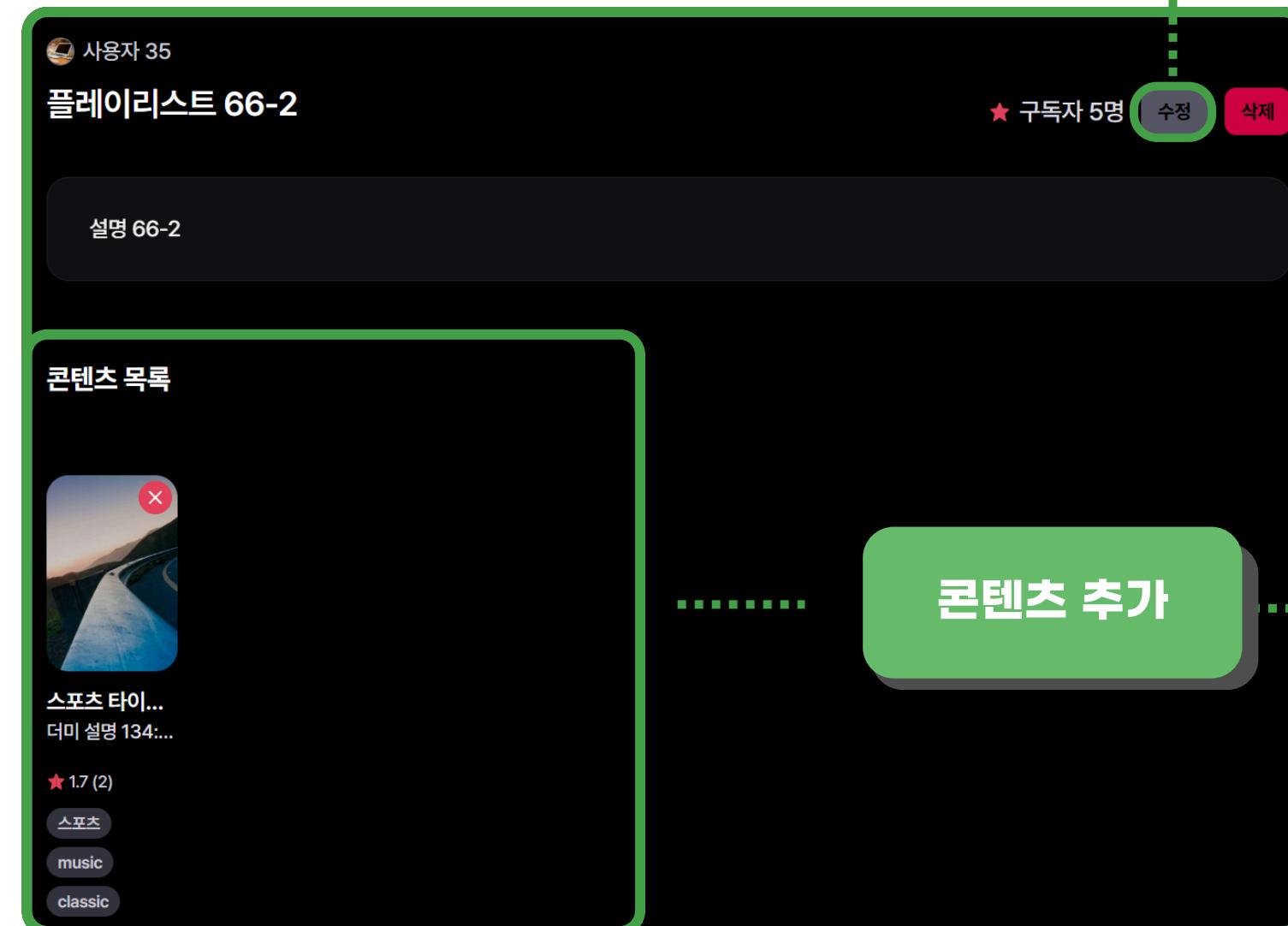
구독 시
소유자에게 알림



소유자 외
사용자의 화면

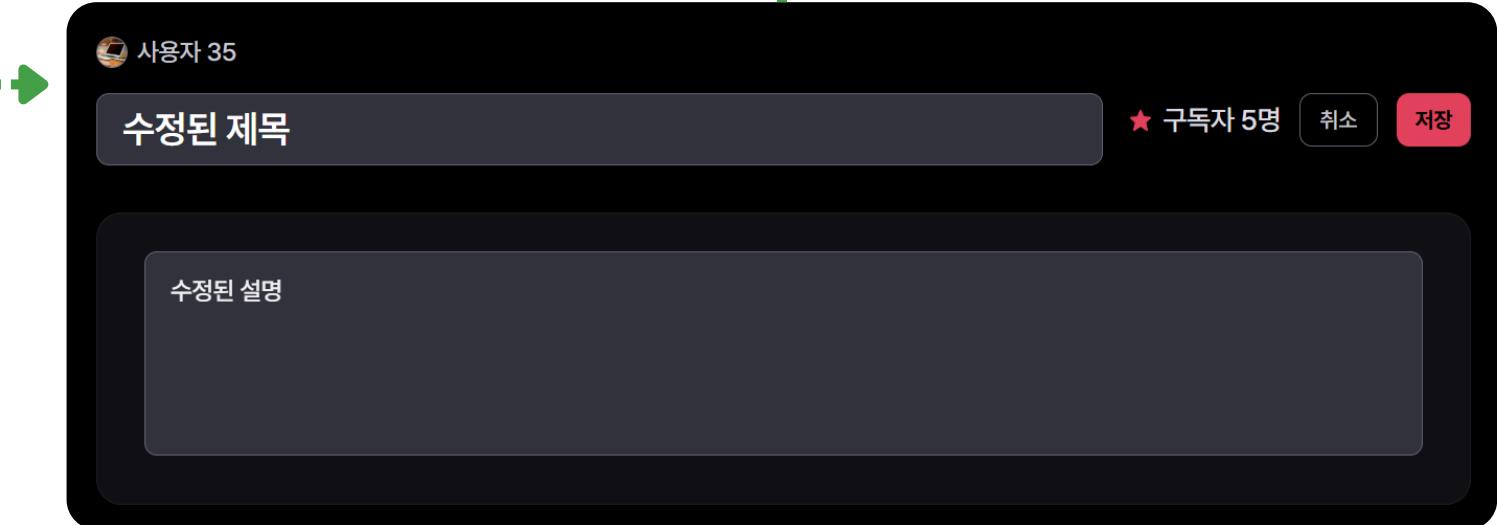
페이지 소개

플레이리스트 - 정보 수정 및 편집

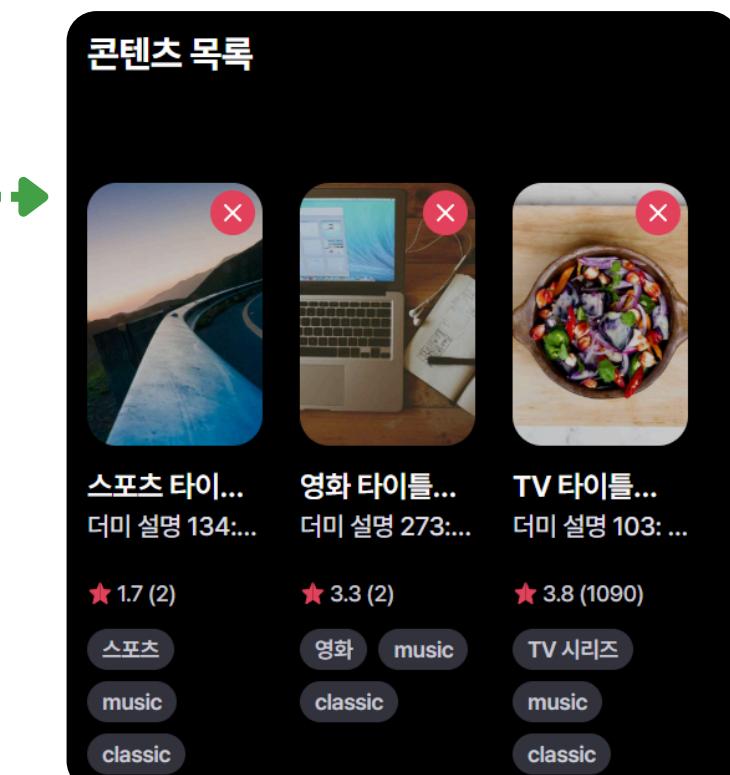


콘텐츠 추가

소유자의 화면



제목과 설명 설정



구독자 알림

페이지 소개

프로필 - 정보 및 플레이리스트 관리

프로필 사진과
사용자명 수정

The diagram illustrates the user profile editing process and the management of playlists.

프로필 사진과 사용자명 수정 (Profile Photo and Username Change):

- Initial State:** A circular profile picture of a laptop, labeled "사용자 35" (User 35) and "팔로우 7".
- Editing Step:** An arrow points from the initial state to a screen where the user is changing the profile photo to a landscape image and the username to "변경" (Change). It also shows the follower count as "팔로우 7".
- Final State:** The profile is updated with the new photo and username, and the follower count remains at "팔로우 7".

플레이리스트 2 (Playlists 2):

- Initial State:** A list of 2 playlists:
 - 수정된 제목 (Edited Title):** 스포츠 타이틀 134, 영화 ... (3개의 콘텐츠). Last updated 9시간 전 (9 hours ago).
 - 플레이리스트 66-1:** 스포츠 타이틀 134, T... 등 10개의 콘텐츠. Last updated 12월 27일 (December 27).
- Final State:** A green callout box labeled "구독 중인 플레이리스트 확인" (Check Subscribed Playlists) points to the list of playlists.

구독 중인 플레이리스트 5 (Subscribed Playlists 5):

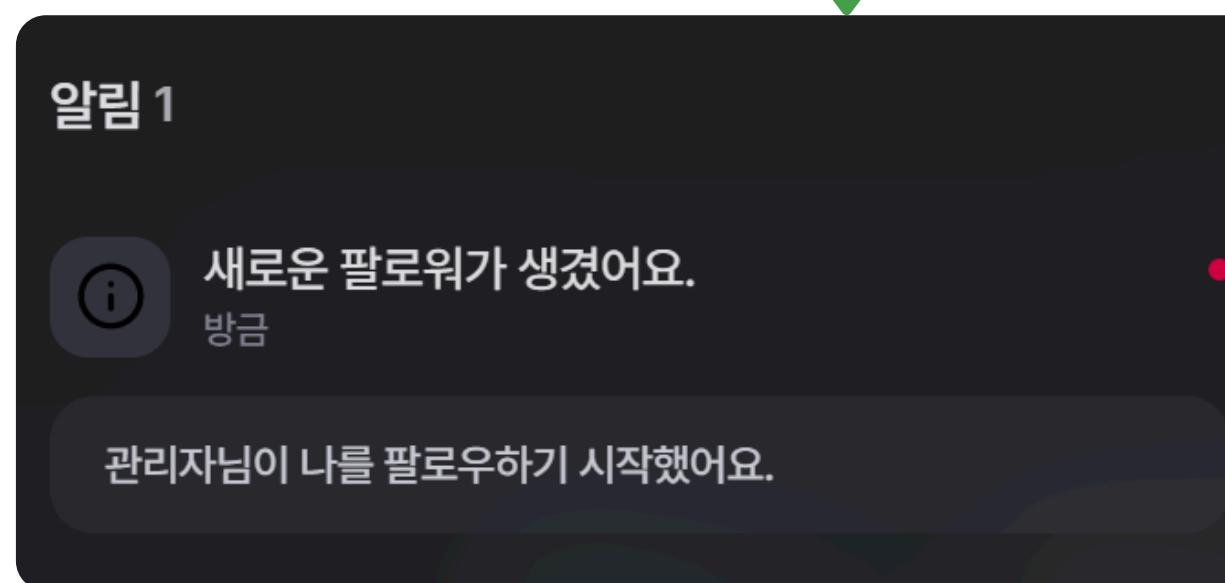
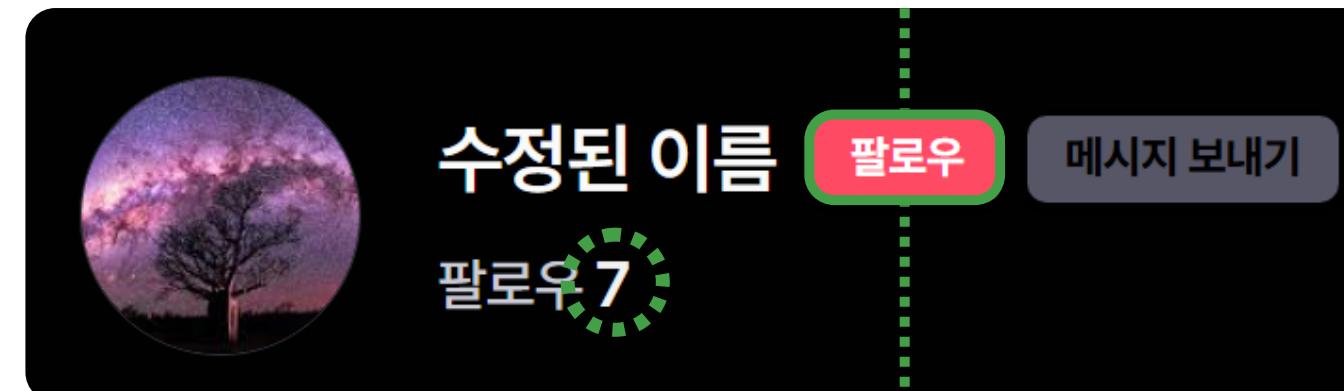
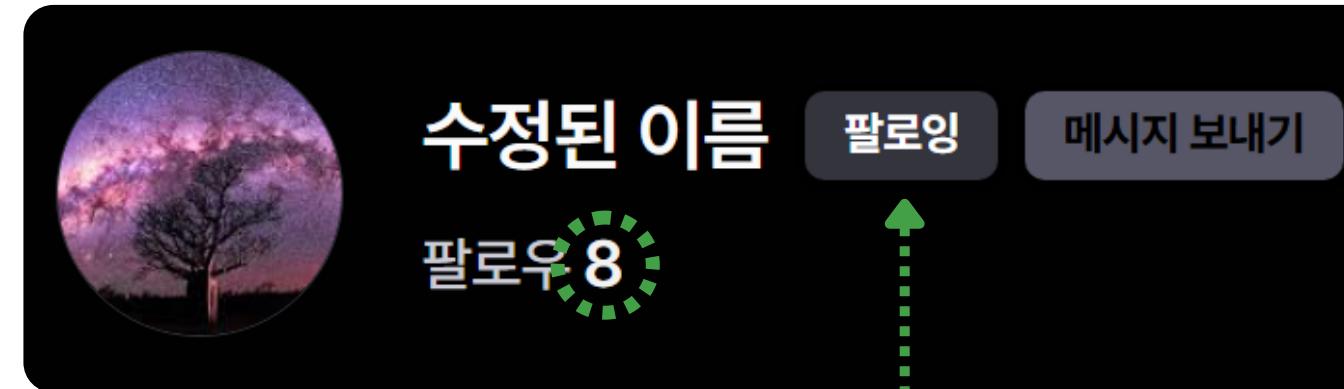
- Initial State:** A list of 5 playlists:
 - 플레이리스트 69-1:** 스포츠 타이틀 134, T... 등 10개의 콘텐츠. Last updated 어제 (yesterday).
 - 플레이리스트 85-1:** 스포츠 타이틀 134, T... 등 10개의 콘텐츠. Last updated 1월 16일 (January 16).
- Final State:** A green callout box labeled "다른 사용자의 프로필 확인" (Check Other User's Profile) points to the list of playlists.

