10. 이벤트

version 2018.37

시스템 간 강결합의 문제

쇼핑몰 도메인에서 환불 기능을 구현한다고 가정해보겠습니다. 이때 환불 기능(도메인 서비스)의 실행 주체는 주문 도메인 엔티티가 될 수 있습니다. 아래의 코드는 이를 구현한 것입니다.

```
public class Order {
  ...

  // 외부 서비스를 실행하기 위해 도메인 서비스를 파라미터로 전달받음
  public void cancel(RefundService refundService) {
    ----------------------주문 로직--------------------
    verifyNotYetShipped();
    this.state = OrderState.CANCELED;

    this.refundStatus = State.REFUND_STARTED;
    -------------------------------------------------

    ----------------------결제 로직--------------------
    try {
      refundService.refund(getPaymentId);
      this.refundStatus = State.REFUND_COMPLETED;
    } catch(Exception ex) {
      ???
```

```
    }
    ----------------------------------------------
  }

  ...
}
```

> 환불 기능은 도메인 서비스이므로 실행 주체는 도메인 엔티티 뿐만 아니라 응용 서비스가 될 수도 있습니다.

보통 결제 시스템은 외부에 존재하므로 RefundService는 외부의 환불 시스템 서비스를 호출합니다. 때문에 이때 두 가지 문제가 발생합니다. **첫 번째 문제**는 환불 기능을 실행 하는 도중 외부 시스템에 문제가 생겨 익셉션이 발생하는 경우, 트랜잭션을 롤백을 해야할지 커밋을 해야할지에 대한 고민이 생깁니다. 롤백을 하는것이 맞아 보일지 모르나 주문 상태만 취소 상태로 변경하고 환불은 나중에 시도하는 방식으로 처리할 수 도 있습니다. **두 번째 문제**는 외부 시스템의 응답 시간이 길어지면 그만큼 대기 시간이 발생한다는 것입니다. 만약 환불 처리 기능이 30초가 걸리면 주문 취소 기능 또한 30초 길어집니다.

두 가지 문제 뿐만 아니라 설계상의 문제가 발생 할 수도 있습니다. 앞서 코드에서 확인한 것 처럼 서로 다른 도메인 로직(주문, 결제)이 하나의 도메인 객체에 존재 하므로써 도메인 개념을 흐리게 됩니다. 때문에 결제 기능의 수정이 필요한 경우 주문 도메인 객체가 수정 되어야하는 문제가 발생 할 수도 있습니다.

지금까지 언급한 문제가 발생하는 이유는 주문 BOUNDED CONTEXT와 결제 BOUNDED CONTEXT 간의 **강결합(high coupling)** 때문입니다. 이러한 강결합을 없애는 방법은 **이벤트**를 사용하는 것인데 특히 비동기 이벤트를 사용하면 두 시스템 간의 결합을 크게 낮출 수 있습니다.

이제 이벤트에 대해 살펴보겠습니다.

이벤트 개요

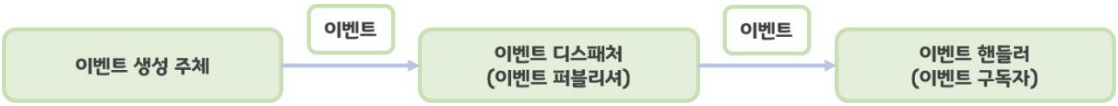
이 절에서 사용하는 **이벤트(event)**라는 용어는 **'과거에 벌어진 어떤 것'**을 뜻합니다. 예를 들어, 사용자가 주문을 취소했다면 '주문을 취소했음 이벤트'가 발생했다고 할 수 있습니다.

이벤트가 발생한다는 것은 상태가 변경됐다는 것을 의미합니다. '주문 취소됨 이벤트'가 발생한 이유는 주문이 취소 상태로 바뀌었기 때문입니다. 도메인 모델에서 도메인의 상태 변경을 이벤트로 표현 할 수 있습니다. 보통 '~할 때', '~가 발생하면', '만약 ~하면'과 같은 요구사항은 도메인의 상태 변경과 관련된 경우가 많고 이런 요구사항을 이벤트를 이용해서 구현할 수 있습니다.

> 예를 들어, '주문을 취소할 때 이메일을 보낸다'라는 요구사항에서 '주문을 취소 할때'는 '주문 취소됨 이벤트'를 활용하여 구현할 수 있습니다.

이벤트 관련 구성요소

도메인 모델에 이벤트를 도입하려면 다음과 같은 네 개의 구성요소를 구현해야합니다.



도메인 모델에서 **이벤트 생성 주체**는 엔티티, 밸류, 도메인 서비스와 같은 도메인 객체입니다. 이들 도메인 객체는 도메인 로직을 실행해서 상태가 바뀌면 관련 이벤트를 발생합니다.

