# dailyfeed backend

## 주요 백엔드 기능 및 설계 관련 설명

2025/10/24 (정순구)

# 핵심 포인트

**설계시 핵심아이디어**

+ 카프카 통신 시 에러 처리 방식

+ 논리적 도메인 구조의 서비스를 스케일링 용도(저장 vs 통계/조회)별 분리

  :: 논리적인 구조로만 분해할 경우 예상치 못한 트래픽 급증시 특정 서비스 전체가 스케일 아웃 되는 현상 발생

  :: Read / Write 용도의 레플리케이션 그룹을 분리해서 스케일링되도록 구성 (e.g. Read 부하가 심할때는 timeline 그룹의 스케일 아웃, Write 부하가 심할때는 content 그룹의 스케일 아웃)

  :: 저장소까지 분리한 완전한 MSA 는 아니지만, 부하(로드)의 성격별 레플리케이션 그룹을 지정

  :: 7인8각 게임과 같은 모놀리딕 구조는 배제 (내가 디지면 너도 디지는거야. 잘해. vs 너는 이거해, 나는 이거할께)

+ 통신레벨과 애플리케이션레벨 결합도 분리 (L4 레벨과 L7 레벨의 분리)

  :: 애플리케이션에서는 통신레벨을 알 필요가 없고, 통신레벨에서는 전달받은 값만 받아서 통신이라는 역할만을 수행

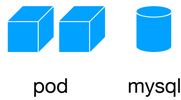
  :: 통신레벨에서는 통신만 수행하고, 익셉션은 애플리케이션 계층으로 throw

+ no common → 주요 서브모듈로 모듈화

# Overview

**dailyfeed-member-svc**

+ 회원가입,로그인,로그아웃




min: 1
max: 2

**dailyfeed-content-svc : 글 생성/수정/삭제 전용 서비스**

+ 글 작성/수정/삭제

+ 댓글 작성/수정/삭제

+ 댓글의 답글 작성/수정/삭제


min: 1
max: 3

**dailyfeed-image-svc : 이미지 조회/업로드/삭제 서비스**

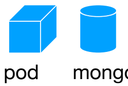
+ 프로필/썸네일 이미지 업로드/수정/삭제/조회

+ timeline 조회, 추천회원 프로필 목록 조회, 나의 프로필에서 이미지 조회


min: 3
max: 5

**dailyfeed-activity-svc**

+ 활동기록 저장/조회/수정




min: 1
max: 2

season2

- season2 폐지 예정

**dailyfeed-timeline-svc : 사용자 맞춤 조회/통계 쿼리 전용 서비스**

+ 팔로우 중인 멤버들의 최근 작성글(follow feed)

+ 지금 가장 인기있는 글 (most popular)

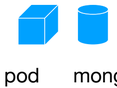
+ 댓글 많은 글 (most commented)

+ 나의 작성글들 (my feed)


min: 1
max: 7

**dailyfeed-search-svc : 본문 검색 서비스**

+ Full Text Search 를 위한 별도의 서비스




min: 1
max: 3

# 참고사항

## dailyfeed-activity-svc

# dailyfeed-activity-svc

**dailyfeed-activity-svc 관련**

+ dailyfeed-activity-svc 는 kafka 사용시 이런 구조로 사용할 것이라는 예시를 남기기 위해 작성한 예제입니다.

+ 사용자가 서비스에 인입해서 어떤 활동을 했는지를 기록하는 서비스인데, 현존 플랫폼 들 중 '알림 배너' 라는 기능에 이웃/친구 관계의 멤버가 어떤 활동(좋아요/좋아요취소/글작성/글수정)을 했는지를 보여주는 기능이 있는데, 이 기능에 대해 어떤 식으로 기록을 할지에 대한 예제 프로젝트입니다.

+ 만약 블로그 서비스가 있다고 해보겠습니다. 블로그 서비스의 운영 년수가 오래되어가고, 중요한 사업아이템이 되었을 때 블로그 서비스에서 글을 작성할 때 사용자의 활동 기록까지 수정하기에는 활동 기록 시에 필요한 부수적인 작업으로 블랙리스트 체크, 욕설 체크, 음란물 필터링, 광고 추천시스템 업데이트 등 복합적인 작업의 요구가 발생할 수 있는데 이 경우 글 쓰기/수정/삭제 기능에서 이 모든 기능을 수행하기 어려워집니다.

+ 이번 프로젝트에서는 이런 경우에 대해 활동기록, 블랙리스트 체크, 광고추천 시스템 업데이트 등의 작업을 dailyfeed-activity-svc 로 이관한 케이스를 가정합니다.

+ 그리고 사용자의 글 삽입/수정/삭제에 대해서는 POST_CREATE, POST_UPDATE, POST_DELETE 이벤트를 발생시켜 Kafka Topic 으로 해당 이벤트를 발행하고, 이 것을 구독하는 dailyfeed-activity-svc 가 관련된 처리를 하도록 합니다. 즉, MSA 간 통신을 Kafka 를 통해 수행하는 과정을 예제의 전제조건으로 가정했습니다.