구독 중인 플레이리스트 3 (Subscribed Playlists 3):

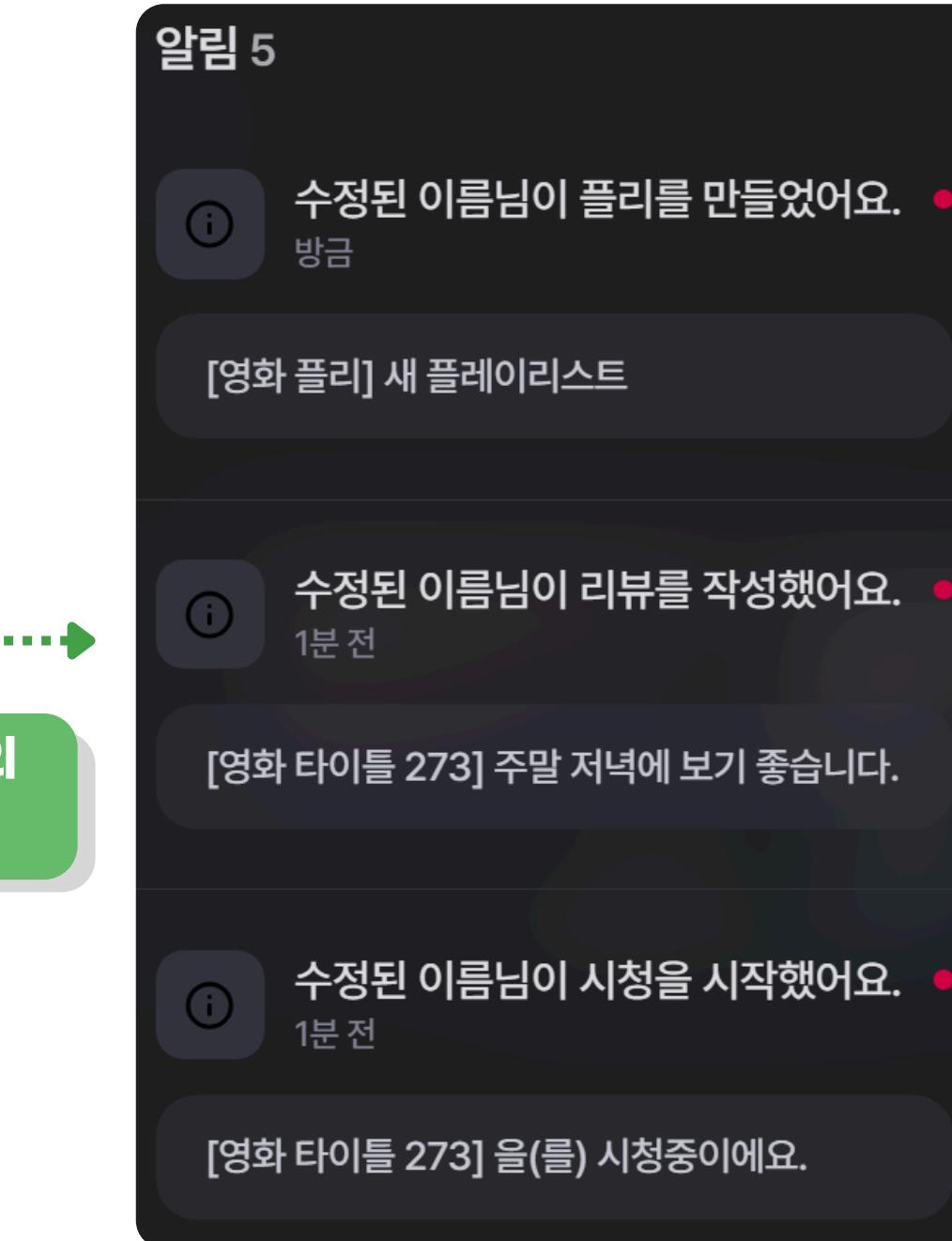
- Initial State:** A list of 3 playlists:
 - 플레이리스트 12-1:** 스포츠 타이틀 134, T... 등 10개의 콘텐츠. Last updated 1월 14일 (January 14).
 - 플레이리스트 100-1:** 스포츠 타이틀 134, T... 등 10개의 콘텐츠. Last updated 1월 2일 (January 2).

페이지 소개

팔로우와 활동 알림



Followee가
받는 알림



Follower가
받는 활동 알림

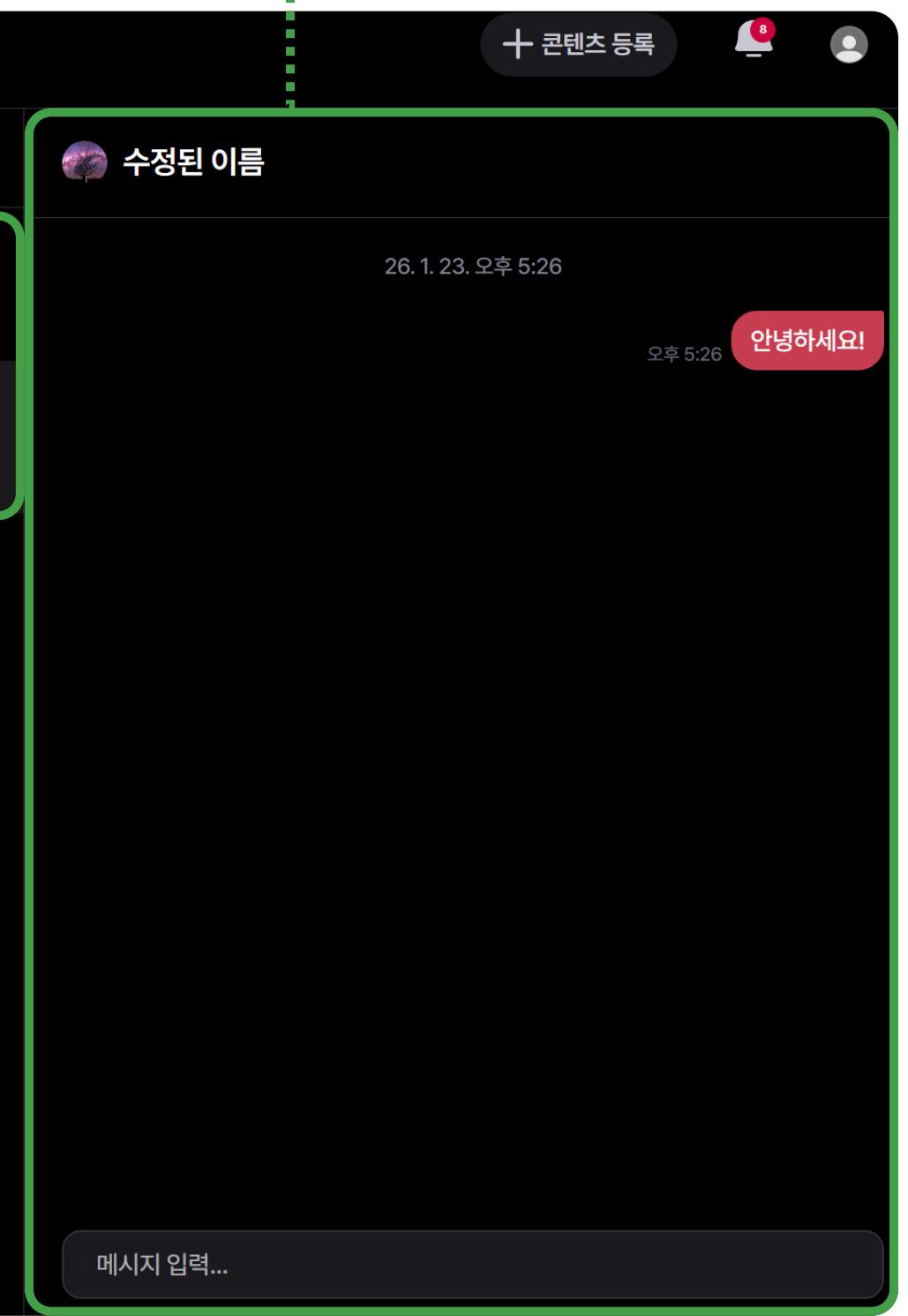
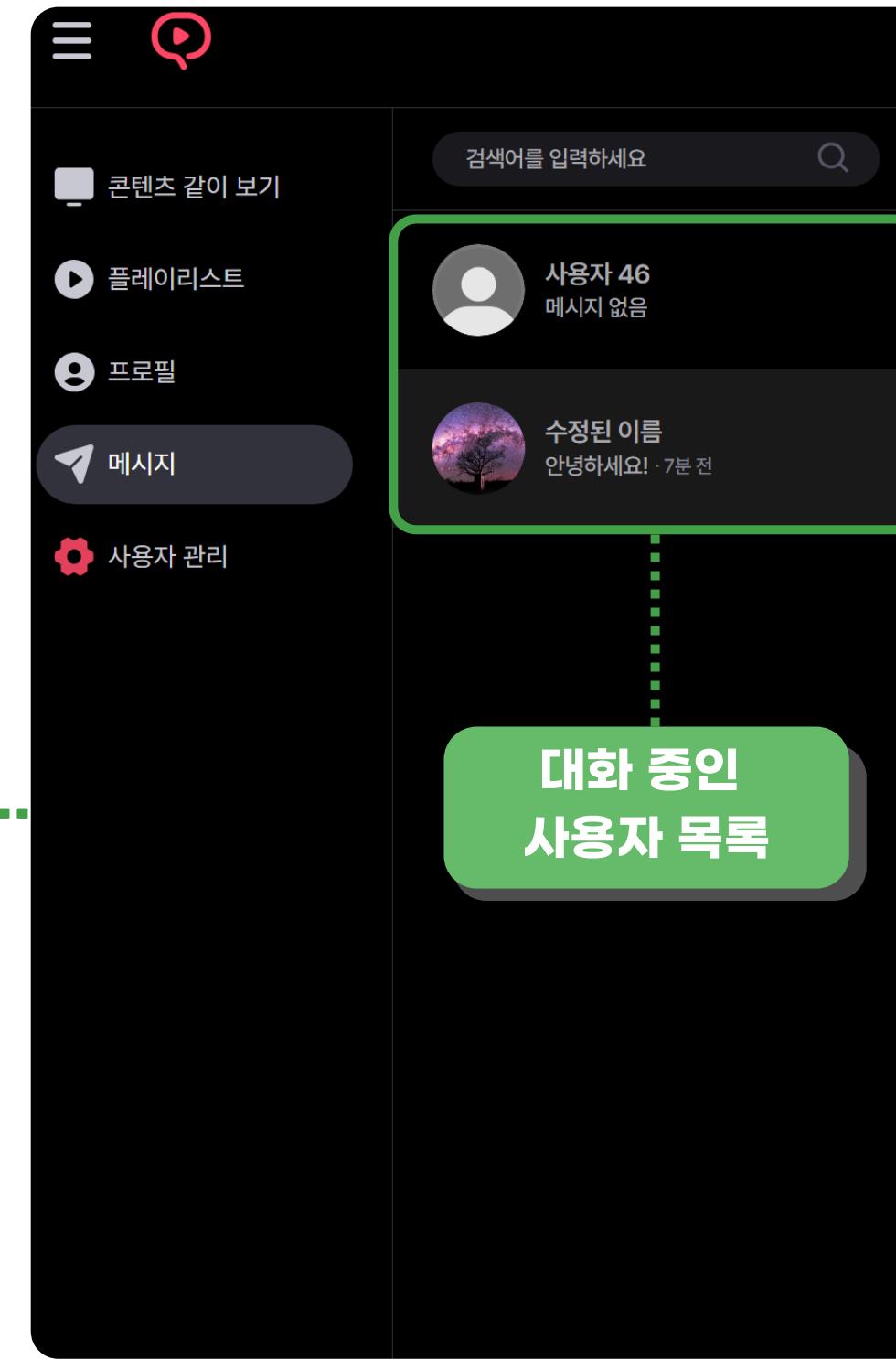
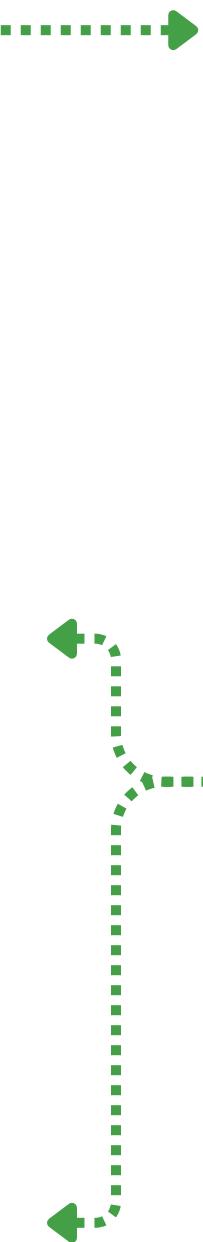
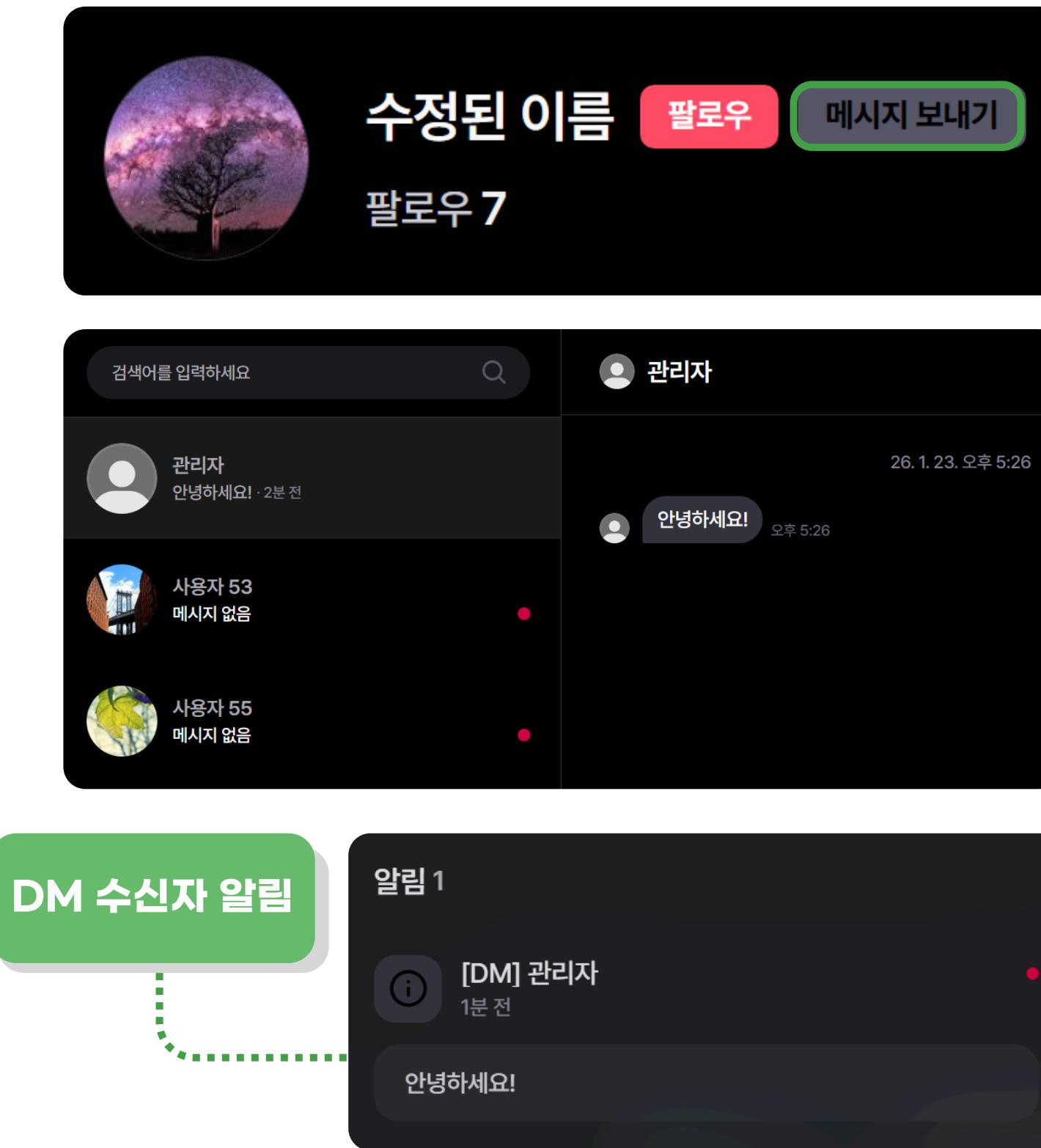
플레이리스트 생성

