dailyfeed backend

주요 백엔드 기능 및 설계 관련 설명

2025/11/02 (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

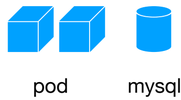
  :: 통신레벨에서는 통신만 수행하고, 익셉션은 애플리케이션 계층으로 throw

+ 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

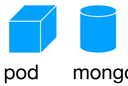
+ 프로필/썸네일 이미지 업로드/수정/삭제/조회

+ 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)

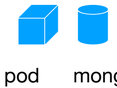
+ 댓글 많은 글 (most commented)

+ 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

git clone

- 각 서브모듈 재귀적 clone 방법
- git submodule update/pull 방법
- 커밋/업데이트 방법

새로운 환경에서 클론(clone)

새로운 환경에서 클론 (clone)

Submodule과 함께 클론 (content-svc, timeline-svc, activity-svc, search-svc, image-svc 도 모두 이와 같이 해주시면 됩니다.)

git clone **--recurse-submodules** dailyfeed-member-svc

또는 클론 후 submodule 초기화

git clone dailyfeed-member-svc

cd dailyfeed-member-svc

git submodule init

git submodule update

서브모듈 업데이트

모든 submodule 을 최신 상태로 업데이트

git submodule update —remote

특정 submodule 만 업데이트

git submodule update —remote dailyfeed-code

또는 다음과 같이 main 브랜치를 pull 하는 코드를 실행해주세요.

source git-pull-all.sh

새로운 원격 submodule repository 를 현재 repository에 submodule 로 추가

e.g. dailyfeed-redis-support

모든 submodule 을 clone 및 gitmodules 등록

git submodule add https://github.com/alpha3002025/dailfyeed-redis-support dailyfeed-redis-support

settings.gradle.kts 에는 다음 구문을 추가합니다.

include(“dailyfeed-redis-support”)

설치방법

(dailyfeed-installer)

설치방법 (local k8s)

local PC 내에 kind 클러스터기반의 kubernetes 환경을 구축하는 방식입니다.

(1) orbstack 또는 docker desktop 설치

orbstack 또는 docker desktop 을 설치합니다.

(2) dailyfeed-installer 리포지터리 clone

dailyfeed-installer github : https://github.com/alpha3002025/dailyfeed-installer

(3.1) install 스크립트 실행

source local-install-infra-and-app.sh main

또는 다음과 같이 수행합니다.

source local-install-infra-and-app.sh {이미지태그명}

(3.2) 만약 infra 만 설치하려 할 경우

cd dailyfeed-infrastructure

source install-local.sh

(3.3) 애플리케이션만 설치하려 할 경우

cd dailyfeed-app-helm

source install-local.sh {이미지 태그명}

삭제 (local k8s)

(1) dailyfeed-infrastructure

cd dailyfeed-infrastructure

source uninstall-all-infra.sh

설치방법 (local was)

local PC 내에 infra(mysql,redis,kafka,mongodb) 를 docker-compose 로 설치한 후 개별 서비스들은 WAS 로 각각 구동할때의 실행방법입니다.

(1) orbstack 또는 docker desktop 설치

orbstack 또는 docker desktop 을 설치합니다.

(2) dailyfeed-installer 리포지터리 clone

dailyfeed-installer github : https://github.com/alpha3002025/dailyfeed-installer

(3) infra (mysql,redis,kafka,mongodb) 설치

cd dailyfeed-infrastructure/docker/mysql-mongodb-redis

docker-compose up -d

(4) 각각의 개별 서비스에서는 다음의 명령을 수행하시면 됩니다.

e.g. dailyfeed-member-svc

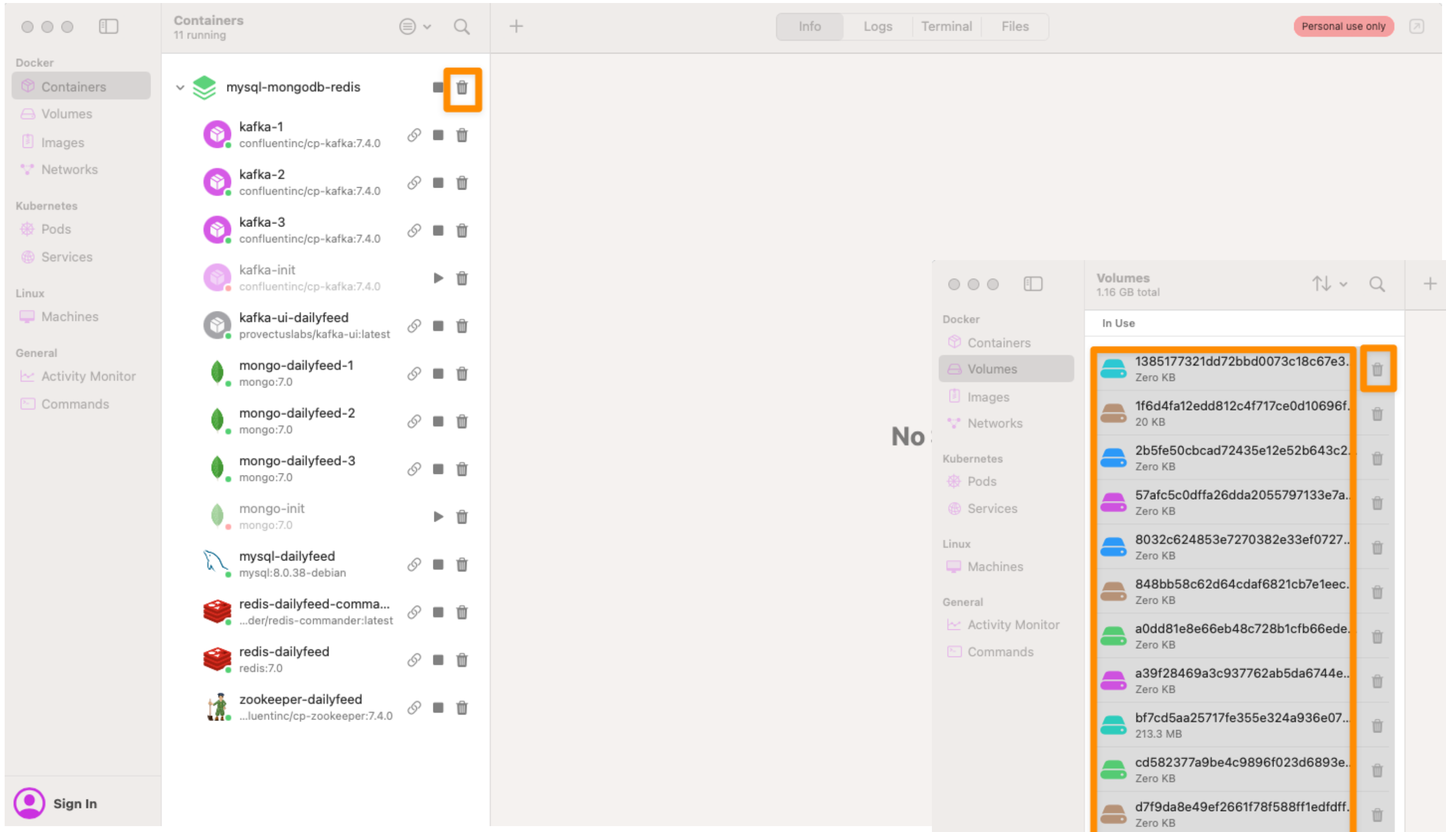
cd dailyfeed-member-svc

source git-pull-all.sh

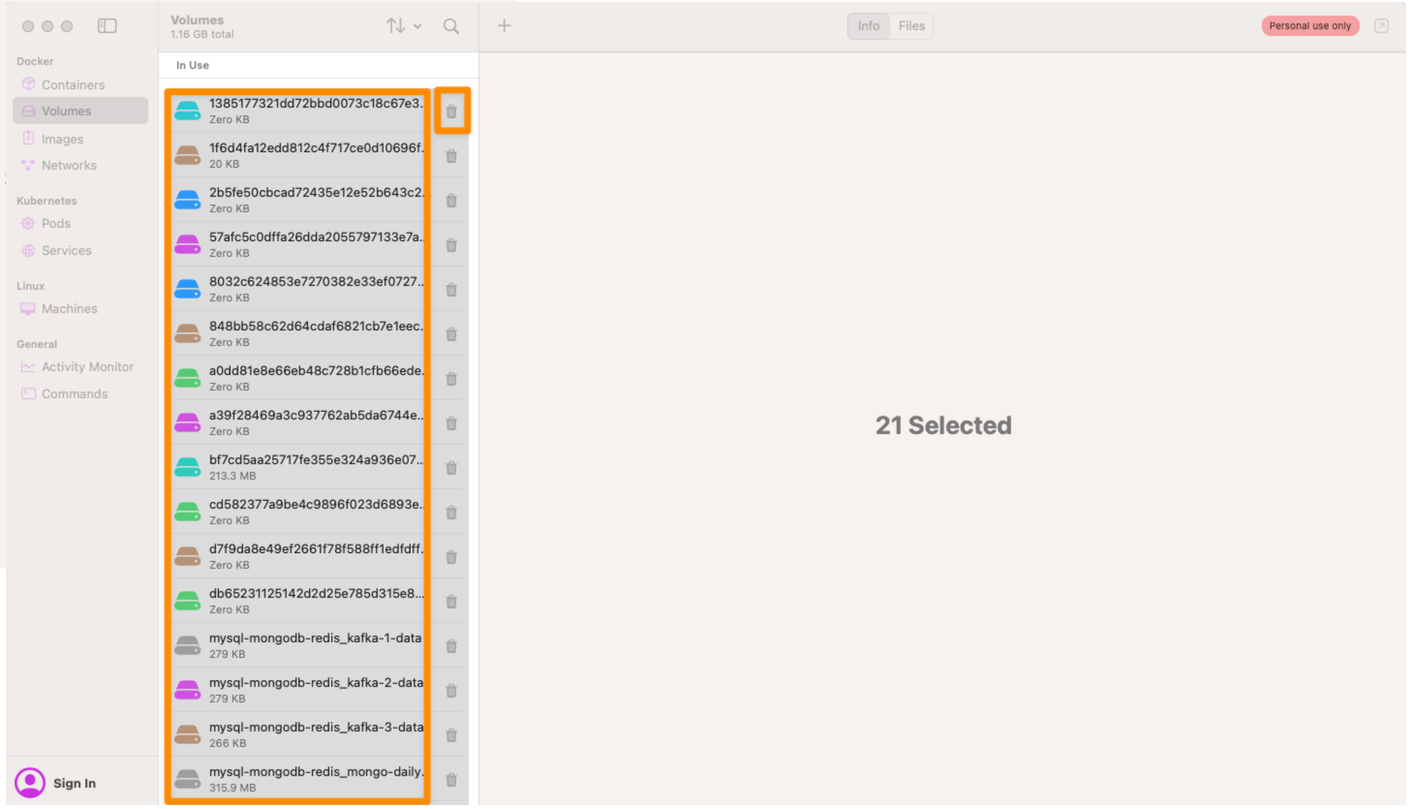
source run-local-was.sh

삭제 (local was)

orbstack 에서 컨테이너, volume 삭제



container 삭제



volume 삭제

cmd + A 를 통해 모든 볼륨 선택 후 휴지통 버튼을 클릭해서 모든 불륨을 삭제

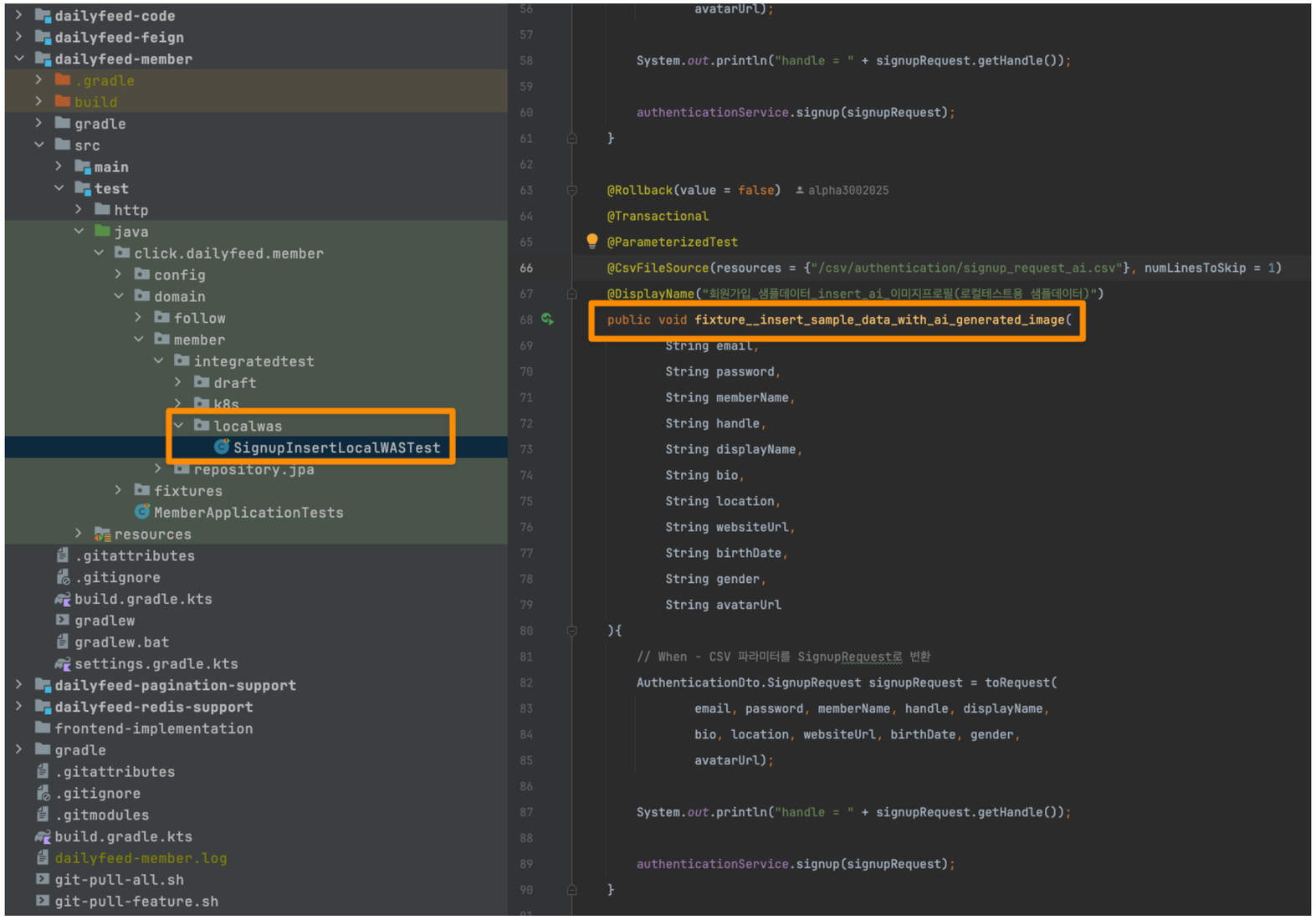
테스트 회원 데이터 insert

테스트 회원 데이터 insert

회원을 50 명 정도를 직접 사람이 일일이 가입해서 follow/unfollow 하는 것을 테스트하기는 쉽지 않습니다.

