



Implementing EventSourcing & CQRS





```
var me = old Developer(){  
    Name = "심천보",  
    Company = "coupang",  
    Title = "Backend Developer"  
}
```

개인적이 행복지니스...

Event Sourcing & CQRS

모르시는 분

Put your hands UP!

뭔지?



What is Event Sourcing?

“

Capture all changes to an application state as a sequence of events.

”

“

어플리케이션의 모든 상태 변화를
순서에 따라 이벤트로 보관한다. ”



Martin Fowler

1. 데이터 저장방식의 새로운 Pattern

전통적인 데이터 저장 방법

account_id	balance	last_update
45678	2000	20170103010101
12345	51000	20170103145510
23456	40000	
.	.	.
.	.	.
.	.	.

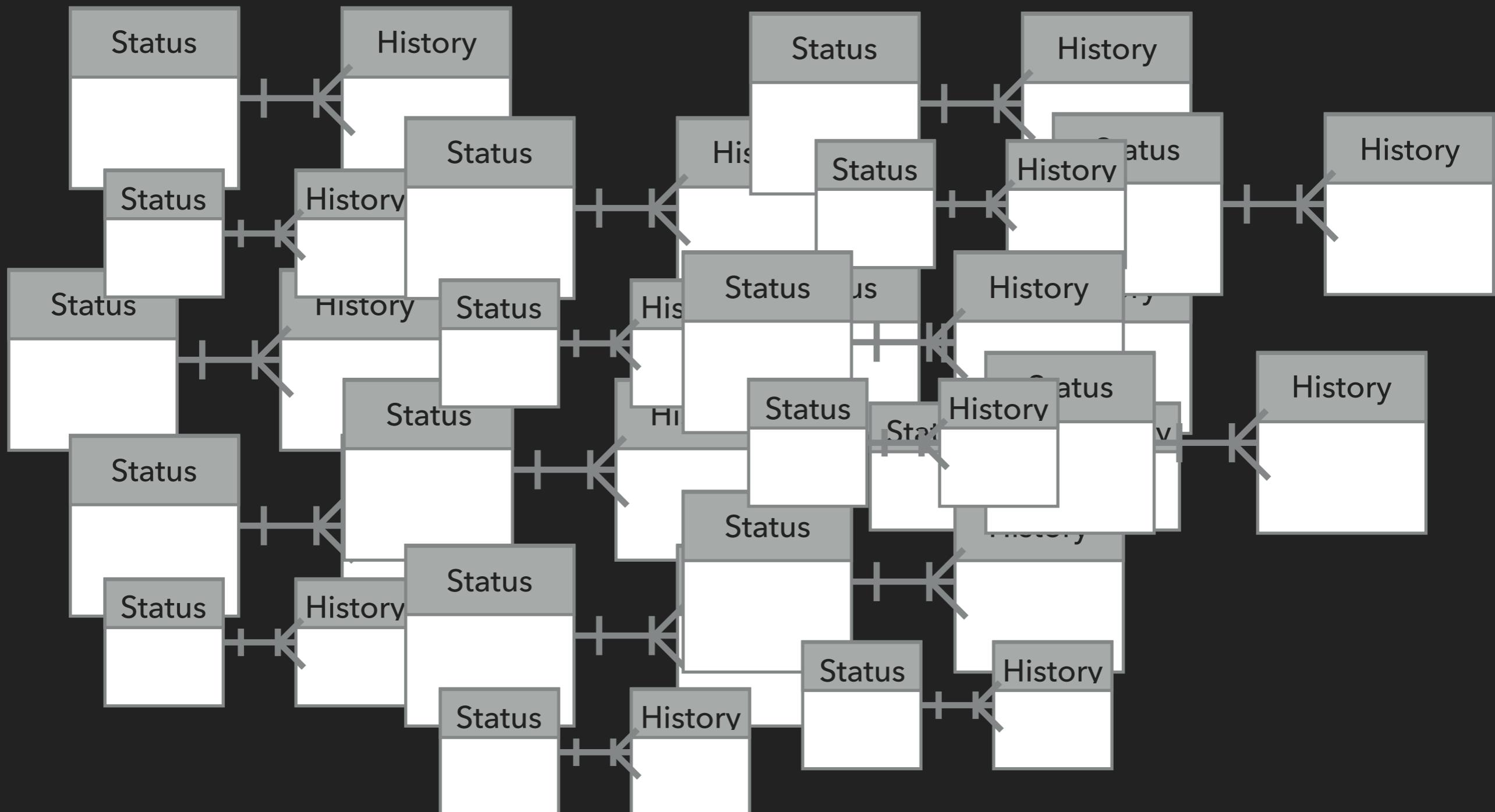
update table_name
set balance=balance+1000,
last_update=now()
where account_id = 12345



seq	account_id	amount	update_date
1	12345	1000	20170103145510
2	12345	5000000	20170103160010
3	23456	2000	20170103183010
.	.	.	.

전통적인 데이터 저장 방법

비지니스가 확장 될 수록 이력 테이블은 점점 증가하고 복잡해짐



2. 모든 상태변화를
Event로 관리

Event 특징 & 구성

특징

- Immutable
- Append Only
- (No! UPDATE / DELETE)

구성

- 이벤트 식별자 (Aggregate Id)
- 이벤트 타입
- 버전,
- 발생시간
- 내용(payload)

관계가 없는
매우 Simple한 구성!

