회원 시스템 마이그레이션(-ing) 구축 부제 : 이제 AWS DMS와 kafka를 곁들인

- 들어가며
- 기존 회원 시스템 Event 발생 시키기
  - 분산 처리를 위한 DMS 설정
- 기존, 신규 회원 연동 마이그레이션 구축
  - 서로 다른 DB의 정합성을 맞추기 위한 기술 2PC
  - Spring Boot의 JTA Atomikos 설명
  - Spring Boot에 Kafka, DMS 적용
- 번외 : 신규 회원 Event 발행과 안정성 강화
- 정리
- 링크

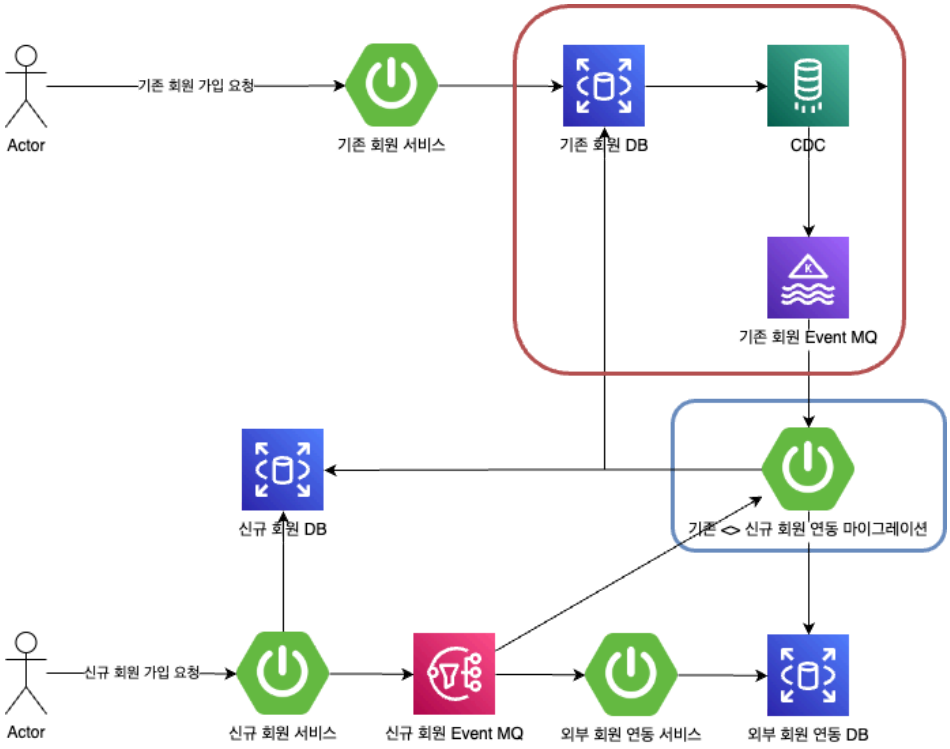
안녕하세요 아이들나라 플랫폼팀 김종인 입니다. 이번 글에서는 신규 시스템 구축 후 레거시 시스템과 어떻게 통합을 했는지, 어떤 기술을 사용하여 실제 코드로 어떻게 구현 했는지 설명하고자 작성하게 되었습니다.

기존 아이들나라 통합 회원 시스템을 소개합니다. (강력한 의존관계의 시스템에서 조금씩 느슨해지기) 글에서 실제 구현에 대한 설명 입니다.

들어가며

저희는 신규 회원 시스템을 구축 함과 동시에 레거시 회원 시스템을 신규 회원 시스템으로 마이그레이션을 계획 했습니다.

이때 사용하게 된 기술이 AWS DMS와 Spring Cloud Stream 인데요. 먼저 전체 구성도를 보면서 설명을 시작 하겠습니다.