이벤트 핸들러(handler)는 이벤트 생성 주체가 발생한 이벤트에 알맞은 처리를 합니다. 이벤트 핸들러는 이벤트 생성 주체가 발생한 이벤트를 전달받고 이벤트에 담긴 데이터를 이용해서 원하는 기능을 실행합니다. 예를 들어, '주문 취소됨 이벤트'를 전달 받은 이벤트 핸들러는 해당 주문의 주문자에게 SMS로 주문 취소 사실을 통지할 수 있습니다.

이벤트 디스패처(dispatcher)는 이벤트 생성 주체와 이벤트 핸들러의 연결 다리 역할을 합니다. 이벤트 생성 주체는 이벤트를 생성해서 디스패처에게 전달하면 디스패처는 해당 이벤트를 처리하는 이벤트 핸들러에게 이벤트를 전달합니다.

> 이벤트 디스패처는 구현 방식에 따라 이벤트 생성과 처리를 **동기**나 **비동기**로 실행합니다.

이벤트의 구성

이벤트는 발생한 이벤트에 대한 정보를 담습니다. 예를 들어 이벤트는 다음과 같은 정보를 가질 수 있습니다.

- **이벤트 종류:** 클래스 이름으로 이벤트 종류를 표현
- **이벤트 발생 시간**
- **추가 데이터:** 주문번호, 신규 배송지 정보 등 이벤트와 관련된 정보

배송지를 변경할 때 발생하는 이벤트를 다음과 같이 구현할 수 있습니다.

```
public class ShippingInfoChangedEvent {

  private String orderNumber;
  private long timestamp;
  private ShippingInfo newShippingInfo;

  // 생성자, getter
}
```

> 이벤트는 현재 기준으로(바로 직전이라도) 과거에 벌어진 것을 표현하기 때문에 **이벤트 이름에는 과거 시제를 사용합니다.**

이 이벤트를 발생시키는 이벤트 생성 주체는 Order 애그리거트입니다. Order 애그리거트의 배송지 변경 기능을 구현한 메서드는 다음 코드처럼 배송지 정보를 변경한 뒤에 이벤트 객체를 생성하고 디스패처를 사용하여 이벤트 객체를 전달 할 것입니다.

```
public class Order {
  ...

  public void changeShippingInfo(ShippingInfo newShippingInfo) {
    verifyNotYetShipped();
    setShippingInfo(newShippingInfo);
    Events.raise(new ShippingInfoChangedEvent(orderNumber.getNumber(),
newShippingInfo));
  }

  ...
}
```

예를 들어, 변경된 배송지 정보를 물류 서비스에 재전송하는 이벤트 핸들러는 다음과 같이 구현할 수 있습니다.

```
public class ShippingInfoChangedHandler implements
EventHandler<ShippingInfoChangedEvent> {
  ...

  public void handle(ShippingInfoChangedEvent evt) {
    shippingInfoSynchronizer.sync(evt.getOrderNumber(),
evt.getNewShippingInfo());
  }

  ...
}
```

이벤트는 이벤트 핸들러가 작업을 수행하는 데 필요한 최소한의 데이터를 담아야합니다. 이 데이터가 부족할 경우 핸들러는 필요한 데이터를 읽기 위해 관련 API를 호출하거나 DB에서 데이터를 직접 읽어와야 합니다.

이벤트 용도

이벤트는 두 가지 용도로 쓰일 수 있습니다. 첫 번째 용도는 트리거입니다. 도메인의 상태가 바뀔 때 후처리를 해야 할 경우 후처리를 실행하기 위한 트리거로 이벤트를 사용할 수 있습니다.

> **트리거의 사전적 의미:** 계기, 방아쇠

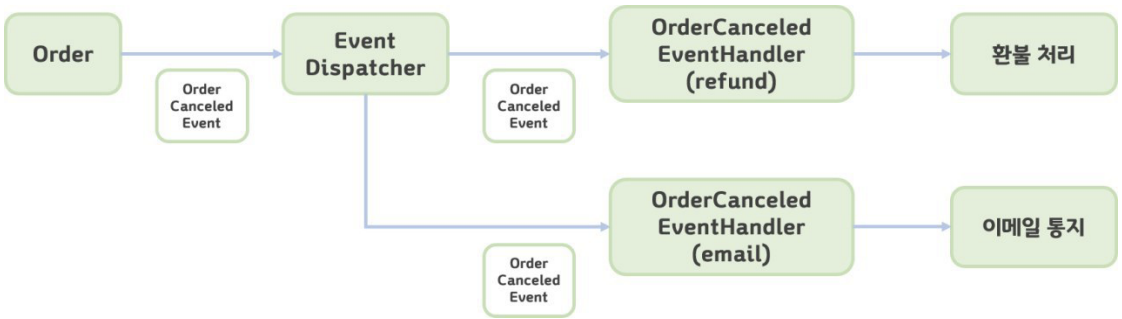
두 번째 용도는 서로 다른 시스템 간의 데이터 동기화입니다. 배송지를 변경하면 외부 배송 서비스에 바뀐 배송지 정보를 전송해야 합니다. 이 경우, 주문 도메인은 배송지 변경 이벤트를 발생시키고 이벤트 핸들러는 외부 배송 서비스와 배송지 정보를 동기화합니다.