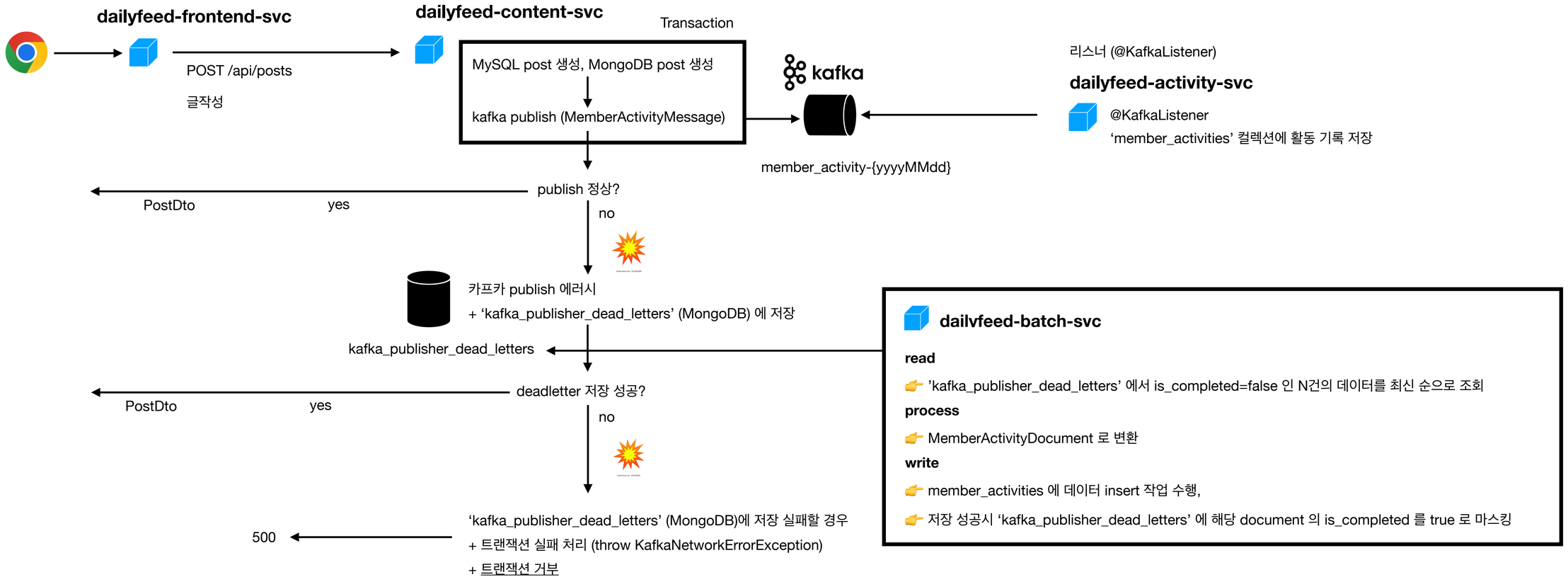
+ 카프카 운영시 카프카 증설 및 업그레이드 작업으로 인해 카프카 업그레이드 이슈 등이 있을 때는 세컨더리 저장소를 통해 Fail Over를 하는 것도 중요한 요소 중 하나입니다.

+ dailyfeed-content-svc 내에서 쓰기 작업 수행 도중 dailyfeed-activity-svc 가 장애가 나서 저장을 못하는 케이스 역시 가정합니다. dailyfeed-activity-svc 가 장애가 발생해도 dailyfeed-content-svc 의 글 쓰기/수정/삭제 작업에는 장애가 나지 않도록 하는 상황을 가정했습니다.