기존 회원 시스템에서 코드 수정 없이 Event를 발생 시키키 위해 AWS DMS(Database Migration Service)를 사용 했습니다. AWS DMS는 홈페이지 설명 그대로 가동 중지 시간을 최소화 하면서 데이터베이스를 안전하게 마이그레이션 할 수 있는 서비스 인데요. DMS에 있는 CDC(change data capture)는 DB to DB 마이그레이션 을 실시간으로 진행 할 수 있는 유용한 서비스 입니다.(AWS를 사용하고 있지 않으면 Debezium 을 직접 사용 하시면 됩니다.) 기존 회원 시스템의 코드를 건드리지 않아도 DB 마이그레이션, 회원 Event Trigger로 사용하게 되었습니다.

그 후 신규 회원 Event와 기존 회원 Event를 Micro Batch성으로 처리를 담당해주는 Spring Cloud Stream을 사용하여 지속적으로 통합을 해주었습니다. 작업 시 어떤점을 고려 하면서 진행 했는지 자세히 설명 드리겠습니다.

3가지 챕터로 나누어서 설명 했습니다.

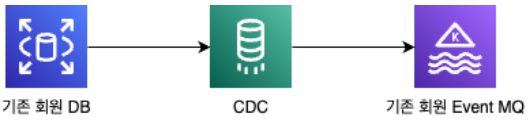
작업 내용

- 기존 회원 서비스 Event 발생 시키기
- 기존, 신규 회원 연동 마이그레이션 구축
- 번외: 신규 회원 Event 발행과 안정성 강화

기존 회원 시스템 Event 발생 시키기

저희는 다행이도(?) AWS 를 사용하고 있기에 기존 AWS에서 제공하는 managed CDC Service인 DMS로 테이블의 변경 내용을 Message로 받을 수 있었습니다.

DMS Task 생성 시 Source를 RDS로 했고, target을 Kafka로 생성 했습니다.



DMS Task 생성 시 Source에서 DB를 사용하기 위해서는 DB마다 특정한 기능을 활성화 시켜 주어야 하며, PostgreSQL 사용 시 WAL(Write Ahead Log)기능을 활성화 하시면 됩니다.
(`wal_level=logical`)

분산 처리를 위한 DMS 설정

그 다음 위에서 연동이된 Source와 Target을 연결해주는 Task를 만들면 됩니다. Task 생성 시 중요한 설정이 필요한데, 그건 `Paralle*` 로 시작하는 속성들을 설정 해줘야 합니다. 그 이유는 Task에서 Event 발행 시, Kafka의 Topic Partition에 Key를 `schema.table` 형식으로 보내기 때문에 이벤트의 순서 보장이 되지 않아 Target을 Consumer에서 사용 시 순서가 꼬일 가능성이 있습니다. 이벤트 순서 보장을 하기 위해 `schema.table-primaryKey` 형식으로 처리 할수 있게, Task 설정 시 아래 설정을 넣어줄 필요가 있습니다. (key hashing 방식으로 같은 key는 같은 parition에 적재됨)

```
"TargetMetadata": {
  "ParallelApplyBufferSize": 100,
  "ParallelApplyQueuesPerThread": 1,
  "ParallelApplyThreads": 2
}
```