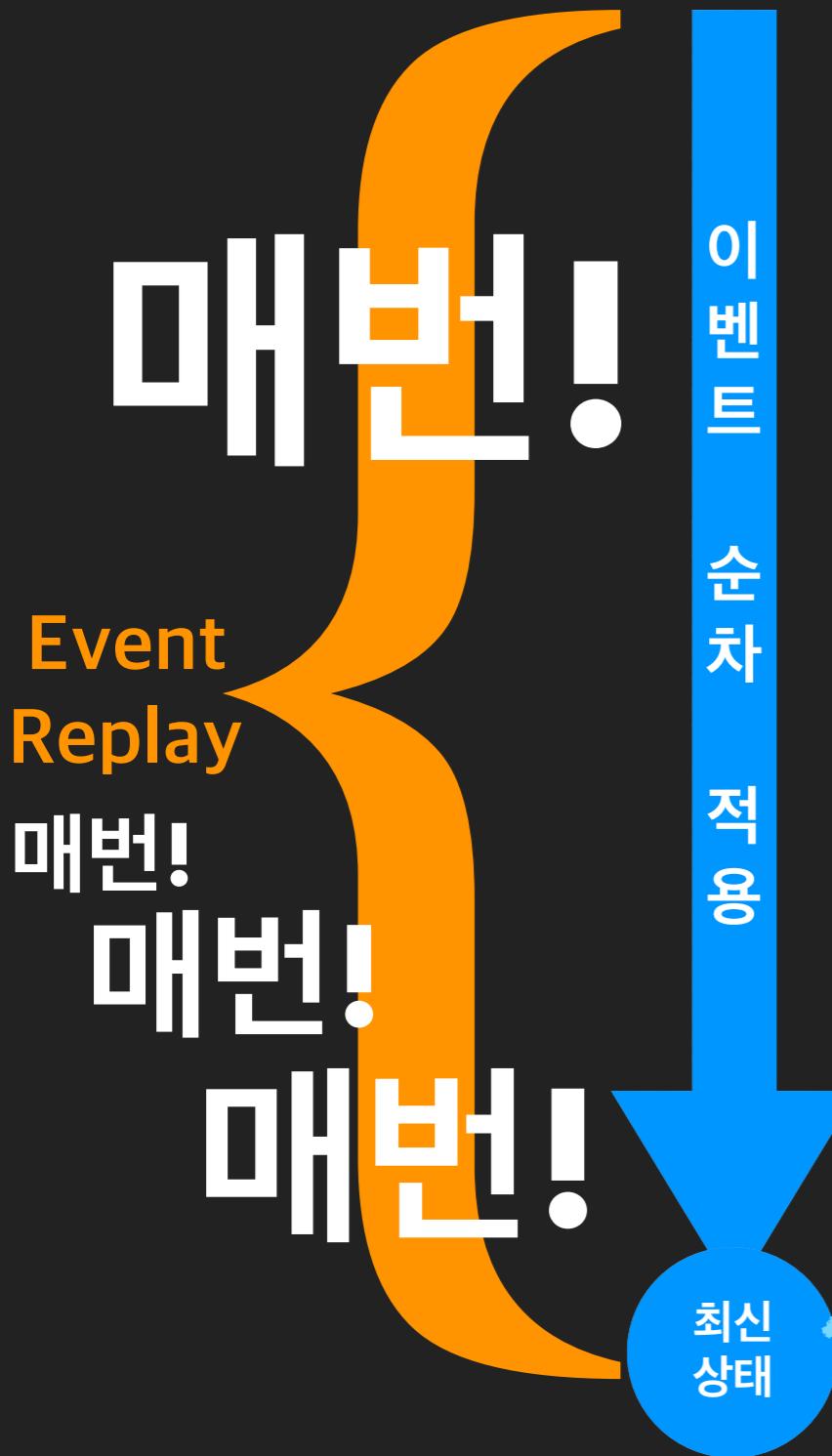
3. 모든 Event는 순서대로
영구 저장소에 저장

이벤트 저장



이벤트 조회 (객체 로딩)

도메인 객체(Aggregate)의 이벤트 핸들러 메소드



```
// 계좌생성
apply(AccountCreated event) {
    this.accountId = event.getAccountId();
    this.balance = event.getBalance();
    this.stopped = false;
}

// 입금
apply(AccountDeposited event) {
    this.balance = event.deposit();
}

// 출금
apply(AccountWithdrawn event) {
    this.balance = event.withdraw();
}

// 계정정지
apply(AccountBlocked event) {
    this.stopped = true;
}
```

Account 객체

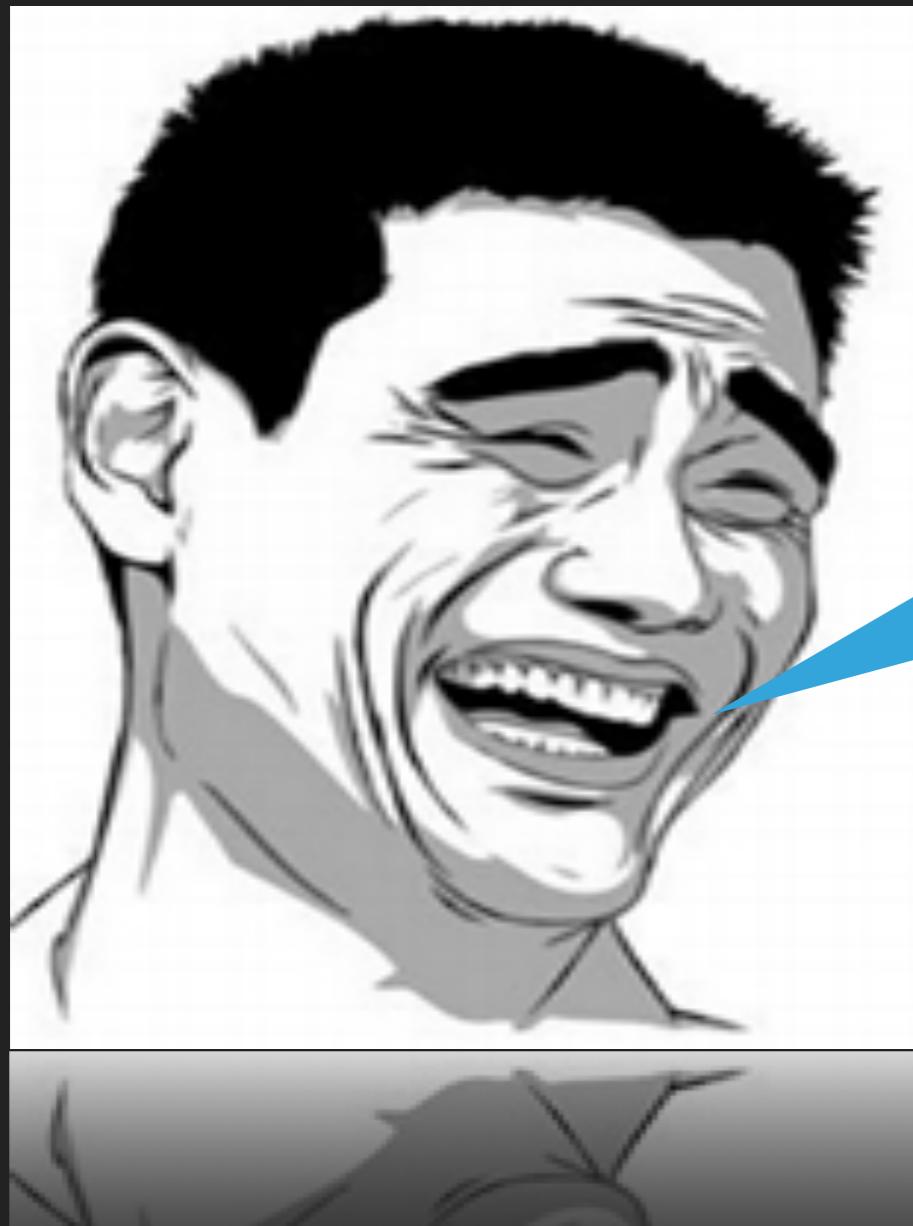
accountId = 123456
balance = 1000
stopped = true

```
apply(AccountDeposited event) {
    this.stopped = true;
}
```

저기요 잠시만요~

어때요?

