dailyfeed backend

주요 백엔드 기능 및 설계 관련 설명

2025/10/27 (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

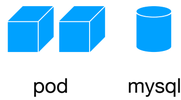
  :: 통신레벨에서는 통신만 수행하고, 익셉션은 애플리케이션 계층으로 throw

+ 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

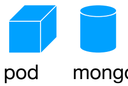
+ 프로필/썸네일 이미지 업로드/수정/삭제/조회

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

min: 3
max: 5

dailyfeed-activity-svc

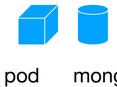
+ 활동기록 저장/조회/수정



min: 1
max: 2

dailyfeed-search-svc : 본문 검색 서비스

+ Full Text Search 를 위한 별도의 서비스



min: 1
max: 3

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)

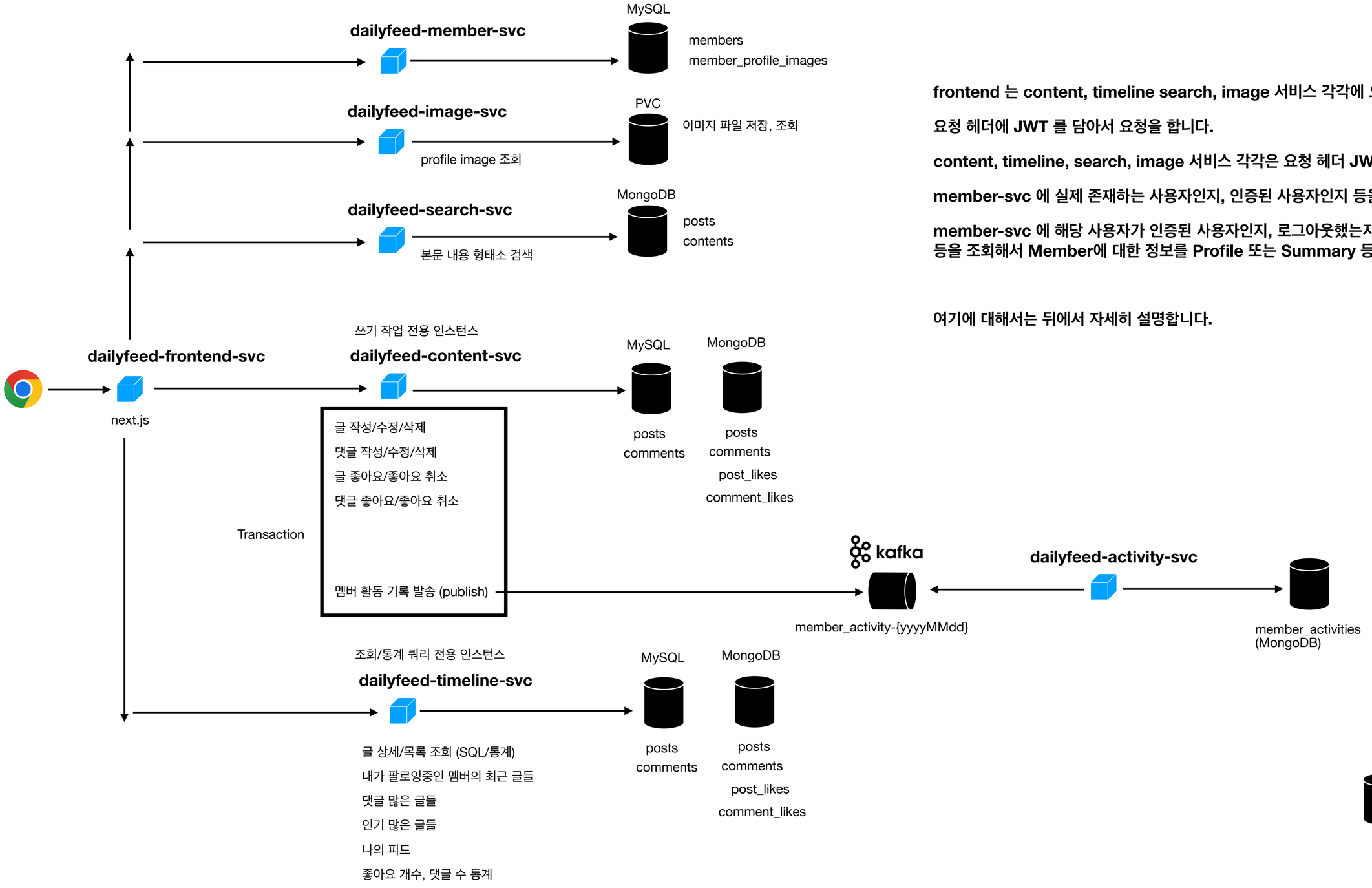
min: 1
max: 7

설치방법

(dailyfeed-installer)

전반적인 흐름

전반적인 흐름



frontend 는 content, timeline search, image 서비스 각각에 요청을 보낼때

요청 헤더에 JWT 를 담아서 요청을 합니다.

content, timeline, search, image 서비스 각각은 요청 헤더 JWT 를 담아서

member-svc 에 실제 존재하는 사용자인지, 인증된 사용자인지 등을 조회합니다.

member-svc 에 해당 사용자가 인증된 사용자인지, 로그아웃했는지, 만료된 토큰을 가지고 있는지 등을 조회해서 Member에 대한 정보를 Profile 또는 Summary 등으로 return 해줍니다.

여기에 대해서는 뒤에서 자세히 설명합니다.

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.

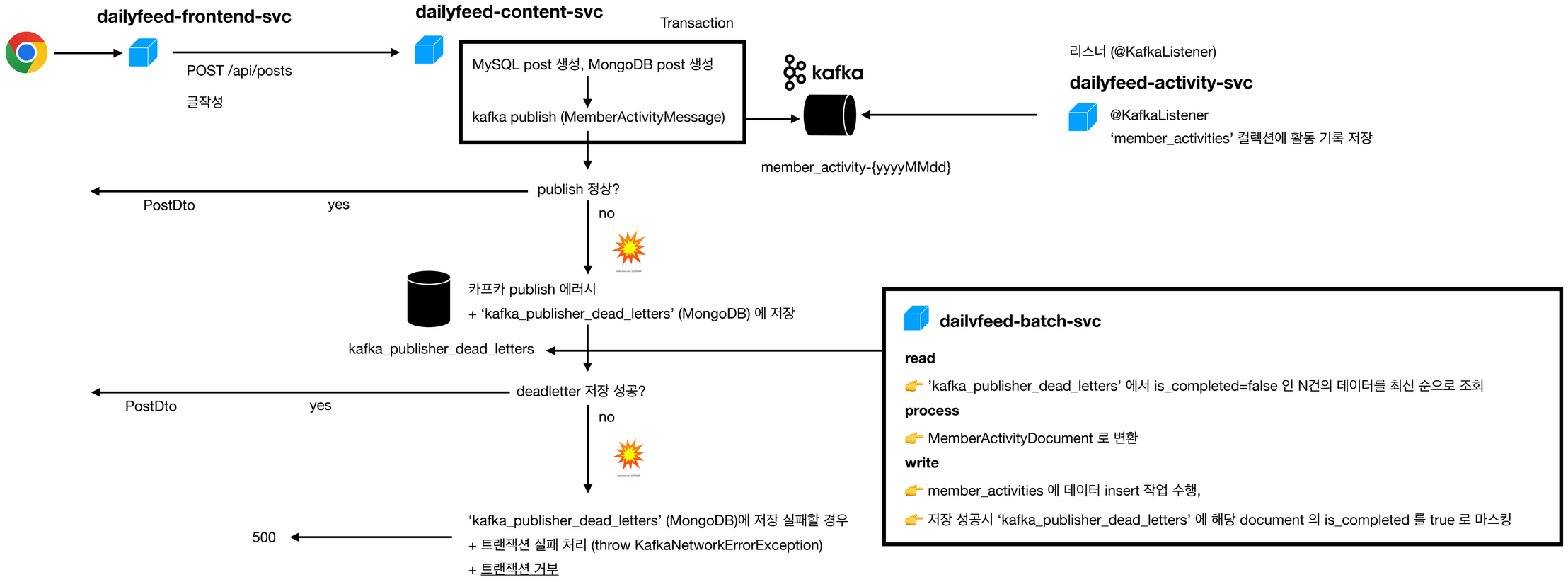
+ 카프카 운영시 카프카 증설 및 업그레이드 작업으로 인해 카프카 업그레이드 이슈 등이 있을 때는 세컨더리 저장소를 통해 Fail Over를 하는 것도 중요한 요소 중 하나입니다.