Kafka Partition에 Key분배를 위한 Task 설정

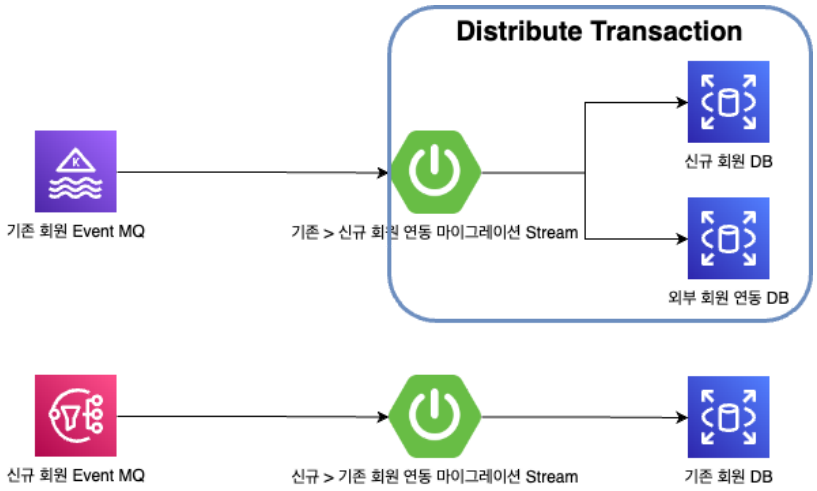
parallel 적용 전

	Offset	Partition	Timestamp	Key Preview	Value Preview
	931	0	8/21/2023, 08:29:50	기존회원Schema.기존회원Table	{ "data": { "기존회원Key": "wonik-test", "회원field": "...
	1013	2	8/21/2023, 08:29:24	기존회원Schema.기존회원Table	{ "data": { "기존회원Key": "wonik-test", "회원field": "...
	981	1	8/21/2023, 08:29:09	기존회원Schema.기존회원Table	{ "data": { "기존회원Key": "wonik-test", "회원field": "...
	930	0	8/21/2023, 08:28:35	기존회원Schema.기존회원Table	{ "data": { "기존회원Key": "wonik-test", "회원field": "...

parallel 적용 후

	Offset	Partition	Timestamp	Key Preview	Value Preview
	1046	1	8/21/2023, 08:34:48	기존회원Schema.기존회원Table-wonik-test	{ "data": { "기존회원Key": "wonik-test", "회원field": "...
	1045	1	8/21/2023, 08:34:48	기존회원Schema.기존회원Table-wonik-test	{ "data": { "기존회원Key": "wonik-test", "회원field": "...
	1044	1	8/21/2023, 08:34:48	기존회원Schema.기존회원Table-wonik-test	{ "data": { "기존회원Key": "wonik-test", "회원field": "...
	1043	1	8/21/2023, 08:34:48	기존회원Schema.기존회원Table-wonik-test	{ "data": { "기존회원Key": "wonik-test", "회원field": "...

기존, 신규 회원 연동 마이그레이션 구축



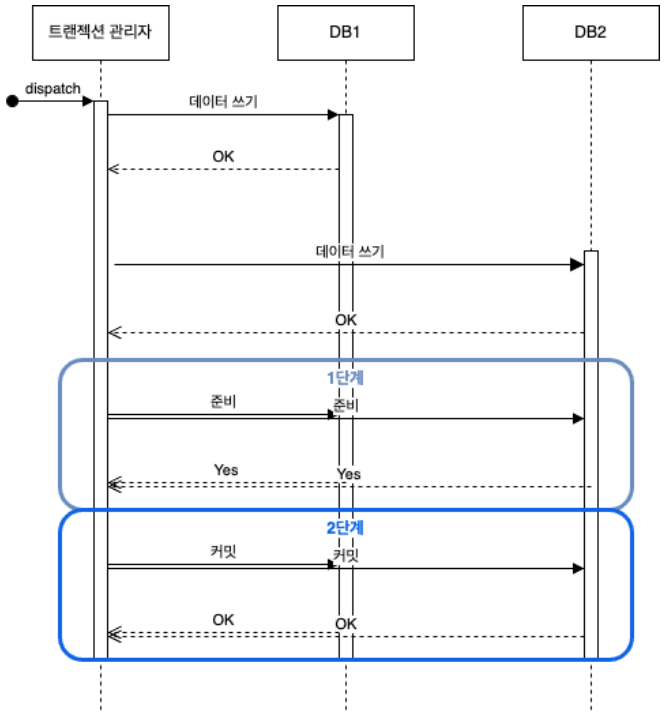
특히 여러 시스템을 처리할 때 데이터 일관성과 순서를 유지하는 것이 중요합니다. 기존 회원 시스템과 신규 회원 시스템 간의 원활한 Event 기반 스트리밍을 보장하기 위해 Kafka의 Topic Partition을 사용하고 병렬 처리를 위한 특정 속성을 설정 했습니다.