네! 어때요~



ㅋㅋㅋㅋㅋ

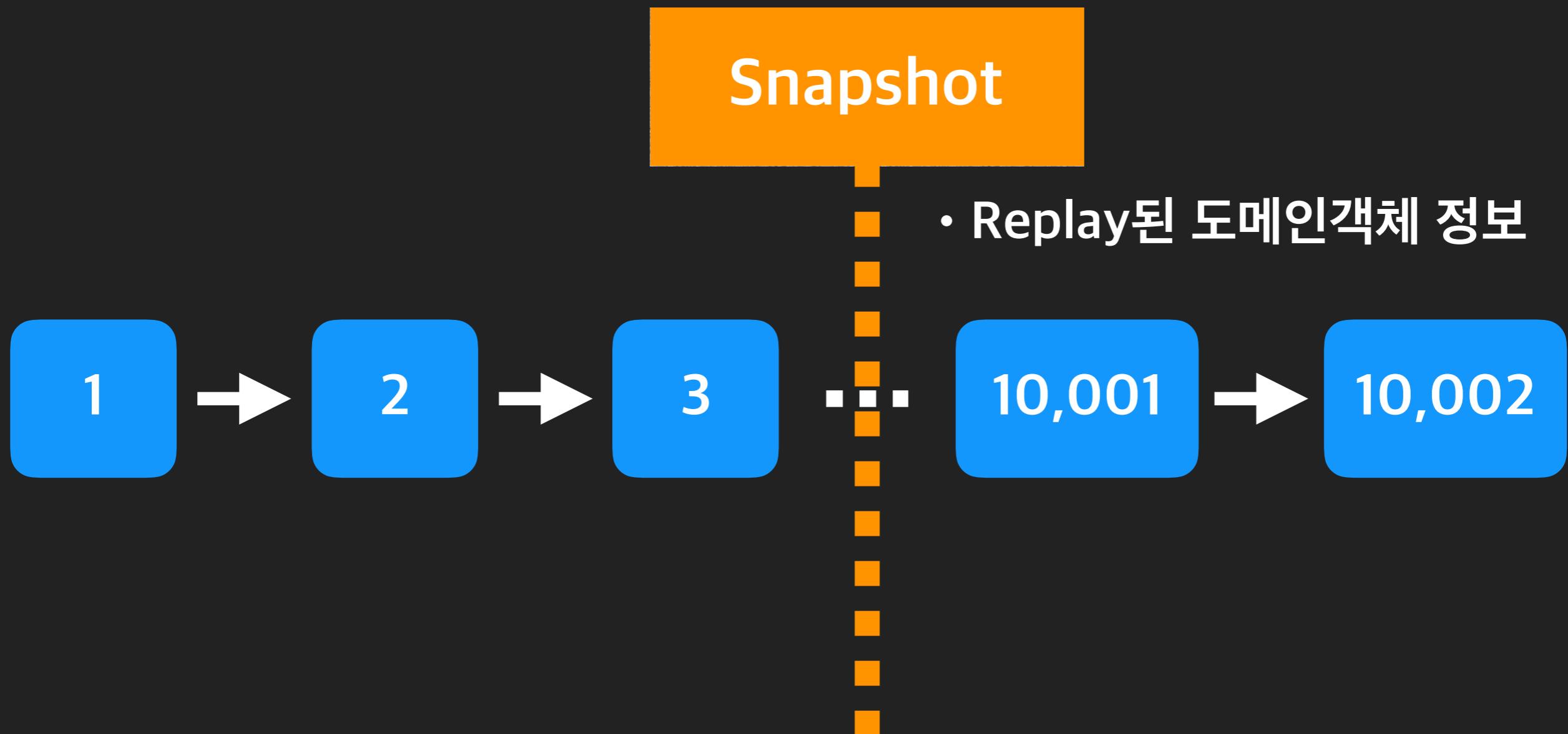
Event Sourcing 성능

- 특정 계좌에 10,000개 이상의 이벤트가 저장되어 있다면….
- 계좌의 현재 상태를 알기 위해선 매번 10,000개 이상의 이벤트를 replay해야 함

처리시간



Event Sourcing 성능



Snapshot 구성

구성

- 이벤트 식별자 (Aggregate Id)
- 이벤트 version (replay된 마지막 Ver.)
- Replay된 도메인 객체

- 이벤트 저장시 생성
- 시간(time) or 이벤트 발생 개수를 기준으로 생성
- Snapshot은 EventStore와 분리된 저장소에 저장
 - 성능상 주로 in Memory 저장소를 사용함

Event Sourcing의 성능

100만개 계좌를 대상으로, 최근 일주일안에
거래정지된 계좌찾기

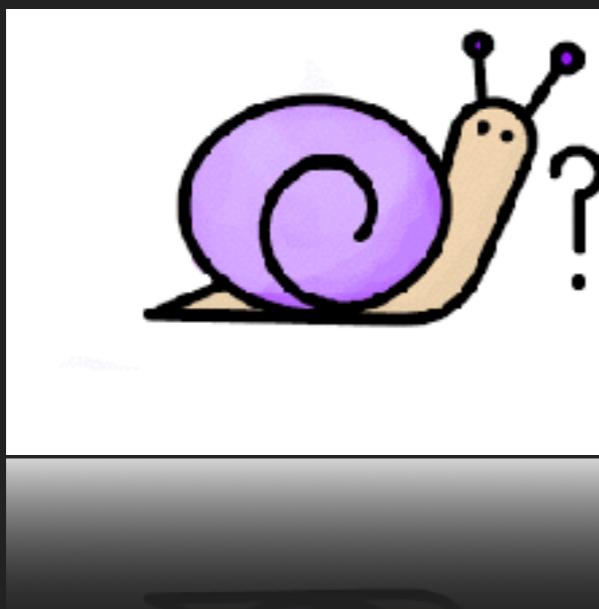
1번 객체
로딩,검사

2번 객체
로딩,검사

3번 객체
로딩,검사

.....

100만번 객체
로딩,검사



겁나게 느린 조회 속도

이벤트 소식에는 진미방의 양꼬개은...

치맥개은 환상의 궁합

CORS

가 있다.



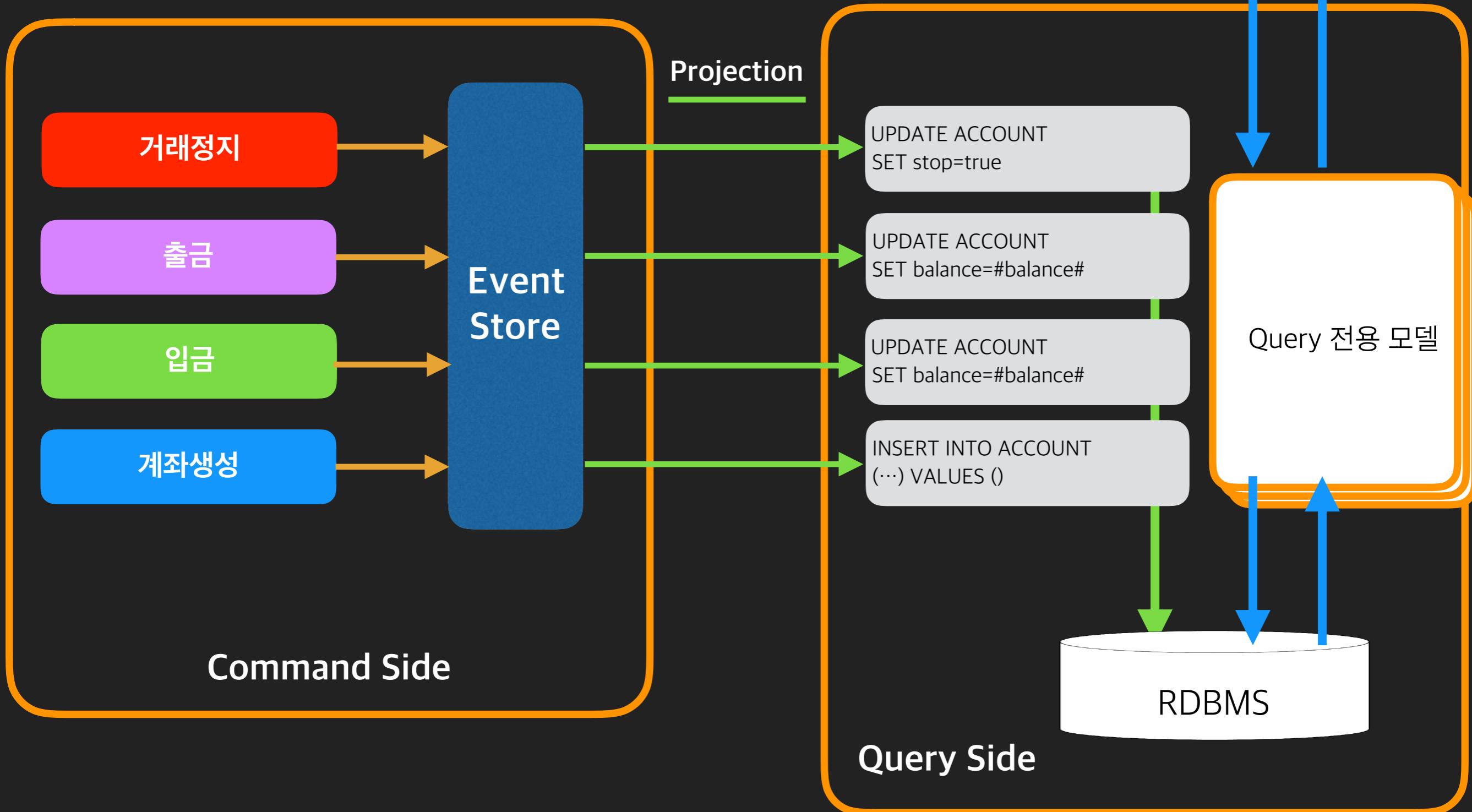
CQRS Command Query Responsibility Segregation

- 명령과 조회의 책임 분리
- Commands -> Writes
- Queries -> Read
- 상태변경을 처리하는 **Command Model**와 데이터 조회 **Query Model**을 분리 구현
- 이벤트 소싱에선 사실상 필수

Event Projection

계좌조회 UI / REST API

조회(Query) 전용 모델을 구성하여 조회 속도 향상



Event Sourcing 구현

주의! 매우 지루 할 수 있음....