콘텐츠 리뷰 작성

콘텐츠 시청 시작

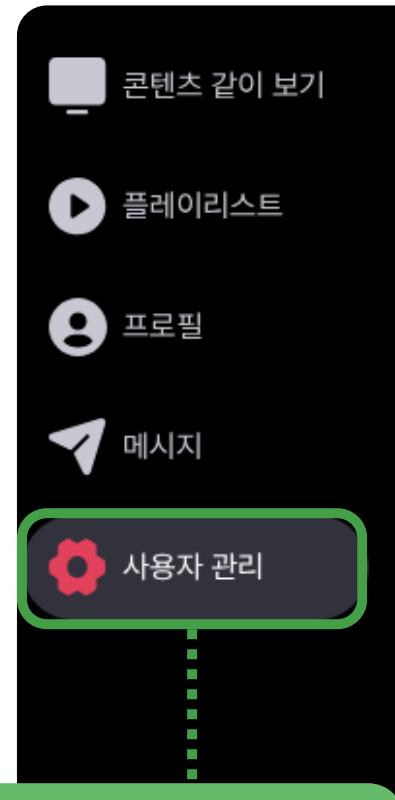
페이지 소개

메시지 - DM으로 개인 간 대화



페이지 소개

사용자 관리 - 조회 및 권한 조정 (관리자 계정)



관리자 권한
계정에만 노출

사용자 관리

Name, Email,etc...



이름 ↑	이메일 ↑	가입일 ↑
관리자	admin@mopl.com	2026. 1. 27
정기주	halogiju123@gmail.com	2026. 1. 27
정기주	4689974291@kakao.com	2026. 1. 27
정기주	raed123456@knu.ac.kr	2026. 1. 27
홍길동	gildong@mopl.com	2026. 1. 27
테스트 사용자	user@mopl.com	2026. 1. 27
test	test@email.com	2026. 1. 27
test0001	user0001@example.com	2026. 1. 27
test0002	user0002@example.com	2026. 1. 27

사용자의 계정 잠금
또는 권한 변경 가능

계정 잠금 ↑	ADMIN ↑
잠금	<input checked="" type="checkbox"/>
잠금	<input type="checkbox"/>
잠금	<input checked="" type="checkbox"/>
잠금	<input checked="" type="checkbox"/>
잠금	<input type="checkbox"/>
잠금	<input checked="" type="checkbox"/>
잠금	<input checked="" type="checkbox"/>
잠금	<input type="checkbox"/>
잠금	<input checked="" type="checkbox"/>
잠금	<input type="checkbox"/>

권한 변경 대상
사용자 알림 전송

알림 1

내 권한이 변경되었어요.

4분 전

내 권한이 USER에서 ADMIN(으)로 변경되었어요.

권한 변경

이 사용자의 권한을 ADMIN으로 변경하시겠습니까?

취소

변경

시청 세션

세션 구현 설계

시청 세션 요구 사항

- ✓ 콘텐츠 별 시청자 수를 알아야함
- ✓ 시청자 별 현재 시청 콘텐츠를 알아야함
- ✓ 콘텐츠 채팅에 참가한 유저의 요약 정보를 알아야함

특성	RDB	Redis
접속/종료 빈번	매번 INSERT/DELETE + 디스크 I/O	인메모리 연산, 고속
영속성	영구 저장 (불필요)	휘발성 (적합)
비정상 종료 대책	만료 컬럼 + cleanup 배치 필요	TTL 자동 만료

시청 세션 분석

- ✓ 모두의플리 서비스는 콘텐츠의 대한 정보, 평가, 실시간 채팅이 핵심 서비스들이기에 상세보기 페이지 접속과 종료가 빈번할 것으로 예상됨
- ✓ 비정상종료에 대한 대책이 필요로함

✓ 저장소는 Redis로 결정

- 저장소는 Redis로 결정빈번한 업데이트와 데이터 영속성 특징, 비정상 종료 시 유령데이터 제거에 적합한 레디스로 채택
- 시청 세션에 대한 조회는 레디스에서 이뤄져서 RDB의 읽기 부하 감소

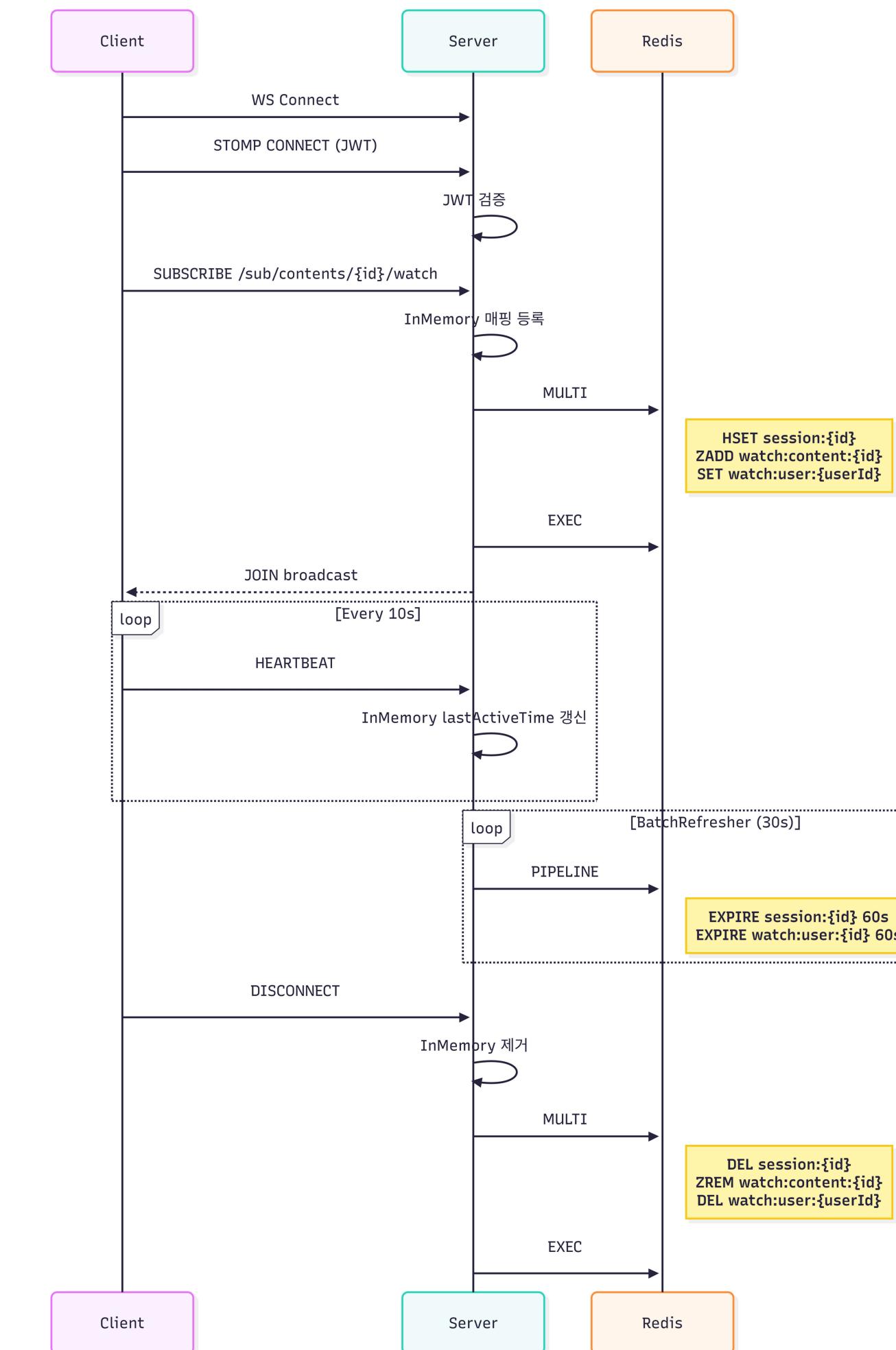
시청 세션

웹소켓과 시청 세션 흐름도

시퀀스 다이어그램

1. 클라이언트가 서버에 웹소켓 연결시도(헤더에 JWT 포함)
2. 연결을 성공하면 서버에선 인메모리에 웹소켓세션 정보를 저장하고 레디스에 시청 세션정보 저장
3. 클라이언트와 서버는 10초 간격으로 HEARTBEAT 신호 전송, 인메모리에서 활동시간 기록
4. 30초마다 TTL 갱신
5. 접속 종료 신호를 받으면 인메모리와 레디스 정보 삭제