+ 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

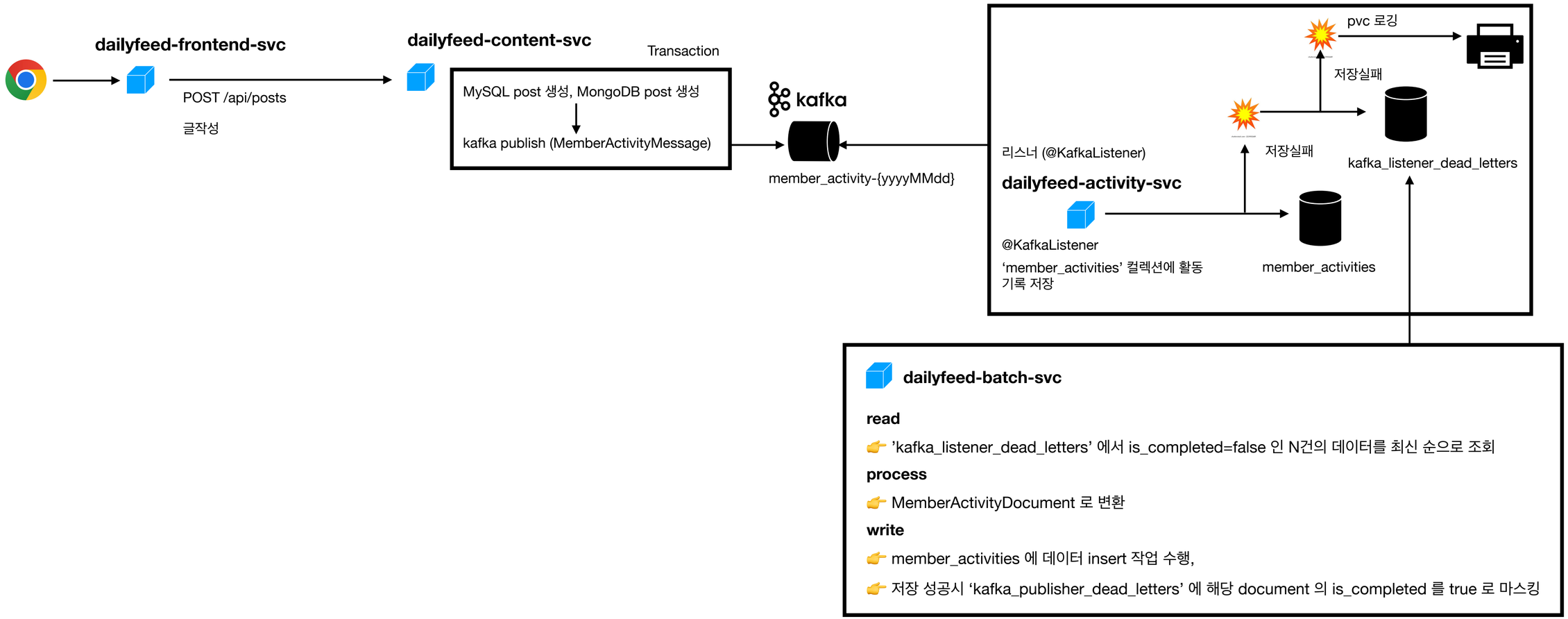
Case (1) Kafka : publisher 측에서의 통신에러 처리



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



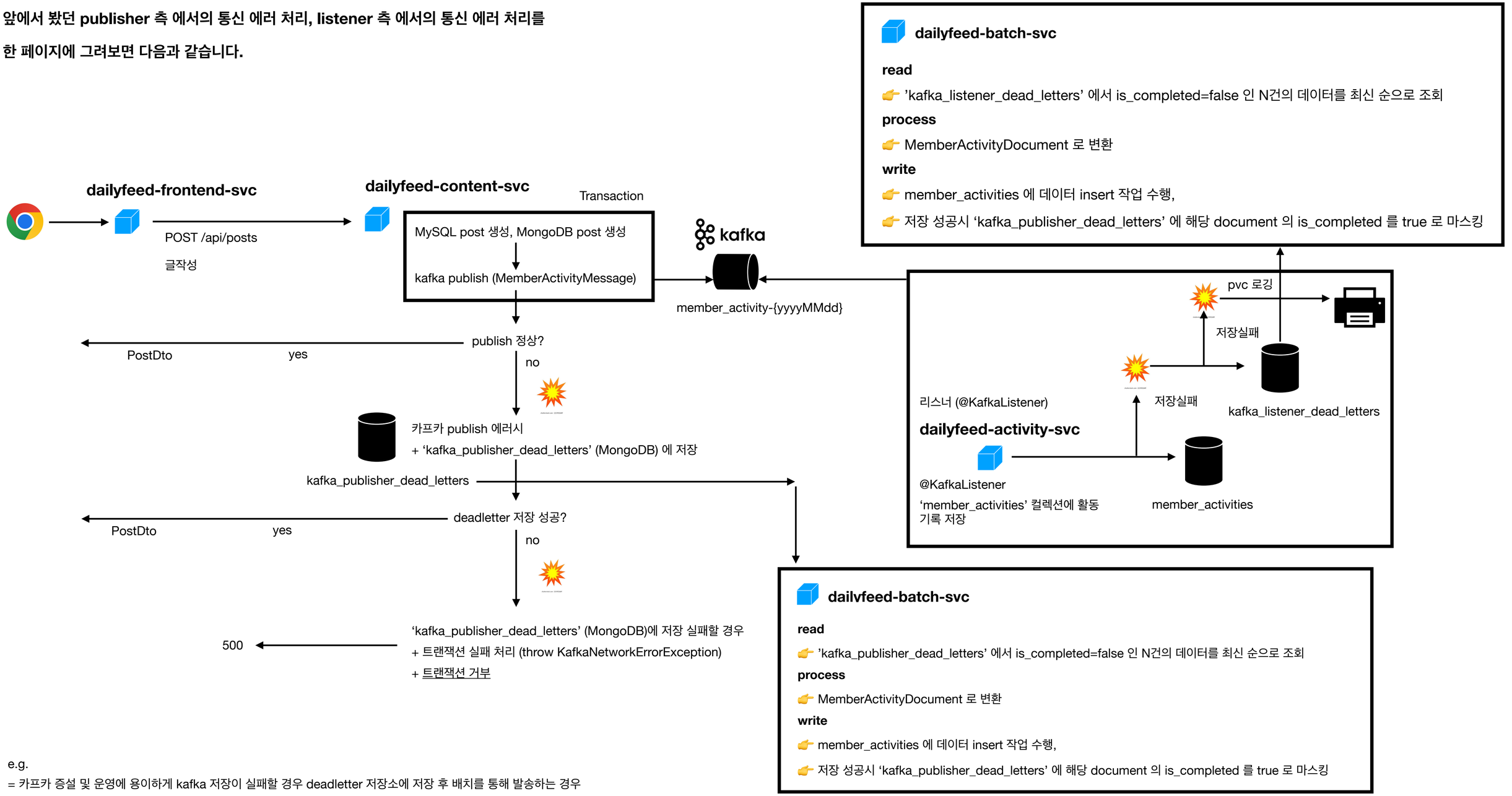
e.g.