이런 이유로 Test 코드로 테스트 데이터를 insert 하는 코드를 만들어두었습니다.

(1) local-was 프로필 (로컬에서 docker-compose 로 인프라 설치한 곳에서 개별 was 들을 구동시)



src/test/java/click.dailyfeed.member/domain/member/integratedtest/localwas/SignupInsertLocalWASTest.java 를 선택합니다.

테스트 클래스 내에 다음 테스트 메서드를 실행합니다.

fixture__insert_sample_data_with_ai_generated_image()

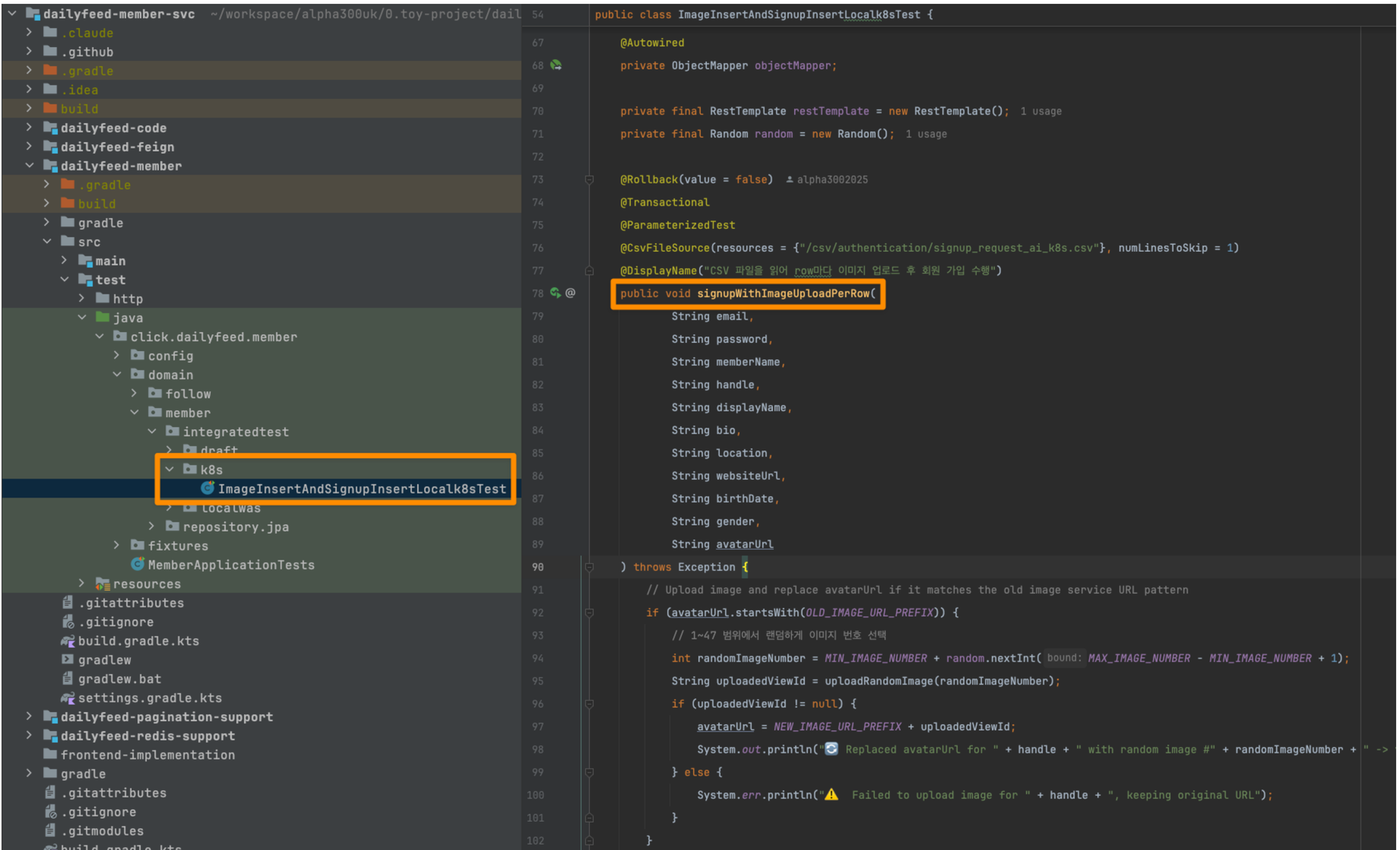
이렇게 하면 docker-compose 로 설치한 infra (mysql) 에 회원 데이터들이 insert 됩니다.

테스트 회원 데이터 insert

회원을 50 명 정도를 직접 사람이 일일이 가입해서 follow/unfollow 하는 것을 테스트하기는 쉽지 않습니다.

이런 이유로 Test 코드로 테스트 데이터를 insert 하는 코드를 만들어두었습니다.

(2) local-k8s 프로필 (로컬에서 kind 클러스터로 인프라(mysql,redis,kafka) 설치한 환경에서 deployment 로 각 리소스들을 구동했을때)



src/test/java/click.dailyfeed.member/domain/member/integratedtest/k8s/ImageInsertAndSignupInsertLocalk8sTest.java 를 선택합니다.

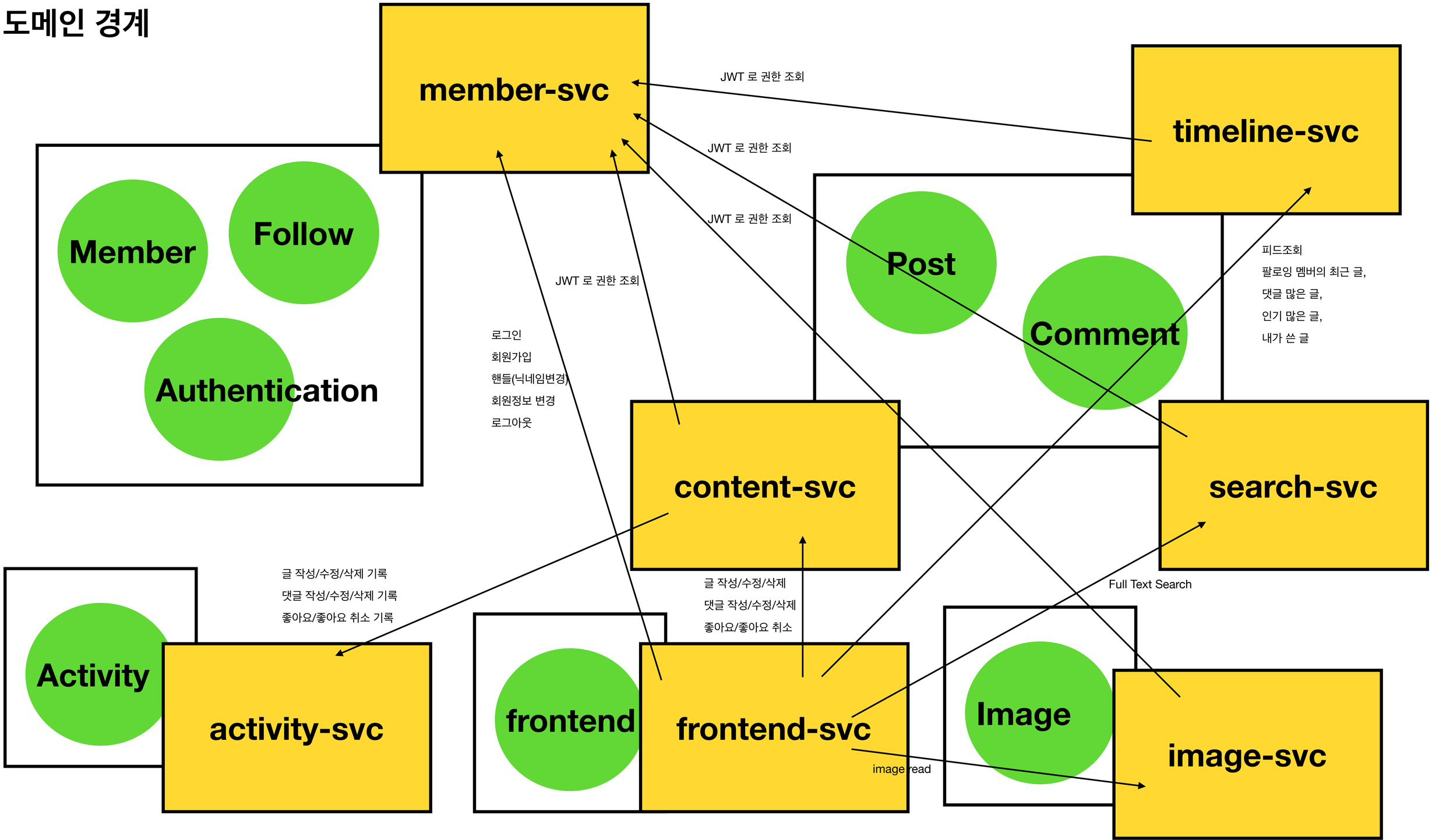
테스트 클래스 내에 다음 테스트 메서드를 실행합니다.

signupWithImageUploadPerRow()

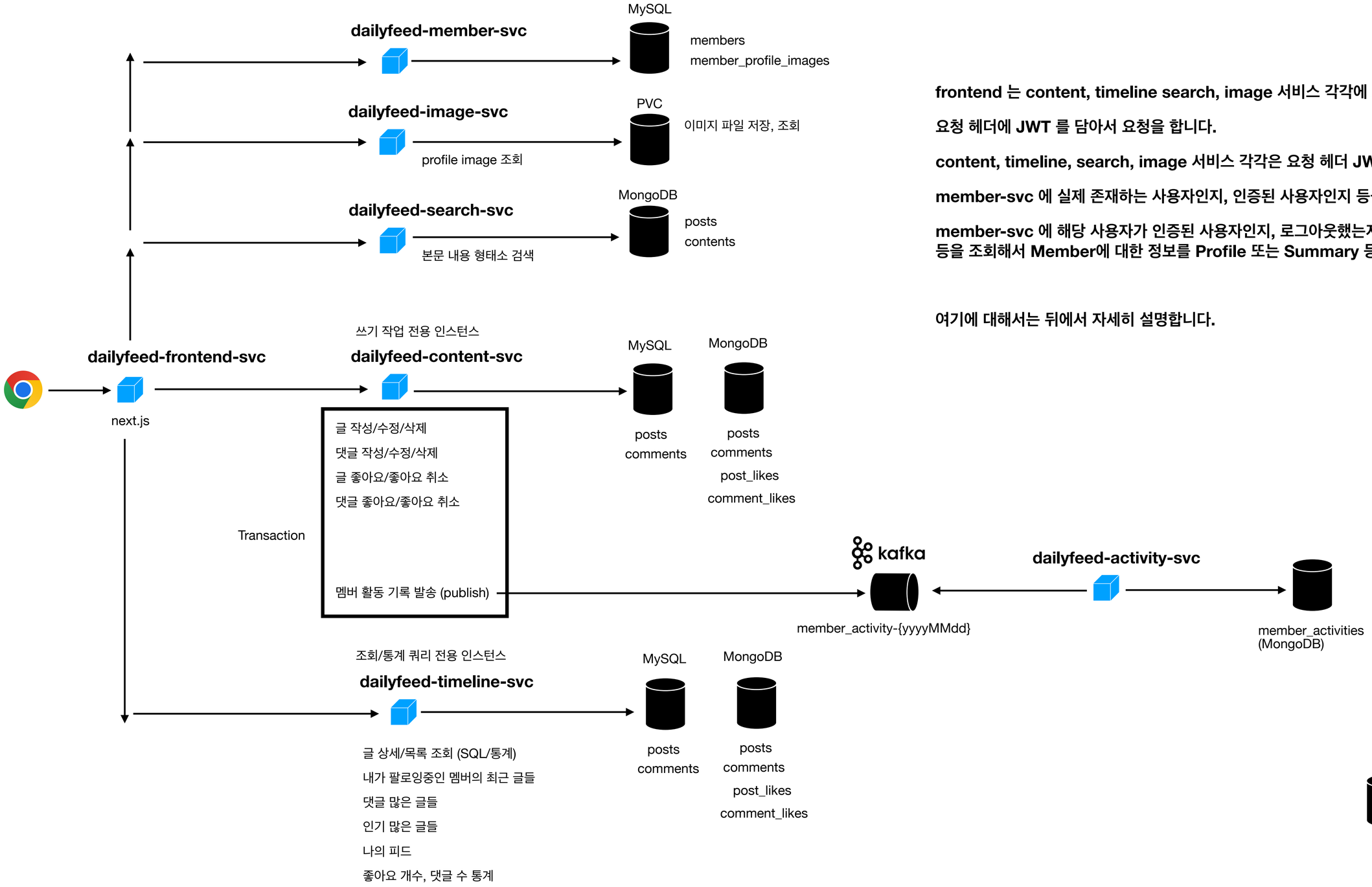
이렇게 하면 k8s 클러스터 내의 infra (mysql) 에 회원 데이터들이 insert 됩니다.