- Redis Namespace 구조
- session:{sessionId}
개별 세션 데이터 Hash(userId, contentId, createdAt)
- watch:content:{contentId}
콘텐츠별 시청자 목록 ZSet(정렬 + 카운트)
- watch:user:{userId}
유저세션 역인덱스 String



알림 생성 및 관리

SSE 및 Kafka 채택 흐름

알림 기획 및 설계 (1)

- 알림 기능은 사용자에게 정상 도착이 중요
- 추가적인 사용자 입력 또는 반응은 불필요
- 알림은 각 도메인에서 다양한 형태로 발생
- 전송 단계의 문제 발생 시, 비즈니스 로직에 영향을 주지 않고 재시도하여 문제 해결 필요



통신 방식	WS	SSE	Polling
흐름	서버 ↔ 클라 양방향	서버 → 클라 단방향 스트리밍	클라 → 서버 주기적 요청
실시간성	실시간 통신	실시간에 가까운 푸시	지연 발생 가능
자원 소모	높은 편	낮음	높음
구조 적합성	2위	1위	3위



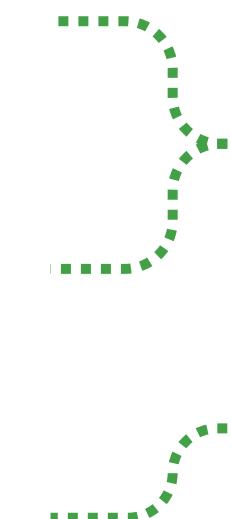
팔로워나 구독자에게
대용량 fan-out 발생



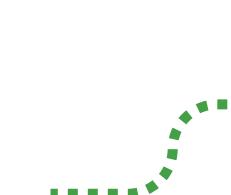
도메인 핵심 기능과
알림 이벤트의 분리



기준을 잡아 재시도



비동기 처리 및
순간적 부하 감당



면등성 설계 및
SSE와의 연계

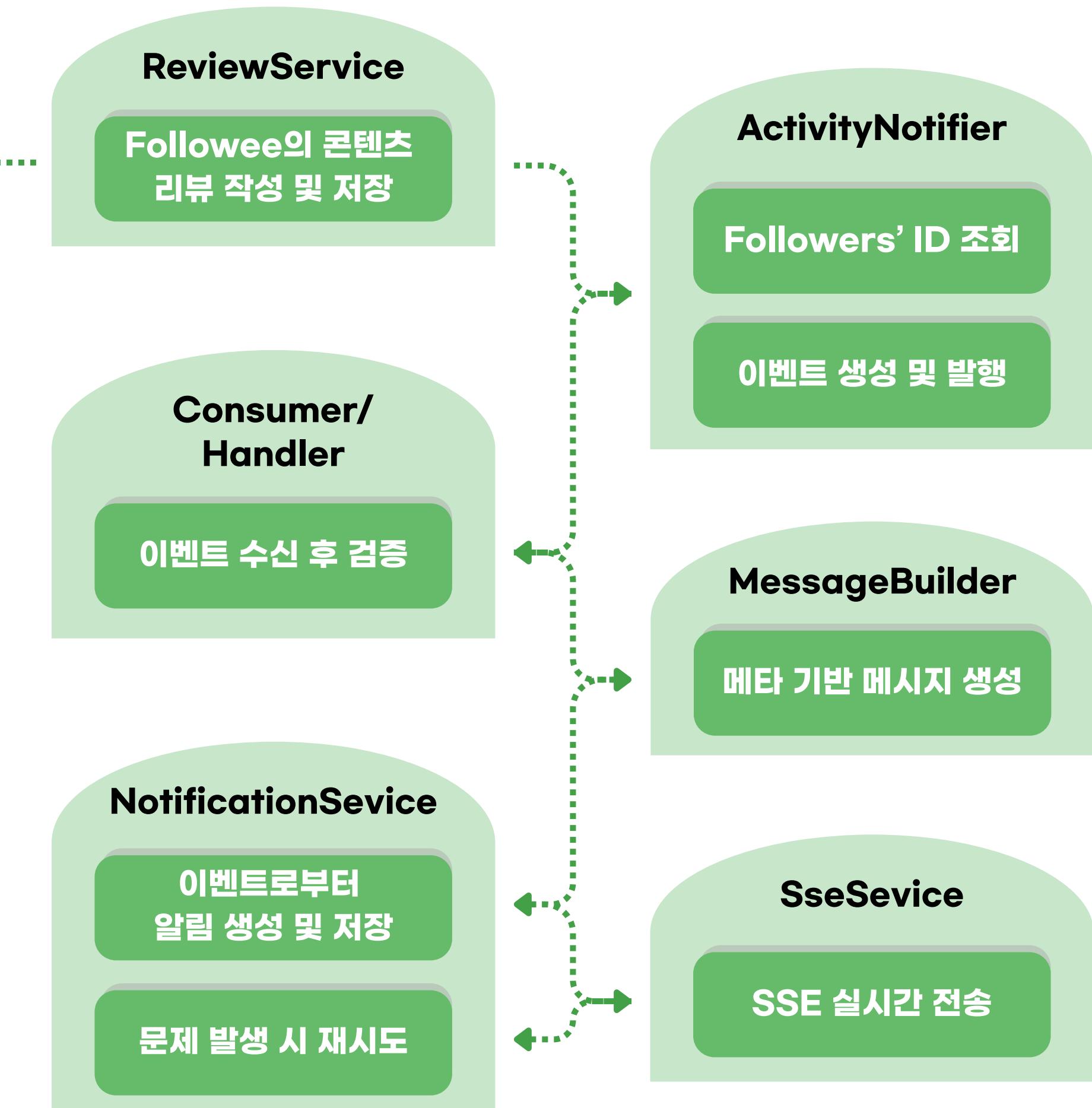
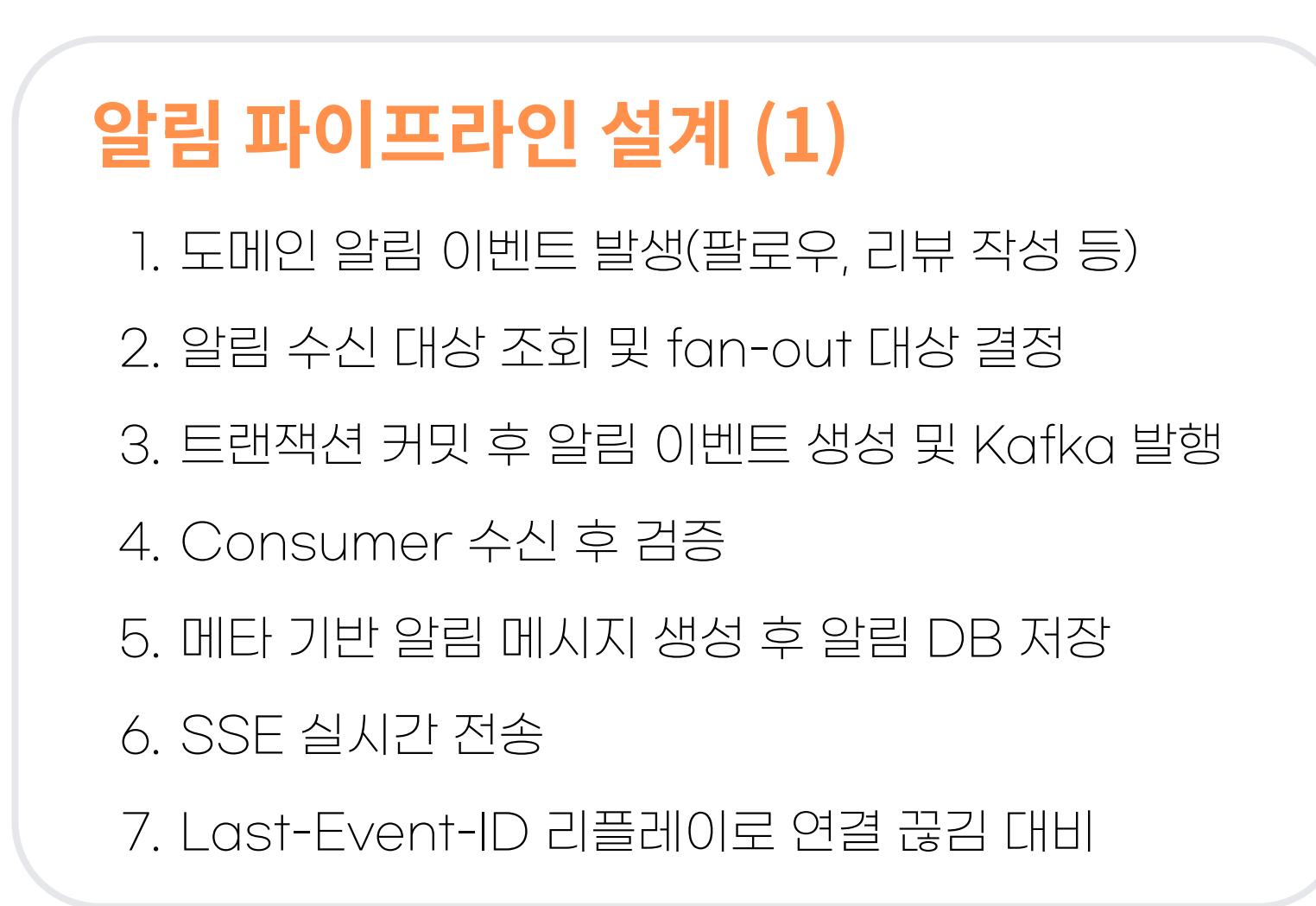


✓
Server Sent Events

kafka ✓

알림 생성 및 관리

알림과 이벤트의 처리 과정

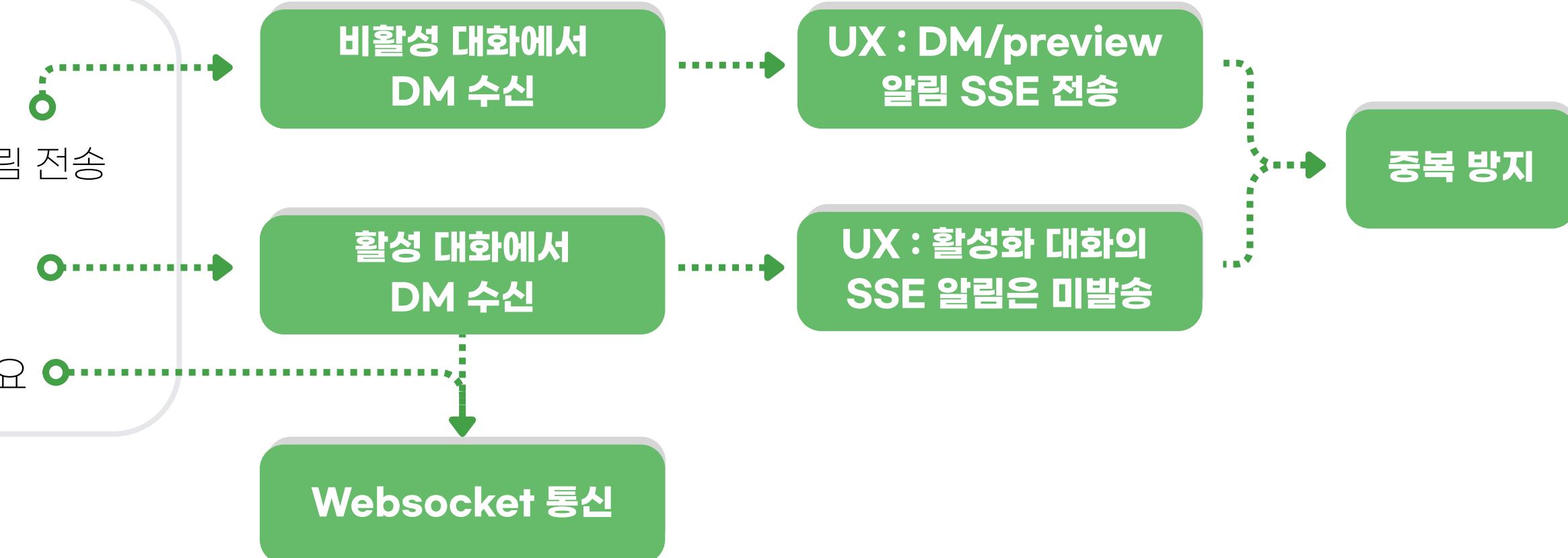


알림 생성 및 관리

DM 전송 및 알림 처리

알림 기획 및 설계 (2)