= 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}”

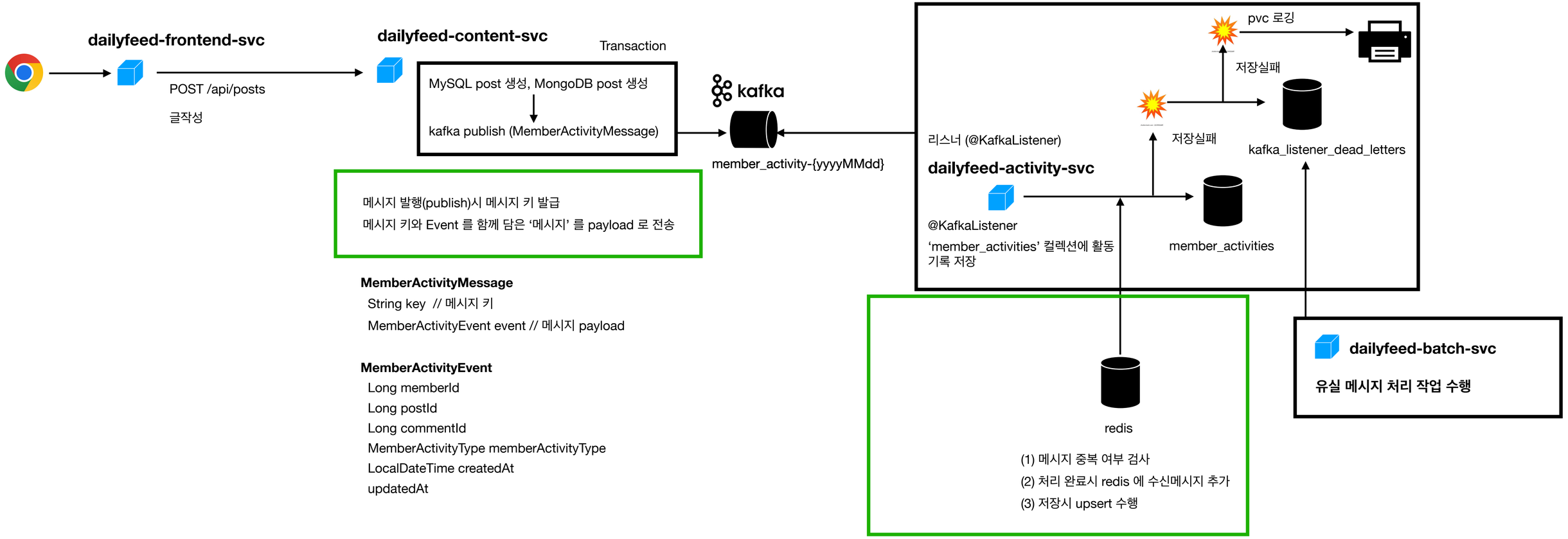
LIKE_COMMENT, LIKE_COMMENT_CANCEL

“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

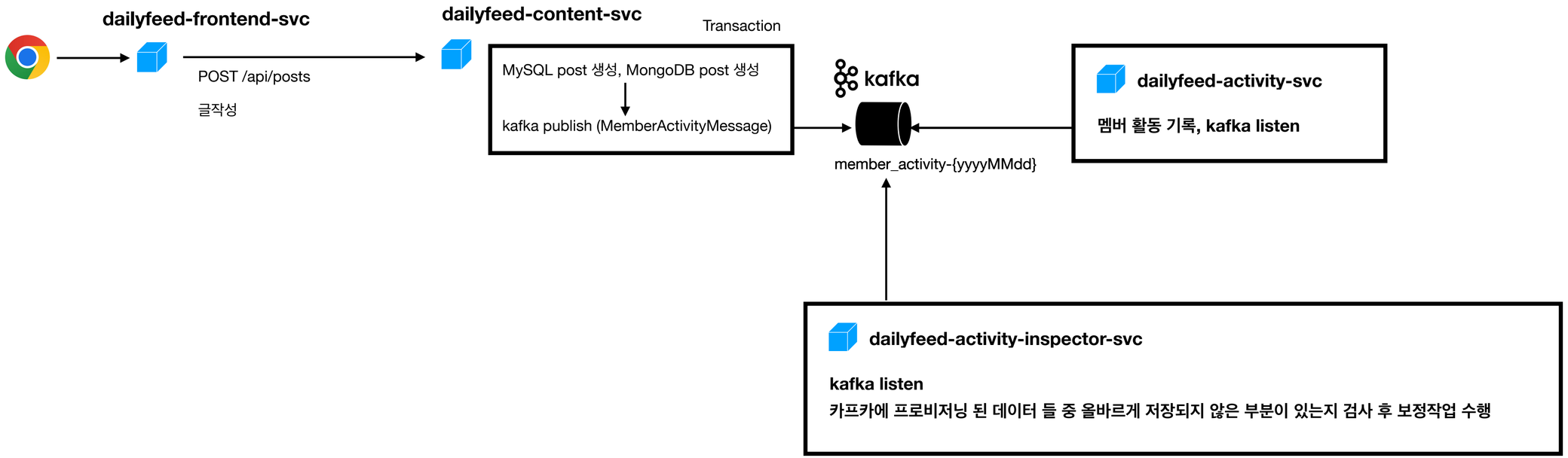
: 날짜별 토픽 ({topicName}-yyyyMMdd)

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)