전반적인 흐름

도메인 경계


member-svc
timeline-svc
Member
Follow
Authentication
Post
Comment
content-svc
search-svc
Activity
activity-svc
frontend
frontend-svc
Image
image-svc
JWT 로 권한 조회
JWT 로 권한 조회
JWT 로 권한 조회
JWT 로 권한 조회
로그인
회원가입
핸들(닉네임변경)
회원정보 변경
로그아웃
피드조회
팔로잉 멤버의 최근 글,
댓글 많은 글,
인기 많은 글,
내가 쓴 글
글 작성/수정/삭제 기록
댓글 작성/수정/삭제 기록
좋아요/좋아요 취소 기록
글 작성/수정/삭제
댓글 작성/수정/삭제
좋아요/좋아요 취소
Full Text Search
image read

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.

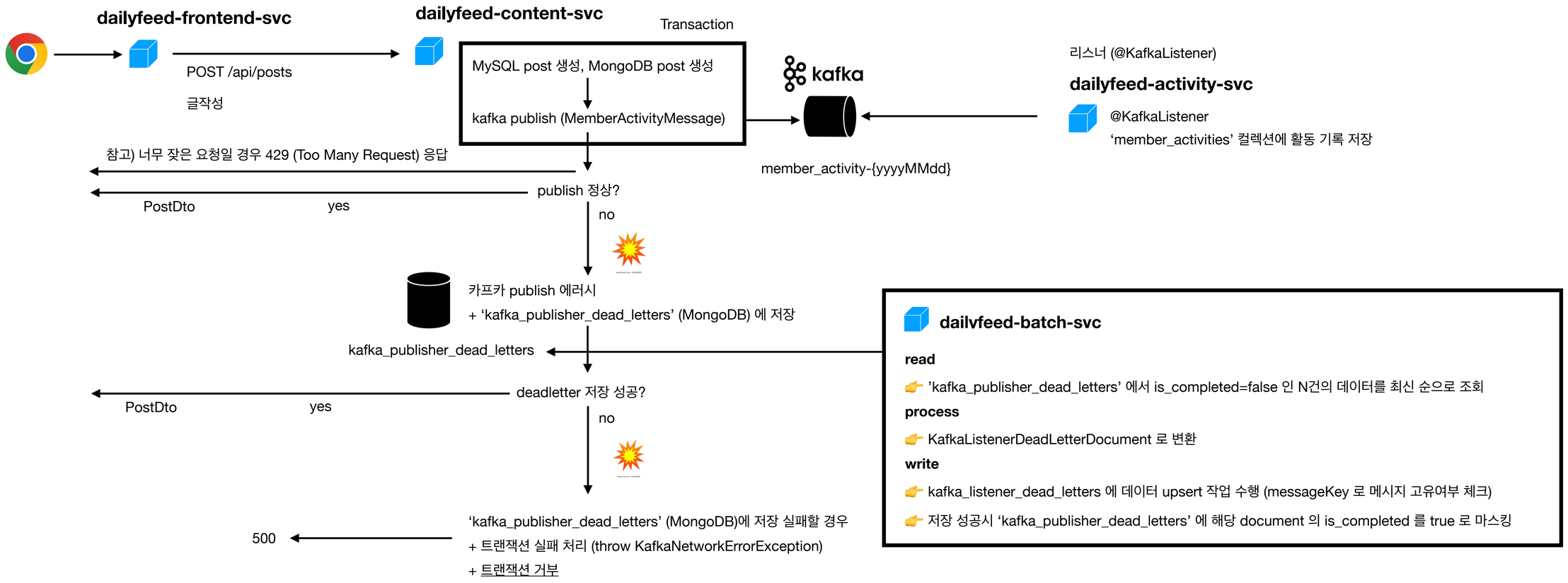
+ 카프카 운영시 카프카 증설 및 업그레이드 작업으로 인해 카프카 업그레이드 이슈 등이 있을 때는 세컨더리 저장소를 통해 Fail Over를 하는 것도 중요한 요소 중 하나입니다.

+ 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

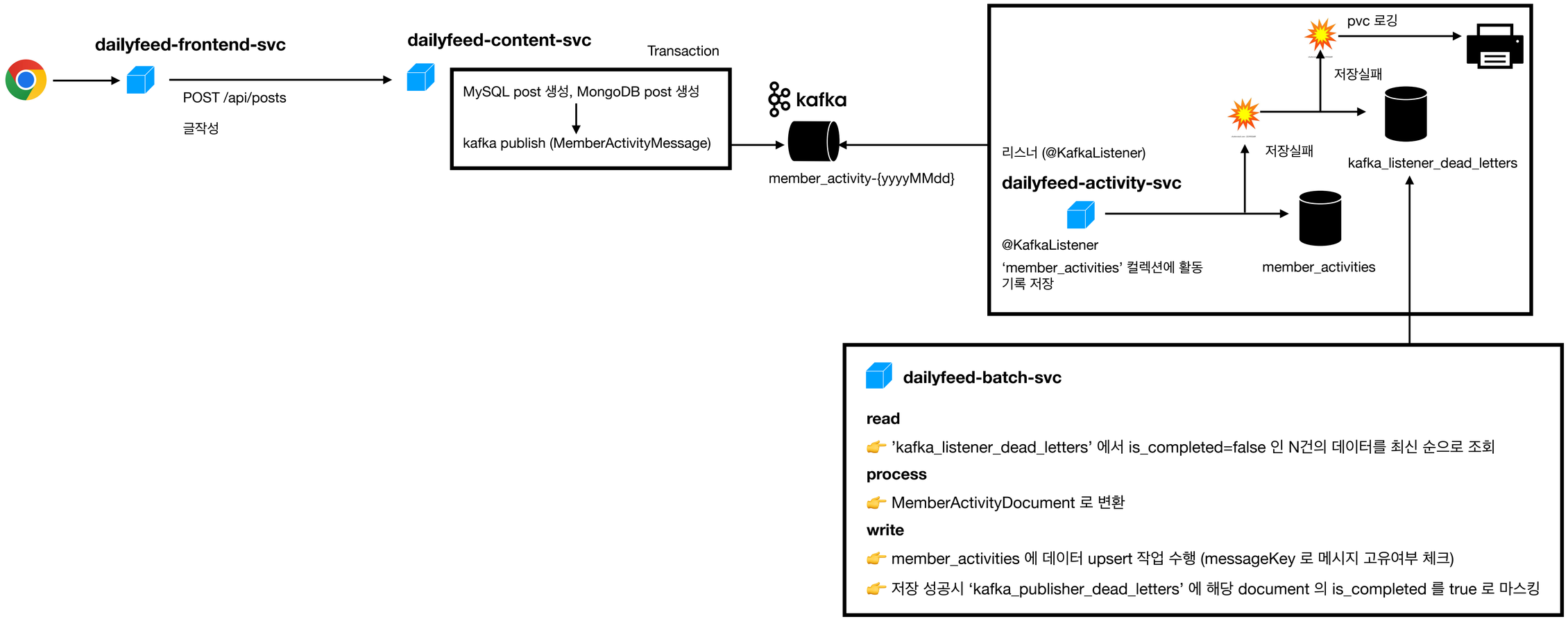
Case (1) Kafka : publisher 측에서의 통신에러 처리



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



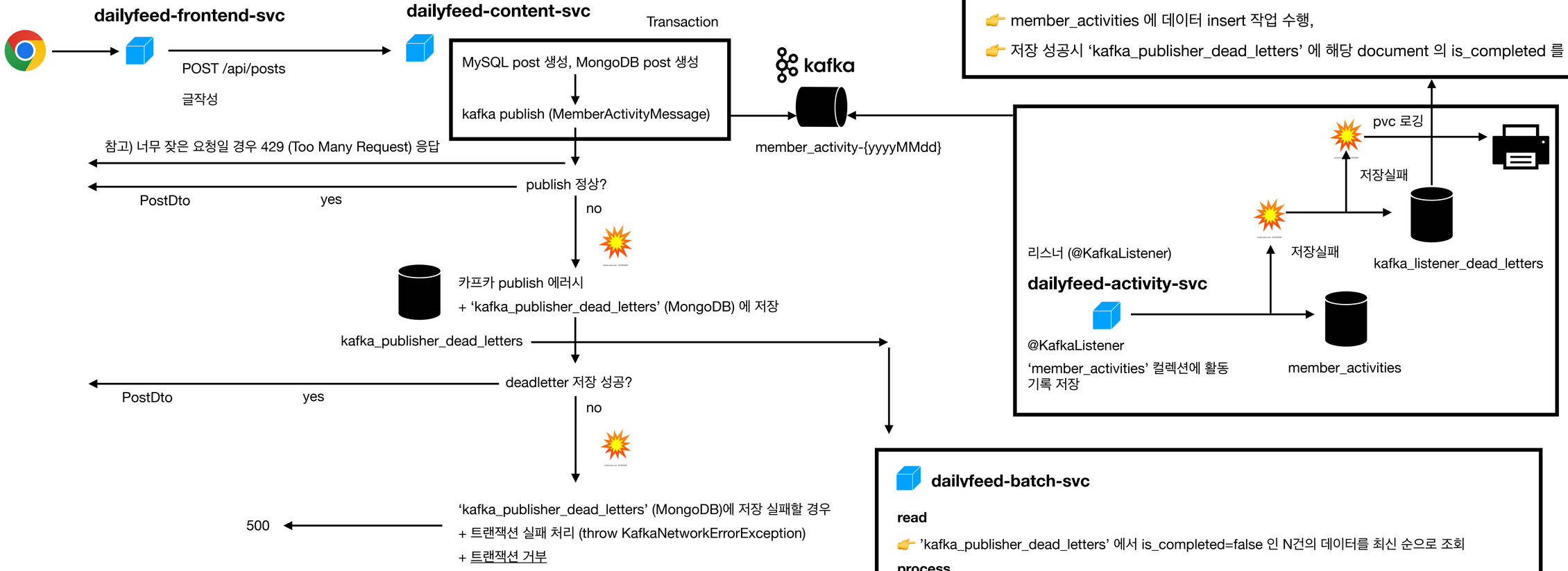
e.g.

= 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.



dailyfeed-batch-svc

read

👉 'kafka_listener_dead_letters' 에서 is_completed=false 인 N건의 데이터를 최신 순으로 조회

process

👉 MemberActivityDocument 로 변환

write

👉 member_activities 에 데이터 insert 작업 수행,

👉 저장 성공시 'kafka_publisher_dead_letters' 에 해당 document 의 is_completed 를 true 로 마스킹

dailyfeed-batch-svc

read

👉 'kafka_publisher_dead_letters' 에서 is_completed=false 인 N건의 데이터를 최신 순으로 조회

process

👉 KafkaListenerDeadLetterDocument 로 변환

write

👉 kafka_listener_dead_letters 에 데이터 upsert 작업 수행 (messageKey 로 메시지 고유여부 체크)

👉 저장 성공시 'kafka_publisher_dead_letters' 에 해당 document 의 is_completed 를 true 로 마스킹

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}”