이벤트 장점

이벤트를 사용하면 아래의 코드와 같이 서로 다른 도메인 로직이 섞이는 것을 방지할 수 있습니다.

```
public class Order {
  ...

  public void cancel(/*RefundService refundService*/) {
    verifyNotYetShipped();
    this.state = OrderState.CANCELED;

    this.refundStatus = State.REFUND_STARTED;
    Events.raise(new ShippingInfoCanceledEvent(orderNumber.getNumber()));
    /*
    ---------------------결제 로직-------------------
    try {
      refundService.refund(getPaymentId);
      this.refundStatus = State.REFUND_COMPLETED;
    } catch(Exception ex) {
      ???
    }
    ------------------------------------------------
    */
  }

  ...
}
```

이벤트 핸들러를 사용하면 기능 확장도 쉬워집니다. 구매 취소 시 환불과 함께 이메일로 취소 내용을 보내고 싶다면 이메일 발송을 처리하는 핸들러를 구현하고 디스패처에 등록하면 됩니다. 기능을 확장하더라도 구매 도메인 로직은 수정할 필요가 없어집니다.



이벤트, 핸들러, 디스패처 구현

실제 이벤트와 관련된 코드를 구현해봅시다. 이벤트와 관련된 코드는 다음과 같습니다.

- **이벤트 클래스**
- **EventHandler(인터페이스):** 이벤트 핸들러를 위한 상위 타입으로 모든 핸들러는 이 인터페이스를 구현해야합니다.
- **Events:** 이벤트 디스패처입니다. 이벤트 발행, 이벤트 핸들러 등록, 이벤트를 핸들러에 전달하는 등의 기능을 제공합니다.

이벤트 클래스

앞서 설명한 것처럼 이벤트 클래스는 이벤트를 처리하는데 필요한 최소한의 데이터를 포함해야 합니다. 예를 들어, 주문 취소됨 이벤트는 적어도 주문번호를 포함해야 관련 핸들러에서 후속 처리를 할 수 있습니다.

```
public class OrderCanceledEvent {

  private String orderNumber;

  public OrderCanceledEvent(String orderNumber) {
    this.orderNumber = orderNumber;
  }

  public String getOrderNumber() {
    return orderNumber;
  }
}
```

모든 이벤트가 공통으로 갖는 프로퍼티가 존재한다면 관련 상위 클래스를 만들 수도 있습니다. 예를 들어, 모든 이벤트가 발생 시간을 갖도록 하려면 아래와 같은 상위 클래스를 만들고 각 이벤트 클래스가 상속 받도록 할 수 있습니다.

```
public abstract class Event {

  private long timestamp;

  public Event() {
    this.timestamp = System.currentTimeMillis();
  }

  public long getTimestamp() {
    return timestamp;
  }
}
```

```
public class OrderCanceledEvent extends Event {

  private String orderNumber;
```

```
  public OrderCanceledEvent(String orderNumber) {
    super();
    this.orderNumber = orderNumber;
  }

  public String getOrderNumber() {
    return orderNumber;
  }
}
```

EventHandler 인터페이스

EventHandler 인터페이스는 이벤트 핸들러를 위한 상위 인터페이스입니다. 아래의 코드는 이를 구현한 것입니다.

```
public interface EventHandler<T> {

  void handle(T event);

  //핸들러가 이벤트를 처리할 수 있는지 여부를 검사
  default boolean canHandle(Object event) {
    Class<?>[] typeArgs =
TypeResolver.resolveRawArguments(EventHandler.class, this.getClass());
    return typeArgs[0].isAssignableFrom(event.getClass());
  }
}
```

EventHandler 인터페이스를 구현한 클래스는 handle() 메서드를 이용해서 필요한 기능을 구현하면 됩니다.

이벤트 디스패처인 Events 구현

도메인을 사용하는 응용 서비스는 Events.handle() 메서드를 이용하여 이벤트 핸들러를 등록하고, 도메인 기능을 실행합니다. 이벤트 핸들러 등록을 쉽게 하기 위해 다음과 같이 정적 메서드를 이용해서 구현할 수 있습니다.

```
public class CancelOrderService {

  private RefundService refundService;
  private OrderRepository orderRepository;

  @Transactional
  public void cancel(Order orderNo) {
    Events.handle(
        (OrderCanceledEvent evt) ->
refundService.refund(evt.getOrderNumber())
    );
```

```
    Order order = findOrder(orderNo);
    order.cancel();

    Events.reset();
  }

  ...
}
```

이 코드에서 Events.handle() 메서드에 OrderCanceledEvent를 처리할 이벤트 핸들러를 전달합니다. Events.handle() 메서드는 인자로 전달 받은 이벤트 핸들러를 List에 보관합니다. 이 후 디스패처(Events)는 이벤트가 발생하면 이에 맞는 이벤트 핸들러의 handle() 메서드를 호출합니다.