- DM 수신 시, 수신자에게 발신자와 내용 알림 전송
- 활성화된 대화의 DM 알림 전송은 불필요
- 1대 1대화의 특성 상 실시간 양방향 통신 필요

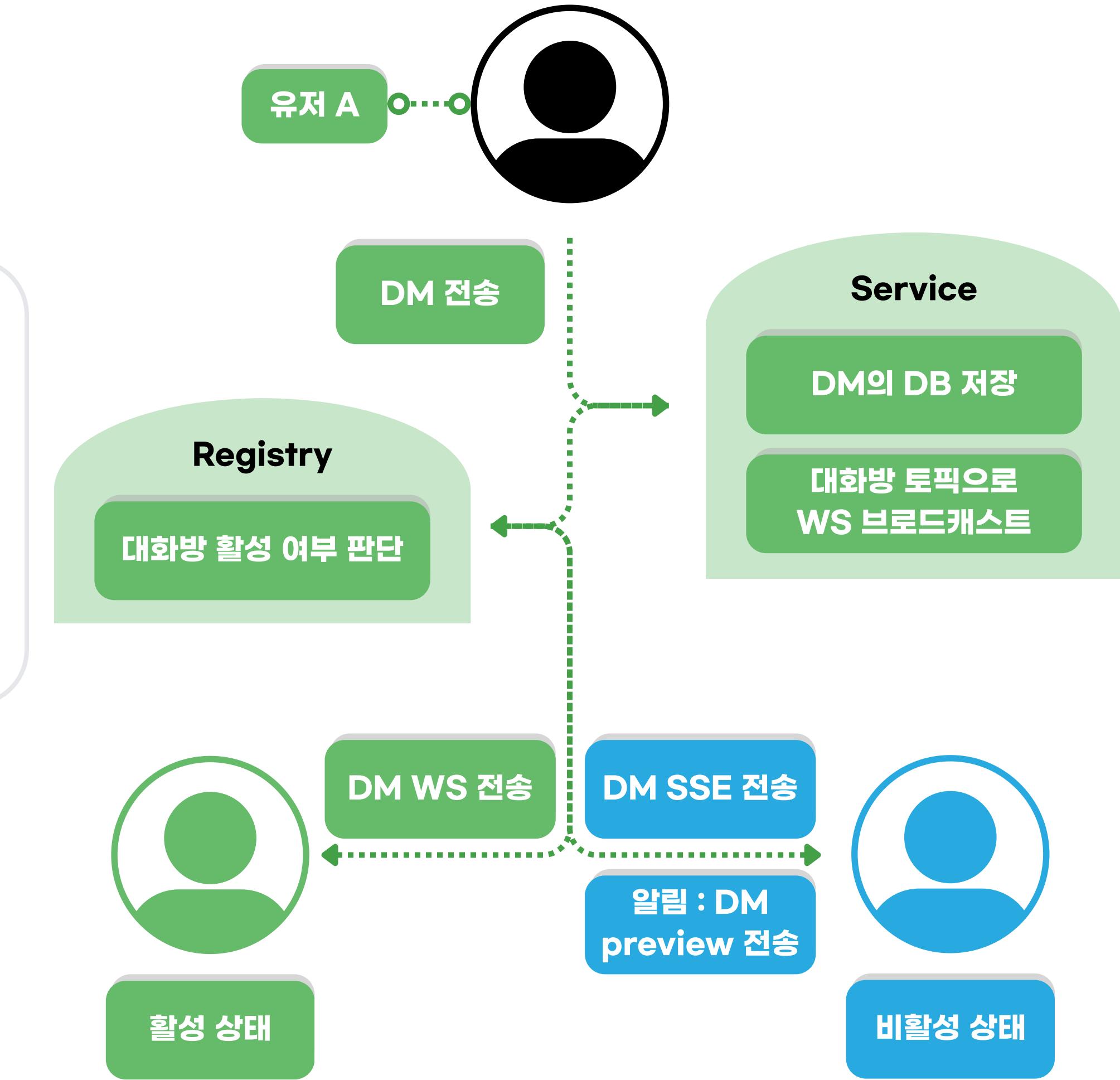
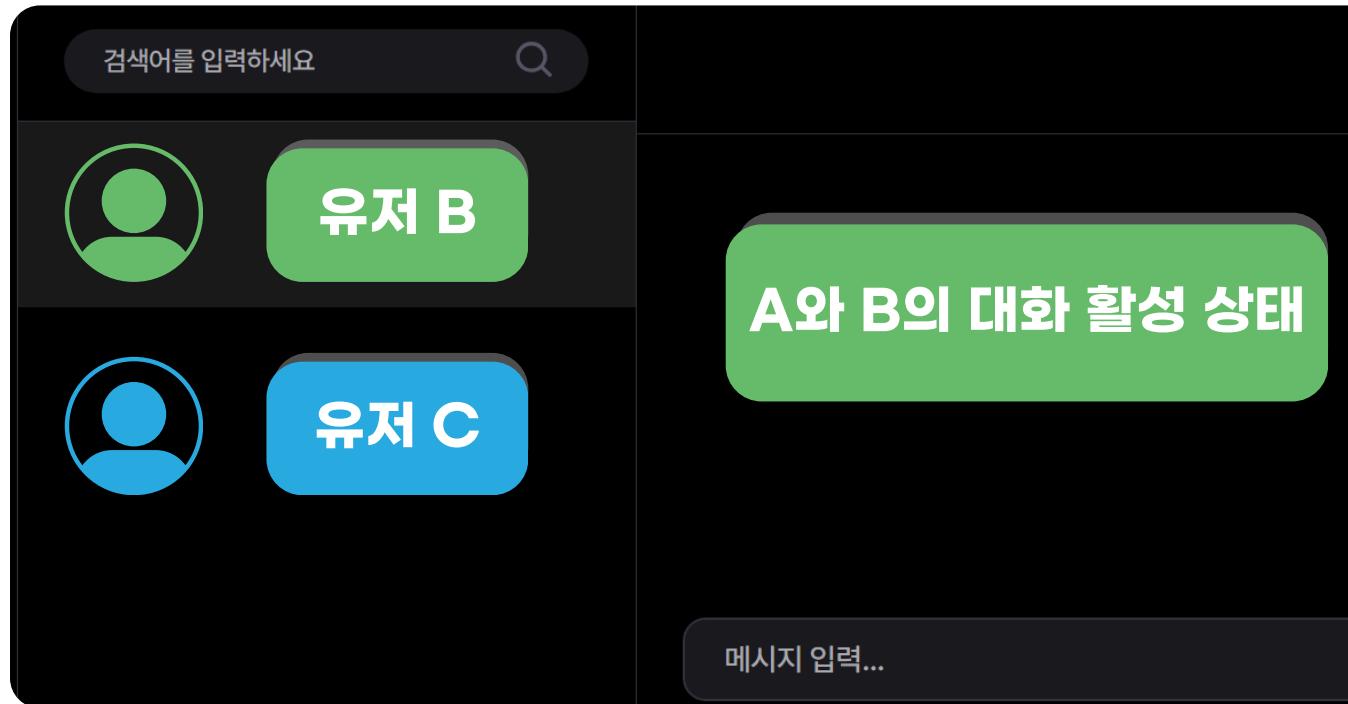


알림 생성 및 관리

DM 전송 및 알림 처리

알림 파이프라인 설계 (2)

1. A가 B, C에게 DM 전송
2. DM을 DB에 저장 후 대화방 토픽 WS 브로드캐스트
3. 수신자(B, C)의 **대화방 활성 여부**를 Registry로 확인
- 4-1. 활성 상태(B)일 시 알림 이벤트 발생 및 SSE 전송 생략
- 4-2. 비활성 상태(C)일 시 알림 이벤트 발생 및 SSE 전송 실행



알림 생성 및 관리

DM 전송 및 알림 처리

기능 필요 사항

- WS 브로드캐스트는 항상 수행
- 대화방의 **활성화 여부 확인** 기능 필요
- 브라우저 환경의 중복 구독으로 인한 문제 방지

기능 구현

- Registry에서 **ConcurrentHashMap**을 사용해 인메모리 방식으로 대화 활성화 여부 판단
- Count를 삽입하여 **구독 수가 1 이상일 때 활성화** 처리하고, 다중 탭이나 세션으로 인한 중복 구독(2 이상)도 동일하게 처리

DirectMessageSubscriptionRegistry.java

```
// 구독 수 카운트 ConcurrentHashMap  
private final Map<UUID, Map<UUID, Integer>> active  
= new ConcurrentHashMap<>();  
  
public void subscribe(UUID conversationId, UUID userId) {  
    active.compute(conversationId, (cid, users) -> {  
        if (users == null)  
            users = new ConcurrentHashMap<>();  
        users.merge(userId, 1, Integer::sum);  
        return users;  
    }...}
```

대화방 구독(활성)

```
public void unsubscribe(UUID conversationId, UUID userId) {  
    // conversationId가 active에 있을 때만 변경  
    active.computeIfPresent(conversationId, (cid, users) -> {  
        Integer count = users.get(userId);  
        if (count == null) return users;  
        // 카운트가 1 이하면 구독이 0이 되므로 userId 제거  
        if (count <= 1) users.remove(userId);  
        // 카운트가 1 초과(2 이상)면 구독 중이므로 count 1 감소  
        else users.put(userId, count - 1);  
        // conversationId에 대해 구독자가 없으면 map을 null로 반환  
        // computeIfPresent가 null일 시 key 제거  
        return users.isEmpty() ? null : users;  
    }...}
```

대화방 구독 해지
(비활성)

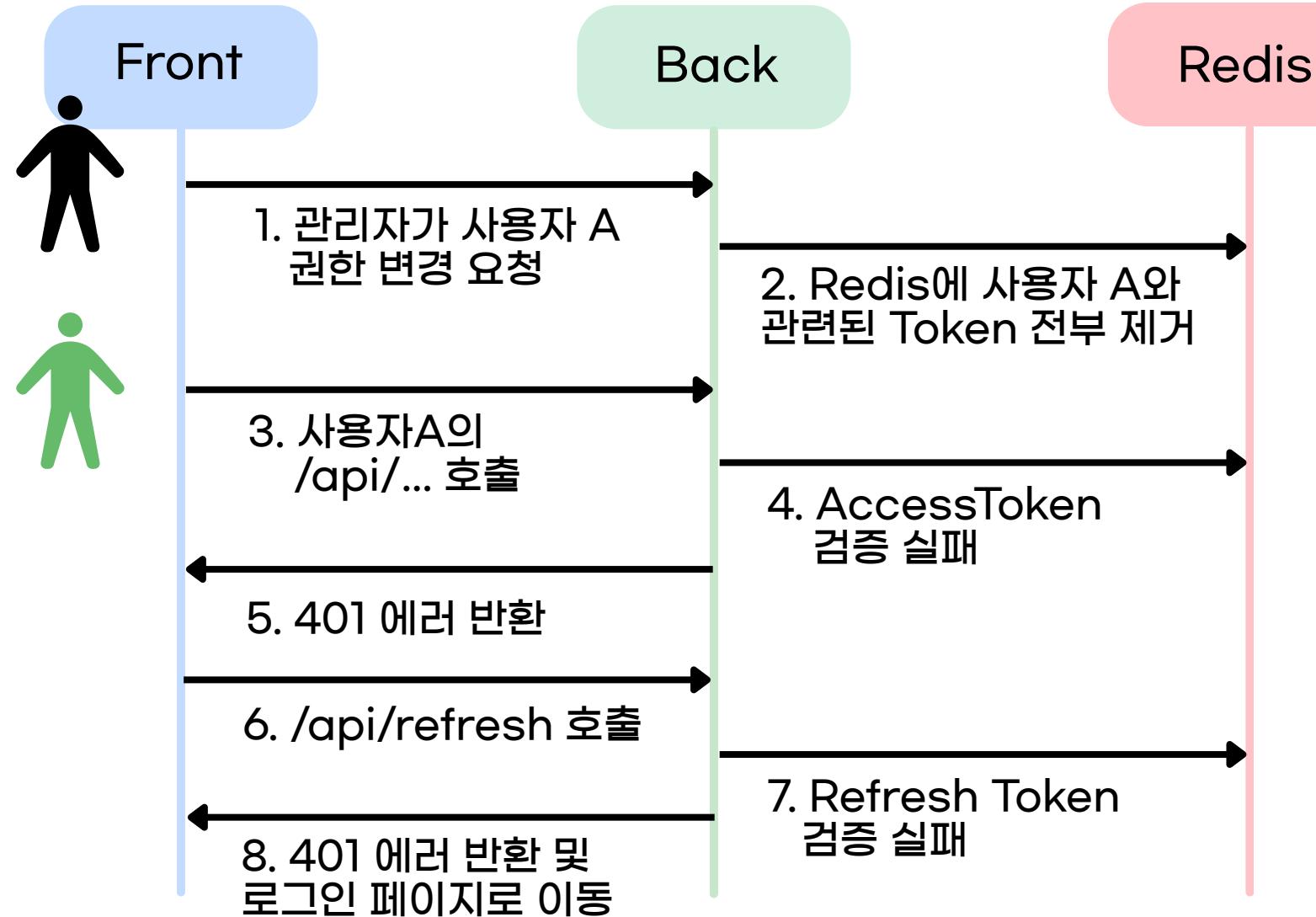
Security 강제 로그아웃 설계

JWT의 Stateless의 한계를 “Redis”로 보완

주어진 요구사항

- ✓ 관리자가 권한 변경 또는 계정 잠금 시, 기존 권한을 막아야함
- ✓ 동시 로그인 제한(중복 로그인 방지)

“만료를 기다리지 않고” 서버가 즉시 세션을 끊을 수 있어야함.