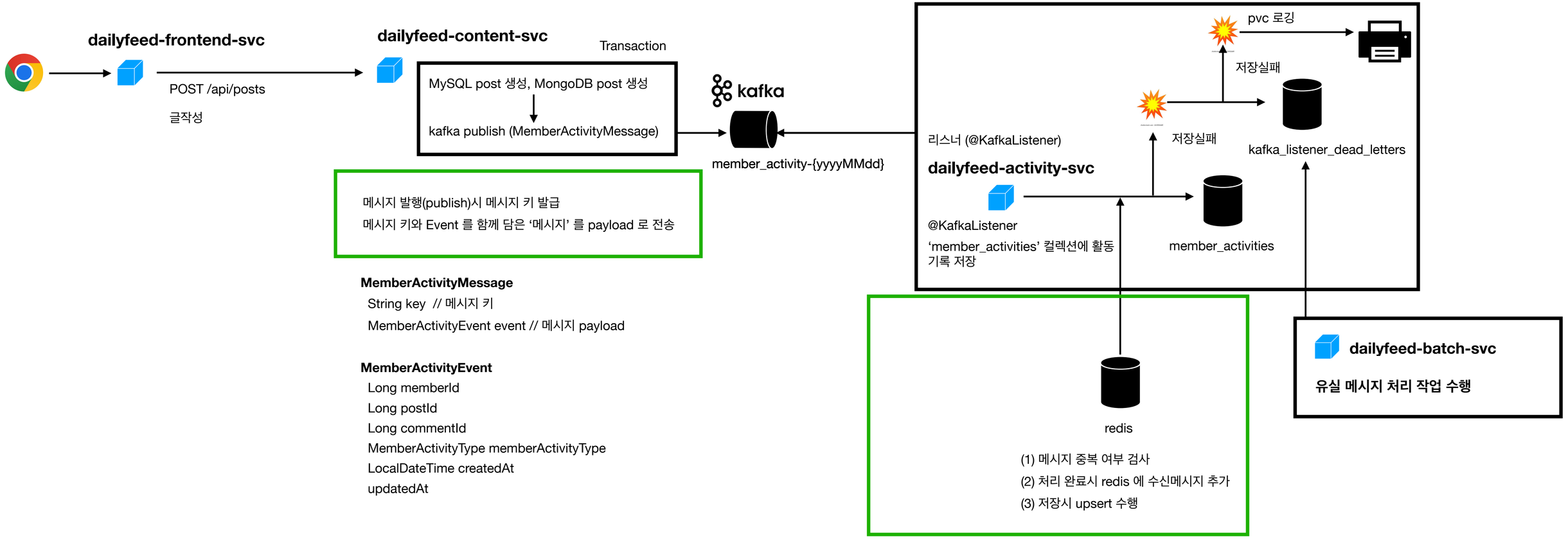
LIKE_COMMENT, LIKE_COMMENT_CANCEL

“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

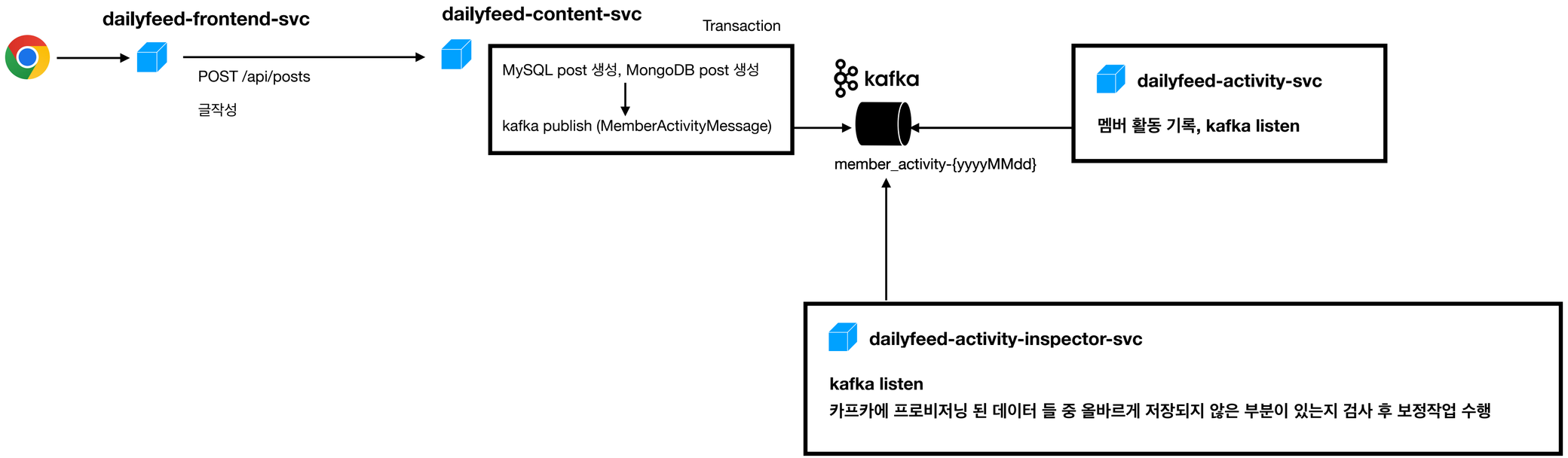
: 날짜별 토픽 ({topicName}-yyyyMMdd)

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)



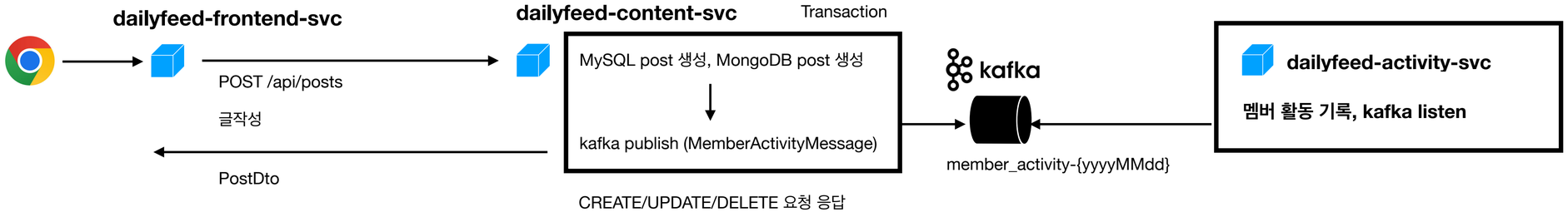
Case (4) 스케일아웃 그룹 분류

: Read 스케일아웃 그룹

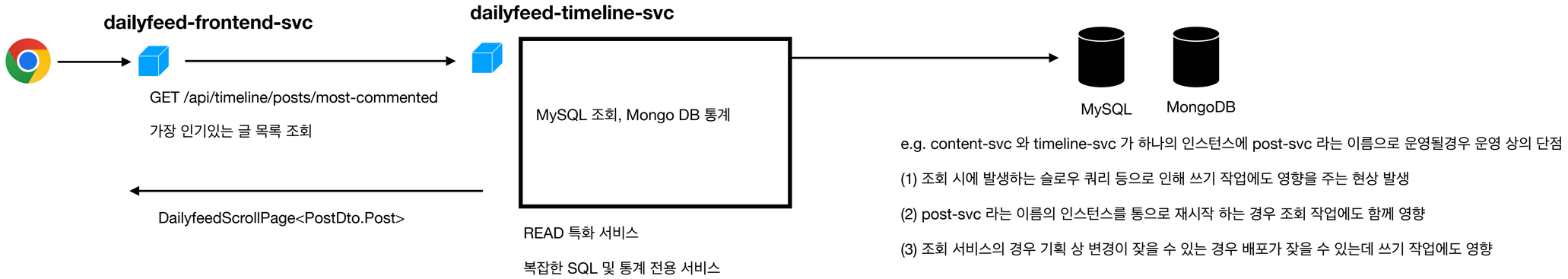
: Create/Update/Delete 스케일아웃 그룹

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?

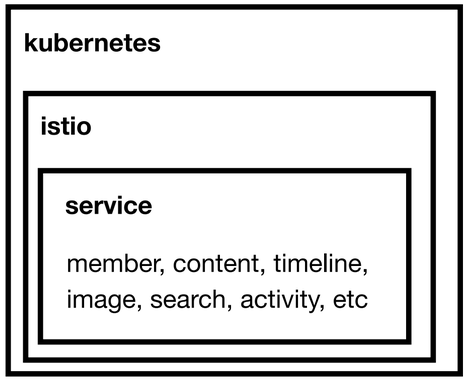
- 혹시라도 istio가 무엇인지 모르는 분들을 위한 설명 섹션입니다.

Istio 란?

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

istio 란?

참고 : https://istio.io



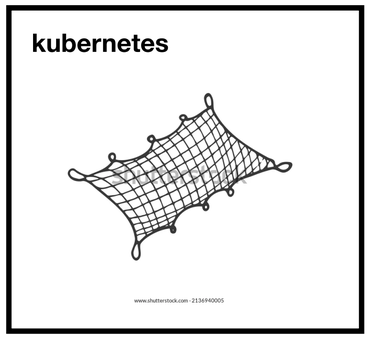
kubernetes 또는 여러 컨테이너 오케스트레이션 등에서 Envoy Proxy 역할을 수행하는 sidecar 를 통해

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.

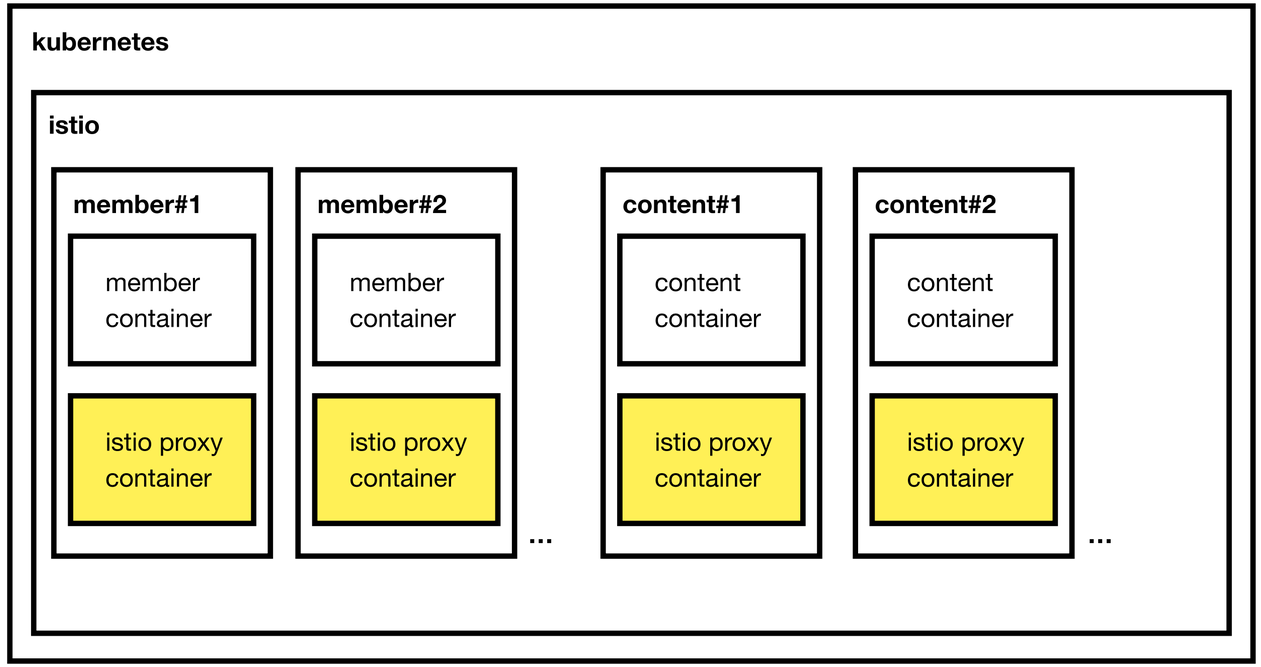
Service Mesh 는 이런 기능에서부터 여러가지 기능들을 제공합니다.

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.

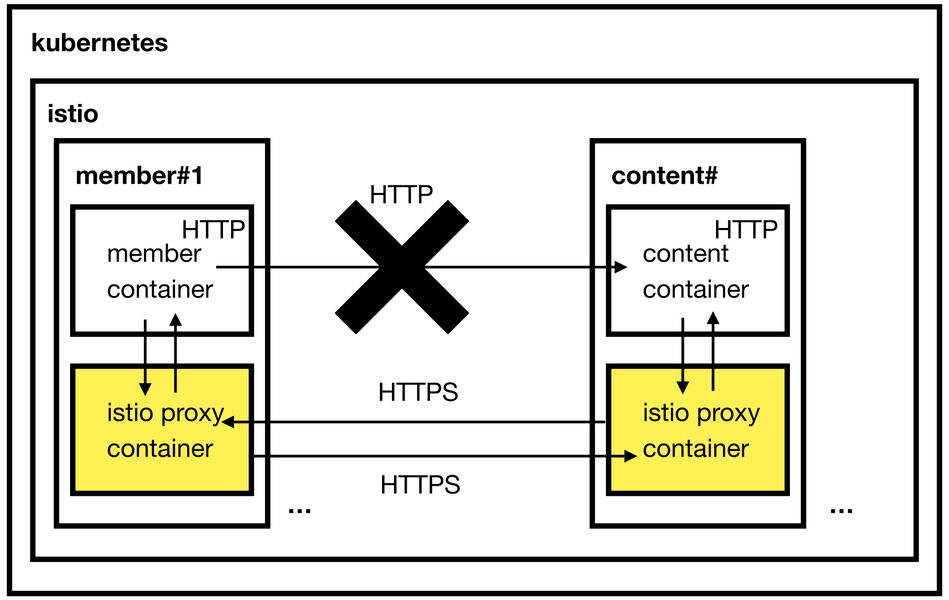
istio 는 이렇게 각각의 pod 내에 istio가 관리하는 container 를 주입해서 각각의 pod 에 대한 네트워크 제어를 수행합니다. 이렇게 주입된 'istio proxy container' 들은 각각 자신의 파드를 대표해서 다른 서비스와 통신을 하는 '외교관', 'proxy' 같은 역할을 수행합니다.

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)

istio 에서는 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.

이 설정은 PeerAuthentication 이라는 리소스를 통해 정의되며, 더 세부적인 AuthorizationPolicy 등을 적용하는 것 역시 가능합니다.

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

상황에 따라 pod 와 다른 pod 는 물리적으로 데이터 센터 중 다른 서버 건물 등에 배치될 수도 있습니다. 이 경우 외부와 통신할 때 HTTPS 가 아닌 HTTP 로 통신을 수행할 경우 통신 내용이 평문으로 전달되면서 탈취될 우려가 있는데, Istio 를 사용한다면 혹시라도 모를 1%의 유출 가능성 까지도 모두 암호화를 적용할 수 있다는 점은 장점입니다.