## Case (1) Kafka

## : publisher, listener 데이터 처리 + 통신 에러 처리

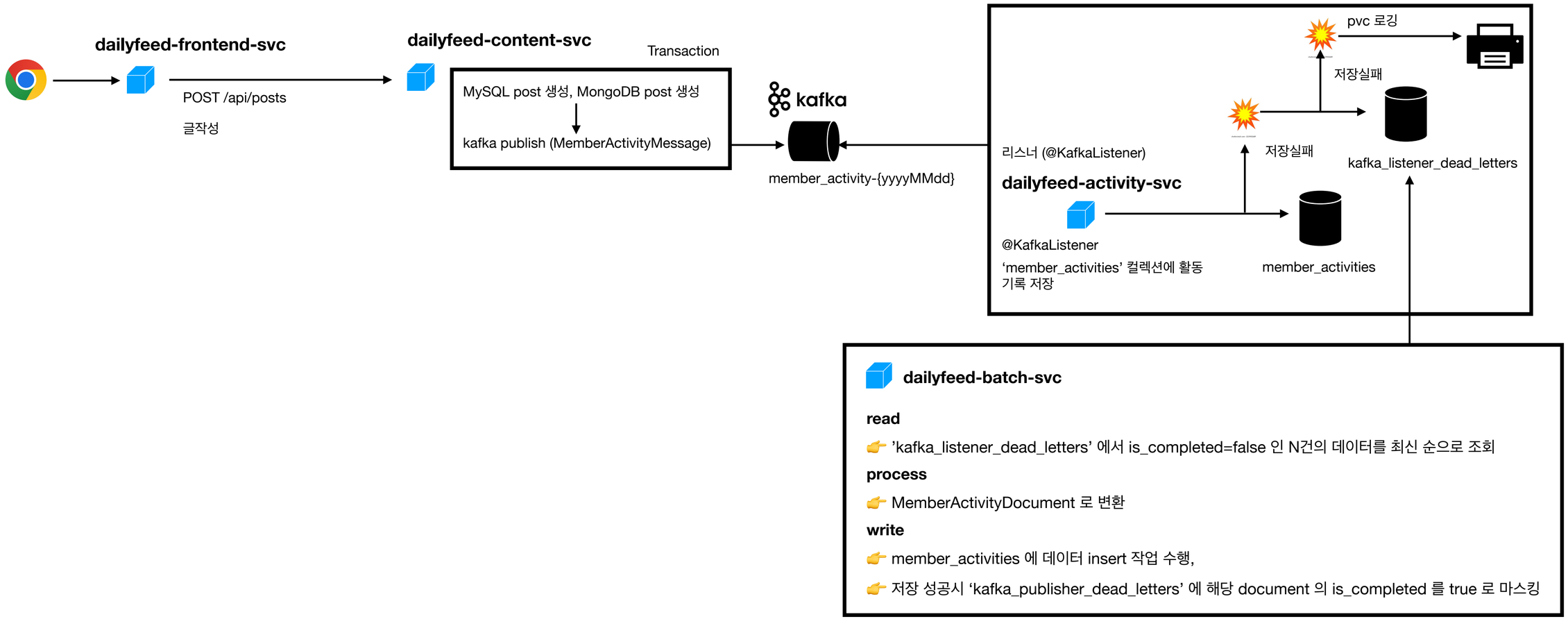
# Case (1) Kafka : publisher 측에서의 통신에러 처리




e.g.