Event 기반 Stream 으로 서로 다른 시스템에 데이터 일관성과 순서를 유지하기 위해서 Event의 Key와 Event 시간을 잘 정의 해야 합니다.

Key는 Table에서 Primary Key를 기준으로 잡으면 되고, Event 시간은 여러 시간 중에 Table의 last_modified시간으로 결정 했습니다. Reqeust Time, Event Publish Time, Response Time 등이 있지만 Entity 데이터의 지속성과 영속성을 부여한 순간이 Event 처리 완료 시간이라고 생각되어 결정 했습니다.

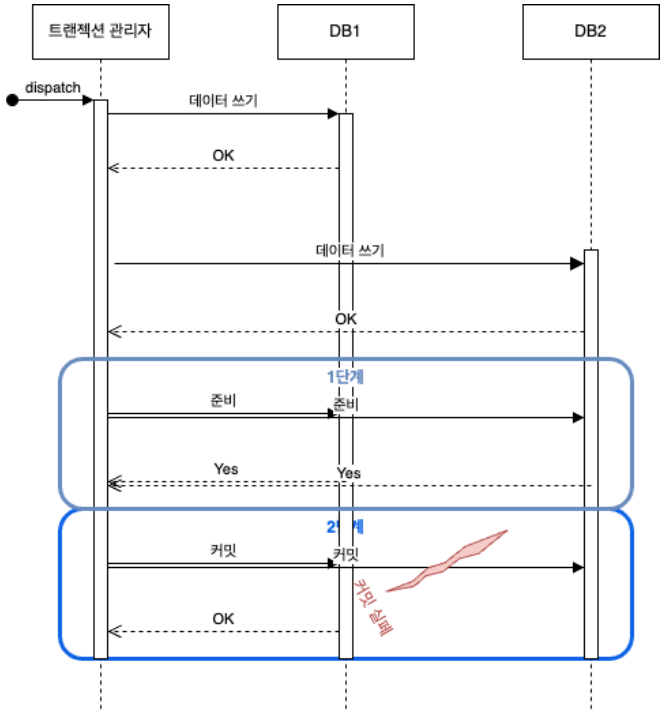
서로 다른 DB의 정합성을 맞추기 위한 기술 2PC

기존 회원 시스템과 신규 회원 시스템 연동 시 서로 다른 DB의 정합성을 맞추기 위해 distributed transaction(또는 XA Transaction)인 JTA(Java Transaction API) Atomikos를 활용 했습니다. JTA는 2PC(2 Phase Commit)를 지원하기 위한 표준 프로토콜이며 2PC는 서로 다른 DB뿐만 아니라 JMS(Java Message Service)와의 정합성을 맞출 수 있는 프로토콜 입니다.