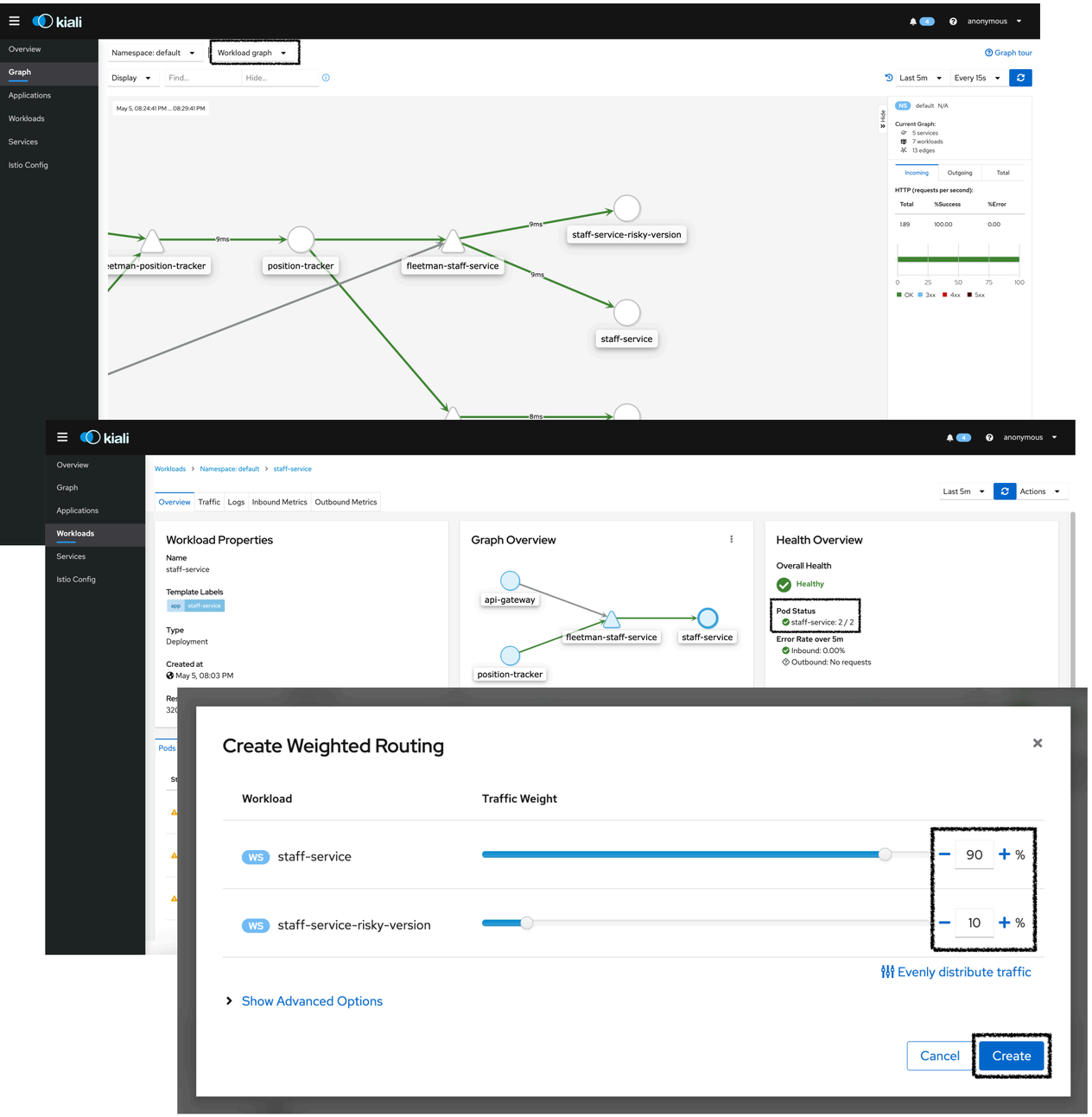
kubernetes 를 도입한다고 해도, istio 등에 대해서 잘 모른채로 부딪히기 식으로만 접근하면

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 (5) - DestinationRule

Destination Rule 은 connectionPool, outlierDetection 을 정의함

member

connectionPool

maxConnections: 100, connectionTimeout: 3s, tcpKeepalive (time = 7200s, interval = 75s, probes = 9), http1MaxPendingRequests:100, http2MaxRequests:500, maxRequestsPerConnection:10, maxRetries:3, idleTimeout:30s, h2UpgradePolicy: UPGRADE

outlierDetection

consecutive5xxErrors: 5, consecutiveGatewayErrors: 3, interval: 1m, baseEjectionTime: 5m, maxEjectionPercent: 50, minHealthPercent: 30, splitExternalLocalOriginErrors: true

content

connectionPool

maxConnections: 100, connectionTimeout: 3s, tcpKeepalive (time = 7200s, interval = 75s, probes = 9), http1MaxPendingRequests:100, http2MaxRequests:500, maxRequestsPerConnection:10, maxRetries:3, idleTimeout:30s, h2UpgradePolicy: UPGRADE

outlierDetection

consecutive5xxErrors: 5, consecutiveGatewayErrors: 3, interval: 1m, baseEjectionTime: 5m, maxEjectionPercent: 50, minHealthPercent: 30, splitExternalLocalOriginErrors: true

timeline

connectionPool

maxConnections: 100, connectionTimeout: 3s, tcpKeepalive (time = 7200s, interval = 75s, probes = 9), http1MaxPendingRequests:100, http2MaxRequests:500, maxRequestsPerConnection:10, maxRetries:3, idleTimeout:30s, h2UpgradePolicy: UPGRADE

outlierDetection

consecutive5xxErrors: 5, consecutiveGatewayErrors: 3, interval: 1m, baseEjectionTime: 5m, maxEjectionPercent: 50, minHealthPercent: 30, splitExternalLocalOriginErrors: true

search (검색 특성상 pending 에 대해 허용치를 조금 크게 지정)

connectionPool

maxConnections: 120, connectionTimeout: 3s, tcpKeepalive (time = 7200s, interval = 75s, probes = 9), http1MaxPendingRequests:120, http2MaxRequests:550, maxRequestsPerConnection:10, maxRetries:3, idleTimeout:45s, h2UpgradePolicy: UPGRADE

outlierDetection

consecutive5xxErrors: 5, consecutiveGatewayErrors: 3, interval: 1m, baseEjectionTime: 5m, maxEjectionPercent: 50, minHealthPercent: 30, splitExternalLocalOriginErrors: true

image (이미지 특성상 pending 에 대해 허용치를 크게 지정)

connectionPool

maxConnections: 150, connectionTimeout: 3s, tcpKeepalive (time = 7200s, interval = 75s, probes = 9), http1MaxPendingRequests:150, http2MaxRequests:600, maxRequestsPerConnection:5, maxRetries:2, idleTimeout:60s, h2UpgradePolicy: UPGRADE

outlierDetection

consecutive5xxErrors: 5, consecutiveGatewayErrors: 3, interval: 1m, baseEjectionTime: 5m, maxEjectionPercent: 50, minHealthPercent: 30, splitExternalLocalOriginErrors: true

activity

connectionPool

maxConnections: 100, connectionTimeout: 3s, tcpKeepalive (time = 7200s, interval = 75s, probes = 9), http1MaxPendingRequests:10, http2MaxRequests:500, maxRequestsPerConnection:10, maxRetries:3, idleTimeout:30s, h2UpgradePolicy: UPGRADE

outlierDetection

consecutive5xxErrors: 5, consecutiveGatewayErrors: 3, interval: 1m, baseEjectionTime: 5m, maxEjectionPercent: 50, minHealthPercent: 30, splitExternalLocalOriginErrors: true

Case (6) HPA

Case (6) - HPA란?

HPA 가 무엇인지 모를 수 있는 분들을 위해 간단한 설명을 정리합니다.

HPA (Horizontal Pod Authoscaler)

```
➜  dailyfeed-installer git:(main) kubectl get po -n dailyfeed
NAME                                  READY   STATUS             RESTARTS      AGE
dailyfeed-activity-69844c4499-5fktm   2/2     Running            0             4m29s
dailyfeed-activity-69844c4499-79lhg   2/2     Running            0             4m14s
dailyfeed-activity-69844c4499-gfhwh   0/2     Init:1/2           0             102s
dailyfeed-content-5f88cc6bb9-5hwx8    0/2     Pending            0             87s
dailyfeed-content-5f88cc6bb9-7km7n    2/2     Running            0             4m15s
dailyfeed-content-5f88cc6bb9-fbhkg    1/2     Running            2 (28s ago)   2m14s
dailyfeed-content-5f88cc6bb9-mwwwh    1/2     CrashLoopBackOff   2 (13s ago)   4m30s
dailyfeed-frontend-5f58b55f68-lgzr6   2/2     Running            0             4m28s
dailyfeed-frontend-5f58b55f68-rck8d   2/2     Running            0             4m28s
dailyfeed-image-79c6459d4b-fgpqd      2/2     Running            0             4m28s
dailyfeed-image-79c6459d4b-qcsc6      2/2     Running            0             4m28s
dailyfeed-image-79c6459d4b-t6846      2/2     Running            0             4m28s
dailyfeed-member-7d59798c58-7qs96     2/2     Running            0             4m15s
dailyfeed-member-7d59798c58-ddqpn     1/2     CrashLoopBackOff   1 (9s ago)    2m59s
dailyfeed-member-7d59798c58-hbxqg     1/2     Running            2 (24s ago)   2m29s
dailyfeed-member-7d59798c58-qkxrk     0/2     Init:1/2           0             102s
dailyfeed-member-7d59798c58-zt2gt     2/2     Running            0             4m30s
dailyfeed-search-5d668444cd-dztlf     2/2     Running            0             4m28s
dailyfeed-search-5d668444cd-k4pct     2/2     Running            0             4m13s
dailyfeed-timeline-84c67c7f89-8x6hk   1/2     CrashLoopBackOff   1 (16s ago)   2m13s
dailyfeed-timeline-84c67c7f89-9rsrb   2/2     Running            0             4m29s
dailyfeed-timeline-84c67c7f89-nn9wx   2/2     Running            0             3m14s
dailyfeed-timeline-84c67c7f89-pj5mn   2/2     Running            0             4m29s
dailyfeed-timeline-84c67c7f89-w7tj8   0/2     Init:1/2           0             16s
```

배포되어 있는 deployment 의 replicas 의 수를 조정하려면 어떻게 해야할까요?

일반적인 방법은 deployment 를 kubectl edit 을 통해 수정하면 됩니다. 하지만, 예상하지 못한 트래픽의 급증 상황이고 현재 중요한 회의에 참석 중이어서 대응을 못해서 애플리케이션으로의 요청이 계속 실패하거나 슬로우 쿼리 등으로 인해 애플리케이션의 성능이 계속 느려진다면 어떻게 될까요?

HPA 를 적용하면 이런 문제가 해결됩니다. Horizontal Scaling 을 Auto 로 하겠다는 의미의 Horizontal Pod Autoscaler 는 Pod 의 메트릭을 검사해서 특정 수치에 도달했을때 scale out 을 진행하게 됩니다. 사람이 수동으로 진행하는 것에 비해 동적으로 대응할 수 있다는 점에서 장점이 있습니다.

metrics-server

```
➜  dailyfeed-installer git:(main) kubectl top nodes
NAME                           CPU(cores)   CPU(%)   MEMORY(bytes)   MEMORY(%)
istio-cluster-control-plane    140m         1%       819Mi           6%
istio-cluster-worker           450m         5%       4677Mi          38%
istio-cluster-worker2          260m         3%       4408Mi          36%
```

HPA 를 사용하기 위해서는 metrics-server 를 설치해야합니다.

HPA 는 pod 또는 node 의 상태 데이터 메트릭을 보고 사용자가 지정한 임계값에 따라 Auto Scaling 을 진행하기 때문입니다.

설치 후에는 kubectl top nodes, kubectl top pods -n {네임스페이스} 명령으로 관련 리소스들의 상태를 확인할 수 있습니다.

Case (6) - 현재 HPA 설정

현재 지정한 HPA 설정을 설명합니다.

서비스	Min/Max Replicas	CPU 임계값	메모리 임계값	Scale Up 특징	비고
activity-svc	2-10	70%	90%	표준 (60초)	표준 설정
content-svc	2-10	70%	90%	표준 (60초)	표준 설정
frontend-svc	**2-15**	**60%**	**70%**	**즉시 (0초)**	사용자 대면, 가장 공격적
image-svc	**2-15**	**65%**	**90%**	표준 (60초)	CPU/메모리 집약적
member-svc	2-12	70%	90%	표준 (60초)	인증 가용성 중요
search-svc	2-12	**65%**	**90%**	표준 (60초)	빠른 응답 필요
timeline-svc	2-10	70%	90%	표준 (60초)	표준 설정

(1) 안정성 우선

모든 서비스에 대해 **최소 2개**의 replica 부여

scale down 시 5분의 안정화 윈도우(stabilization window)를 부여해 급격한 scale down 방지

(2) 서비스 특성 별 세분화

frontend-svc : 사용자 경험을 위해 빠른 스케일 업 (스케일업 안정화 윈도우 : 0초)

image-svc, search-svc : CPU 워크로드 임계값을 낮은 값을 부여

(3) 점진적 스케일링

1분 마다 스케일업 검사, 1분마다 임계값에 도달시 2개, 또는 5개의 Pod(frontend)를 스케일업

1분마다 스케일다운 검사, 1분마다 임계값에 도달시 2개, 50% 축소

(4) 리소스 requests, limits 세부 정의

리소스마다 top 명령을 통해 파악한 최소,최대의 범위를 대략적으로 파악 후 적용 (가장 오래 걸렸던 노가다 작업)

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

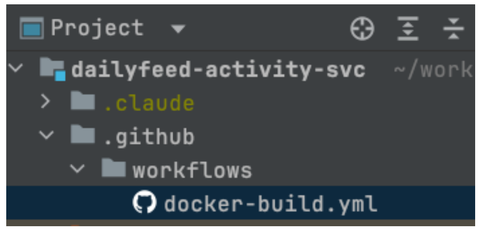
: 자동 모드

: 수동 모드

Case (7) - 도커 이미지 빌드 자동화 (github workflow) - 자동 모드

workflow 대상 브랜치 : main/develop/hotfix

commit & push

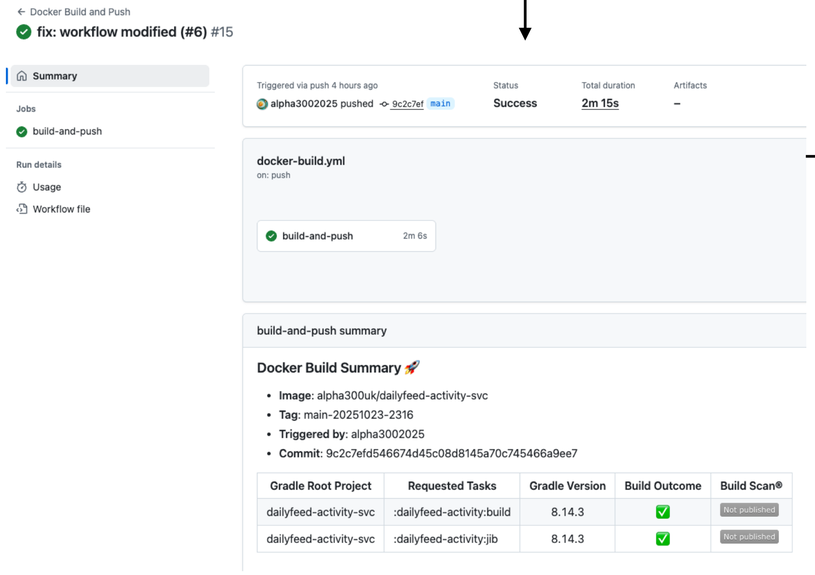


github repository 내의 workflow 발동 (trigger)