Event Sourcing 처리 흐름



ES & CQRS 주요 구현

직접 Source^을 보시는게 더 이해가 빠르시겠군?

<https://github.com/jaceshim/springcamp2017>

Plz...

Click "★"



Command Model 구현

- Request Parameter Mapping
- Validation

```
public ResponseEntity<Product> changeName(@PathVariable final Long productId,  
@RequestBody @Valid ProductCommand.ChangeName productChangeNameCommand,  
BindingResult bindingResult) {  
    if (bindingResult.hasErrors()) {  
        throw new InvalidRequestException("Invalid Parameter!", bindingResult);  
    }  
    final Product product  
        = productService.changeName(productId, productChangeNameCommand);  
    return new ResponseEntity<>(product, HttpStatus.OK);  
}
```

Command Model 구현

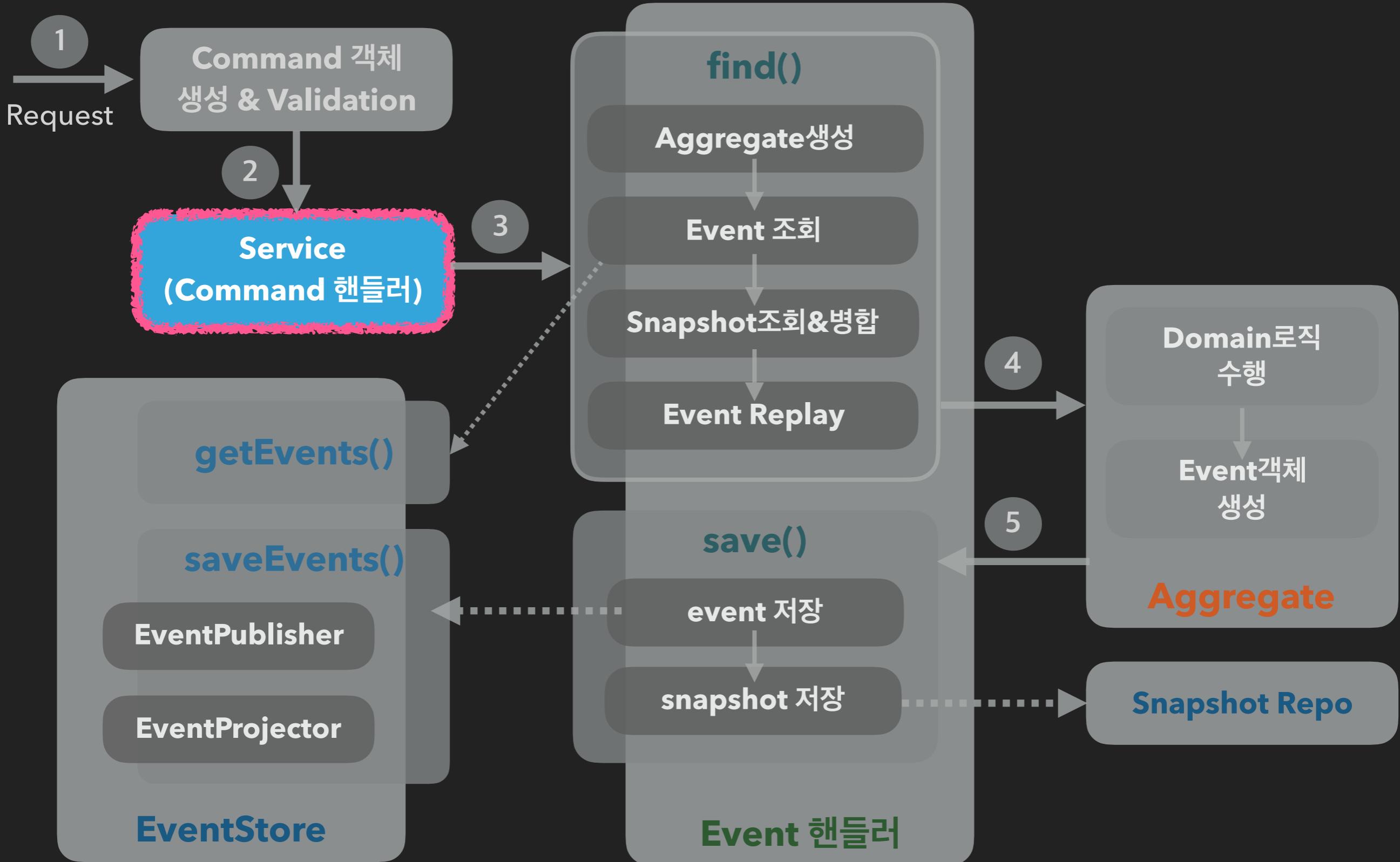
```
@Getter  
@NoArgsConstructor  
public static class CreateProduct {  
    /** 상품 아이디 */  
    private Long productId;  
    /** 상품 명 */  
    @NotNull @Size(min = 1)  
    private String name;  
    /** 상품 가격 */  
    @Min(1)  
    private int price;  
    중략...
```

가능한 불변객체

즉 **public Setter**는 구현하지 않음

```
public CreateProduct(Long productId, String name, int price, int quantity, . . .  
    this.productId = productId;  
    this.name = name;  
    this.price = price;  
    this.quantity = quantity;  
    this.imagePath = imagePath;  
    this.description = description; } } }
```