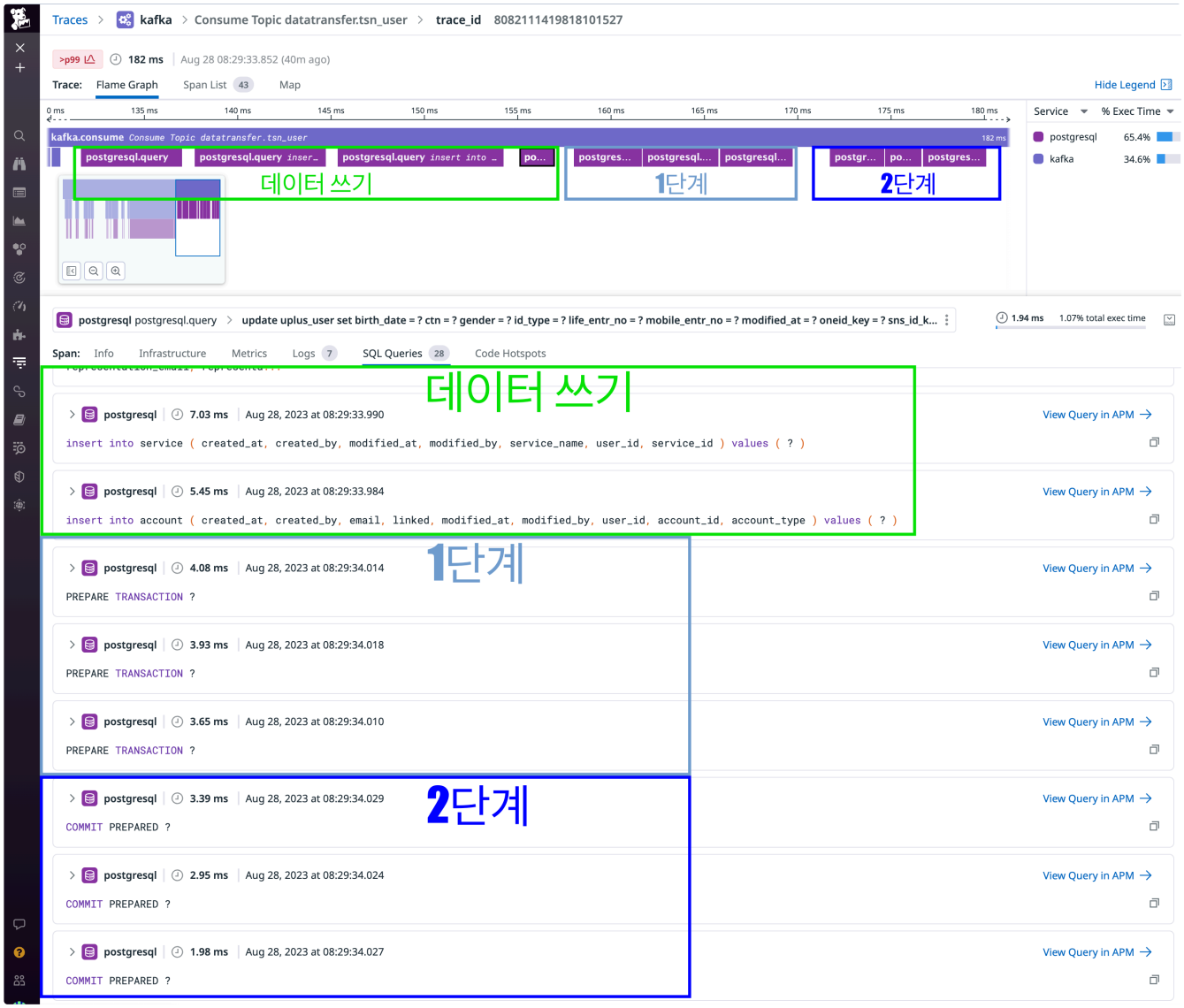


2PC 성공 Flow

2PC도 조심해서 사용해야 하는데요. 1단계 에서 실패 할때에는 Rollback이 가능 하지만 2단계에서 실패 시에는 Rollback이 불가 합니다. 그래서 이문제를 해결하고자 트랜젝션 관리자에서도 디스크에 요청결과를 저장하는데요 실패한 결과에 대해서는 성공할때까지 무한으로 요청을해서 2단계의 실패처리에 대한 안정성을 강화하게 됩니다. 그래서 2PC를 사용하는 Application의 경우 디스크를 가지고 있어야 하기에 Pod에 AWS EBS(Elastic Block Store)를 붙여서 관리 합니다.



2PC 실패 Flow



2PC적용된 Datadog Trace

Spring Boot의 JTA Atomikos 설명

Spring Boot3.0 기반 정식 버전은 2023년 5월 6일에 출시 하여 적용 했습니다.(ExtremeTransactions 6.0) PostgreSQL에서 JTA를 사용하기 위한 설정이 필요 한데요. 그건 `max_prepared_transactions` 입니다. `max_prepared_transactions`을 `max_connections`과 동일 하게 맞추고 사용 하게 되었습니다.
(`max_prepared_transactions`는 distributed transaction을 사용할 때 필요한 옵션 입니다.) 그리고 마이크로서비스에서는 2PC 대신에 Saga 패턴을 적용 해야 하지만, Saga 패턴에 이해가 부족한 상태에서 적용하기엔 다소 시간이 걸릴것으로 판단되어 2PC를 적용 하게 되었습니다.