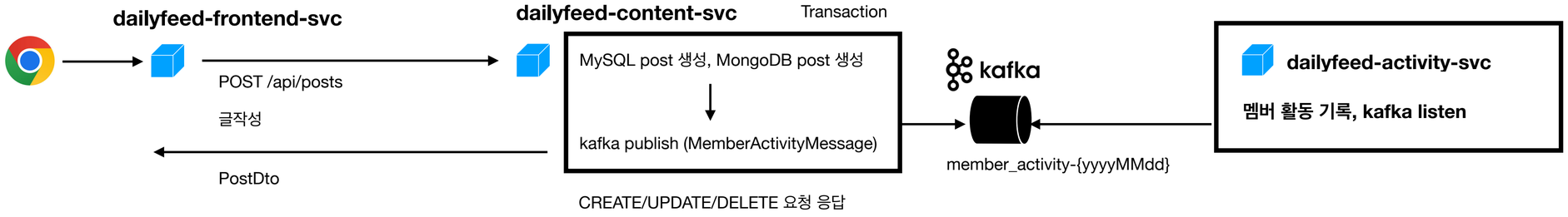
Case (4) 스케일아웃 그룹 분류

: Read 스케일아웃 그룹

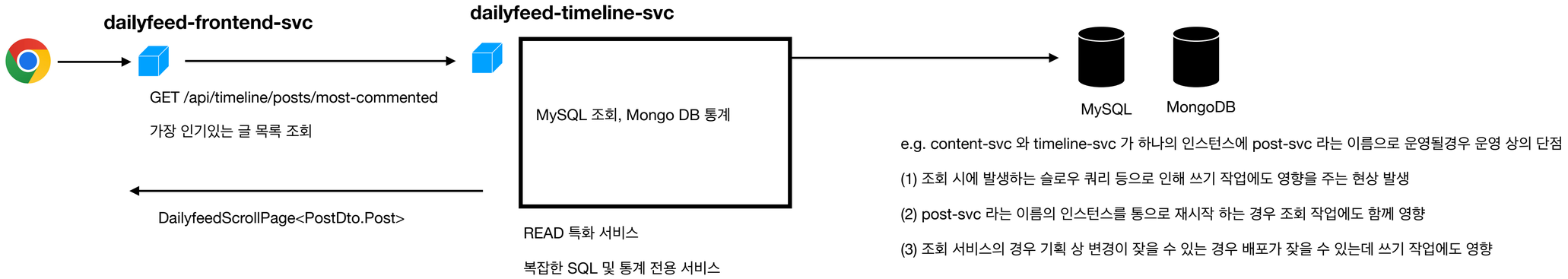
: Create/Update/Delete 스케일아웃 그룹

Case (4) - Create/Update/Delete 트래픽과 Read 트래픽의 스케일아웃 그룹 분류

Create/Update/Delete 트래픽



Read 트래픽



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

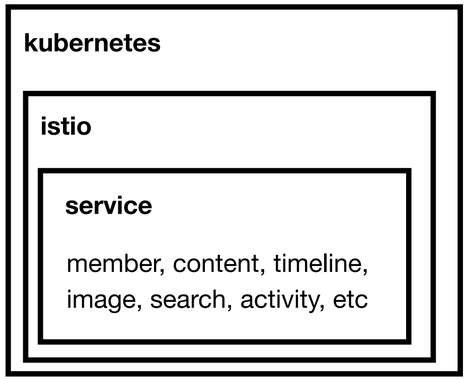
참고) istio 란?

Istio 란?

혹시라도 istio 가 무엇인지 모를 수도 있는 분들을 위해 간단한 설명을 추가합니다.

istio 란?

참고 : https://istio.io



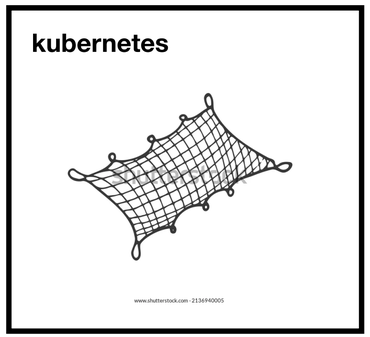
kubernetes 또는 여러 컨테이너 오케스트레이션 등에서 Envoy Proxy 역할을 수행하는 sidecar 를 통해

전체 네트워크의 서비스 메시 (Service Mesh) 레벨에서

라우팅, HTTPS 상호 보안, HttpRetry, Timeout, CircuitBreaker, HTTP Connection Pool 등의 기능을 제공하는

네트워크 유틸리티 플랫폼 입니다.

서비스 메시 (Service Mesh) 란?



Mesh 는 network 의 ‘그물’을 의미하는데, 이 ‘Service Mesh’ 라는 것은

kubernetes 의 Network 레벨을 의미하는 ‘Mesh’ 를 의미합니다.

Istio 를 사용하면, 1차적으로 ‘Mesh’라고 부르는 Network 레벨에서

통합적인 제어 (라우팅, HTTS, Timeout, HTTP Retry, CircuitBreaker, Connection Pool 등) 를 수행할 수 있습니다.

요청이 너무 많을 경우 그 요청을 모두 처리할 필요가 없을수도 있습니다.

너무 잦은요청은 네트워크 레벨에서 거부하고, 다음 타임윈도우에 재요청하게 하는 것이 나을 수 있습니다.

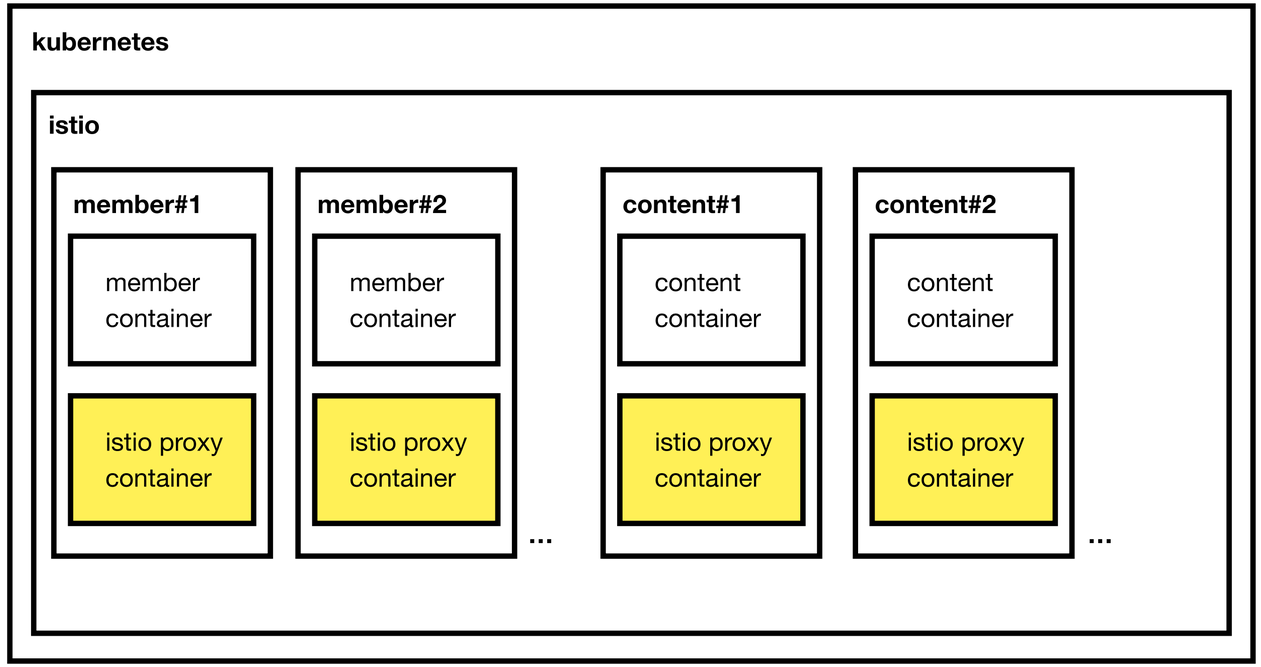
Service Mesh 는 이런 기능에서부터 여러가지 기능들을 제공합니다.