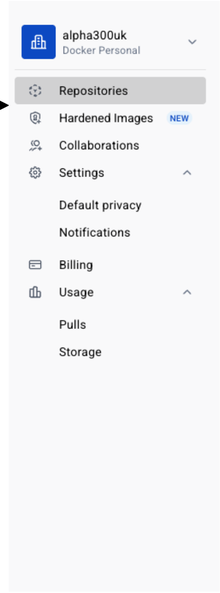
참고 : https://github.com/alpha3002025/dailyfeed-activity-svc/blob/main/.github/workflows/docker-build.yml

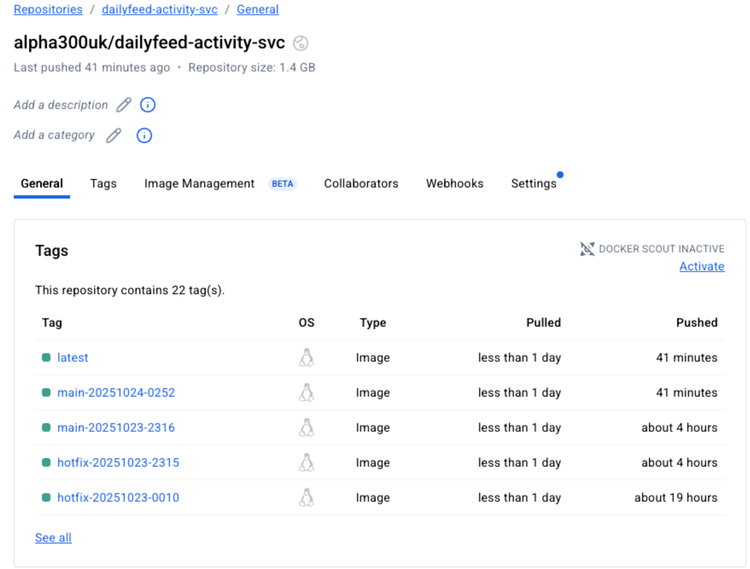
이미지 태그 형식

: {branchName}-{yyyyMMdd}-{build번호}



docker 이미지 jib build & push

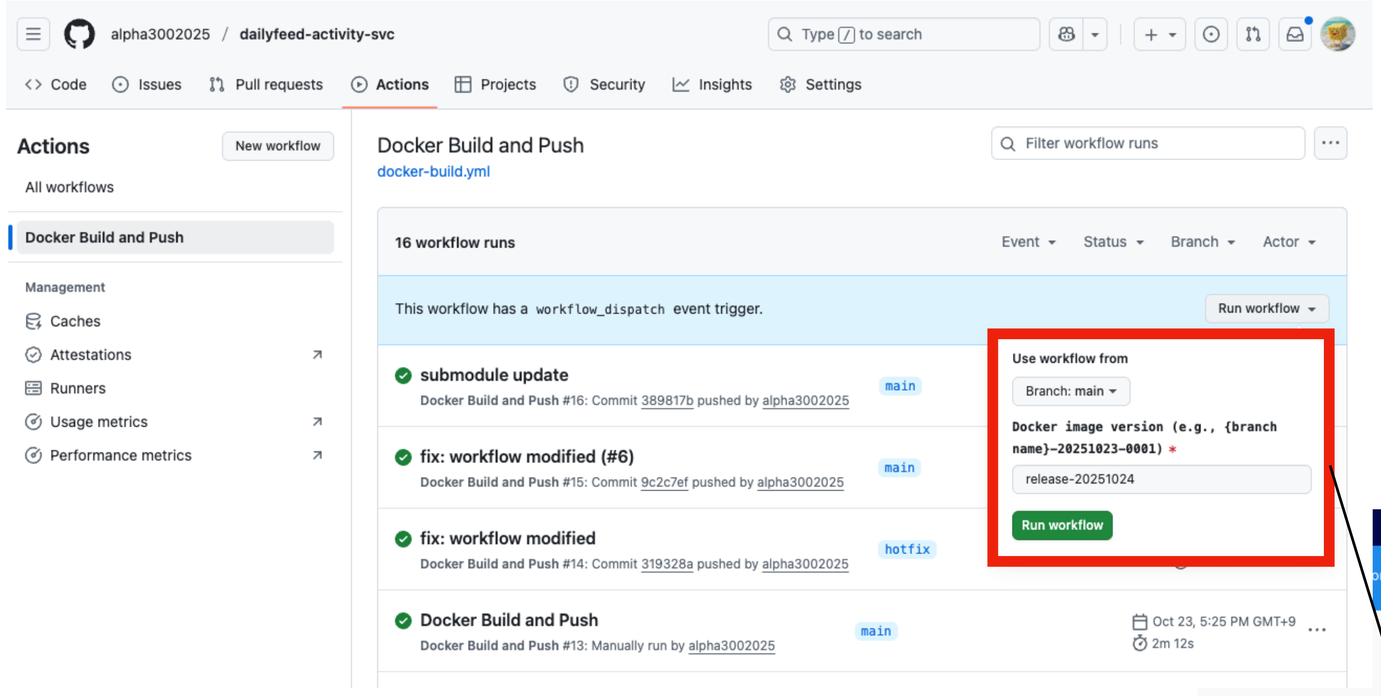




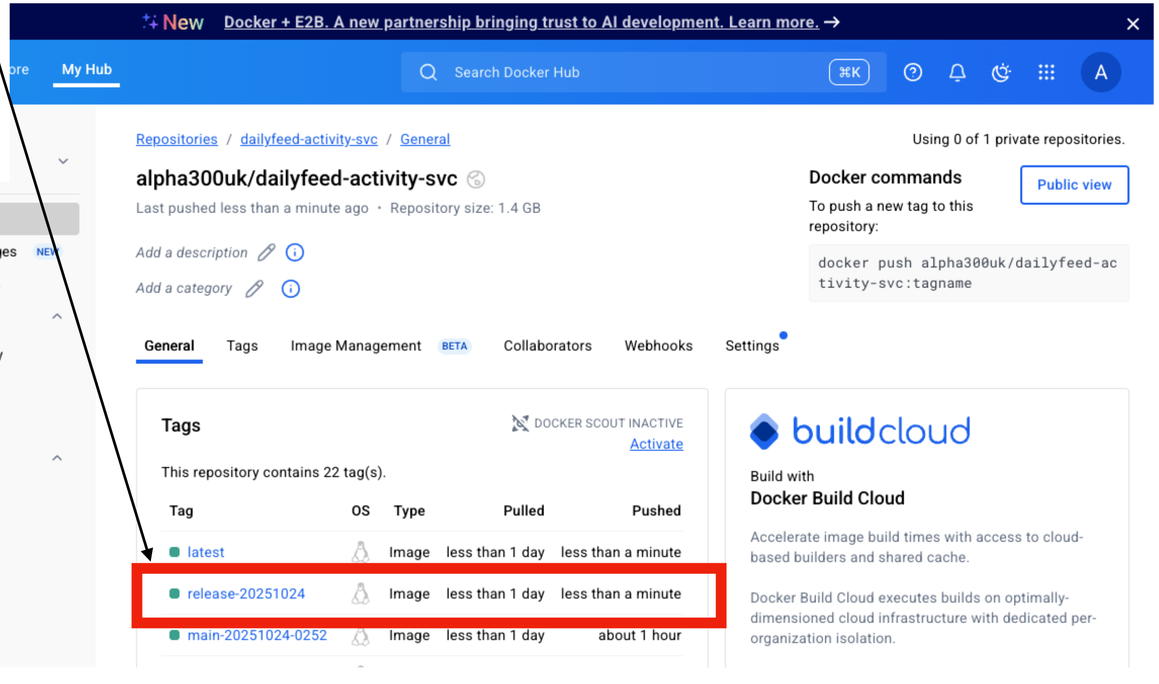
Case (7) - 도커 이미지 빌드 자동화 (github workflow) - 수동 모드

workflow 대상 브랜치 : main/develop/hotfix

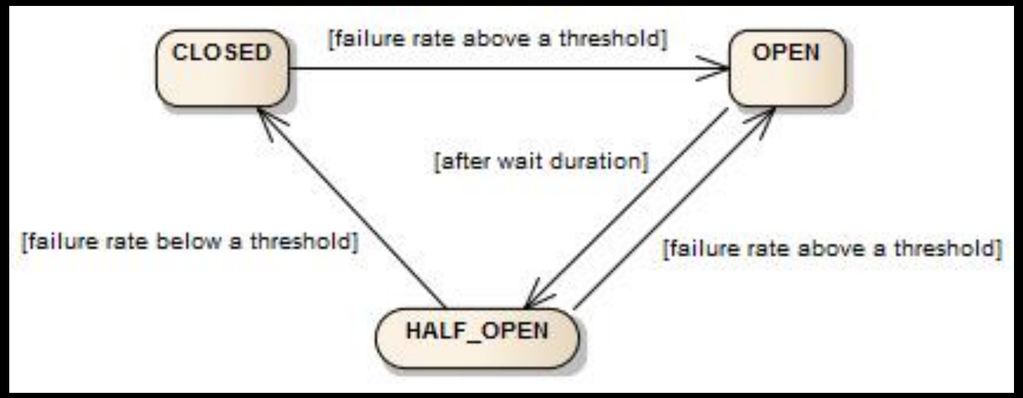
github workflow 실행



docker hub (push 된 이미지 확인)



Case (8) resillience4j feign
: circuitbreaker, rate limiter



Case (8) - rate limiter

rate limiter : 현재 서비스에서 다른 마이크로서비스로 Feign 을 통해 요청을 호출하는 outbound 요청의 빈도(rate)를 제한하는 기능

- self-protection : 다른 서비스에 과부하를 주지 않도록 보호해줄수 있습니다.
- 연쇄 장애 방지 : 현재 서비스에서의 과도한 요청으로 인해 전체 시스템에 영향을 주지 않도록 합니다.
- 호출 주기 안정화 : 특정 서비스로의 급격한 트래픽 증가시 제한을 걸수 있습니다.
- 일정한 양의 최소/최대 호출 가능 rate 를 지정하면, 현재 시스템 내에서 해당되는 대상 서비스 역시 어느 정도의 트래픽을 허용치로 둘지 명확해집니다.

설정의 정도에 따라 default, critical, conservative 로 분류해두었고, 각 서비스별로 다음과 같이 지정했습니다.

critical : timelineService, memberService

default : contentService, activityService

conservative : imageService

다음은 default 에 대한 설정입니다.

limit-refresh-period : 1s

- 제한 갱신 주기
- 1초로 지정했습니다.
- 1초마다 기록했던 rate 가 리셋됩니다.

limit-for-period : 100

- 갱신 주기 당 허용 호출 수
- limit-refresh-period 구간 내에서 '허용할 호출 수' 를 의미합니다.
- 1초에 100번 호출 가능하다는 것을 의미합니다.

timeout-duration : 0s

- 허용량 초과시 대기시간
- 0s 는 대기하지 않고 즉시 실패하는 것을 의미합니다.

```yaml
ratelimiter:
  configs:
    # 기본 Rate Limiter 설정
    default:
      # 제한 갱신 주기: 1초마다 제한이 리셋됨
      limit-refresh-period: 1s
      # 갱신 주기당 허용 호출 수: 1초에 100번 호출 가능
      limit-for-period: 100
      # 허용량 초과 시 대기 시간: 대기하지 않고 즉시 실패
      timeout-duration: 0s

    # 중요 서비스용 (더 많은 요청 허용)
    critical:
      limit-refresh-period: 1s
      limit-for-period: 200  # 1초에 200번 호출 가능 (부하가 높을 수 있음)
      timeout-duration: 100ms  # 100ms까지 대기 후 실패

    # 보수적 Rate Limiter (외부 서비스 보호)
    conservative:
      limit-refresh-period: 1s
      limit-for-period: 50  # 1초에 50번으로 제한 (외부 서비스 부담 감소)
      timeout-duration: 0s

  instances:
    timelineService:
      base-config: critical  # Timeline은 트래픽이 많으므로 여유있게
    memberService:
      base-config: critical
    contentService:
      base-config: default
    activityService:
      base-config: default
    imageService:
      base-config: conservative  # Image 서비스는 부담이 크므로 제한
```

Case (8) - retry

retry : 현재 서비스에서 다른 마이크로서비스로 Feign 을 통해 요청을 호출시 에러 발생시 재시도 횟수, 재시도 전 timeout 정도, 어떤 Exception 에 대해 재시도할지 등에 대한 재시도 설정

- 불필요한 재시도 방지 (e.g. 4xx 등에 대해서는 즉시 실패 처리)
- 의미있는 재시도 (e.g. 네트워크 장애, 타임아웃의 경우 재시도하도록 설정)
- Backoff 전략으로 부하 분산 (백오프 : 시스템이 실패 후 재시도할 때 즉시 재시도하지 않고 일정시간 대기하는 전략을 의미합니다.)