Redis 저장 구조

✓ jwt:access:{token}	->	userId (TTL: 15분)
✓ jwt:refresh:{token}	->	userId (TTL: 24시간)
✓ jwt:user:access:{userId}	->	[access1, access2, ...]
✓ jwt:user:refresh:{userId}	->	[refresh1, refresh2, ...]

조회 성능

- 활성 확인
jwt:access:{token} 을 통한 존재 여부 확인 O(1)
- 필터 위치
JwtAuthenticationFilter
단계에서 차단

무효화 전략

- 일괄 삭제
jwt:user:* 목록 기반으로 전부 삭제
- 정리 범위
Access/Refresh + 목록 키 자체 삭제

“개별 토큰 키” + “유저별 목록”을 같이 둘으로써

- 빠른 검증과 빠른 일괄 삭제를 동시에 만족
- 동시접속의 확장시에도 유리함을 가짐

Batch 아키텍처

Job Layer → Step Layer(Reader → Processor → Writer)

주어진 요구사항

- ✓ TMDB 를 활용해 영화/TV 데이터 수집
- ✓ Sports 경기 데이터 수집

API별 Reader 전략 분리 (Processor/Writer 공통)

Job 구조

Job Layer

fetchTmdbContentsJob

tmdbMovieStep → tmdbTvStep

chunk 1000

fetchSportContentsJob

sportApiStep

chunk 100



Batch 구현-1

API 특성에 따른 Chunk 사이즈 설정

Chunk: 배치에서 Reader->Processor->Writer 흐름을 N개 단위로 묶어 한 번에 처리하는 (커밋하는 작업 단위)

Chunk 사이즈의 영향

- Chunk가 클수록 트랜잭션 횟수 감소 → 전체 처리 속도 향상, 실패 시 더 많은 데이터 재처리
- Chunk가 작을수록 실패 영향 범위 최소화 → 안정성 증가, 실패 시 트랜잭션 오버헤드 증가

🎬 TMDB API → 1,000

- ✓ 많은 사용자들이 찾는 사이트의 API → 상대적으로 안정적
- ✓ 20 페이지 × 최대 500 페이지 → 최대 10,000건/Step
- ✓ Step 구성 : MovieStep → TvStep
- ✓ Commit 횟수 : $20,000 / 1,000 = 20\text{회 Commit}$

⚽ Sport API → 100

- ✓ 무료 API일 경우 30번 요청 시 429 에러 발생 → 1분 대기
- ✓ 소량 수집: 하루 10~30번 경기 × 100일
→ 약 1,000~3,000건
→ $3,000 / 100 = 30\text{회 Commit}$

Batch 구현-2

아이템 1건마다 DB 조회 → Chunk 사이즈만큼의 쿼리 발생

구현 배경

- 외부 API(TDMB, SPORT API)를 호출하여 가져온 데이터의 **중복여부**를 판별해야하는 상황
- 기존의 중복 여부 판별은 아이템 하나당 쿼리를 날려 중복을 판단함.

ContentsProcessor.java

```
@Override  
public ContentFetchDto process(ContentFetchDto item) {  
    boolean isDuplicate =  
        contentRepository  
            .existsBySourceIdAndType(item.getSourceId(),  
                item.getType());  
    ...  
}
```

- 영향
 - ✓ 처리 시간이 데이터 개수에 비례해서 증가(선형)
 - ✓ **DB 부하 증가** → 다른 작업까지 느려질 수 있음
 - ✓ 병목이 명확(중복 체크)해서 최적화 우선순위가 높음

최종 구현

- Batch에서는 Processor는 단건 처리, Writer는 다건 처리를 수행하기에 중복 판별 책임을 옮김
- Processor 대신 Writer에서 chunk 단위로 **IN 쿼리로 필터링**

ContentsWriter.java

```
// Repository 추가  
// Set findSourceIdsByTypeAndSourceIdIn(Type type, Collection  
//     sourceIds);  
// Writer에서 일괄 필터링  
Set existingIds = contentRepository  
    .findSourceIdsByTypeAndSourceIdIn(type, allSourceIds);  
List newItems = items.stream()  
    .filter(item ->  
        !existingIds.contains(item.getSourceId()))  
    .toList();
```

결론

- ✓ 청크가 1000으로 설정된 경우 1000번의 SELECT 쿼리 → **1번**
- ✓ Chunk 단위 “**일괄 조회 + 필터링**”으로 병목 제거

06. 트러블 슈팅



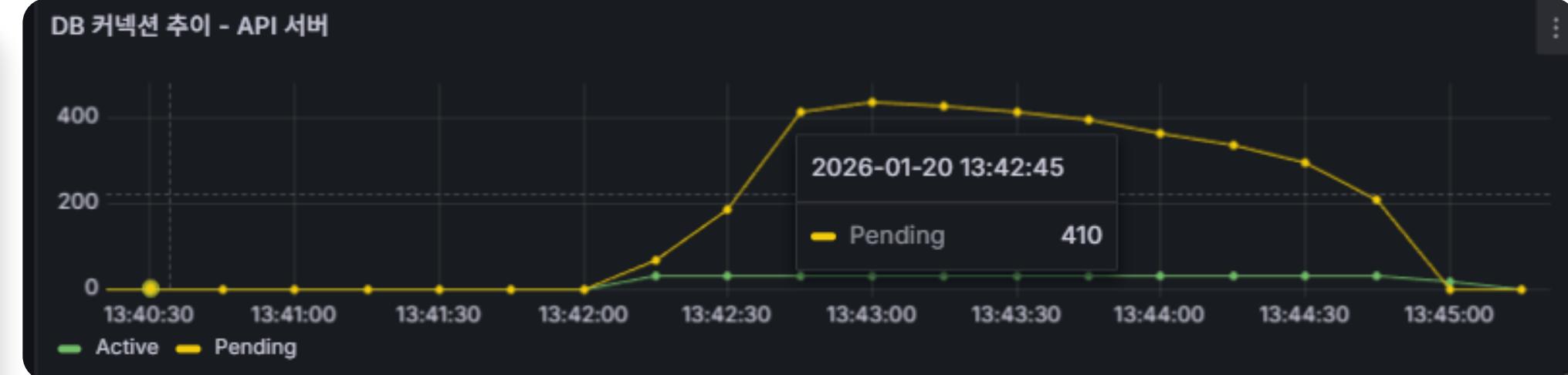
N+1 문제

모니터링을 통한 N+1 문제 발견 및 해결

문제 상황

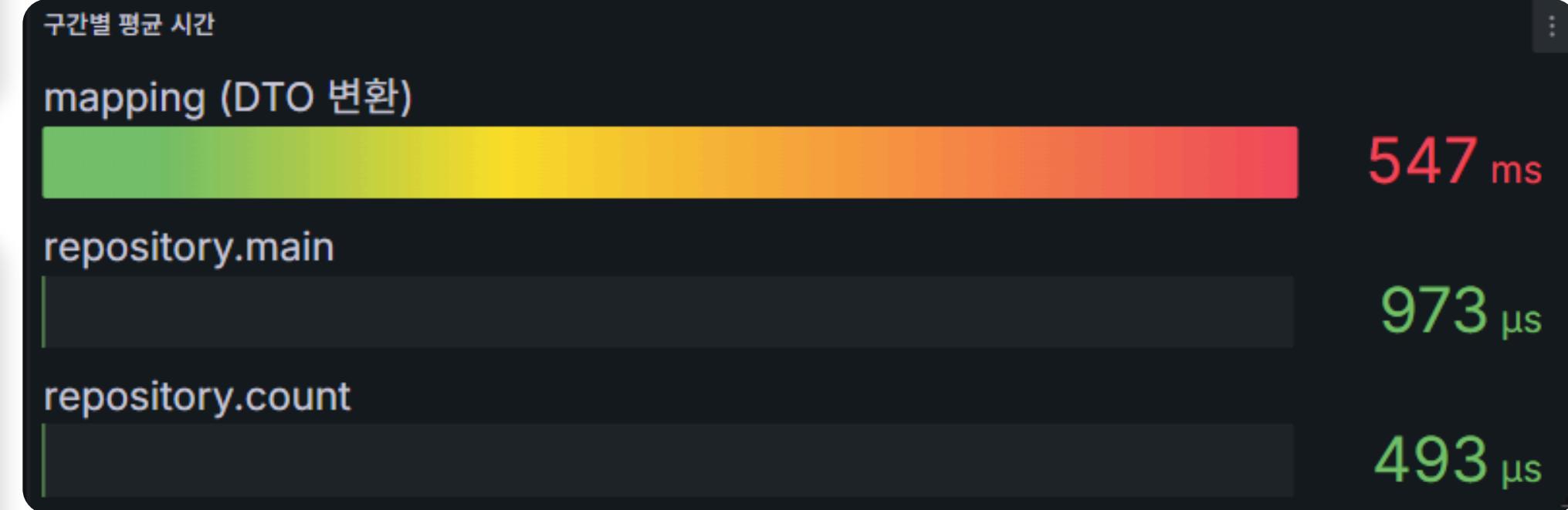
- ✓ 부하 테스트 중 Error율 증가,
- ✓ TPS 하락, 응답시간 하락, DB timeout
- ✓ grafana에서 hikariCP Pending 확인

테스트 환경 : Jmeter, Local docker에서 테스트
Thread 500, Ramp period 60s, loop 10회



원인 분석

- ✓ 특정 API(playlist조회) 성능이 낮아서 원인 분석을 위
해서 매틱스 수집한 결과
- ✓ mapping(관련 컨텐츠 조회 후 dto 변환)에서 느림
- ✓ Hibernate 결과 N+1 문제 발견



```
Caused by: java.sql.SQLTransientConnectionException: HikariPool-1 - Connection is not available, request timed out after 30000ms (total=30, active=30, idle=0, waiting=811)
    at com.zaxxer.hikari.pool.HikariPool.createTimeoutException(HikariPool.java:714)
    at com.zaxxer.hikari.pool.HikariPool.getConnection(HikariPool.java:184)
    at com.zaxxer.hikari.pool.HikariPool.getConnection(HikariPool.java:142)
    at com.zaxxer.hikari.HikariDataSource.getConnection(HikariDataSource.java:127)
```

N+1 문제

N+1 문제 해결 이후 재측정

솔루션

- ✓ Playlist 내 콘텐츠 정보를 Stream으로 개별 조회하는 것을 확인
- ✓ 배치 조회를 사용해서 연관 데이터들을 한번에 조회 한 뒤 dto로 변환
- ✓ 대시보드는 각각 데이터 조회에 대한 시간 측정

결과

- ✓ 547ms → 7.2ms로 개선
- ✓ 속도 테스트로는 **98.7** 퍼센트 개선
- ✓ hikariCP 병목 개선



쿼리 분리에 따른 기존 Mapping과 비교를 위한 시간은
약 7.2ms

DB 인덱스

playlist 목록 조회에서 content_tag 인덱스 누락

문제 상황

- ✓ 테스트 환경
Thread 2500/ ramp 60s/ loop 10회 테스트 중
- ✓ playlist 그라파나 모니터링에서 tags만 유독 더 느린 것을 확인



원인 분석

- ✓ 실행 계획을 분석한 결과, 해당 쿼리는 Sequential Scan 방식으로 전체 테이블을 스캔
- ✓ 인덱스 구성을 점검한 결과, content_tags 테이블의 content_id 컬럼에 인덱스가 존재하지 않음을 확인

Hash Join (cost=13.15..1653.47 rows=16 width=596) (actual time=0.209..2.674 rows=12 loops=1)
 Hash Cond: (ct1_0.tag_id = t1_0.id)
 -> Seq Scan on content_tags ct1_0 (cost=0.00..1640.28 rows=16 width=56) (actual time=0.189..2.648 rows=12 loops=1)
 Filter: (content_id = ANY ('{019be468-afc0-7e2f-9d33-90cca2ad7088, 019be469-546c-7829-b04b-09dad493fa4, 019be46c-b0c6-7887-848e-ea98541d2698, 019be468-fde8-7f63-8031-199996caf5ae, 019be469-546c-7e90-b0de-b2b57c0c7e05}':uuid[]))"
 Rows Removed by Filter: 61728
 -> Hash (cost=11.40..11.40 rows=140 width=540) (actual time=0.013..0.014 rows=29 loops=1)
 Buckets: 1024 Batches: 1 Memory Usage: 10kB
 -> Seq Scan on tags t1_0 (cost=0.00..11.40 rows=140 width=540) (actual time=0.005..0.006 rows=29 loops=1)
Planning Time: 0.412 ms
Execution Time: 2.711 ms