기존 회원과 신규 회원의 Event가 동시에 발생 할 경우 Event 발생 시간을 기준으로 최신 메시지를 덮어 씌우는 식으로 작업하여 Event Consum이 지연되더라도 Event 시간 기반으로 최신 상태를 반영 하기에 만약 메시지가 지연 되더라도 데이터를 최신 상태로 유지 할 수 있게 됩니다.

Spring Boot에 Kafka, DMS 적용

DMS의 데이터 포멧은 Table의 data와 DMS metadata로 이루어져 있으며, 그 중 metadata.operation으로 데이터의 CUD를 알 수 있습니다.(insert, update, delete, load=전체데이터 Load)

```
{
    "data": {
        "고객데이터Key": "고객데이터Value"
    },
    "metadata": {
        "timestamp": "2023-08-03T03:27:45.193054Z",
        "record-type": "data",
        "operation": "insert",
        "partition-key-type": "primary-key",
        "schema-name": "고객DB스키마",
        "table-name": "고객DB테이블",
        "transaction-id": 0
    }
}
```

DMS 데이터 포멧

없어도될 Koltin Code

위의 DMS 데이터 포멧으로 Kotlin Code로 옮겼습니다.

```
data class Dms기존유저Data(
  val metadata: DmsMetaData,
  val data: 기존유저Dto,
)

@JsonNaming(PropertyNamingStrategies.KebabCaseStrategy::class)
data class DmsMetaData(
  val operation: OperrationType,
  val tableName: String,
)

@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy::class)
data class 기존유저Dto(
  val 유저Key: String,
  val 유저이름: String,
  private val lastModifiedAt: LocalDateTime,
) {
  val lastModifiedAtInstant: Instant = this.lastModifiedAt.toInstant(ZoneOffset.of("+09:00
}

enum class OperrationType {
  load, insert, update, delete
}
```

DMS 데이터 포멧에 대한 Kotlin Code

기존 회원 시스템에서 Code 수정 없이 Event를 발행하는 방법에 대해 알아 보았고, 이 Event를 사용하여 신규 회원 시스템에 통합 하는 방법에 대해 알아 보겠습니다.

기존유저To신규유저 마이그레이션은 Spring Cloud Stream으로 개발 했습니다. Spring Cloud Stream으로 개발한 이유는 자유로운 MQ(Message Queue) 바인더 선택과 다양한 Function들의 조

합으로 인해 기능을 작은 단위로 분리 할 수 있으며, 재 사용성이 높아져 다양하게 활용이 가능 하였습니다. 그리고 사내에 SCDF(Spring Cloud Data Flow)도 구축 되어 있어 Stream Processor를 쉽게 배포 할 수 있는 환경이 되어 있어 선택 하게 되었습니다.

```
@Bean
fun 기존유저To신규유저Consumer(messageMemeberHandlers: List<MessageMemeberHandler>): Consumer<
  return Consumer { message: Message<Dms기존유저Data> ->
    try {
      val payload: Dms기존유저Data = message.payload
      messageMemeberHandlers // OperrationType + 고객상태로 회원가입, 수정, 탈퇴 로직 실행
        .filter { it.isSurport(payload) }
        .forEach { it.handle(payload) }
    } catch (e: Exception) {
      logger().error(e) { e.localizedMessage }
    }
  }
}