이벤트를 발생시킬 때에는 Events.raise() 메서드를 사용합니다. 다음 코드와 같이 '구매 취소 되었음' 이벤트를 발생 시킬 수 있습니다.

```
public class Order {
  ...

  public void cancel() {
    verifyNotYetShipped();
    this.state = OrderState.CANCELED;

    this.refundStatus = State.REFUND_STARTED;
    Events.raise(new ShippingInfoCanceledEvent(orderNumber.getNumber()));
  }

  ...
}
```

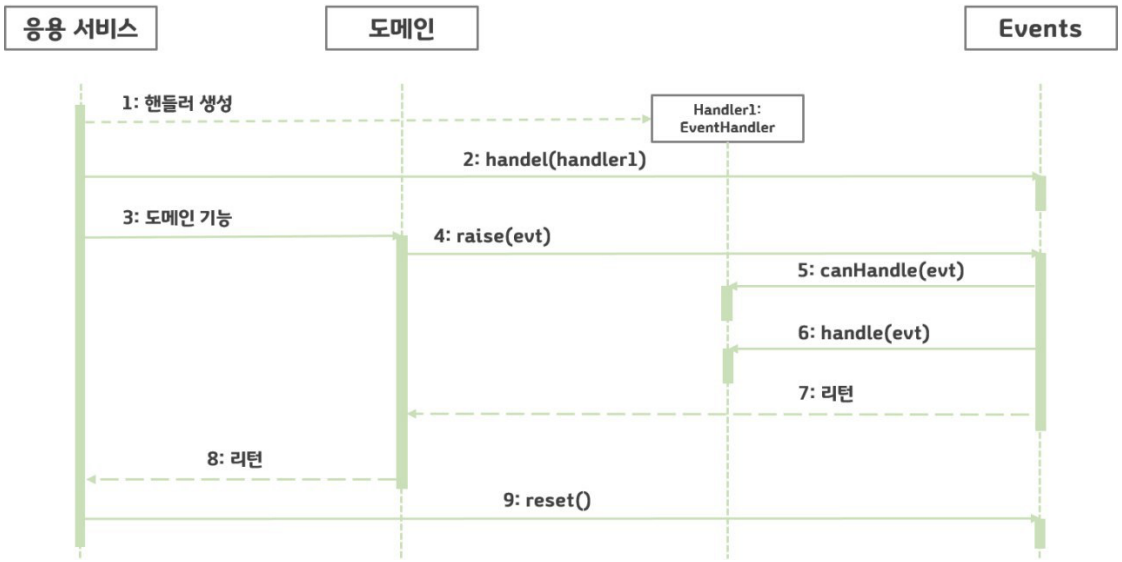
Events.raise()를 이용해서 이벤트를 발생시키면 Events.raise() 메서드는 이벤트를 처리할 핸들러를 찾아 handle() 메서드를 실행합니다.

Events 클래스의 구현 코드는 아래와 같습니다.

```
public class Events {

  private static ThreadLocal<List<EventHandler<?>>> handlers = new
ThreadLocal<>();
  private static ThreadLocal<Boolean> publishing =
      new ThreadLocal<Boolean>() {
        @Override
```

```
        protected Boolean initialValue() {
          return Boolean.FALSE;
        }
      };

  public static void raise(Object event) {
    if (publishing.get()) return;

    try {
      publishing.set(Boolean.TRUE);

      List<EventHandler<?>> eventHandlers = handlers.get();
      if (eventHandlers == null) return;
      for (EventHandler handler : eventHandlers) {
        if (handler.canHandle(event)) {
          handler.handle(event);
        }
      }
    } finally {
      publishing.set(Boolean.FALSE);
    }
  }

  public static void handle(EventHandler<?> handler) {
    if (publishing.get()) return;

    List<EventHandler<?>> eventHandlers = handlers.get();
    if (eventHandlers == null) {
      eventHandlers = new ArrayList<>();
      handlers.set(eventHandlers);
    }
    eventHandlers.add(handler);
  }

  public static void reset() {
    if (!publishing.get()) {
      handlers.remove();
    }
  }
}
```

흐름 정리



동기 이벤트 처리 문제

이벤트를 사용해서 강결합 문제는 해소했지만 아직 남아 있는 문제가 하나 있습니다. 바로 외부 서비스에 영향을 받는 문제입니다.

```
public class CancelOrderService {

  private RefundService refundService;
  private OrderRepository orderRepository;

  @Transactional // 외부 연동 과정에서 익셉션이 발생하면 트랜잭션 처리는?
  public void cancel(Order orderNo) {
    Events.handle(
        // RefundService.refund()가 오래 걸리면?
        (OrderCanceledEvent evt) ->
refundService.refund(evt.getOrderNumber())
    );

    Order order = findOrder(orderNo);
    order.cancel();

    Events.reset();
  }

  ...
}
```

외부 시스템과의 연동을 동기로 처리할 때 발생하는 성능문제와 트랜잭션 범위 문제를 해소하는 방법중 하나가 이벤트를 비동기로 처리하는 것입니다. 이어서 비동기 이벤트 처리에 대해 알아보겠습니다.