예를 들면 논리적인 라우팅을 통해 여러 버전의 인스턴스를 공존하게 하는 것 역시 가능합니다.

Envoy Proxy 란 ?

혹시라도 istio 가 무엇인지 모를 수도 있는 분들을 위해 간단한 설명을 추가합니다.

Envoy Proxy 란?



좌측 그림의 노란색 상자에는 'istio proxy container' 라는 것이 각각 존재합니다.

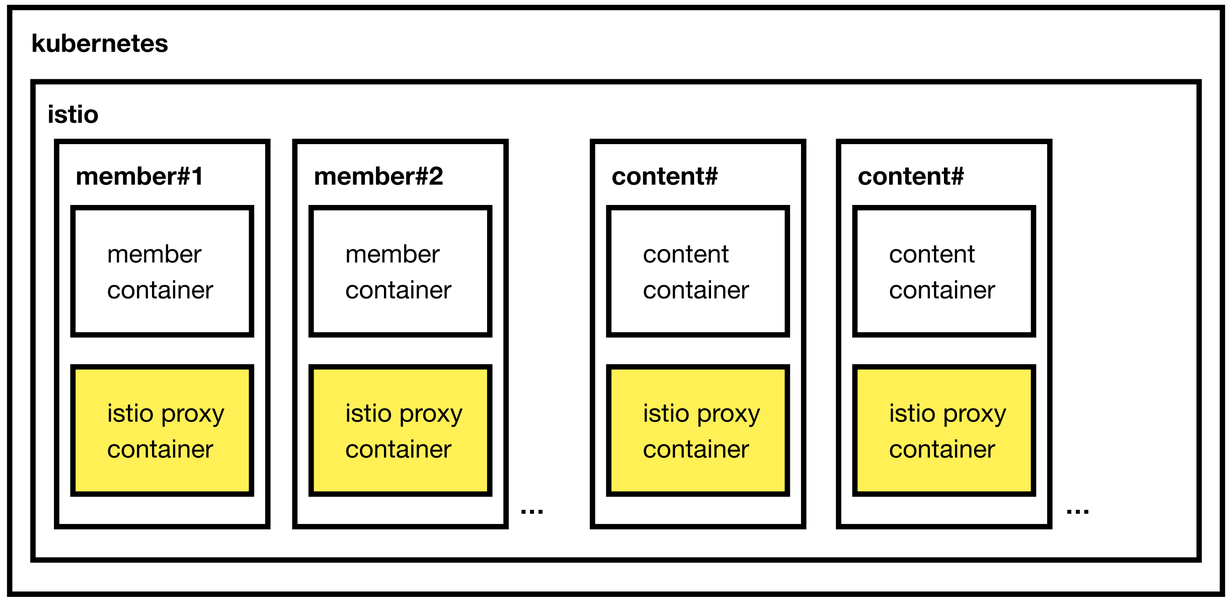
istio 는 이렇게 각각의 pod 내에 istio가 관리하는 container 를 주입해서 각각의 pod 에 대한 네트워크 제어를 수행합니다. 이렇게 주입된 'istio proxy container' 들은 각각 자신의 파드를 대표해서 다른 서비스와 통신을 하는 '외교관', 'proxy' 같은 역할을 수행합니다.

예를 들면 HTTPS 가 각각의 pod 에 적용이 안되어 있더라도 istio proxy container 가 주입되어 있고 PeerAuthentication 을 SRICT 로 설정해둔 상태라면, pod 간의 통신시 서비스 컨테이너가 HTTPS 요청을 하지 않더라도 istio proxy container 가 네트워크 레벨에서 HTTPS 암호화를 통해 파드간의 통신을 암호화합니다.

이렇게 각각의 파드에 대해 대사(Envoy) 역할을 하는 Proxy 를 Envoy Proxy 라고 일반적으로 부르며, 일반적으로는 Sidecar 라고 부르기도 하고, Proxy 라고 부르기도 하기도 합니다.

Istio 사용의 장점 - (1) HTTPS

혹시라도 istio 가 무엇인지 모를 수도 있는 분들을 위해 간단한 설명을 추가합니다.



(1) HTTPS (PeerAuthentication)

위에서 이야기했듯, deployment, pod 내에 배포한 container 가 기본적으로 https 를 적용하지 않았더라도 istio 의 proxy 는 기본적으로 pod 와 pod 간 통신을 HTTPS 로 암호화해서 통신할 수 있도록 네트워크 레벨에서 암호화를 처리해서 통신을 합니다.

kubernetes 를 도입을 하더라도, 각각의 애플리케이션 서비스마다 HTTPS 인증서를 관리하는 것은 쉬운 일이 아닙니다. 만약 Istio 를 사용하게 된다면 이런 작업들을 최소화 할 수 있게 됩니다.