Event Sourcing 처리 흐름



Service 구현

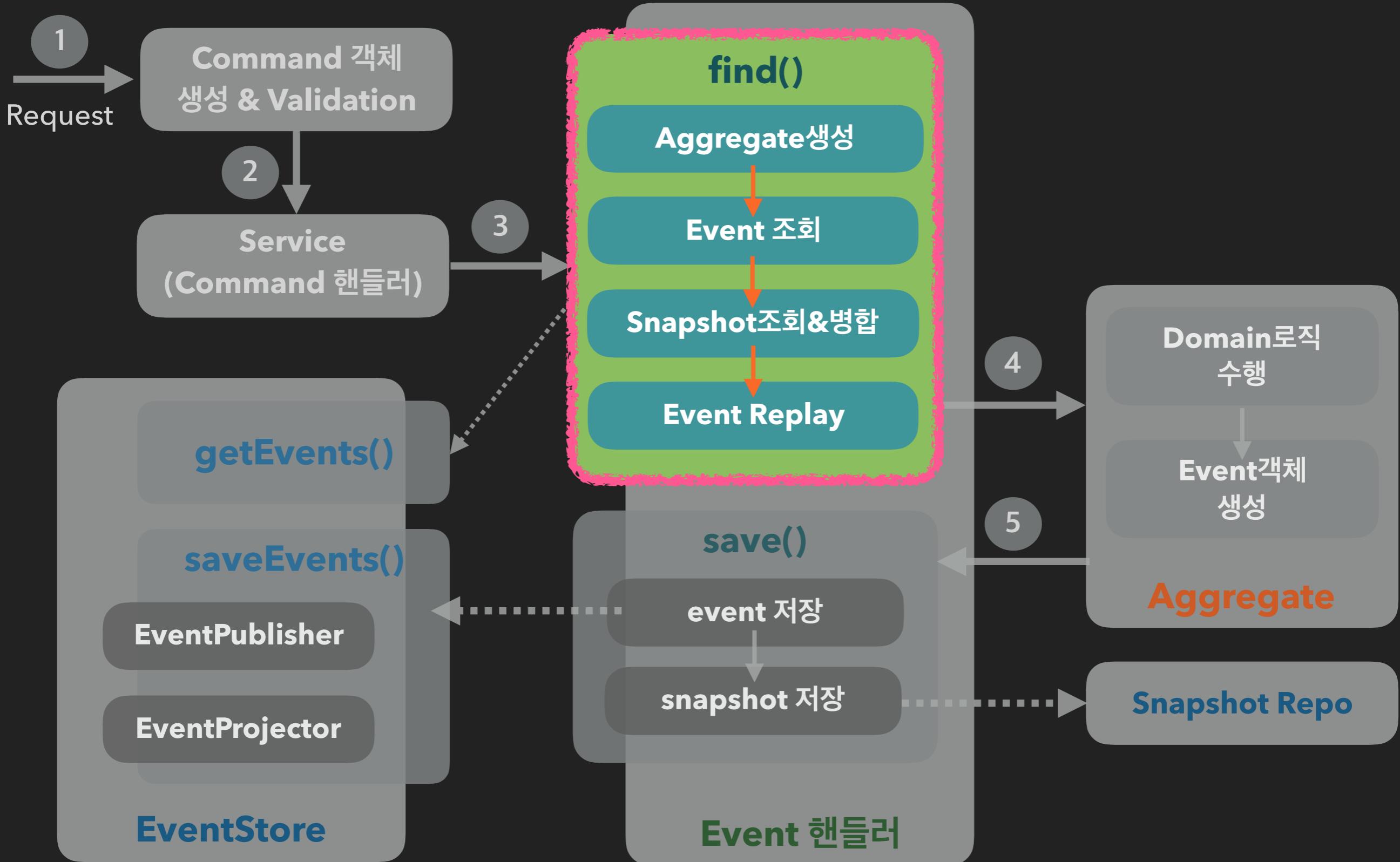
- Command Handler 개념으로 구현
- Domain 객체 생성 or 로딩 및 Domain 로직 호출
- EventHandler 호출

```
public Product createProduct(ProductCommand.CreateProduct productCreateCommand) {  
  
    // DB sequence를 통해서 유일한 productId값 생성  
    final Long productId = productEventStoreRepository.createProductId();  
  
    final Product product = Product.create(productId, productCreateCommand);  
  
    productEventHandler.save(product);  
  
    return product;  
}
```

이벤트 저장

도메인 객체 생성 (Aggregate 생성)
도메인 로직 호출

Event Sourcing 처리 흐름



EventHandler 구현 - find()

- 도메인 객체(Aggregate) 생성 / Event 조회 / Snapshot 조회 & 병합

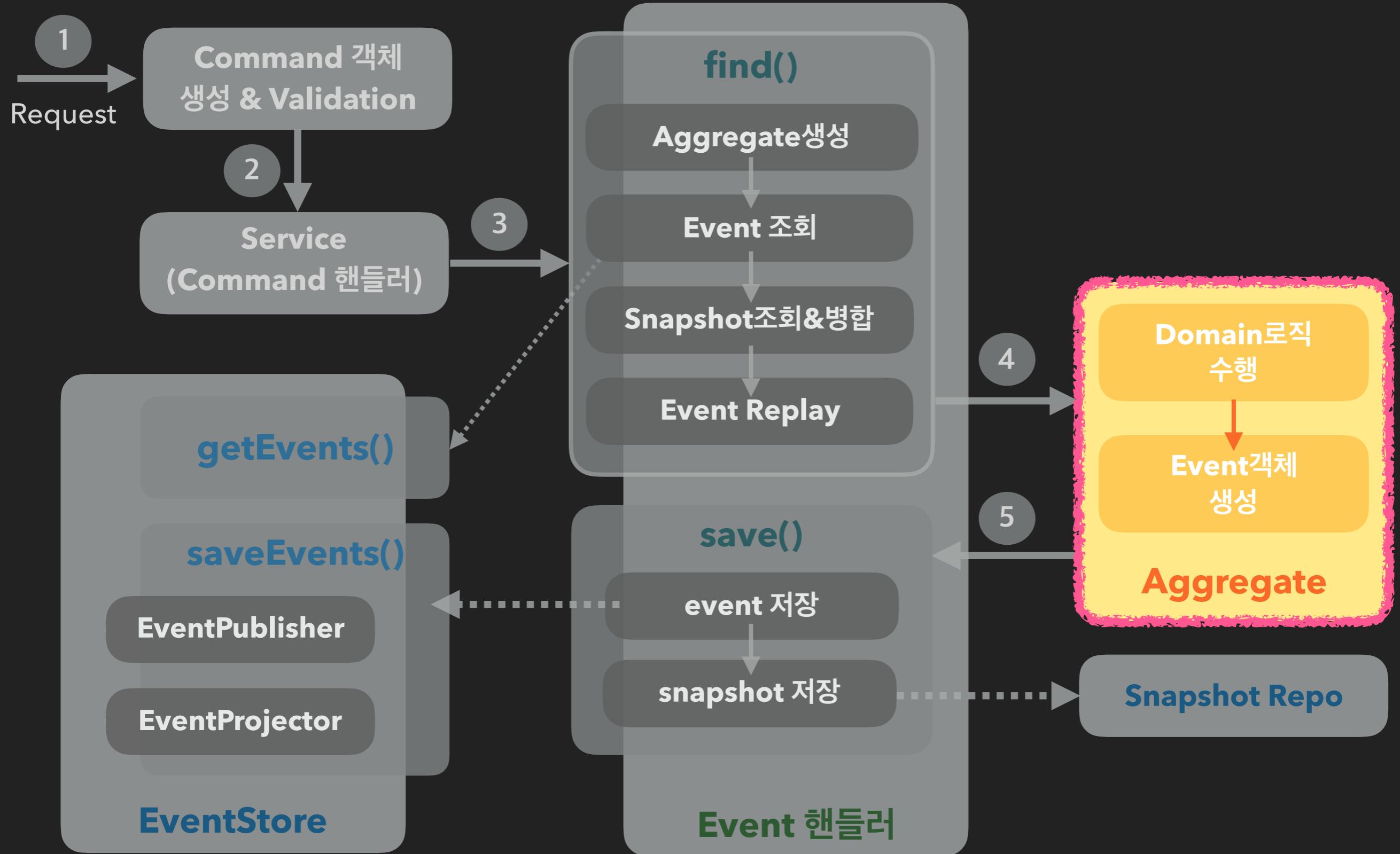
```
public class MemberEventHandler extends AbstractEventHandler<Member, String> {  
    public A find(ID identifier) {  
        // 리플렉션을 통해서 AggregateRoot 객체 생성  
        A aggregateRoot = createAggregateRootViaReflection(identifier);  
  
        Optional<Snapshot<A, ID>> retrieveSnapshot = retrieveSnapshot(identifier);  
        List<Event<ID>> baseEvents;  
        if (retrieveSnapshot.isPresent()) {  
            Snapshot<A, ID> snapshot = retrieveSnapshot.get();  
            baseEvents = eventStore.getEventsByAfterVersion(snapshot.getIdentifier(), snapshot.getVersion());  
            // 생성된 aggregateRoot를 snapshot에 저장된 aggregateRoot 객체로 대체  
            aggregateRoot = snapshot.getAggregateRoot();  
        } else {  
            // 모든 event 목록 호출  
            baseEvents = eventStore.getEvents(identifier);  
        }  
  
        if (baseEvents == null || baseEvents.size() == 0) {  
            return null;  
        }  
        // 이벤트 replay 호출  
        aggregateRoot.replay(baseEvents);  
  
        return aggregateRoot;  
    }
```