설정의 정도에 따라 default, fast, conservative 로 분류해두었고, 각 서비스별로 다음과 같이 지정했습니다.

fast : timelineService, memberService

default : contentService, activityService

conservative : imageService

```yaml
retry:
  configs:
    # 기본 재시도 설정
    default:
      max-attempts: 3
      wait-duration: 500ms
      retry-exceptions:
        - feign.RetryableException
        - java.io.IOException
        - java.net.SocketTimeoutException
        - java.util.concurrent.TimeoutException
      ignore-exceptions:
        - feign.FeignException$BadRequest
        - feign.FeignException$Unauthorized
        - feign.FeignException$Forbidden
        - feign.FeignException$NotFound
```

다음은 default 에 대한 설정입니다.

max-attempts: 3

- 최대 3번까지 시도 (최초 1번 + 재시도 2번), 실패 시 총 3번의 호출 시도 후 최종 실패 처리

wait-duration: 500ms

- 각 재시도 사이의 대기 시간 (실패 후 재시도 하지않고 500ms 를 기다린 후 재시도)
- 장점
  - Backoff : 대상 서비스에 즉시 재시도하지 않고 일정시간 대기하는 것을 Backoff 라고 합니다.
  - 일시적 장애 회복 시간 제공 : retry 하기 전에 term 을 두어서 회복시간 부여

retry-exception:

- 어떤 exception 발생시 재시도 할지

ignore-exceptions:

- 어떤 exception 발생시 재시도를 하지 않을지

Backoff 에는 다음의 종류들이 있습니다.

Fixed Backoff (고정 백오프) : 매번 동일한 시간 대기 (e.g. 500ms → 500ms → 500ms)

Exponential Backoff (지수 백오프) : 재시도마다 대기 시간이 지수적으로 증가 (e.g. 100ms → 200ms → 400ms → 800ms)

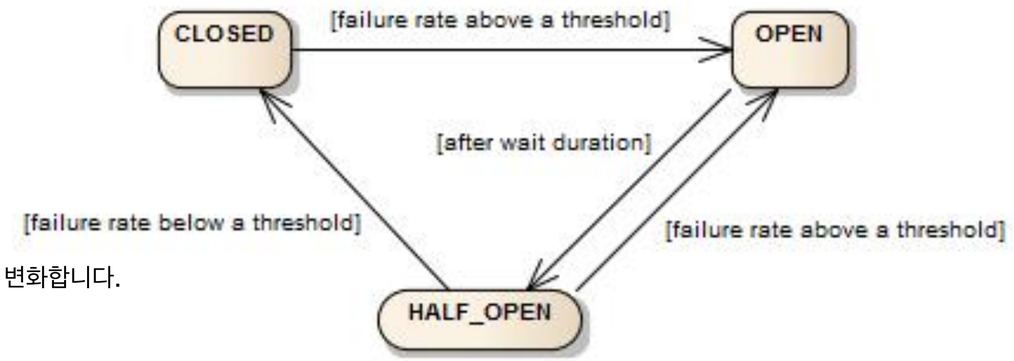
Linear Backoff (선형 백오프) : 재시도마다 대기 시간이 선형적으로 증가 (e.g. 100ms → 200ms → 300ms → 400ms)

Case (8) - circuit breaker

circuit breaker : 현재 서비스에서 다른 마이크로서비스로 Feign 을 통해 요청을 호출시 장애가 났다고 판정을 하고 해당 서비스로의 호출을 차단하거나 자동복구되는 차단을 하거나 장애 회복시 복구를 하는 등의 설정을 제공하는 기능

장점

- 연쇄 장애(Cascading Failure) 방지, 빠른 실패 (Fail Fast), 자동 복구 (Self Healing)
- 사람이 즉시 대응할 수 없는 요소에 대해 장애를 감지해서 차단을 하거나 자동 복구가 되도록 지정 가능



주요 상태

- CLOSED (닫힘 - 정상), OPEN (열림 - 차단), HALF_OPEN(반열림 - 테스트)
- HALF_OPEN 시에 서비스 회복 여부를 체크하는 네트워크 테스트가 내부적으로 수행되며, 테스트 결과에 따라 CLOSED(정상복구), OPEN(차단)으로 변화합니다.

설정의 정도에 따라 default, fast, conservative 로 분류해두었고, 각 서비스별로 다음과 같이 지정했습니다.

critical : timelineService, memberService

default : contentService, activityService

bulk : imageService

다음은 default 에 대한 설정입니다.

max-attempts: 3

- 최대 3번까지 시도 (최초 1번 + 재시도 2번), 실패 시 총 3번의 호출 시도 후 최종 실패 처리

wait-duration: 500ms

- 각 재시도 사이의 대기 시간 (실패 후 재시도 하지않고 500ms 를 기다린 후 재시도)
- 장점
  - Backoff : 대상 서비스에 즉시 재시도하지 않고 일정시간 대기하는 것을 Backoff 라고 합니다.
  - 일시적 장애 회복 시간 제공 : retry 하기 전에 term 을 두어서 회복시간 부여

retry-exception:

- 어떤 exception 발생시 재시도 할지

```yaml
retry:
  configs:
    # 기본 재시도 설정
    default:
      max-attempts: 3
      wait-duration: 500ms
      retry-exceptions:
        - feign.RetryableException
        - java.io.IOException
        - java.net.SocketTimeoutException
        - java.util.concurrent.TimeoutException
      ignore-exceptions:
        - feign.FeignException$BadRequest
        - feign.FeignException$Unauthorized
        - feign.FeignException$Forbidden
        - feign.FeignException$NotFound
```

Case (8) - circuit breaker

다음은 default 에 대한 설정입니다.

sliding-window-type: COUNT_BASED

- 슬라이딩 윈도우 타입 (COUNT_BASED(호출횟수 기반), TIME_BASED(시간 기반))

sliding-window-size: 100

- 슬라이딩 윈도우 크기. 최근 N개의 호출을 기준으로 실패율을 계산

minimum-number-of-calls: 10

- 최소 호출 횟수. 이 횟수 이상 호출되어야 circuit breaker 가 실패율을 계산

failure-rate-threshold: 50

- 실패율 임계값(%) : 이 비율 이상 실패시 Circuit OPEN

slow-call-rate-threshold: 80

- 느린 호출 기준 시간. 이 시간 이상 걸릴 경우 느린 호출로 간주

wait-duration-in-open-state: 60s

- OPEN 상태 대기 시간. Circuit 이 OPEN 된 후 HALF_OPEN 으로 전환되기까지의 대기 시간

permitted-number-of-calls-in-half-open-state: 10

- HALF_OPEN 상태에서 허용할 호출 수. 이 횟수 만큼 테스트 호출 후 CLOSED/OPEN 결정

automatic-transition-from-open-to-half-open-enable: true

- 자동 전환 활성화 : OPEN → HALF_OPEN 자동 전환 여부

```yaml
# Resilience4j 설정
resilience4j:
  circuitbreaker:
    configs:
      # 기본 설정 (일반적인 서비스)
      default:
        # 슬라이딩 윈도우 타입: COUNT_BASED(호출 횟수 기반) 또는 TIME_BASED(시간 기반)
        sliding-window-type: COUNT_BASED
        # 슬라이딩 윈도우 크기: 최근 N개의 호출을 기준으로 실패율 계산
        sliding-window-size: 100
        # 최소 호출 횟수: 이 횟수 이상 호출되어야 Circuit Breaker가 실패율을 계산
        minimum-number-of-calls: 10
        # 실패율 임계값(%): 이 비율 이상 실패 시 Circuit OPEN
        failure-rate-threshold: 50
        # 느린 호출 비율 임계값(%): 이 비율 이상이 느린 호출이면 Circuit OPEN
        slow-call-rate-threshold: 80
        # 느린 호출 기준 시간: 이 시간 이상 걸리면 느린 호출로 간주
        slow-call-duration-threshold: 5s
        # OPEN 상태 대기 시간: Circuit이 OPEN된 후 HALF_OPEN으로 전환되기까지 대기 시간
        wait-duration-in-open-state: 60s
        # HALF_OPEN 상태에서 허용할 호출 수: 이 횟수만큼 테스트 호출 후 CLOSED/OPEN 결정
        permitted-number-of-calls-in-half-open-state: 10
        # 자동 전환 활성화: OPEN → HALF_OPEN 자동 전환 여부
        automatic-transition-from-open-to-half-open-enabled: true

      # 중요 서비스용 (Timeline, Member 등 사용자 UX에 직접 영향)
      critical:
        sliding-window-type: COUNT_BASED
        sliding-window-size: 50        # 빠른 감지를 위해 작은 윈도우 사용
        minimum-number-of-calls: 5     # 5번만 호출되어도 패턴 감지
        failure-rate-threshold: 40     # 더 민감한 임계값 (40% 실패 시 OPEN)
        slow-call-rate-threshold: 70   # 느린 호출에 대해서도 민감하게 반응
        slow-call-duration-threshold: 3s  # 3초 이상이면 느린 호출
        wait-duration-in-open-state: 30s  # 빠른 복구 시도 (30초)
        permitted-number-of-calls-in-half-open-state: 5  # 적은 테스트 호출로 빠른 결정
        automatic-transition-from-open-to-half-open-enabled: true
```

Case (9) JWT

: Key Refresh

: JWT ID, Black List

Case (9) JWT - Key Refresh, Refresh Token, Blacklist

Access Token, Refresh Token

- Access Token 은 사용자가 인증되었음을 의미하는 JWT 입니다. 짧은 만료기한을 가지며, Access Token 이 만료되었을 경우 401 UnAuthorized 를 응답하며, 응답 헤더로 {‘X-Token-Refresh-Needed’: ‘true’} 를 return 합니다.
- Refresh Token 은 새로운 accessToken, refreshToken 을 부여받을때 사용하는 토큰입니다. 위와 같이 {‘X-Token-Refresh-Needed’: ‘true’} 를 응답받을때 refreshToken 으로 서버에 POST /api/token/refresh 요청을 합니다.

e.g.

- 로그인: 로그인 시 응답헤더에는 {‘Authorization’: ‘Bearer {토큰값}’} 을 담아서 응답, cookie 에는 Refresh Token 을 응답 (만료: 30Day)
- AccessToken 만료시 : 클라이언트에서는 만료된 AccessToken 에 대해 401 UnAuthorized, {‘X-Token-Refresh-Needed’: ‘true’} 를 응답받습니다.

  이후 frontend 는 member-svc 백엔드에 POST /api/token/refresh 를 호출해서 새로운 accessToken, refreshToken 을 받은 후 하던 작업을 재요청합니다.
- Server 측 동작 :

  이 때 member-svc (서버) 에서는 기존 refresh token 철회(revoke)하고, 새로운 refresh token 을 발급, 그리고 새로운 access token 역시 발급해서 access token(응답헤더), refresh token(쿠키) 을 응답하는 과정을 거칩니다.

Blacklist

- 로그아웃을 했을 때 로그아웃된 해당 토큰을 Blacklist 에 추가해서, 중복된 토큰을 사용하지 않도록 보장합니다.
- 만료된 토큰,로그아웃된 토큰을 악용하는 케이스에 대응할수 있습니다.
- 만약 Blacklist 에 해당되는 토큰으로 API 요청이 발생할 경우 서버에서는 401 UnAuthorized 를 응답하며 응답헤더로 {‘X-Relogin-Required’: ‘true’} 를 응답하게 됩니다. (feign 호출의 경우 Header/Status 전파)
- 클라이언트는 401 UnAuthorized, 응답헤더 {‘X-Relogin-Required’: ‘true’} 를 응답받으면 sessionStorage, localStorage 를 무효화하고, /login 페이지로 사용자를 리다이렉트 시킵니다.

JWT Key, Key Refresh

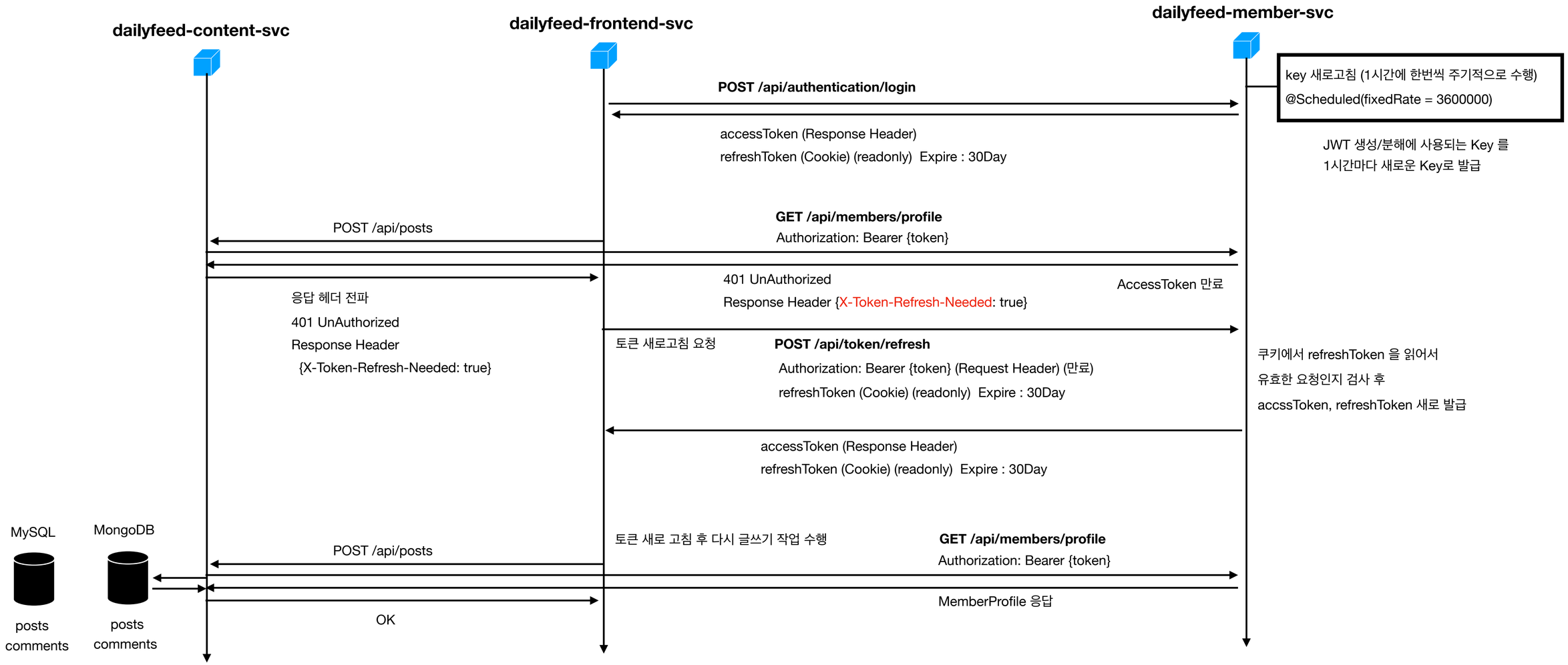
- JWT 를 만드는 Key 는 1시간에 한번씩 새로운 Key 로 바뀌며 해당 key 는 ‘jwt_keys’ 라는 테이블에 저장됩니다.
- 서버로 JWT를 담아 요청 시에 해당 JWT 내의 ‘kid’ 의 값이 오래된 값일 경우 (이미 새로운 Key 가 생성되어 있는 경우) 서버에서는 응답 헤더에 {‘X-Token-Refresh-Needed’: ‘true’} 를 담아서 정상 응답을 합니다.
- 클라이언트는 이 응답을 받으면 하던 작업을 모두 완료한 후 별도로 POST /api/token/refresh 로 요청을 보내 accessToken, refreshToken 을 새롭게 부여받습니다.