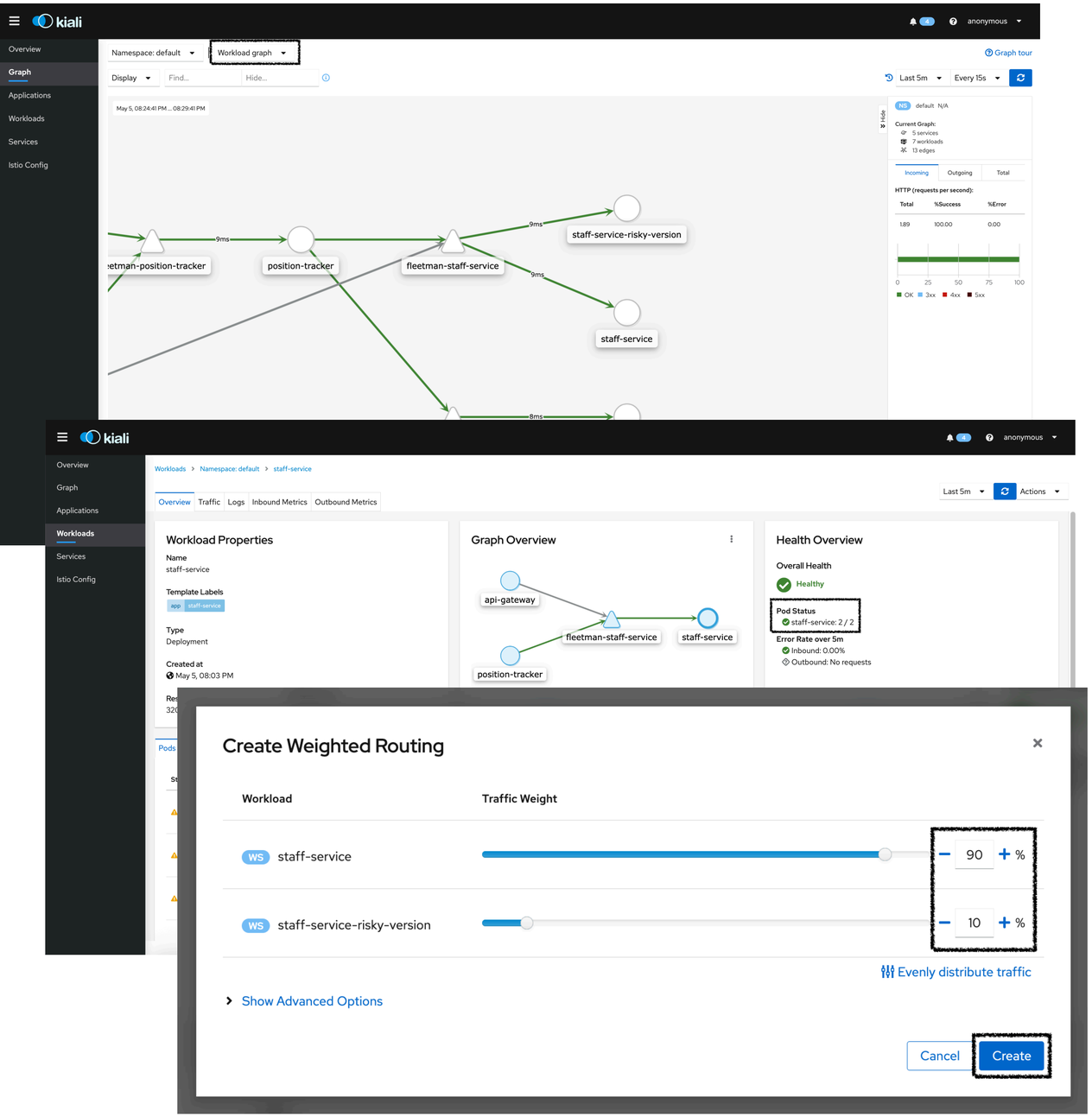
kubernetes 를 도입한다고 해도, istio 등에 대해서 잘 모른채로 부딪히기 식으로만 접근하면

결국 관리가 쉽지 않아질 가능성이 있습니다.

운영 레벨에서 kubernetes 를 제어하기 위해서는 최소한 istio 에 대한 관리 능력 정도는 필요하다고 생각합니다.

Istio 사용의 장점 - (2) Kiali, virtual service, routing

혹시라도 istio 가 무엇인지 모를 수도 있는 분들을 위해 간단한 설명을 추가합니다.



(2) kiali

istio 에서는 kiali 를 별도의 addon 으로 제공하는데, 각각의 서비스가 어디로 향하는지를 시각화해서 표현한 대시보드를 제공합니다. 이 대시보드 내에서 긴급상황 등에 대해 트래픽의 제어를 할수 있고, 전체 트래픽의 흐름을 확인하는 것 역시 가능합니다.

예를 들어 timeline 애플리케이션의 v2 버전의 디플로이먼트가 슬로우쿼리로 인해 장애가 나고 있다고 해보겠습니다. 이에 대응하기 위해 kiali 대시보드에서는 다음의 대응을 취할수 있습니다.

- v1 의 timeline 디플로이먼트를 배포

- timeline-svc 서비스로의 요청에 대해 v2 버전의 timeline 디플로이먼트로는 요청을 끊기

- timeline-svc 서비스로의 요청에 대해서는 앞으로는 v1 버전의 디플로이먼트로 트래픽이 향하도록 제어

물론 이런 기능은 kiali 에서만 제공되는 기능은 아닙니다. istio 의 yaml 편집을 통해서도 제공할 수 있으며 git 등의 버전관리 등이 적용되도록 일반적인 운영상황에서는 yaml 편집으로 인프라에 대한 내력을 관리합니다.

위와 같은 장애 상황에서만 사용하지는 않습니다. 신규버전 배포시 카나리 배포와 유사한 방식을 istio 내에서 적용할 수 있습니다. 예를 들면 다음과 같은 방식입니다.

- v2 버전 배포

- virtual service 는 5% 의 트래픽은 v2 를 라우팅, 95% 의 트래픽은 v1 을 라우팅

- (1일 후) virtual service 는 10% 의 트래픽은 v2 를 라우팅, 90% 의 트래픽은 v1 을 라우팅

- (7일 후) virtual service 는 50% 의 트래픽은 v2 를 라우팅, 50% 의 트래픽은 v1 을 라우팅

- (30일 후) virtual service 는 100% 의 트래픽은 v2 를 라우팅, 0% 의 트래픽은 v1 을 라우팅

이렇게 istio 를 사용하면 각 네트워크 트래픽에 weight 을 부여해서 virtual service 내에서 각 destination 에 대해 weight 을 부여해서 몇 퍼센트의 네트워크를 v2 로 흘려보내서 장애가 나는지 아닌지를 미리 파악해보면서 천천히 배포해볼수 있게 됩니다.

Istio 사용의 장점 - (3) Destination Rule - CircuitBreaker, OutlierDetection

혹시라도 istio 가 무엇인지 모를 수도 있는 분들을 위해 간단한 설명을 추가합니다.

```yaml
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: dailyfeed-activity
  namespace: dailyfeed
spec:
  host: dailyfeed-activity
  trafficPolicy:
    # Load balancing: 최소 요청 수를 가진 인스턴스로 라우팅 (기본 ROUND_ROBIN보다 효율적)
    loadBalancer:
      simple: LEAST_REQUEST
      leastRequestLbConfig:
        choiceCount: 2  # 2개 인스턴스 중 최소 요청 선택

    # Connection Pool Settings
    connectionPool:
      tcp:
        maxConnections: 100           # TCP 최대 연결 수
        connectTimeout: 3s            # 연결 타임아웃
        tcpKeepalive:                 # TCP Keepalive 설정
          time: 7200s                 # 2시간
          interval: 75s
          probes: 9
      http:
        http1MaxPendingRequests: 100  # HTTP/1.1 대기 요청 수
        http2MaxRequests: 500         # HTTP/2 최대 요청 수
        maxRequestsPerConnection: 10  # 연결당 최대 요청 수 (연결 재사용 제한)
        maxRetries: 3                 # 최대 재시도 수
        idleTimeout: 30s              # 유휴 연결 타임아웃
        h2UpgradePolicy: UPGRADE      # HTTP/2 업그레이드 허용

    # Outlier Detection (Circuit Breaker)
    outlierDetection:
      consecutive5xxErrors: 5         # 연속 5xx 에러 5회 시 제거
      consecutiveGatewayErrors: 3     # 연속 게이트웨이 에러 3회 시 제거 (더 민감하게)
      interval: 1m                    # 분석 간격
      baseEjectionTime: 5m            # 기본 제거 시간
      maxEjectionPercent: 50          # 최대 제거 비율 (50% 이상 제거 방지)
      minHealthPercent: 30            # 최소 건강한 인스턴스 비율 (30% 보장)
      splitExternalLocalOriginErrors: true  # 외부/내부 오류 구분

  # Subsets (버전별 라우팅용 - 필요시 사용)
  subsets:
  - name: v1
    labels:
      version: v1
```

(3) DestinationRule

Destination Rule 은 VirtualService 가 관리하는 목적지 룰을 의미하는데, 이 Destination Rule 은 Connection Pool 의 갯수부터 어떤 에러가 나타났을 때 잠시 Eject 할지, 얼마 동안 Eject 할지 등을 의미하는 Circuit Breaking, LoadBalaner 기능 까지도 지원합니다.