```

기존 유저에서 신규 유저로 Migration 하는 Spring Cloud Stream

```
@Bean
fun 신규유저To기존유저Consumer(): Consumer<Message<신규유저Data>> {
  return Consumer { message: Message<신규유저Data> ->
    try {
      val payload: 신규유저Data = message.payload
      val eventType: EventType? = EventType.of(message.headers.get("eventType", String::cla
      when (eventType) {
        EventType.SIGN_UP -> existingMemberRegistration(payload)
        EventType.WITHDRAW -> existingMemberWithdrawal(payload)
        EventType.UPDATE -> modifyExistingMembers(payload)
        else -> logger().warn { "Undefined type $eventType" }
      }
    } catch (e: Exception) {
      logger().error(e) { e.localizedMessage }
    }
  }
}

```

신규 유저에서 기존 유저로 Migration 하는 Spring Cloud Stream

이로써 기존 회원 Event와 신규 회원 Event를 활용하여 실시간으로 통합하는 방법에 대해 알아 보았고 추가로 신규 회원 Event 발행 시 실패 할때 어떻게 안정성을 강화 했는지 알아 보겠습니다.

번외 : 신규 회원 Event 발행과 안정성 강화

신규 회원 시스템은 DMS를 사용하지 않고 Event발행을 고려하면서 설계 하여, 자체적으로 회원 Event를 발행 했습니다. 그래서 `Message 발행의 안정성을 강화`하기 위해 `Transactional Outbox Pattern`을 적용 했습니다. 이를 적용 하면 Event 발행이 실패할 경우나 재 발행이 필요한 경우 유용하게 사용할 수 있습니다.

신규 회원 시스템에서 회원 가입 하는 Service로직을 가져 왔습니다. 여기서는 회원 가입 시 Event 발행하는 부분을 보면 되는데요.

```
@Service
```

```kotlin
class SignUpService(
  private val signUpValidator: SignUpValidator,
  private val saveUserPort: SaveUserPort,
  private val applicationEventPublisher: ApplicationEventPublisher,
) : SignUpUseCase {

  @Transactional
  override fun signUp(signUpCommand: SignUpCommand): SignUpResult {
    signUpValidator.validate(signUpCommand)
    val savedUser: User = saveUserPort.save(signUpCommand)

    // 저장 후 신규 가입 Event 발행
    val signUpUserEvent: SignUpUserEvent = SignUpEventFactory.generateSignUpEvent(savedUser)
    applicationEventPublisher.publishEvent(signUpUserEvent)

    return SignUpResult(user = savedUser)
  }

}

```

SignUpUseCase 로직 `SignUpUserEvent` 를 발행

```kotlin
open class UserEvent<T>(
  val userId: String,
  val data: T,
  val publishedAt: Instant,
) {

  override fun toString(): String {
    return "UserEvent(userId=$userId, data=$data, publishedAt=$publishedAt)"
  }
}

class SignUpUserEvent(
  _userId: String,
  _data: SignUpUser,
  _publishedAt: Instant,
) : UserEvent<SignUpUser>(userId = _userId, data = _data, publishedAt = _publishedAt)
class UpdateUserEvent(
  _userId: String,
  _data: UpdateUser,
  _publishedAt: Instant,
) : UserEvent<UpdateUser>(userId = _userId, data = _data, publishedAt = _publishedAt)
class WithdrawUserEvent(
  _userId: String,
  _data: WithdrawUser,
  _publishedAt: Instant,
) : UserEvent<WithdrawUser>(userId = _userId, data = _data, publishedAt = _publishedAt)