클래스 파라미터 타입의 Aggregate 생성

snapshot 호출

snapshot version 이후 이벤트만 호출

Event Sourcing 처리 흐름



Aggregate 구현 - AbstractAggregateRoot

- 연관된 Entity와 Value Object의 묶음
- 데이터 변경시 한 단위로 처리됨
- 이벤트를 Domain객체에 반영하는 handler method 구현

```
public abstract class AggregateRoot<ID> implements Serializable {  
    private ID identifier;  
    private List<Event> changeEvents = Lists.newArrayList();  
    private Long expectedVersion = 0L;  
    ...  
}
```

Aggregate 식별자 Type Parameter

객체상태 일관성 보장을 위한 Version

전통적인 방식으로
상품의 수량을 동시에 변경한다면...



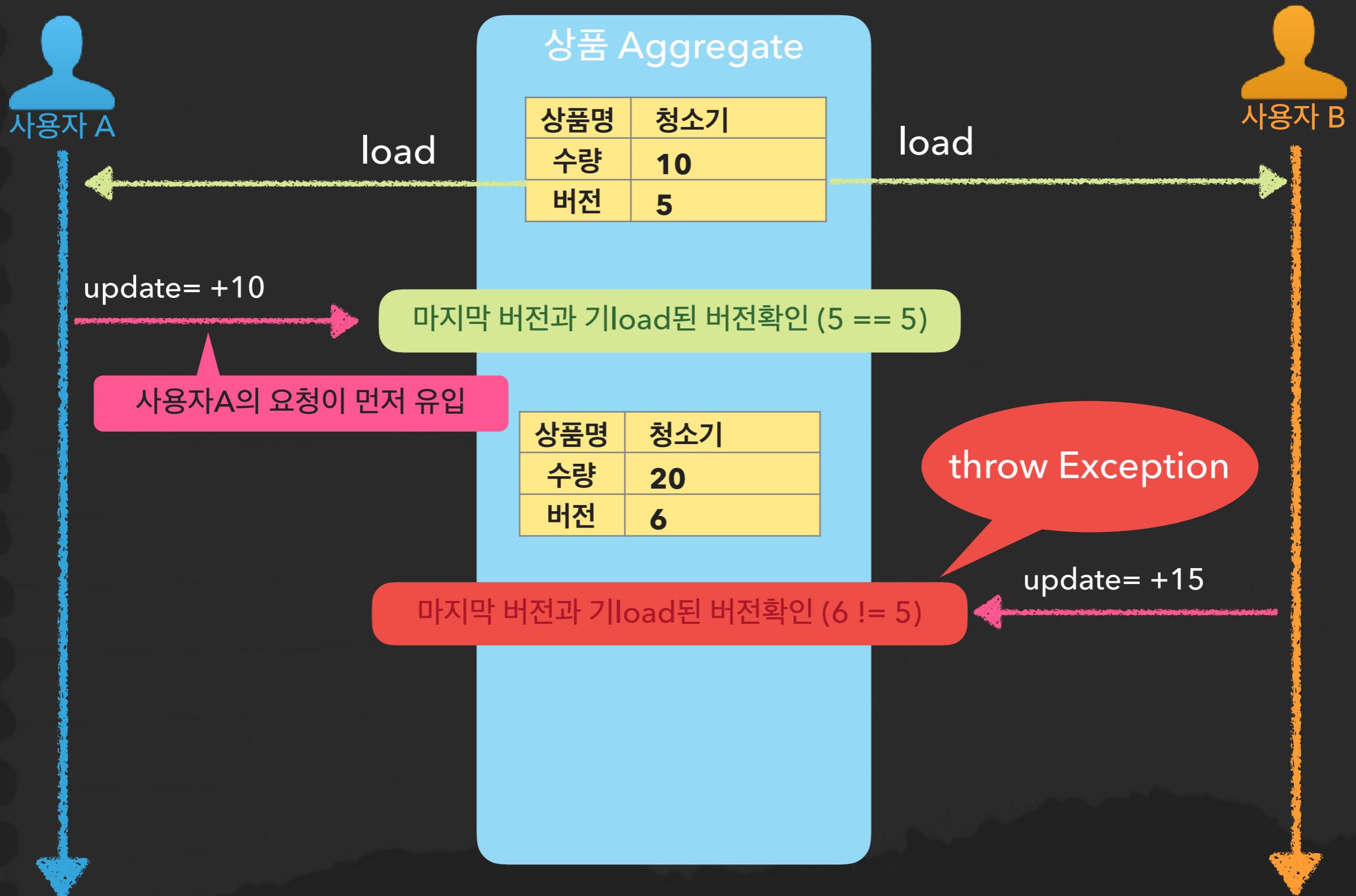
107H에서 157H 추가했는데 357H가 나왔어....

이건 어느나라 산수이야?



니콜라스 사용자B

이벤트 소싱에서
상품의 수량을 동시에 변경한다면...



Aggregate 구현 - event replay

```
public void replay(List<Event> changes) {  
    for (Event event : changes) {  
  
        applyChange(event, false);  
        this.expectedVersion++;  
    }  
  
    private static final String APPLY_METHOD_NAME = "apply";  
    private void applyChange(Event event, boolean isNew) {  
        Method method;  
        try {  
            method = this.getClass().getDeclaredMethod(APPLY_METHOD_NAME, event.getClass());  
            if (method != null) {  
                method.setAccessible(true);  
                method.invoke(this, event);  
            }  
  
            if (isNew) {  
                changeEvents.add(event);  
            }  
        } catch (IllegalAccessException | IllegalArgumentException  
                | InvocationTargetException | NoSuchMethodException e) {  
            log.error(e.getMessage(), e);  
            throw new EventApplyException(e.getMessage(), e);  
        }  
    }  
}
```

EventHandler에서 호출함

// 이벤트 replay 호출
aggregateRoot.replay(baseEvents);

Event발생시점에만 true이고 replay시에는 false

Replay시 version 순차적으로 증가처리

이벤트 핸들러 메소드명은 모두 apply로 정의됨.

true는 이벤트가 발생한 경우이고 저장된 이벤트 목록에 추가.

Aggregate 구현 - AbstractAggregateRoot 구현체

- Domain 로직 구현 / 이벤트 객체 생성

```
public class Order extends AggregateRoot<Long> {  
    /** 주문번호 */  
    private Long orderId;  
  
    /** 주문회원 */  
    private Member orderMember;  
  
    /** 배송정보 */  
    private Delivery delivery;  
    중략...
```

가능한 public Setter는 구현하지 않음

연관된 Entity나 Value Object

```
/** 주문 이벤트 반영 */  
public void apply(OrderCreated orderCreated) {  
    this.orderId = orderCreated.getOrderId();  
    this.orderMember = orderCreated.getOrderMember();  
    this.delivery = orderCreated.getDelivery();  
    this.orderItems = orderCreated.getOrderItems();  
}
```