더 자세한 내용은 좌측의 캡쳐를 참고해주시기 바랍니다.

애플리케이션 레벨에서도 CircuitBreaker 를 제공하는 것 역시 가능합니다.

하지만 혹시라도 모르는 장애 상황에서 CircuitBreaker 가 적용되지 않은 애플리케이션 서비스는 있을수도 있습니다. 항상 모든 애플리케이션에 CircuitBreaker 를 적용할 수 있는 것은 아니기에, 이런 경우에 대해 네트워크 레벨에서 CircuitBreaker 가 작동된다면, 전체 서비스로 장애가 전파되지 않을 수 있다는 안전 대책을 마련할 수 있게 됩니다.

istio 도입에 대해

혹시라도 istio 가 무엇인지 모를 수도 있는 분들을 위해 간단한 설명을 추가합니다.

위에서 살펴봤던 장점들은 istio의 모든 장점을 설명하지는 못합니다.

istio 도입은 필수가 아니지만, 아마도 대부분의 kubernetes 환경에서 제대로된 데브옵스 팀이 운영되고 있다면

아마도 대부분이 istio 를 운영하고 있을 가능성이 큽니다.

쿠버네티스 도입이 배포/장애 관리 등의 이슈로 인해 오히려 운영의 효율성을 떨어뜨릴때도 있는데

이 경우 istio 를 도입한다면 운영상의 어려움을 경감시킬 수 있습니다.

Case (5) Istio VirtualService, DestinationRule

: VirtualService

: DestinationRule

Case (5) - VirtualService

VirtualService 는 HttpTimeout, HttpRetry 를 정의

	member	content	timeline	image	search	activity
timeout	10s	10s	10s	30s (이미지 처리는 시간이 오래 걸릴수도 있다는 점을 감안)	15s (검색은 조금 더 긴 타임아웃 필요)	10s
retries/attempts	5	5	5	3 (이미지 업로드는 재시도를 적게하기 위해)	5	5
retries/perTryTimeout	3s	3s	3s	10s (이미지 처리 시간을 고려)	4s	3s
retries/retryOn	gateway-error, connect-failure, refused-stream, 5xx, retriable-4xx, reset	gateway-error, connect-failure, refused-stream, 5xx, retriable-4xx, reset	gateway-error, connect-failure, refused-stream, 5xx, retriable-4xx, reset	gateway-error, connect-failure, refused-stream, reset	gateway-error, connect-failure, refused-stream, 5xx, retriable-4xx, reset	gateway-error, connect-failure, refused-stream, 5xx, retriable-4xx, reset
retries/retryRemoteLocalities	true	true	true	true	true	true

Case (6) HPA

Case (7) 도커 이미지 빌드 자동화 (github workflow)

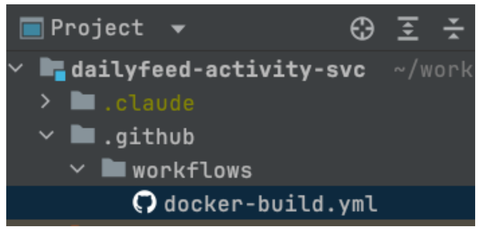
: 자동 모드

: 수동 모드

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

workflow 대상 브랜치 : main/develop/hotfix

commit & push

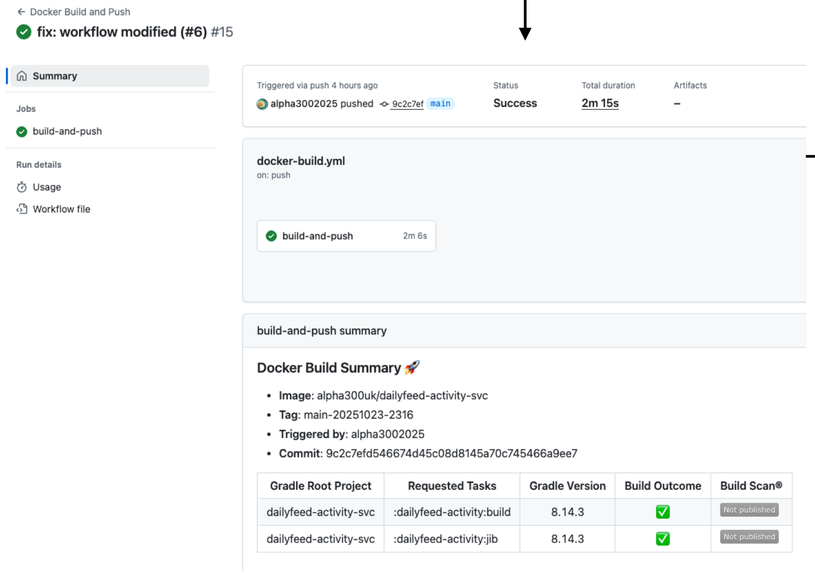


github repository 내의 workflow 발동 (trigger)

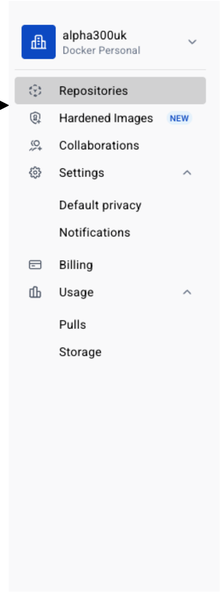
참고 : https://github.com/alpha3002025/dailyfeed-activity-svc/blob/main/.github/workflows/docker-build.yml