DB 인덱스

인덱스 추가

솔루션

- ✓ content_tags 테이블에 content_id 인덱스 추가

```
Hash Join (cost=13.44..38.91 rows=16 width=596) (actual time=0.040..0.068 rows=12 loops=1)
  Hash Cond: (ct1_0.tag_id = t1_0.id)
    -> Index Scan using idx_content_tags_content_id on content_tags ct1_0 (cost=0.29..25.72 rows=16
width=56) (actual time=0.021..0.047 rows=12 loops=1)
        Index Cond: (content_id = ANY ('{019be468-afc0-7e2f-9d33-90cca2ad7088,019be469-546c-7829-b04b-
09dadcd493fa4,019be46c-b0c6-7887-848e-ea98541d2698,019be468-fde8-7f63-8031-199996caf5ae,019be469-546c-
7e90-b0de-b2b57c0c7e05}':uuid[]))"
    -> Hash (cost=11.40..11.40 rows=140 width=540) (actual time=0.013..0.013 rows=29 loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 10kB
        -> Seq Scan on tags t1_0 (cost=0.00..11.40 rows=140 width=540) (actual time=0.005..0.006
rows=29 loops=1)
Planning Time: 0.386 ms
Execution Time: 0.092 ms
```

결과

- ✓ 실행 계획 Index Scan
- ✓ 단일 쿼리 실행 시간 2.711ms → 0.092ms
- ✓ tags 조회속도 개선

배치 쿼리 구간별 평균 시간



LIKE절 검색

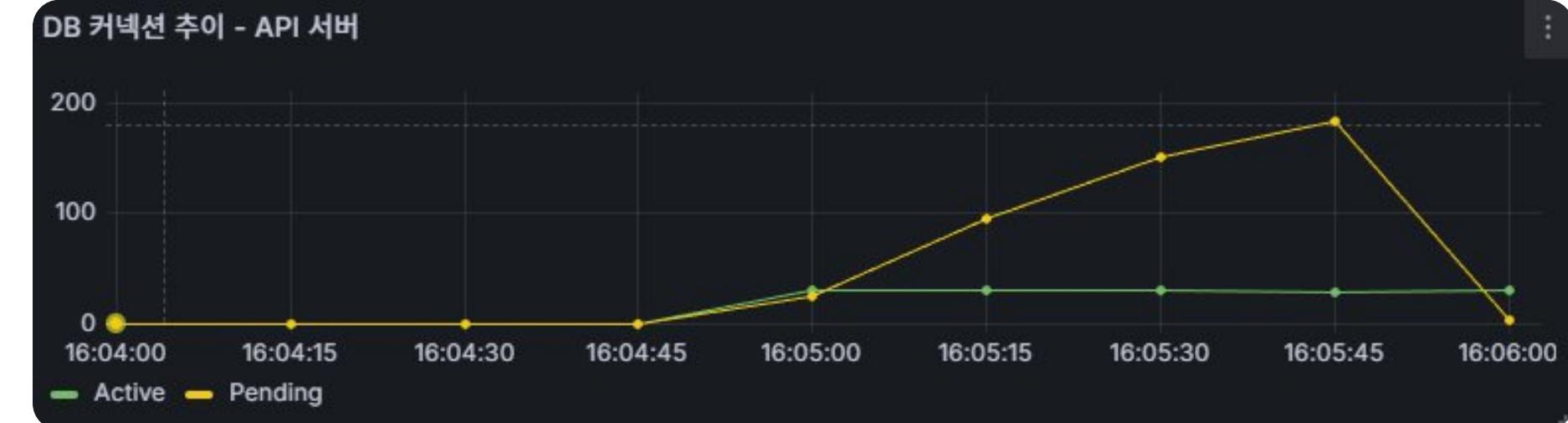
RDB의 LIKE 성능 문제

문제 상황

- ✓ 테스트환경
Thread 1000, ramp 60s, loop 10 테스트 중
- ✓ LIKE절을 이용한 검색이 db 커넥션 팬딩과 응답 속도가 느린 것을 확인

원인 분석

- ✓ RDB에서 LIKE절을 사용하면 Full-scan을 하기 때문에 성능 저하
- ✓ 데이터 많은 경우 검색용으로 RDB를 사용하면 갈 수록 성능저하 우려



RDB 구간별 평균 시간



LIKE절 검색

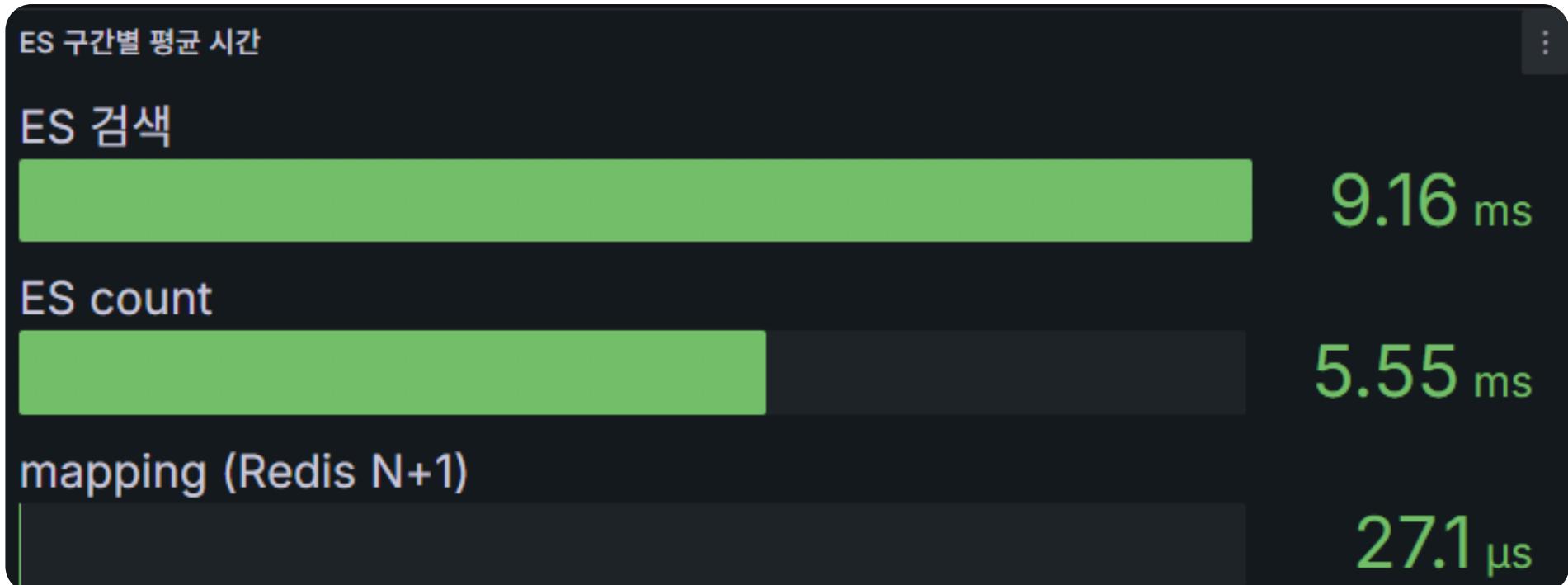
Elasticsearch 도입

솔루션

- ✓ RDB가 아닌 검색 특화 엔진을 가진 Elasticsearch를 사용

결과

- ✓ 기존 약 460 ms → 1.78 ms로 엄청난 성능 향상
- ✓ 이후 5000 쓰레드, ramp 60s, loop 20 회
재측정 시 오른쪽 결과와 같이 빠른 속도로 조회
- ✓ 트래픽 내성 대폭 증가를 확인할 수 있었습니다.
1000쓰레드 → 5000 쓰레드이상 부하 감당 가능



중복 생성 방지

대화방 생성 중복 가능성

문제 상황

- ✓ 두 명의 사용자가 각각 서로에게 DM을 보내기 위해서 대화방 생성을 하는 상황이 오면 대화방의 중복 생성이 되지 않을까라는 문제제기

원인 분석

- ✓ ERD에서 대화 테이블에서 참여자 정보를 직접 가지고 있지 않아서 DB 유니크 제약 형태로 처리할 수 없었어서 동일한 타이밍에 생성을 시도할 시 중복 발생 가능한 걸 테스트 코드 작성하여 확인

대화 참여자								conversation_participants
Key	아이디	id	Domain	UUID	NOT NULL	Default value	Comment	
	생성 시각	created_at	Domain	TIMESTAMP	NOT NULL	Default value	대화 참여시각으로 활용 가능	
Key	참여자 아이디	user_id	Domain	UUID	NOT NULL	Default value	UNIQUE(user_id, conversation_id)	
Key	대화 아이디	conversation_id	Domain	UUID	NOT NULL	Default value	Comment	
	마지막 읽은 시각	last_read_at	Domain	TIMESTAMP	NULL	Default value	유저별 적용	

대화								conversations
Key	아이디	id	Domain	UUID	NOT NULL	Default value	Comment	
	생성 시각	created_at	Domain	TIMESTAMP	NOT NULL	Default value	Comment	
	마지막 메시지 아이디	last_message_id	Domain	UUID	NULL	Default value	Comment	

[대화방 중복 생성 테스트]

전체 시도 횟수: 10

예상 결과 : 성공 1, 실패 9

생성 성공: 10

락으로 인한 실패: 0

중복 생성 방지

레디스 분산락 적용

솔루션

- ✓ DB 테이블 변경없이 해결하기 위해서
- ✓ 레디스에 분산락을 적용하도록 했습니다.
- ✓ key는 userId 정렬, value는 본인 식별 랜덤 uuid

✓ 1 test passed 1 test total, 635 ms

[대화방 중복 생성 테스트]

전체 시도 횟수: 10

예상 결과 : 성공 1, 실패 9

생성 성공: 1

락으로 인한 실패: 9

결과

- ✓ 전체 쓰레드 10개 중 1개만 성공하고
- ✓ 나머지 9개는 락으로 인해서 실패하여
- ✓ 중복 생성이 방지되는 모습

네트워크 I/O 병목

Batch에서 외부 URL 다운로드 + S3 업로드

문제 배경

- 썸네일 처리는 외부 URL 이미지 다운로드 + S3 업로드로 구성된 네트워크 I/O 작업
- CPU 사용은 적고, 응답 대기 시간이 대부분이라 순차 처리 시 대기 누적이 발생
- 결과적으로 배치 전체 성능이 크게 저하

문제 정의

- 기존 방식(순차)

100개 ≈ 130초

가령, 아이템마다 다운로드(800ms) + 업로드(500ms)
≈ 1,300ms가 누적

- 병목 성격

외부 URL 다운로드

S3 업로드

대기 시간 누적

CPU 사용 낮음

- ✓ 순차 처리 시 네트워크 대기가 아이템 수만큼 누적
- ✓ 배치 성능 저하 원인이 명확

목표: 썸네일 처리 병목을 줄이고, Writer 단계에서 처리 시간을 단축

병목을 푸는 3가지 방향

해결 방법 후보

1 Multi-threaded Step

Chunk 병렬 실행

- 기존: **Single-threaded Step**
→ 청크를 순차 처리
- 개선: **TaskExecutor**로 여러 청크를 동시에 실행
- Reader/Processor/Writer가 여러 스레드에서 동시에 호출

전제: Thread-Safe 보장 필요
→ 공유 상태가 있으면 중복/누락 위험

```
.taskExecutor(stepTaskExecutor())  
.throttleLimit(THREAD_COUNT)
```

2 Writer 내부 I/O 병렬화

썸네일 업로드 병렬

- 병목 지점이 **썸네일 네트워크 I/O**로 명확
- 비동기 업로드(스레드 풀)로 **대기 시간 분산**
- 업로드 결과(S3 URL) 수집 후 DB 저장

CompletableFuture

FixedThreadPool(10)

join() 결과 수집

```
CompletableFuture.supplyAsync(..)  
.map(CompletableFuture::join)
```

3 Kafka 도입(비동기 분리)

Producer/Topic/Consumer

- Fetcher(Producer)와 Writer(Consumer)를 **완전히 분리**
- Kafka에 메시지 보관 → Consumer 재시작해도 **이어서 처리**
- 기존의 이벤트 처리를 위해 사용되던 Kafka를 활용 가능

디커플링

속도 차이 흡수

스케일 아웃

장애 복구(디스크 보관)

```
@KafkaListener("contents.fetch")  
void consume(Payload p,  
Acknowledgment ack){ /* S3+DB */  
    ack.acknowledge(); }
```

최종 선택

실패 영향 범위 최소화 + 남은 시간, 경험 부족

Writer 내부 병렬화 선택

도입

- 썸네일 처리는 네트워크 I/O(다운로드 + S3 업로드)
- 스레드 풀: 과도한 동시 요청은 부담, 너무적으면 병렬 효과 감소
→ `FixedThreadPool(10)`

```
try {  
    // 외부 URL 다운로드 + S3 업로드 (I/O)  
    return s3Url;  
} catch (Exception e) {  
    return null; // 실패 격리  
}
```

Kafka 도입(비동기 분리)

미도입

도입 대비 학습/운영 비용과 구조 변경 폭이 크다.