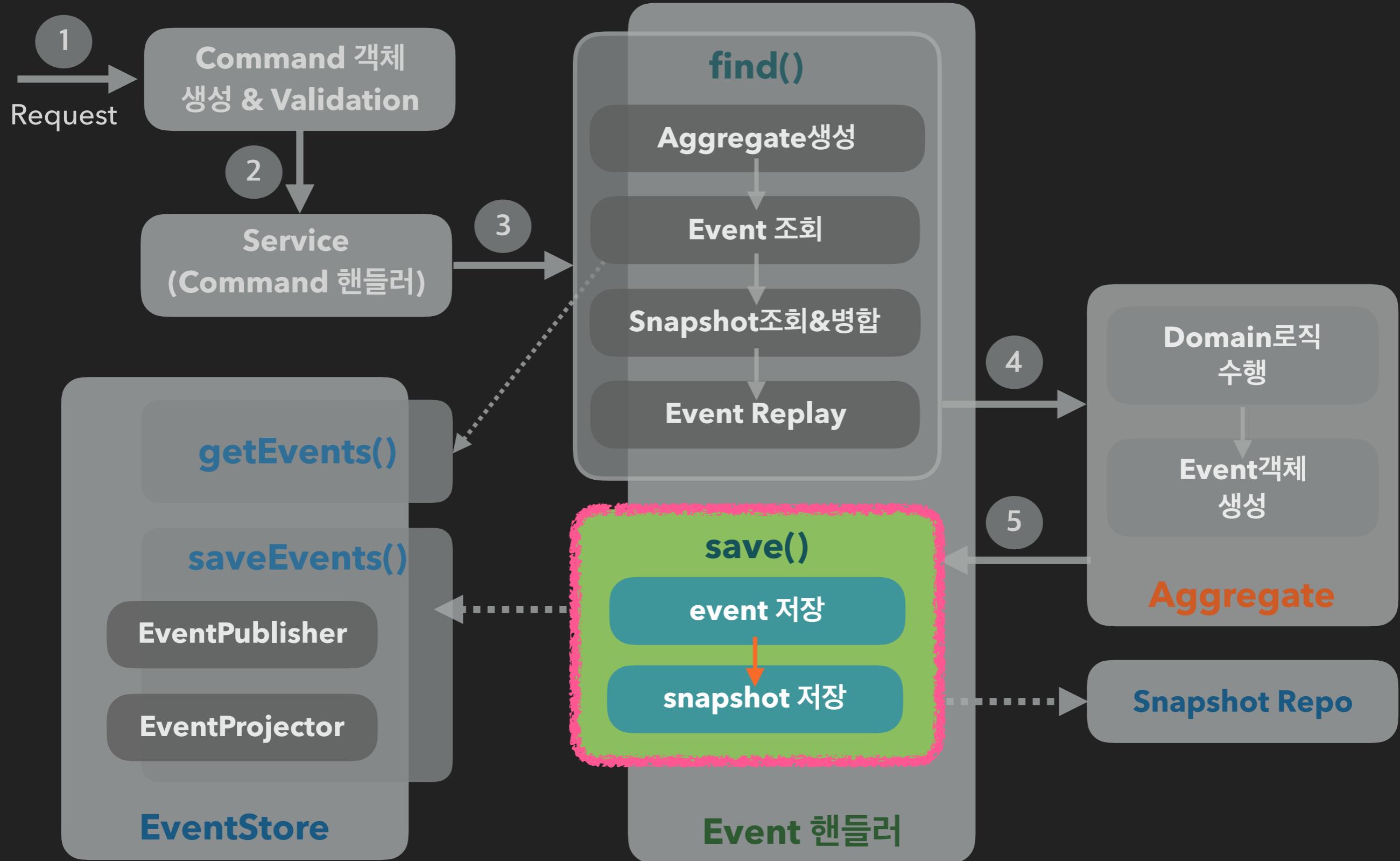
각 이벤트별로 존재하는 이벤트 적용 apply()

비지니스 로직 메소드 구현

```
Order(Long orderId, Member orderMember, Delivery delivery, List<OrderItem> checkoutItems) {  
    super(orderId);  
    // some biz logic  
    중략...  
}  
  
applyChange(new OrderCreated(orderId, orderMember, delivery, orderItems, created));  
}
```

비지니스 처리시 도메인객체의 상태를 변경하지 않는다.

Event Sourcing 처리 흐름



EventHandler 구현

- Aggregate에서 발생한 이벤트 저장 처리

영구 저장소에 저장

```
public void save(A aggregateRoot) {  
    final ID identifier = (ID) aggregateRoot.getIdentifier();  
    // 이벤트 저장소에 이벤트 저장  
    eventStore.saveEvents(identifier,  
        aggregateRoot.getExpectedVersion(), aggregateRoot.getUncommittedChanges());  
}
```

// 저장대상 이벤트 목록 clear

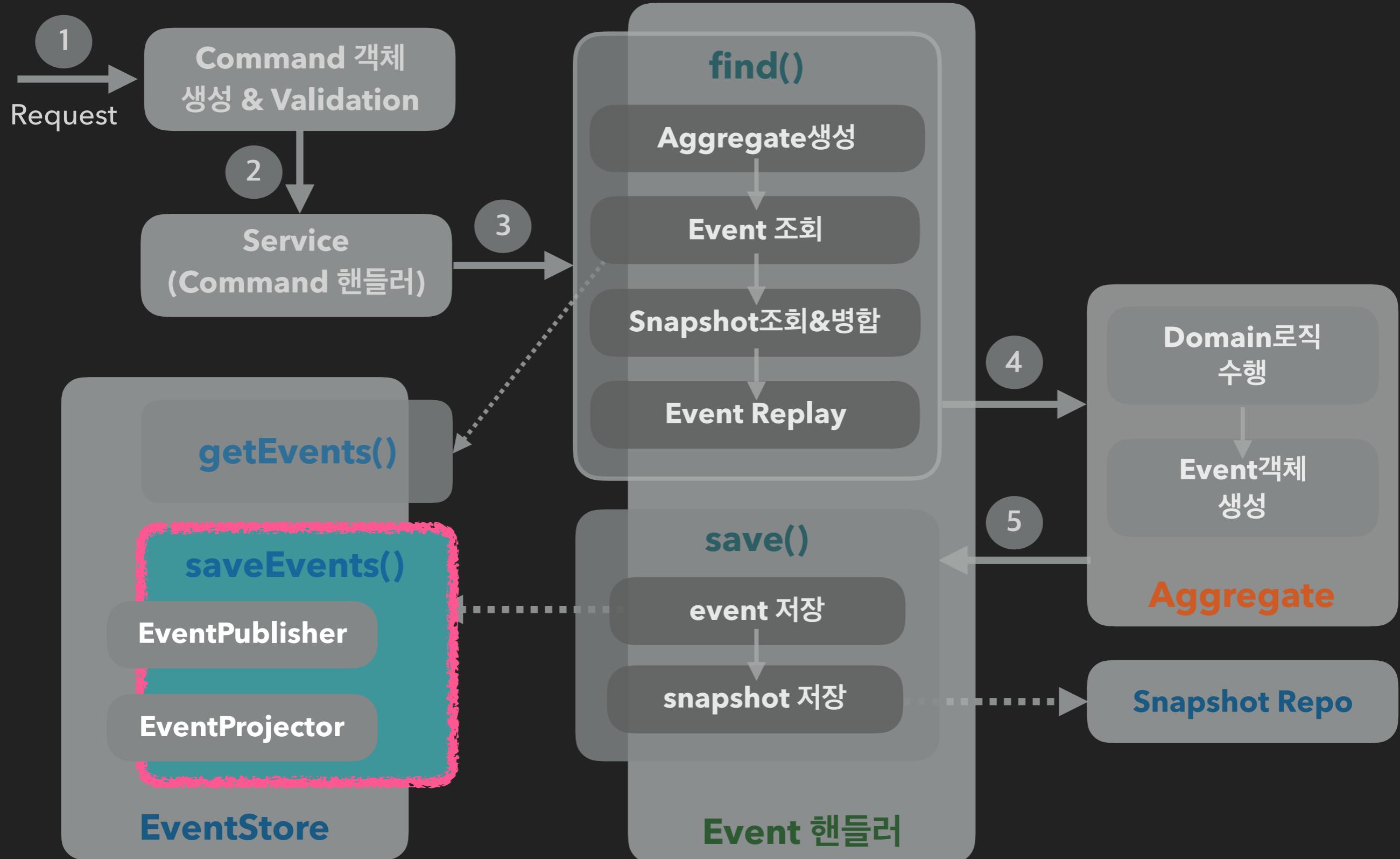
```
aggregateRoot.markChangesAsCommitted();  
AtomicInteger snapshotCount = snapshotCountMap.get(identifier);  
if (snapshotCount == null) {  
    snapshotCount = new AtomicInteger(0);  
}  
}
```