Case (9) JWT - Access Token 만료(Expired) 시 동작

AccessToken 이 만료될 경우

Refresh Token 을 이용해서 POST /api/token/refresh 를 수행

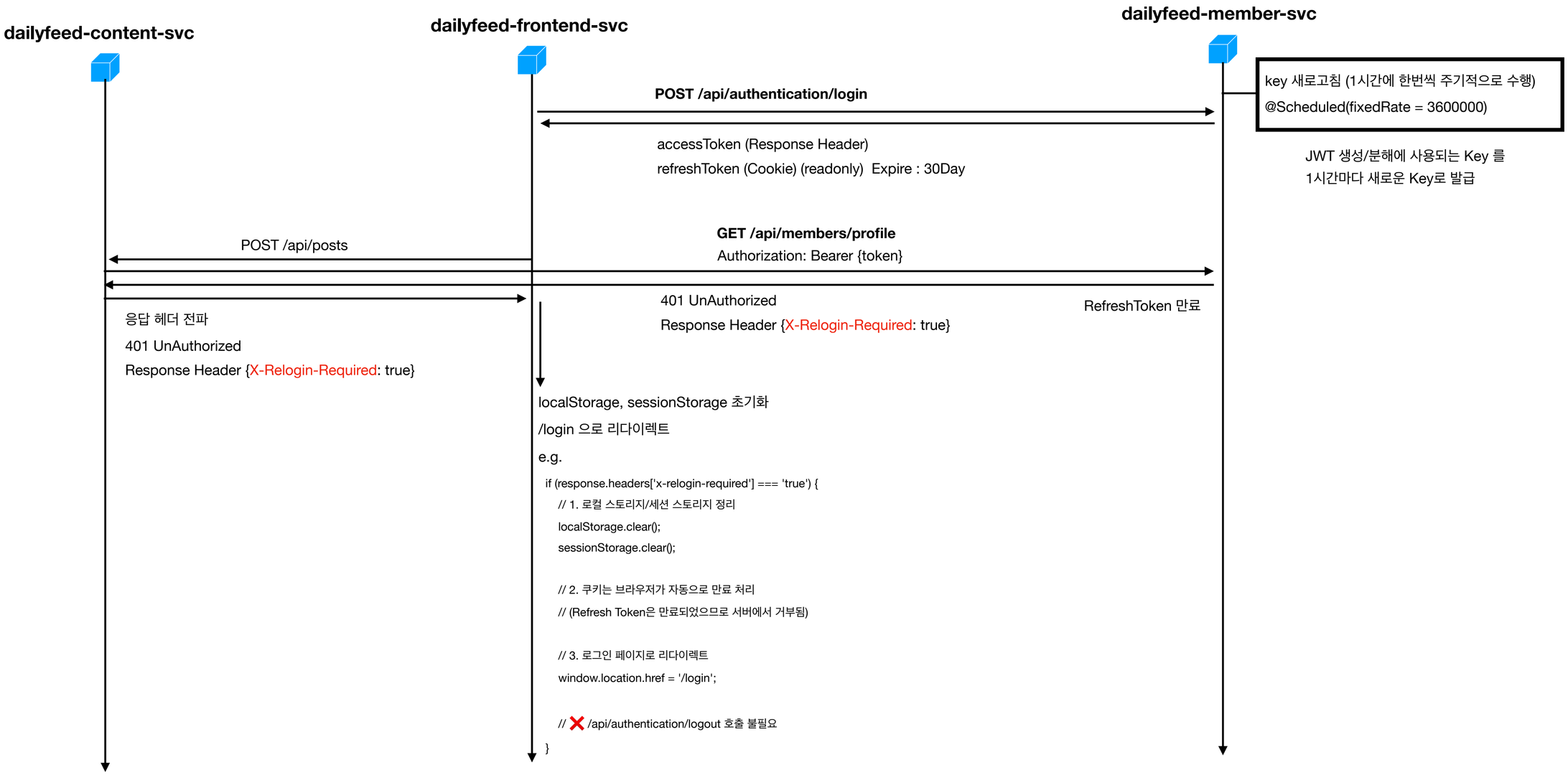
member-svc 에서는 새로운 Access Token 은 응답헤더로, 새로운 Refresh Token 은 cookie 에 세팅



Case (9) JWT - Refresh Token 만료(Expired) 시 동작

서버에서는 RefreshToken 이 만료될 경우 응답 헤더 {X-Relogin-Required: true} 와 함께 401 UnAuthorized 를 응답합니다.

클라이언트에서는 token 무효화 + /login 페이지로 이동합니다.



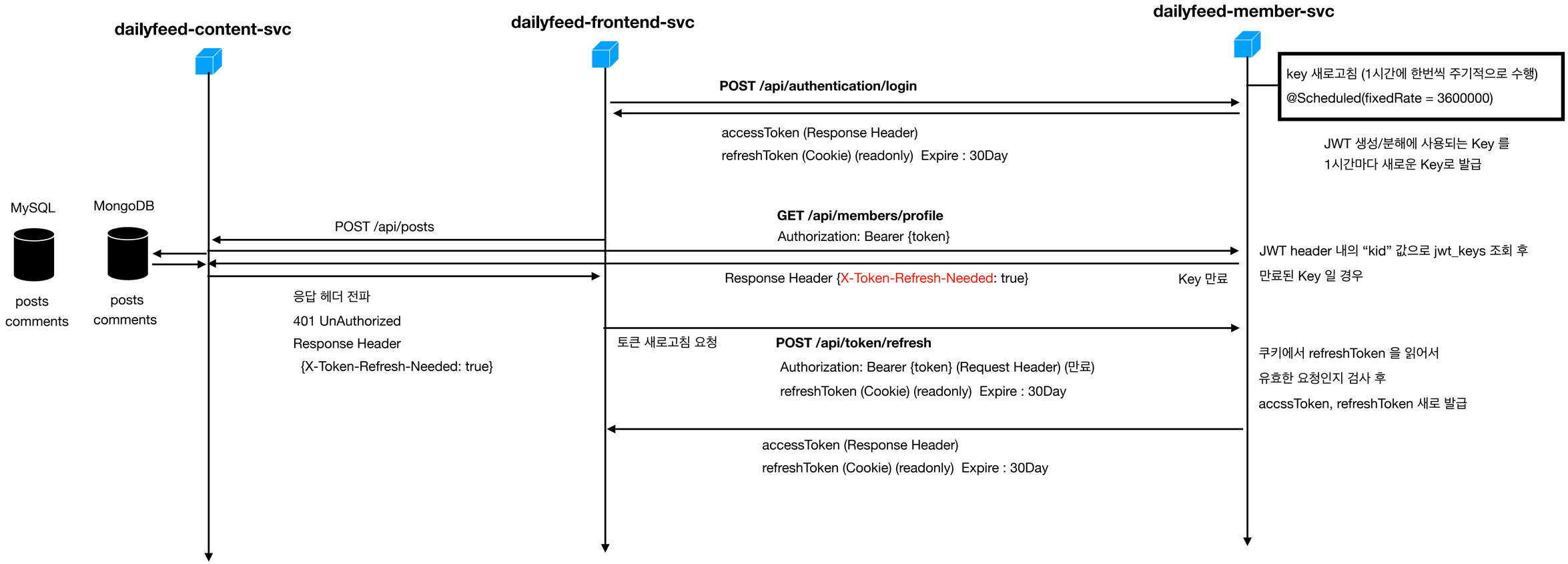
Case (9) JWT - 오래된 “kid” 로 만들어진 JWT 로 요청이 왔을 경우

만료된 Key 로 만든 JWT 로 요청시

JWT Header 내의 “kid” 값으로 “jwt_keys” 테이블 조회 후, 찾은 Key 가 만료된 Key(is_active=false)일 경우

응답 헤더로 {X-Token-Refresh-Needed: true} 를 응답하고 클라이언트에게 정상응답을 하며, 클라이언트 역시 하던 작업을 정상 수행합니다.

해당 작업 수행 후 별도로 클라이언트에서 서버로 **POST /api/token/refresh** 요청을 보내 새로운 accessToken, refreshToken 을 부여받습니다.



Case (9) Authentication - Season2

AuthenticationManager, AuthenticationProvider

- 현재 코드는 Filter 기반의 단순한 기능만을 사용하고 있는데 다양한 인증수단을 지원하는 프로젝트로 발전시켜나갈 예정입니다.

OAuth

- email 을 통한 회원가입 말고도 Google 로그인을 지원할 예정이며, Season 2 까지는 개발 버전의 구글 로그인을 제공할 예정입니다. (심사과정이 복잡하고 오래 걸리는 것으로 인해)
- OAuth 지원이 마무리되면 email 회원 기능은 제거 예정입니다. (개인정보 처리 법규 대응은 개인이 감당할 수 있는 범위의 일의 수준이 아니어서 SNS 로그인만을 지원하도록 변경 예정입니다.)

Case (10) Follow 정책, 댓글/답글 정책

Case (10) 주요 코딩 컨벤션

: no common -> 서브모듈로 공유

: mapper

: get—OrThrow

: boolean (x) -> Predicate enum 정의

: data 로직 공통화 자제 (과도한 리소스 사용방지)

: code (인터페이스 역할의 Dto, Exception,Enum 등 정의)

Case (10) 어려웠던 점들

어려웠던 점들

비용 문제로 kubernetes cluster 안에 infra 를 직접 설치해야 했던 점

: 개인 프로젝트이기에 비용을 최소화 하기 위해 kubernetes cluster 내에 redis,mysql,kafka,mongodb 를 설치했는데, 이 infra 들에 대한 작업을 하는 데에 시간이 정말 많이 걸렸습니다.

: 돈이 충분히 있다면 redis,mysql,mongodb 정도는 클라우드 서비스를 사용했을 것 같습니다.

: 위의 infra 리소스들로 인해 애플리케이션 hpa 설정시 각각의 requests, limits 를 점진적으로 늘리면서 infra 리소스가 죽는 현상이 없는지를 일일이 테스트하고 kubectl top 명령을 통해 마지막으로 안정적이었던 사양을 메모하는 거쳤는데 이 과정이 시간이 꽤 걸렸습니다.

Case (9) JWT - Key Refresh, Refresh Token, Blacklist

JWT Key

- JWT 토큰을 만들때 사용되는 Key 값으로 UUID 기반의 임의로 생성된 문자열입니다.
- ‘jwt_keys’ 라는 테이블에서 보관하며, 주기적으로 1시간에 한번씩 Key 가 만료기한이 지났는지 체크 후 만료기한이 지났을 경우 새로 생성합니다. 그리고 만료된 키값은 disable 처리 합니다.
- 애플리케이션을 새로 기동 시에는 jwt_key 에 key 가 없기에 이 경우에는 jwt_key 에 새로운 jwt_key 를 추가합니다.
- JWT 에 생성되는 Key 는 주기적으로 변경되어야 보안에 취약하지 않기 때문에 Key Refresh 기능을 추가했습니다.
- member-svc 내의 /api/token/refresh API 를 통해 현재 token 을 새로운 key 로 생성한 새로운 Token 을 부여받을 수 있습니다.

Refresh Token

- 로그인을 통해 부여받은 Access Token 은 기한이 짧습니다. 잦은 로그인은 사용자 경험을 떨어뜨리는데, 여기에 대해 RefreshToken 을 도입해서 Refresh Token 을 통해 token 새로 고침 요청을 합니다.
- Access Token 이 만료되었을 때 /api/token/refresh API 를 frontend 에서 호출하면 새로운 Access Token, Refresh Token 을 부여받게 됩니다. 이때 Refresh Token 은 AccessToken 의 jti 를 참조로 갖습니다.
- 만약 Refresh Token 까지도 만료되면 로그인을 다시 해야 합니다.

Blacklist

- 로그아웃을 했을 때 로그아웃된 해당 토큰을 Blacklist 에 추가해서, 중복된 토큰을 사용하지 않도록 보장합니다.
- 만료된 토큰,로그아웃된 토큰을 악용하는 케이스에 대응할수 있습니다.