= 카프카 증설 및 운영에 용이하게 kafka 저장이 실패할 경우 deadletter 저장소에 저장 후 배치를 통해 발송하는 경우

# Case (1) Kafka : listener 측에서의 통신에러 처리



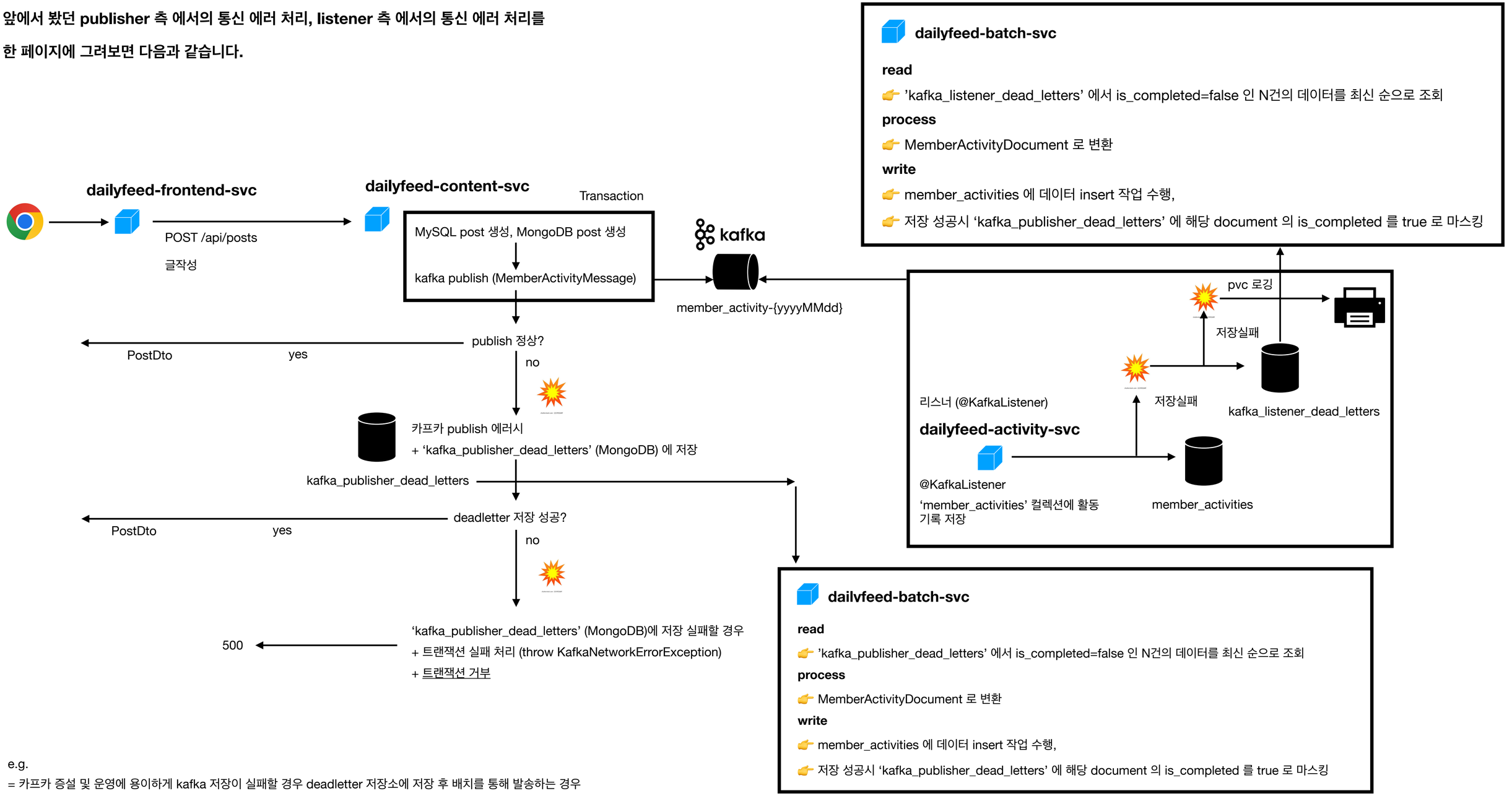

e.g.

= 카프카 증설 및 운영에 용이하게 kafka 저장이 실패할 경우 deadletter 저장소에 저장 후 배치를 통해 발송하는 경우

# Case (1) Kafka : publisher,listener 측에서의 통신에러 처리

**앞에서 봤던 publisher 측 에서의 통신 에러 처리, listener 측 에서의 통신 에러 처리를**

**한 페이지에 그려보면 다음과 같습니다.**




e.g.

= 카프카 증설 및 운영에 용이하게 kafka 저장이 실패할 경우 deadletter 저장소에 저장 후 배치를 통해 발송하는 경우

## Case (2) Kafka

## : Acks = 1, At Least Once

## : 중복메시지 수신여부 체크 및 upsert

# Case (2) - Acks = 1, At Least Once

**Producer Acknowledgement : Acks = 1 선택**

참고)

**acks = 1** : Producer 가 메시지를 브로커에 보낼때 리더파티션에 메시지를 보낸 후, 리더파티션이 메시지를 받아서 로그에 쓴 후 리더로부터 ack 을 받는 방식입니다.

**acks = 0** : Producer 가 메시지를 브로커에 보낼때 메시지를 보내기만 하고 ack 를 기다리지 않는 방식입니다.

**acks = all (-1)** : Producer 가 메시지를 브로커에 보낼때 메시지를 보낸 후 리더파티션 & 모든 ISR 이 메시지를 받은 후 ack 을 받는 방식입니다.

(복제본에 모두 복제되야 acks 를 받음)

acks = all 을 사용할 경우 min.insync.replicas=2 등으로 최소 한도의 복제 본 수를 지정해야 합니다.

현재 프로젝트에서는 일반적으로 많이 설정되는 Acks = 1 로 지정했습니다. Acks = 1 로 지정하면 리더가 죽고 팔로워가 복제를 받지 못한 경우 데이터가 손실될 가능성이 있지만 개발 버전의 환경상 많은 리소스가 불필요하고, 리더파티션 1기만 운영할 경우도 있기에 acks = 1 로 지정했습니다. 만약 acks=all 을 선택할 경우는 꼭 'min.insync.replcas' 를 지정해야 합니다.

**Consumer Offset : At Least Once 선택**

참고)

**At Most Once (최대 한번)** : Offset 을 먼저 커밋하고, 나중에 처리하는 방식입니다. 처리 중 실패할 경우 메시지가 손실됩니다. 최대 1번 처리됩니다.

메시지의 중복처리는 없지만, 데이터 손실가능성이 존재합니다.

**At Least Once (최소 한번)** : 처리를 먼저 하고 Offset을 나중에 커밋하는 방식입니다. 처리 후 커밋 전 실패할 경우 재처리 됩니다.(커밋이 안된 메시지는 재수신)

데이터의 손실은 없지만 메시지가 중복 수신하게 됩니다. 따라서 애플리케이션 레벨에서 메시지 중복 시에 대한 처리를 해주면 메시지 유실 없이 카프카 통신이 가능합니다.

가장 일반적으로 사용되는 방식입니다.

**Exactly Once (정확히 한번)** : 처리와 커밋이 하나의 트랜잭션으로 묶입니다. 둘 다 성공하거나 둘 다 실패합니다. 정확히 1번 처리됩니다.

중복처리가 없고 데이터 손실 역시 없는 방식이지만 성능 오버헤드가 있으며 구현이 복잡하고 추가설정이 필요합니다.

**dailyfeed 프로젝트**

**+ Producer Acknowledgement =  Acks = 1 선택**

: 리소스/비용을 줄이기 위해 리더 파티션 1기만 운영하는 경우까지 고려

**+ Consumer Offset 커밋 방식 = At Least Once 선택**

: 중복된 메시지를 받더라도 Redis 를 통해 중복 메시지를 체크하고, 데이터 저장시에도 같은 메시지일 경우 upsert 를 하도록 지정했습니다.