비동기 이벤트 처리

이벤트를 비동기로 구현할 수 있는 방법은 매우 다양합니다. 이 절에서는 다음의 네 가지 방식으로 비동기 이벤트 처리를 구현하는 방법에 대해 살펴보겠습니다.

- 로컬 핸들러를 비동기로 실행하기
- 메시지 큐를 사용하기
- 이벤트 저장소와 이벤트 포워더 사용하기
- 이벤트 저장소와 이벤트 제공 API 사용하기

네 가지 방식은 각자 구현하는 방식도 다르고 그에 따른 장점과 단점이 있습니다. 각 방식에 대해 차례대로 살펴보겠습니다.

로컬 핸들러의 비동기 실행

우리는 이벤트 핸들러를 별도의 스레드로 실행하므로써 비동기로 실행할 수 있습니다. 스프링에서 서로 다른 스레드에서 실행되는 두 메서드는 서로 다른 트랜잭션을 사용합니다. 때문에 별도의 스레드를 사용하여 트랜잭션 문제도 해결할 수 있습니다. 아래의 코드는 Events 클래스에 비동기로 핸들러를 실행하는 기능을 추가한 것입니다.

```
public class Events {

  private static ThreadLocal<List<EventHandler<?>>> handlers = new
ThreadLocal<>();
  private static ThreadLocal<Boolean> publishing =
      new ThreadLocal<Boolean>() {
        @Override
        protected Boolean initialValue() {
          return Boolean.FALSE;
        }
      };
  ----------------------------------------시작---------------------------------
-
  private static ThreadLocal<List<EventHandler<?>>> asyncHandlers = new
ThreadLocal<>();
  private static ExecutorService executor;

  public static void init(ExecutorService executor) {
    Events.executor = executor;
  }

  public static void close() {
    if (executor != null) {
      executor.shutdown();
      try {
        executor.awaitTermination(10, TimeUnit.SECONDS);
      } catch (InterruptedException e) {
```

```
      }
    }
  }
  ----------------------------------------끝----------------------------------
--

  public static void raise(Object event) {
    if (publishing.get()) return;

    try {
      publishing.set(Boolean.TRUE);
  ----------------------------------------시작----------------------------------
--
      List<EventHandler<?>> asyncEventHandlers = asyncHandlers.get();
      if (asyncEventHandlers == null) return;
      for (EventHandler handler : asyncEventHandlers) {
        if (handler.canHandle(event)) {
          executor.submit(() -> handler.handle(event));
        }
      }
  ----------------------------------------끝----------------------------------
---
      List<EventHandler<?>> eventHandlers = handlers.get();
      if (eventHandlers == null) return;
      for (EventHandler handler : eventHandlers) {
        if (handler.canHandle(event)) {
          handler.handle(event);
        }
      }
    } finally {
      publishing.set(Boolean.FALSE);
    }
  }

  public static void handle(EventHandler<?> handler) {
    if (publishing.get()) return;

    List<EventHandler<?>> eventHandlers = handlers.get();
    if (eventHandlers == null) {
      eventHandlers = new ArrayList<>();
      handlers.set(eventHandlers);
    }
    eventHandlers.add(handler);
  }
----------------------------------------시작----------------------------------
  public static void handleAsync(EventHandler<?> handler) {
    if (publishing.get()) return;
```

```
      List<EventHandler<?>> eventHandlers = asyncHandlers.get();
      if (eventHandlers == null) {
        eventHandlers = new ArrayList<>();
        asyncHandlers.set(eventHandlers);
      }
      eventHandlers.add(handler);
    }
  ----------------------------------------끝----------------------------------
    public static void reset() {
      if (!publishing.get()) {
        handlers.remove();
        asyncHandlers.remove();
      }
    }
  }
```

> 별도 스레드를 이용해서 이벤트 핸들러를 실행하면 이벤트 발생 코드와 같은 트랜잭션 범위에 묶을 수 없기 때문에 한 트랜잭션으로 실행해야 하는 이벤트 핸들러는 비동기로 처리해서는 안됩니다.