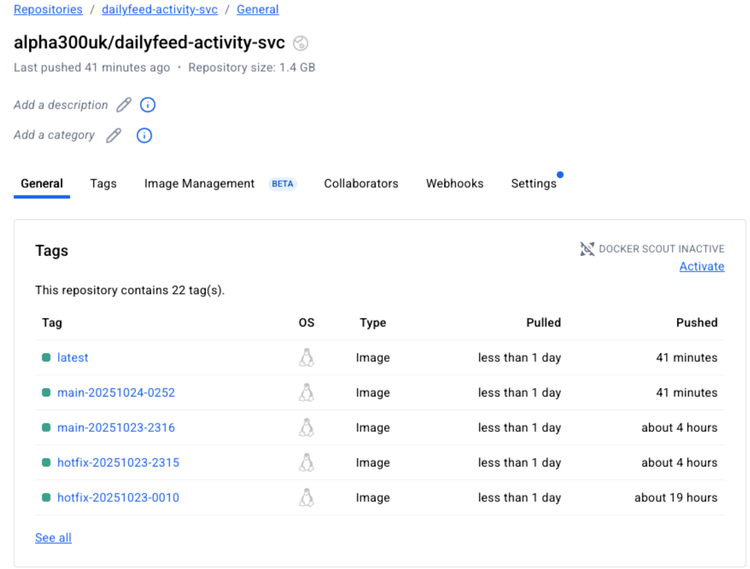
이미지 태그 형식

: {branchName}-{yyyyMMdd}-{build번호}



docker 이미지 jib build & push

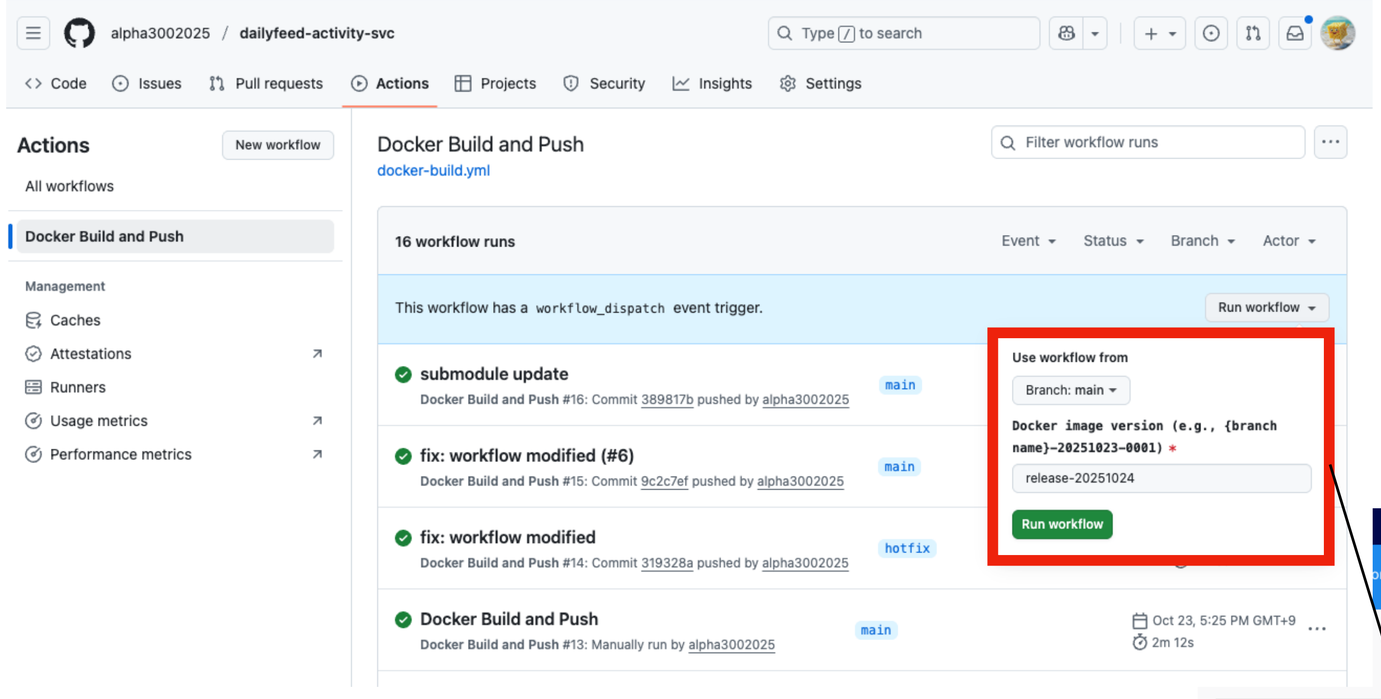




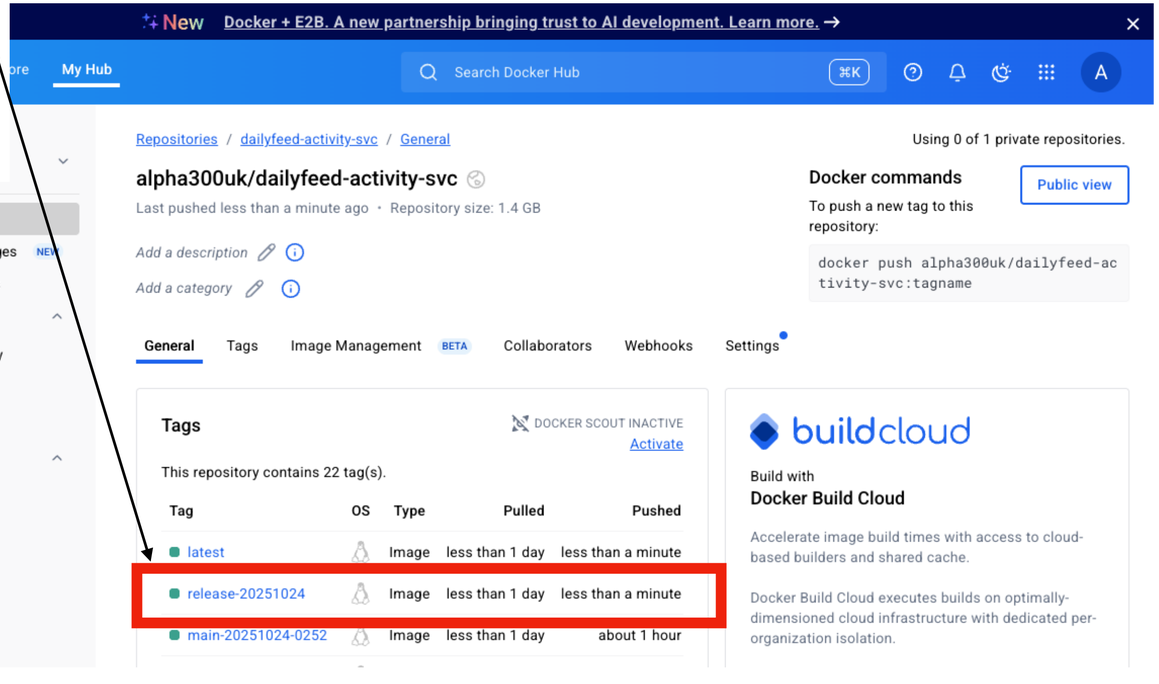
Case (6) - 도커 이미지 빌드 자동화 (github workflow) - 수동 모드

workflow 대상 브랜치 : main/develop/hotfix

github workflow 실행



docker hub (push 된 이미지 확인)



Case (8) resillience4j feign

: circuitbreaker, rate limiter

Case (9) JWT

: Key Refresh

: JWT ID, Black List

Case (10) 주요 코딩 컨벤션

: no common -> 서브모듈로 공유

: mapper

: get—OrThrow

: data 로직 공통화 자제 (과도한 리소스 사용방지)