# Case (2) - 중복 메시지 체크 방식

**메시지 중복체크**

**: 메시지 전송 시 메시지 키를 발급, 메시지 수신시 Redis/Database 에서 중복 수신 여부를 체크하는 방식을 사용**

**메시지 키 형식**

POST_CREATE

“member_activity:kafka_event:POST_CREATE###{postId}###{memberId}”

POST_UPDATE,POST_DELETE,POST_READ

“member_activity:kafka_event:{POST_UPDATE|POST_DELETE|POST_READ}###{postId}###{memberId}###{yyyy-MM-dd HH:mm:ss.SSSSSSSSS”

COMMENT_CREATE

“member_activity:kafka_event:COMMENT_CREATE###{postId}###{memberId}”

COMMENT_UPDATE,COMMENT_DELETE,COMMENT_READ

“member_activity:kafka_event:{COMMENT_UPDATE|COMMENT_DELETE|COMMENT_READ}###{commenttId}###{memberId}###{yyyy-MM-dd HH:mm:ss.SSSSSSSSS”

LIKE_POST, LIKE_POST_CANCEL

“member_activity:kafka_event:{LIKE_POST|LIKE_POST_CANCEL}###{postId}###{memberId}”

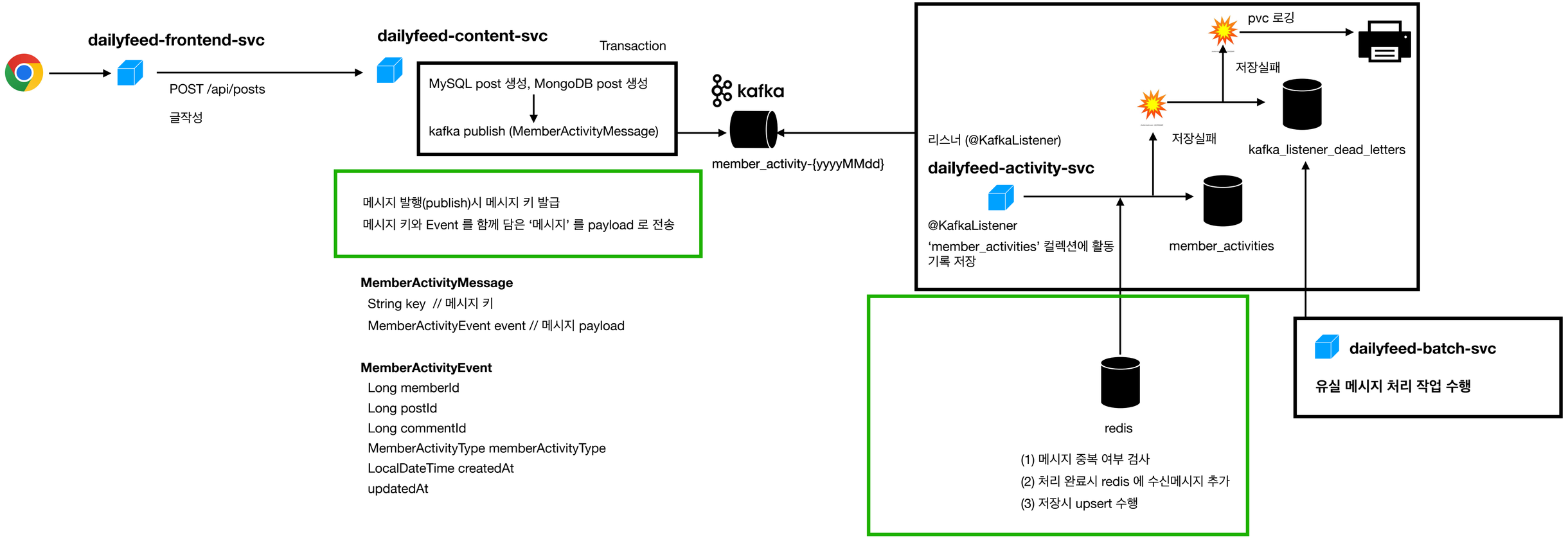
LIKE_COMMENT, LIKE_COMMENT_CANCEL

“member_activity:kafka_event:{LIKE_COMMENT|LIKE_COMMENT_CANCEL}###{commentId}###{memberId}”