- Batch 담당 인원의 경험부족
- 이번 범위는 “배치 내부 최적화”로 충분히 효과를 얻을 수 있음

Multi-threaded Step

미도입

- Thread-safe 보장 전제가 커서, 중복/누락 리스크가 있다.
- Reader/Processor/Writer가 여러 스레드에서 동시에 호출
- Reader가 공유 상태를 가지면 정합성 이슈 가능
- 이번 문제는 ‘Writer’에서 발생하는 문제
→ 기존의 Reader, Processor 코드를 전부 수정해야하는 상황

```
/**  
 * SportApiReader.java 일부로  
 * Reader 내부 상태(현재 날짜/인덱스/버퍼 데이터)를 필드로 유지함  
 * → Multi-threaded Step에서 여러 스레드가 동시에 접근하면  
 * 데이터 중복/누락 위험  
 */  
  
private LocalDate startDate;  
private int currentDate;  
private int currentIndex;  
private List<ContentFetchDto> currentData;
```

최종 결과

100개 아이템:

실제 구현

- 병목 구간인 썸네일 Download → S3 Upload(I/O) 만 Writer에서 병렬 처리

ContentsWriter.java

```
private static final int THREAD_POOL_SIZE = 10;

@Transactional
public void write(Chunk<? extends ContentFetchDto> chunk) {
    ...
    List<CompletableFuture<String>> futures =
        uniqueItems.stream()
            .map(item -> CompletableFuture.supplyAsync(() ->
                uploadThumbnailToS3(item), imageExecutor))
            .toList();

    List<String> s3Urls = futures.stream()
        .map(CompletableFuture::join)
        .toList();
    ...
}
```

Adapt

개선 효과

Before(순차)

~62.9m

9980개 아이템

10 threads

After(병렬)

~4.19m

I/O 대기 분산으로 누적 지연 감소

```
=====
성능 비교 결과
=====
데이터 개수: 9980
스레드 풀 크기: 10
-----
직렬 처리 시간: 3779927ms
병렬 처리 시간: 251470ms
-----
성능 향상: 93.3% 빨라짐
속도 배율: 15.03x 빠름
=====
```

네트워크 I/O의 병렬 처리를 통한 성능이 **93.3%** 향상

배포 이후 OAuth 장애

배포 환경에서의 Kakao/Google OAuth 트러블슈팅

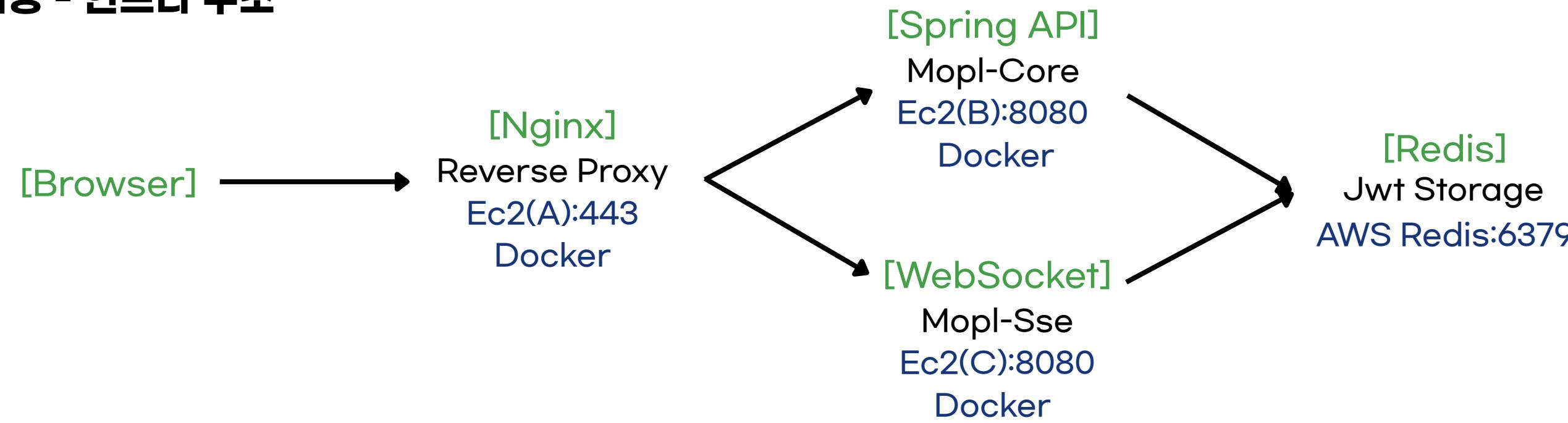
문제

Expected: <https://mopl.cloud/callback>
Actual: <http://mopl.cloud:8080/callback>

로컬 이후 배포환경에서의 장애

- ✓ 로컬: Nginx, Spring, Redis...를 컨테이너로 올린 후 테스트
- ✓ 로컬: http ssl 적용 X

문제 배경 - 인프라 구조



[Nginx]

- ✓ static 파일 캐싱
- ✓ L7 로드 밸런싱

[mopl-core]

- ✓ 메인 API 서버
- ✓ OAuth 인증 처리
- ✓ JWT 토큰 발급 및 검증

[mopl-websocket-sse]

- ✓ 실시간 알림
- ✓ SSE 스트리밍
- ✓ JWT 검증만 수행

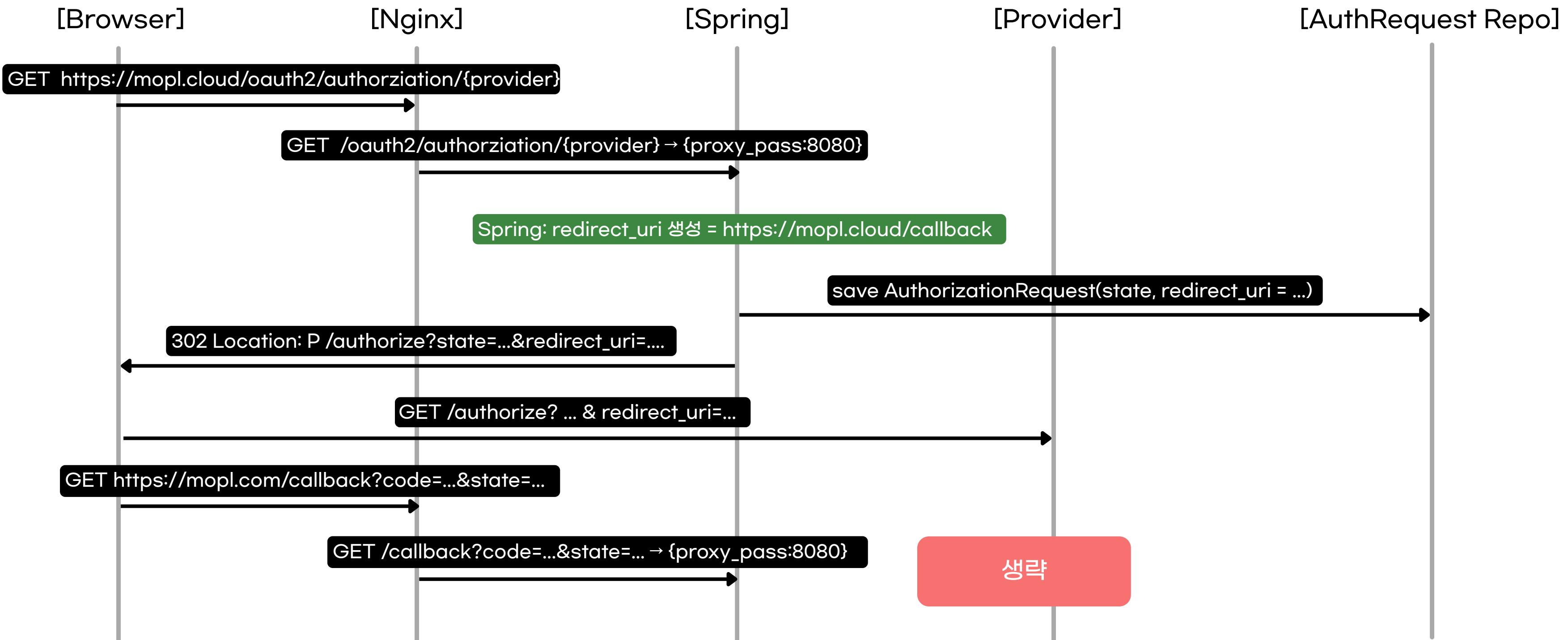
[Redis]

- ✓ JWT 토큰 저장
- ✓ 세션 관리
- ✓ 토큰 로테이션

OAuth 의 흐름

배포 환경에서의 Kakao/Google OAuth 트러블슈팅

원인 분석 - OAuth 절차



기존의 설정에서의 원인

STATELESS 세션 환경에서의 Kakao/Google OAuth 트러블슈팅

근본적인 원인

Expected: https://mopl.cloud/callback Actual:**http://mopl.cloud:8080/callback**

- Spring에서 redirect_uri를 잘못 생성했음을 확인할 수 있다.
- application.yml에서의 redirect-uri

```
registration:  
  google:  
    redirect-uri: "{baseUrl}/callback"
```

- Spring Security가 {baseUrl}을 **현재 요청 기반**으로 자동 생성하는 방식
 - ✓ 내부적으로 Spring Boot는 자신이 **http://localhost:8080**에서 동작한다고 인식
 - ✓ 외부 요청은 Nginx(HTTPS/443)를 거치지만, Nginx에서 헤더를 설정하지 못하면 **http://localhost:8080**으로 인식

```
// Spring이 계산하는 baseUrl  
String scheme = request.getScheme(); // "http" (X-Forwarded-Proto 무시)  
String host = request.getServerName(); // "mopl.cloud" (Host 헤더 읽음)  
int port = request.getServerPort(); // 8080 (X-Forwarded-Port 무시)  
  
String baseUrl = scheme + "://" + host + ":" + port;
```

- ✓ 결과: **{baseUrl}** = **http://mopl.cloud:8080** 으로 계산됨 → Nginx가 기본적으로 Host는 전달

OAuth 장애 해결

배포 환경에서의 Kakao/Google OAuth 트러블슈팅

해결 방안 - 1: Nginx 와 .yml 설정

미도입

- Nginx에서 헤더 추가, Spring API가 헤더를 읽어야 함
 - ✓ X-Forwarded-Proto : https
 - ✓ X-Forwarded-For: 123.45.67.89
 - ✓ X-Forwarded-Host: mopl.cloud

```
# nginx.conf.deploy
location /oauth2 {
    proxy_pass http://spring-api;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

- Spring 쪽에서도 설정이 필요함
 - ✓ Nginx가 헤더를 보내도 기본적으로 Spring은 기본적으로 안 읽음
→ 기본값: **forward-headers-strategy: NONE**
 - ✓ **forward-headers-strategy: native**
→ Forwarded와 관련된 헤더를 **tomcat**이 처리
 - ✓ 설정 전: `request.getSchema() = "http"`
 - ✓ 설정 후: `request.getSchema() = "https"`

```
# application-prod.yml
server:
  forward-headers-strategy: native
  tomcat:
    remoteip:
      protocol-header: X-Forwarded-Proto # 443
      remote-ip-header: X-Forwarded-For
      internal-proxies: "./*"
```

OAuth 장애 해결

배포 환경에서의 Kakao/Google OAuth 트러블슈팅

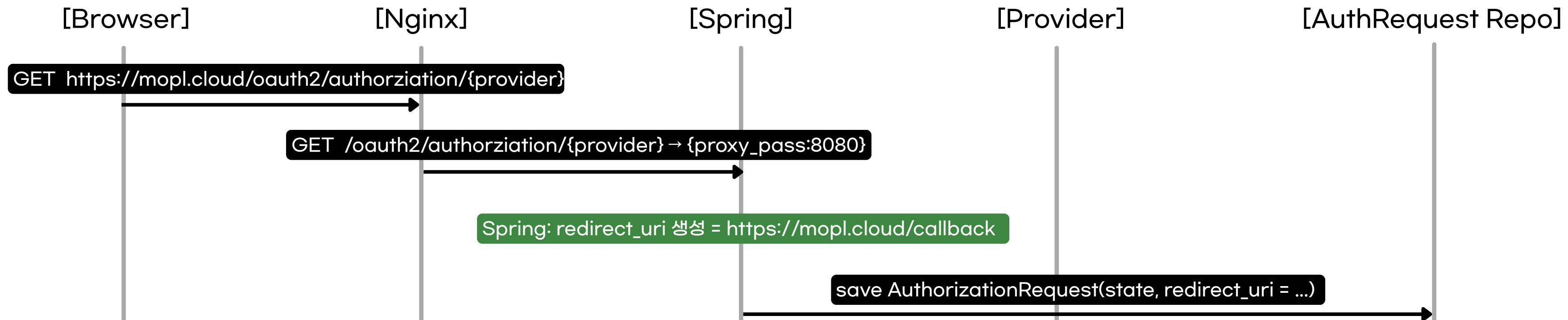
해결 방안 - 2 : 하드코딩

도입

- 기본적으로 redirect-uri를 하드 코딩하면 된다. 해결 방안 1의 경우 복잡한 설정으로 예상치 못한 문제가 발생할 수 있다.

```
# application-prod.yml
google:
  redirect-uri: {baseUrl}/login/oauth2/code/google
kakao:
  redirect-uri: {baseUrl}/login/oauth2/code/kakao
```

```
# application-prod.yml
google:
  redirect-uri: https://mopl.cloud/login/oauth2/code/google
kakao:
  redirect-uri: https://mopl.cloud/login/oauth2/code/kakao
```



시연