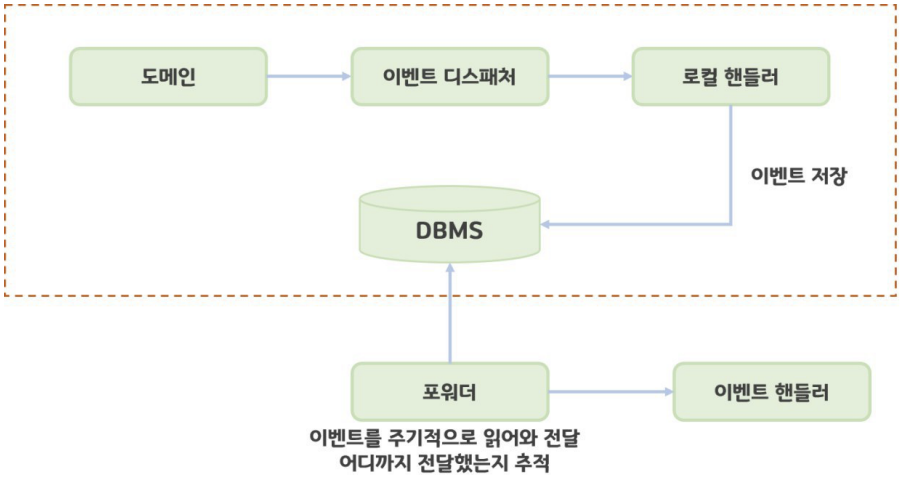
메시징 시스템을 이용한 비동기 구현

RabbitMQ와 같은 메시징 큐를 사용하여 비동기로 이벤트를 처리할 수도 있습니다. 이벤트가 발생하면 이벤트 디스패처는 이벤트를 메시지 큐에 보냅니다. 메시지 큐는 이벤트를 메시지 리스너에 전달하고, 메시지 리스너는 알맞은 이벤트 핸들러를 이용해서 이벤트를 처리합니다. 이때 이벤트를 메시지 큐에 저장하는 과정과 메시지 큐에서 이벤트를 읽어와 처리하는 과정은 별도 스레드나 프로세스로 처리됩니다.

> 많은 경우 메시지 큐를 사용하면 이벤트를 발생하는 주체와 이벤트 핸들러가 별도의 프로세스에서 동작합니다. 이는 자바의 경우 이벤트 발생 JVM과 이벤트 처리 JVM이 다르다는 것을 의미합니다.

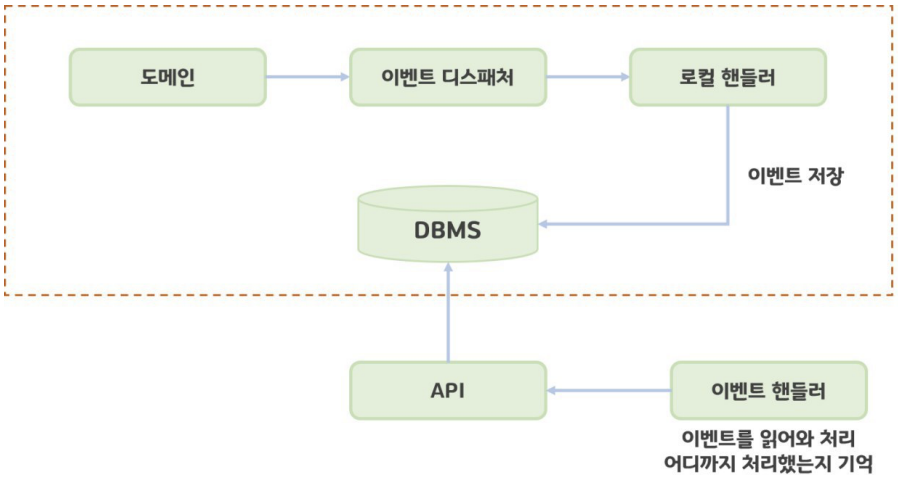
이벤트 저장소를 이용한 비동기 처리

이벤트를 DB에 저장한 뒤에 별도 프로그램을 이용해서 이벤트 핸들러에 전달하므로써 이벤트를 비동기로 처리할 수도 있습니다. 이 방식의 실행 흐름은 다음 그림과 같습니다.



이벤트가 발생하면 핸들러는 데이터베이스에 이벤트를 저장합니다. 포워더는 주기적으로 이벤트 저장소에서 이벤트를 가져와 이벤트 핸들러를 실행합니다. 포워더는 별도의 스레드를 이용하기 때문에 이벤트 발행과 처리가 비동기로 처리됩니다.

이벤트 저장소를 이용한 두 번째 방법은 아래의 그림과 같이 이벤트를 외부에 제공하는 API를 사용하는 것입니다.