# Case (2) - 중복 메시지 체크 방식

**메시지 중복체크**

**: 메시지 전송 시 메시지 키를 발급, 메시지 수신시 Redis/Database 에서 중복 수신 여부를 체크하는 방식을 사용**



## Case (3) Kafka

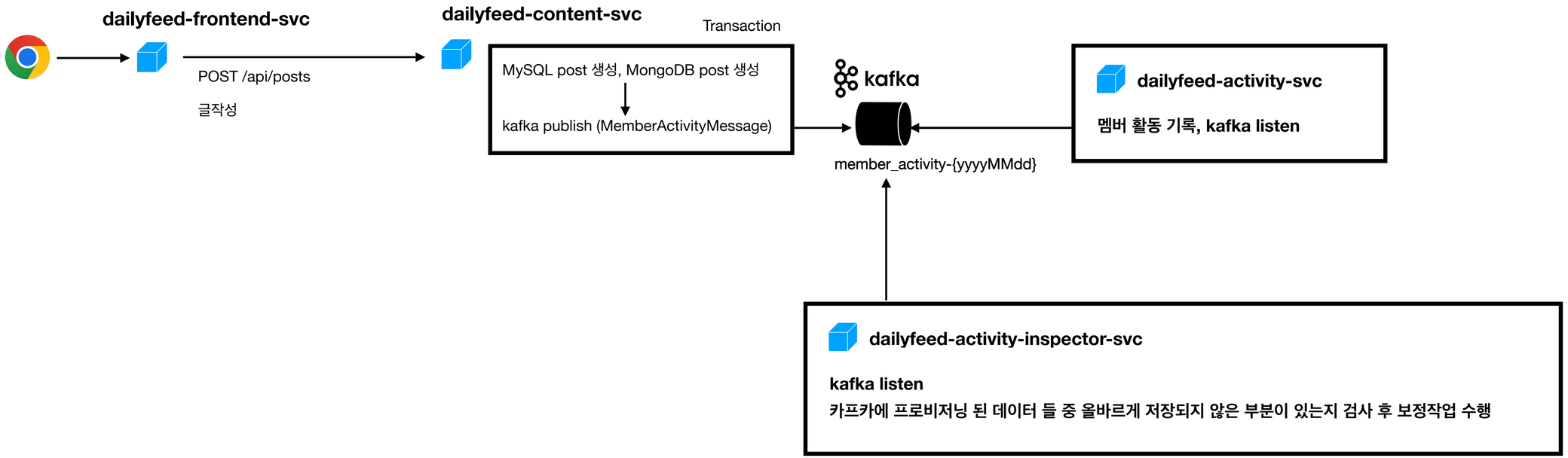
## : 날짜별 토픽 ( {topicName}-yyyyMMdd)