```

UserEvent 정의

Transactional이 포함된 Event를 처리 할때 쓰이는 `TransactionalEventListener` 를 활용 하여 외부 MQ로 Event를 발행 했습니다. `TransactionalEventListener` 를 하용하게 되면 회원 정보가 저장 된 후 Event를 발행하게 도와 주며, 여기서 사용한 AFTER_COMMIT 이 외에, BEFORE_COMMIT, AFTER_ROLLBACK, AFTER_COMPLETION이 있으며 다양하게 활용 할 수 있습니다.

```kotlin
@Component
class UserEventListener(
  private val objectMapper: ObjectMapper,
  private val saveEventPort: SaveEventPort,
  private val eventMessagePublisherPort: EventMessagePublisherPort,
) {
  @Transactional(propagation = Propagation.REQUIRES_NEW)
  @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
  fun listenAndMqPublish(userEvent: UserEvent<*>) {
    val eventData: String = objectMapper.writeValueAsString(userEvent)
    publishedEvent(userEvent.javaClass.simpleName, userEvent.userId, eventData)
  }

  private fun publishedEvent(
    eventType: String,
    entityId: String,
    eventData: String,
  ) {
    val eventId: Long = saveEventPort.save(
      SaveEventCommand(
        eventType = eventType,
        entityType = "User",
        entityId = entityId,
        eventData = eventData
      )
    )
    if (eventMessagePublisherPort.publish(EventCommand(eventType = eventType, eventData = eve
      saveEventPort.published(eventId)
    }
  }
}
```

UserEvent를 Transactional Outbox 적용 부분(`publishedEvent`)

event Table은 `마이크 리처드슨` 의 책 `마이크로서비스 패턴` 의 Table 구조를 참조 하여 구성 하였으며, 책과 다른 부분은 event_id 같은 경우는 Event 특성상 무한으로 발행되기 때문에 event_id_sequence를 cycle로 설정하여 순환하게 구성 하였습니다.

```sql
create sequence event_id_sequence cycle;

create table event
(
    event_id        bigint   default nextval('event_id_sequence'::regclass) not null
        primary key,
    event_type      varchar(20)                                             not null,
    entity_type     varchar(50)                                             not null,
    entity_id       varchar(50)                                             not null,
    event_data      text                                                    not null,
    published       boolean  default false                                  not null,
    published_count smallint default 0                                      not null,
    created_at      timestamp                                               not null
);
alter sequence event_id_sequence owned by event.event_id;

create index idx_event_published on event (published);
```

Event 이력용 Table Schema

위에서 발행에 실패된 Event들은 Spring Cloud Stream을 통해서 재 발행을 하게 됩니다.



Event 발행 실패 시 Stream을 통한 Event 재 발행

Tansactional Outbox를 적용하면 Event 발행에 대한 안정성을 강화하게 되며, 재 발행이 필요할 경우에도 Event Table을 참조하여 쉽게 발행 할 수 있습니다.

정리

통합 회원 시스템을 구축 하면서 연동 시스템과의 의존성을 느슨하게 하였으며 고민한 부분을 설계에 녹였고, 이를 통해서 시스템의 안정성을 강화 했습니다. 또한 유지 보수를 쉽게 할 수 있는 시스템을 구축 했습니다. 저희는 더 큰 서비스로 도약하기 위해 확장성과 유지 보수성이 좋은 시스템을 만들기 위해 노력 할 것이며, 앞으로도 좋은 시스템을 만들기 위해 좋은 방법을 연구해 보겠습니다. 긴 글 읽어 주셔서 감사합니다.

- 레거시 시스템에서 Event 발행이 필요할 경우 CDC를 도입하자.
  - 코드를 건드리지 않는선에서 최선의 방법
- 레거시와 신규 시스템을 실시간으로 연동하고 싶은 경우 Spring Cloud Stream을 도입하자.
  - Saga Pattern 사용 시 고려 해야할 사항이 많지만 2PC를 사용하면 2개의 시스템 DB의 데이터의 일관성을 맞추는데 도움이 된다.
- Event 발행의 안정성을 강화 하고 싶으면 Transactional Outbox를 사용 하자.

링크

- AWS DMS
  - Source PostgreSQL - PostgreSQL 데이터베이스를 AWS DMS 소스로 사용 - AWS 데이터베이스 마이그레이션 서비스
  - Target Fafka - Apache Kafka를의 대상으로 사용 AWS Database Migration Service - AWS 데이터베이스 마이그레이션 서비스
- Tansactional Outbox - Microservices Pattern: Pattern: Transactional outbox
- 책`카프카 핵심 가이드`