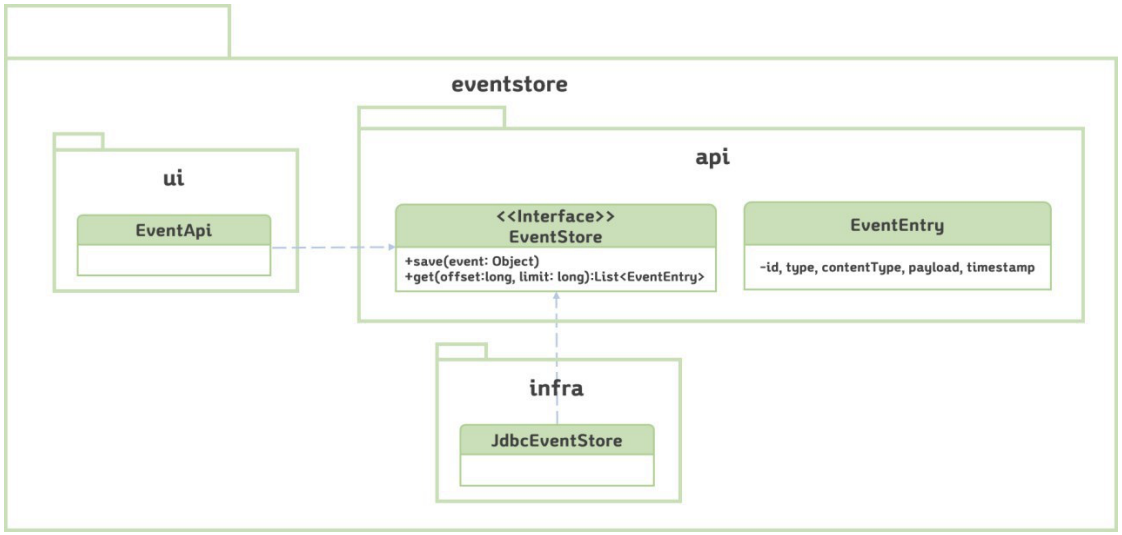


API 방식과 포워더 방식은 이벤트를 전달하는 방식에 있어서 차이가 있습니다. 포워더 방식에서는 포워더가 이벤트를 외부에 전달하는 방식이라면, API 방식은 외부 핸들러가 API 서버를 통해 이벤트 목록을 가져오는 방식입니다.

> 포워더 방식에서 포워더가 이벤트를 어디까지 처리했는지 기억해야 하고, API 방식에서는 외부 핸들러가 자신이 이벤트를 어디까지 처리했는지 기억 해야합니다.

이벤트 저장소 구현

포워더 방식과 API 방식 모두 이벤트 저장소가 필요하므로 이벤트를 저장할 저장소가 필요합니다. 이벤트 저장소를 구현한 코드 구조는 다음과 같습니다.



- **EventEntry:** 이벤트 저장소에 저장할 데이터
  - **id:** 이벤트 식별자
  - **type:** 이벤트의 타입
  - **contentType:** 직렬화한 이벤트 데이터의 형식
  - **payload:** 이벤트의 데이터
  - **timestamp:** 이벤트 발생시간
- **EventStore:** 이벤트를 저장하고 조회하는 인터페이스를 제공
- **JdbcEventStore:** JDBC를 이용한 EventStore 구현 클래스
- **EventApi:** REST API를 이용하여 이벤트 목록을 제공하는 컨트롤러

> **직렬화:** 객체를 전송 가능한 형태로 만드는것

EventEntry 클래스는 다음 코드와 같이 이벤트 데이터를 정의하고 있습니다.

```
public class EventEntry {

  private Long id;
  private String type;
  private String contentType;
  private String payload;
  private long timestamp;

  public EventEntry(String type, String contentType, String payload) {
    this.type = type;
    this.contentType = contentType;
    this.payload = payload;
    this.timestamp = System.currentTimeMillis();
  }

  public EventEntry(Long id, String type, String contentType, String
payload, long timestamp) {
    this.id = id;
```

```
    this.type = type;
    this.contentType = contentType;
    this.payload = payload;
    this.timestamp = timestamp;
  }

  // getter
}
```

EventStore는 이벤트 객체를 직렬화해서 payload에 저장합니다. 이때 이벤트 객체를 JSON 형식으로 payload에 저장했다면 contentType의 값으로 'application/json'을 갖습니다.

EventStore 인터페이스는 다음과 같습니다.

```
public interface EventStore {

  void save(Object event);
  List<EventEntry> get(long offset, long limit);
}
```

EventStore 인터페이스를 구현한 JdbcEventStore 클래스는 다음과 같습니다.

```
public class JdbcEventStore implements EventStore {
  private ObjectMapper objectMapper;
  private JdbcTemplate jdbcTemplate;

  @Override
  public void save(Object event) {
    EventEntry entry = new EventEntry(event.getClass().getName(),
"application/json", toJson(event));

    // insert 쿼리 실행
  }

  private String toJson(Object event) {
    try {
      return objectMapper.writeValueAsString(event);
    } catch (JsonProcessingException e) {
      throw new PayLoadConvertException(e);
    }
  }

  @Override
  public List<EventEntry> get(long offset, long limit) {
```

```
    return jdbcTemplate.query(
          "select * from evententry order by id asc limit ?, ?",
          ps -> {
            ps.setLong(1, offset);
            ps.setLong(2, limit);
          },
          (rs, rowNum) -> {
            return new EventEntry(
                rs.getLong("id"),
                rs.getSting("type"),
                rs.getSting("content_type"),
                rs.getSting("payload"),
                rs.getTimestamp("timestamp").getTime();
                );
          }
        );

  }

  // setter
}
```