# Case (3) - 날짜별 토픽 ({topicName}-yyyyMMdd)

**날짜별 토픽을 도입하게 된 이유**

**: 데이터의 오류가 있는지 등에 대한 후보정 작업에 대해 유연하게 전략을 취할수 있다는 점**

**: 이미 처리 완료된 데이터에 대한 토픽 (e.g. 7일 전)의 경우 토픽 삭제를 통해 운영시 카프카 브로커가 점유하는 디스크 사이즈를 줄일 수 있다는 점 (운영 비용 최적화 가능한 구조를 고려함)**

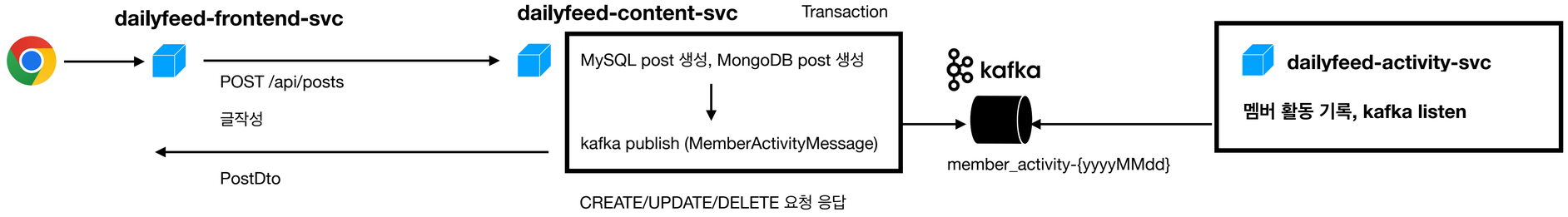


Case (4) 스케일아웃 그룹 분류

: Read 스케일아웃 그룹

: Create/Update/Delete 스케일아웃 그룹

# 설계 Item (2) - Read 트래픽과 Create/Update/Delete 트래픽의 스케일아웃 그룹 분류





e.g. content-svc 와 timeline-svc 가 하나의 인스턴스에 post-svc 라는 이름으로 운영될경우 운영 상의 단점

(1) 조회 시에 발생하는 슬로우 쿼리 등으로 인해 쓰기 작업에도 영향을 주는 현상 발생

(2) post-svc 라는 이름의 인스턴스를 통으로 재시작 하는 경우 조회 작업에도 함께 영향

(3) 조회 서비스의 경우 기획 상 변경이 잦을 수 있는 경우 배포가 잦을 수 있는데 쓰기 작업에도 영향

## Case (5) Istio VirtualService, DestinationRule

## : VirtualService

## : DestinationRule

## Case (6) 도커 이미지 빌드 자동화 (github workflow)

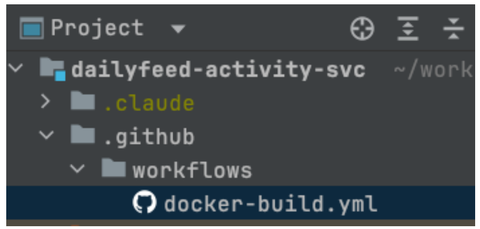
: 자동 모드

: 수동 모드

# Case (6) - 도커 이미지 빌드 자동화 (github workflow) - 자동 모드

**workflow 대상 브랜치 : main/develop/hotfix**


**commit & push**

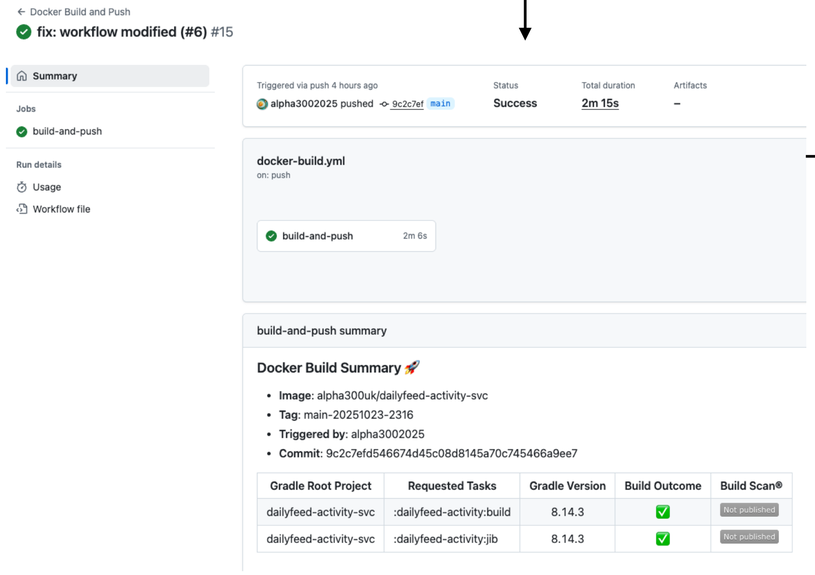



**github repository 내의 workflow 발동 (trigger)**

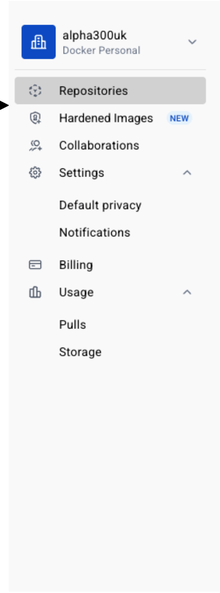
참고 : https://github.com/alpha3002025/dailyfeed-activity-svc/blob/main/.github/workflows/docker-build.yml