EventEntry를 저장할 evententry테이블의 DDL은 다음과 같습니다.

```
create table evententry (
    id int not null AUTO_INCREMENT PRIMARY KEY,
    `type` varchar(255),
    `content_type` varchar(255),
    payload MEDIUMTEXT,
    `timestamp` datetime
) character set utf8;
```

이벤트 저장을 위한 이벤트 핸들러 구현

이벤트 저장소의 기반이 되는 클래스는 모두 구현하였습니다. 이제 남은 것은 발생한 이벤트를 이벤트 저장소에 추가하는 이벤트 핸들러를 구현하는 것입니다. 다음 코드는 이벤트 핸들러를 구현한 것입니다.

```
public class EventStoreHandler implements EventHandler<Object> {

  private EventStore eventStore;

  @Override
  public void handle(Object event) {
    eventStore.save(event);
  }
```

```
  @Autowired
  public void setEventStore(EventStore eventStore) {
    this.eventStore = eventStore;
  }
}
```

EventStoreHandler는 EventHandler<Object>를 상속하고 있으므로 canHandle() 메서드는 모든 객체에 대해 true를 리턴합니다. 즉 이벤트 타입에 상관없이 이벤트는 저장소에 보관됩니다.

REST API 구현

REST API는 간단합니다. offset과 limit의 웹 요청 파라미터를 이용해서 EventStore.get 메서드를 실행하고 그 결과를 JSON으로 리턴하면 됩니다. 다음은 이를 구현한 코드입니다.

```
@RestController
public class EventApi {

  private EventStore eventStore;

  @RequestMapping(value = "/api/events", method = RequestMethod.GET)
  public List<EventEntry> list(@RequestParam(name="offset", required =
true) Long offset,
                               @RequestParam(name="limit", required = true)
Long limit) {
    return eventStore.get(offset, limit);
  }

  public void setEventStore(EventStore eventStore) {
    this.eventStore = eventStore;
  }
}
```

API를 사용하는 클라이언트(외부 이벤트 핸들러)는 일정 간격으로 다음 과정을 실행합니다.

1. 가장 마지막에 처리한 데이터의 오프셋인 lastOffset을 구합니다. 저장한 lastOffset이 없으면 0을 사용합니다.
2. 마지막에 처리한 lastOffset을 offset으로 사용해서 API를 실행합니다.
3. API 결과로 받은 데이터를 처리합니다.
4. offset + 데이터 개수를 lastOffset으로 저장합니다.

포워더 구현

포워더는 일정 주기로 EventStore로 부터 이벤트를 읽어와 이벤트 핸들러에게 전달하면 됩니다. API 방식에서 클라이언트와 마찬가지로 마지막으로 전달한 이벤트의 오프셋을 기억해 두었다가 다음 조회 시점에 마지막으로 처리한 오프셋 부터 이벤트를 가져오면 됩니다. 다음은 포워더를 구현한 코드입니다.

```
public class EventForwarder {

  private static final int DEFAULT_LIMIT_SIZE = 100;

  private EventStore eventStore;
  private OffsetStore offsetStore;
  private EventSender eventSender;

  private int limitSize = DEFAULT_LIMIT_SIZE;

  @Scheduled(initialDelay = 1000L, fixedDelay = 1000L)
  public void getAndSend() {
    long nextOffset = getNextOffset();
    List<EventEntry> events = eventStore.get(nextOffset, limitSize);
    if (!events.isEmpty()) {
      int processedCount = sendEvent(events);
      if (processedCount > 0) {
        saveNextOffset(nextOffset + processedCount);
      }
    }
  }

  private long getNextOffset() {
    return offsetStore.get();
  }

  private int sendEvent(List<EventEntry> events) {
    int processedCount = 0;
    try {
      for (EventEntry entry : events) {
        eventSender.send(entry);
        processedCount++;
      }
    } catch (Exception e) {

    }
    return processedCount;
  }

  private void saveNextOffset(long nextOffset) {
    offsetStore.update(nextOffset);
  }

  // 각 필드에 대한 set 메서드
}
```

> getAndSend() 메서드를 주기적으로 실행하기 위해 스프링의 @Scheduled 애노테이션을 사용

이벤트 적용 시 추가 고려사항

1. **이벤트 소스**
2. **포워더-전송 실패를 얼마나 허용할 것인가**
3. **이벤트 손실**
4. **이벤트 순서**
5. **이벤트 재처리**