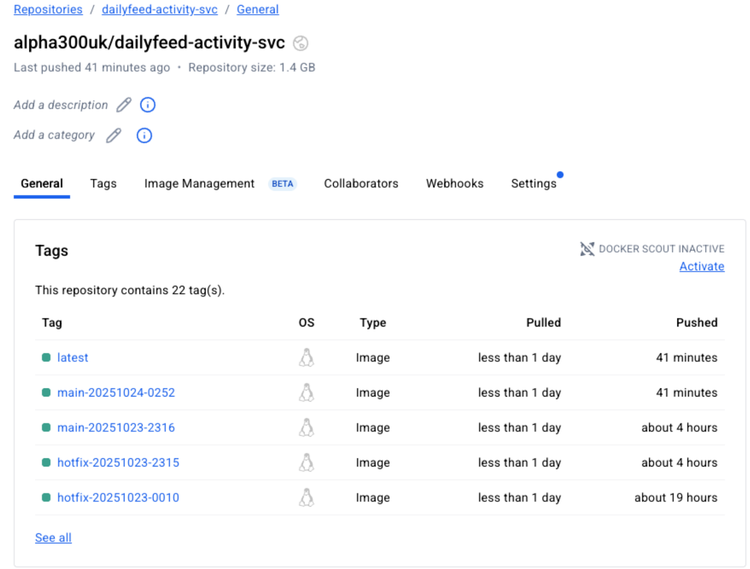
**이미지 태그 형식**

**: {branchName}-{yyyyMMdd}-{build번호}**




**docker 이미지 jib build & push**

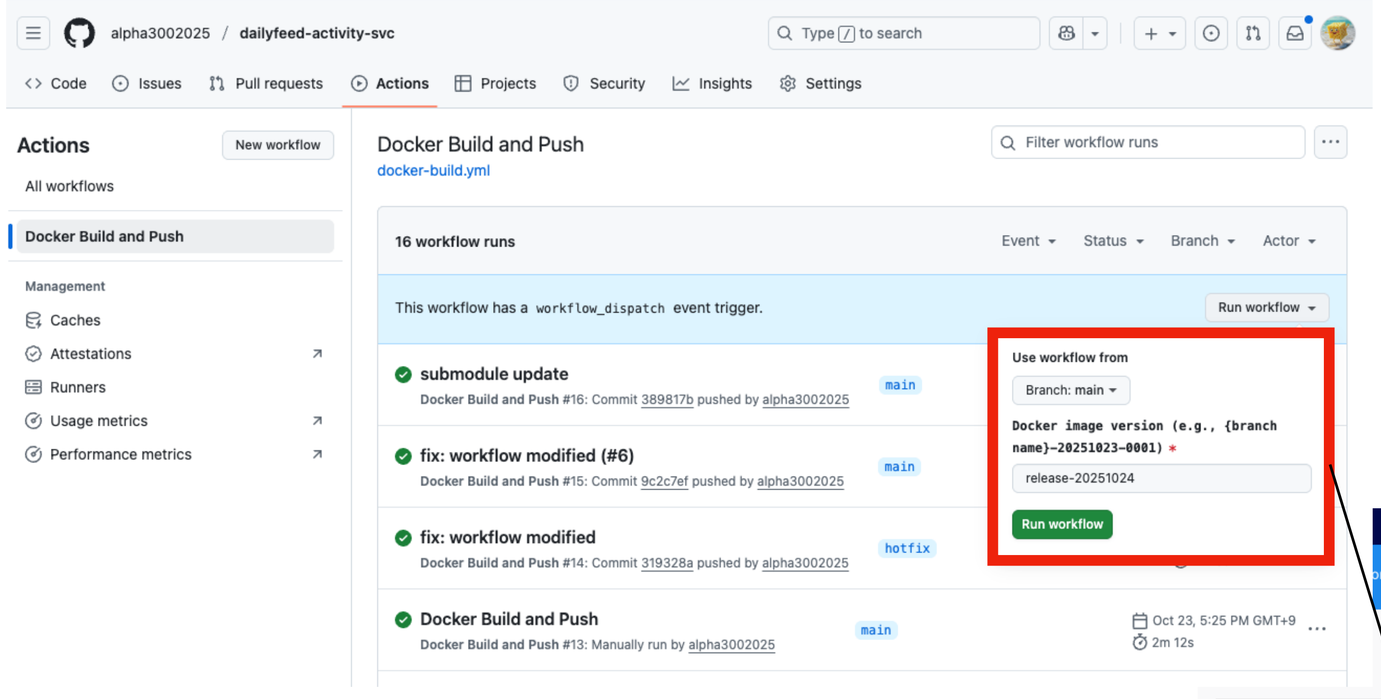


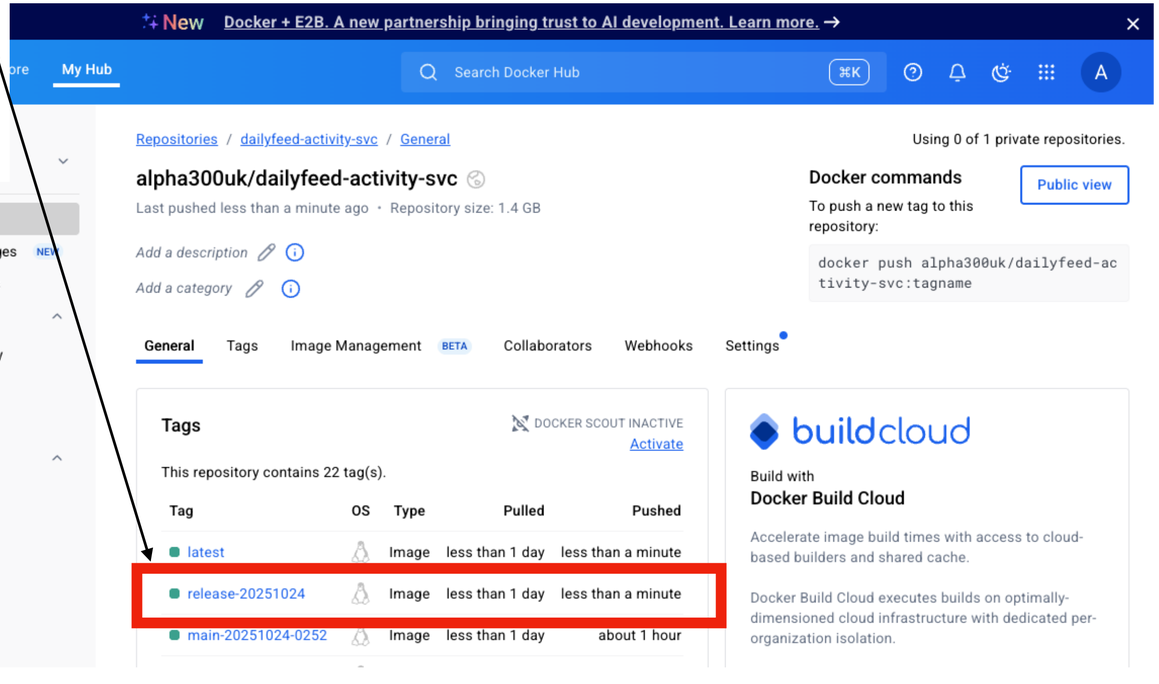
# Case (6) - 도커 이미지 빌드 자동화 (github workflow) - 수동 모드

**workflow 대상 브랜치 : main/develop/hotfix**

**github workflow 실행**




**docker hub (push 된 이미지 확인)**



# Case (7) resillience4j feign

# : circuitbreaker, rate limiter

**Case (8) JWT**

**: Key Refresh**

**: JWT ID, Black List**

Case (8) 주요 코딩 컨벤션

: no common -> 서브모듈로 공유

: mapper

: get—OrThrow

: data 로직 공통화 자제 (과도한 리소스 사용방지)