snapshot 생성 정책에 따라 저장

```
if (snapshotCount.get() == SNAPSHOT_COUNT) {  
    log.debug("{} snapshot count {}", identifier, snapshotCount.get());  
    Snapshot<A, ID> snapshot =   
        new Snapshot<>(identifier, aggregateRoot.getExpectedVersion(), aggregateRoot);  
    snapshotRepository.save(snapshot);  
    snapshotCount.set(0);  
  
    return;  
}  
}
```

```
final int increaseCount = snapshotCount.incrementAndGet();  
log.debug("{} snapshot increase count {}", increaseCount);  
}
```

Event Sourcing 처리 흐름



EventStore 구현

```
public class ProductEventStore implements EventStore<Long> {
```

```
@Autowired
```

```
private ProductEventStoreRepository productEventStoreRepository;
```

Event 영구 저장소 액세스

```
@Autowired
```

```
private EventPublisher eventPublisher;
```

Event 발행처리

```
@Autowired
```

```
private EventProjector eventProjector;
```

Event Projection 처리

중략...

EventStore 구현 - saveEvents()

```
public void saveEvents(final Long identifier, Long expectedVersion, final List<Event> events) {  
    // 신규 등록이 아닌 경우 version확인 후 처리  
    if (expectedVersion > 0) {  
        List<ProductRawEvent> productRawEvents = productEventStoreRepository.findByIdentifier(identifier);  
        Long actualVersion = productRawEvents.stream()  
            .sorted(Comparator.comparing(ProductRawEvent::getVersion))  
            .findFirst().map(ProductRawEvent::getVersion)  
            .orElse(-1L);  
  
        // 버전이 맞지 않는 경우 처리 안함.  
        if (expectedVersion != actualVersion) {  
            String exceptionMessage = String.format("Unmatched Version : expected: {}, actual: {}", expectedVersion,  
                actualVersion);  
            throw new IllegalStateException(exceptionMessage);  
        }  
    }  
    중략...  
}
```

version이 0인 경우 Aggregate의 첫번째 Event임

EventStore 구현 - saveEvents()

```
for (Event event : events) {○
    String type = event.getClass().getName();○
    String payload = null;○

    try {○
        payload = objectMapper.writeValueAsString(event);○
        log.debug("-> event payload : {}", payload);○
    } catch (JsonProcessingException e) {○
        log.error(e.getMessage(), e);○
    }○

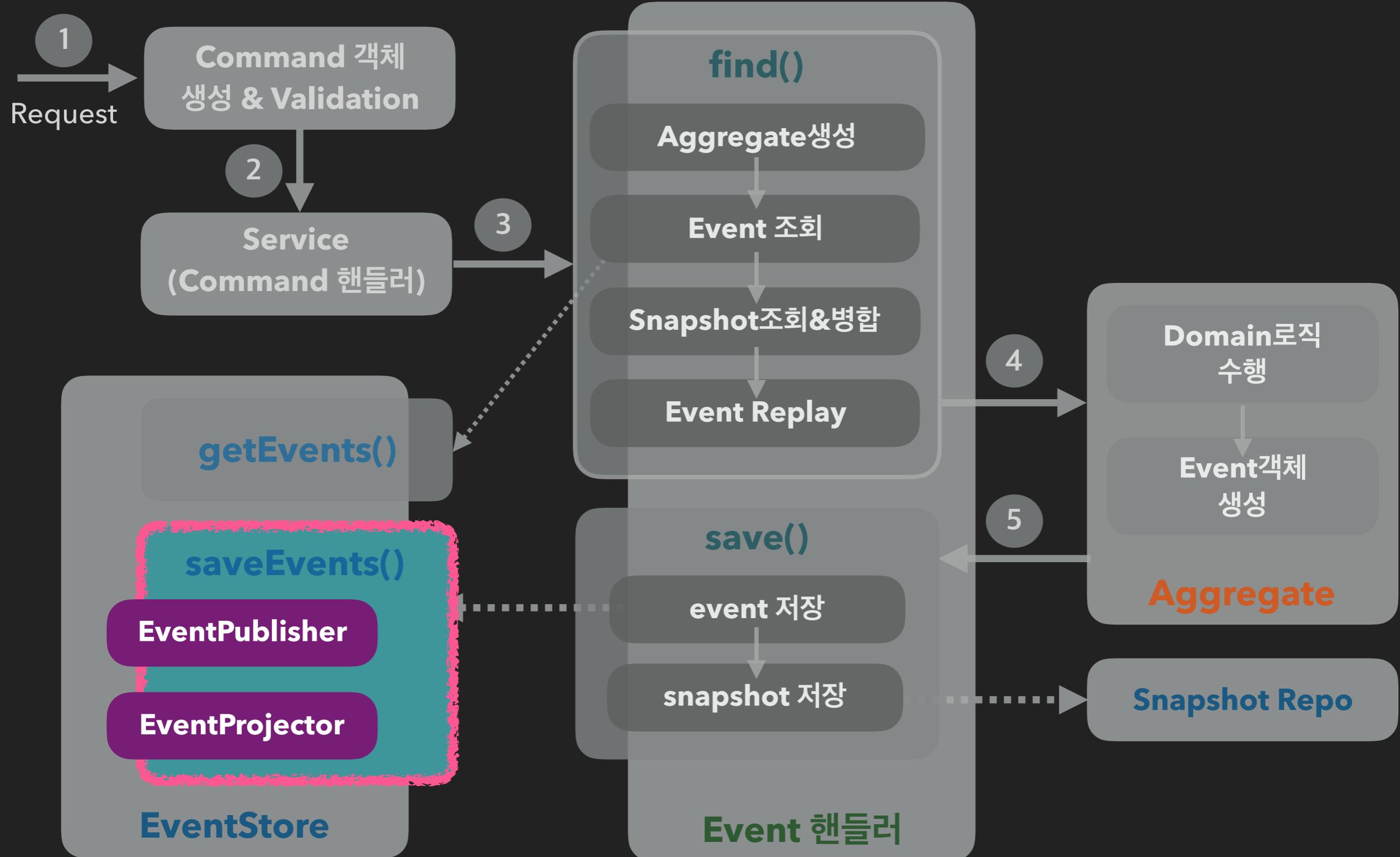
    expectedVersion++;○
    LocalDateTime now = LocalDateTime.now();○
    ProductRawEvent productRawEvent = new ProductRawEvent(identifier, type, expectedVersion,
    payload, now);○

    // event를 영구 저장소에 저장
    productEventStoreRepository.save(productRawEvent);○

    // event 발행
    eventPublisher.publish(productRawEvent);○

    // event projection
    eventListener.handle(event);○
}
```

Event Sourcing 처리 흐름



EventPublisher 구현

- Event를 서비스 외부로 발행
- 일반적으로 Message Broker(Queue)로 Producing

```
public class ProductEventPublisher implements EventPublisher {  
    @Autowired  
    private ObjectMapper objectMapper;  
  
    @Autowired  
    KafkaTemplate<String, String> kafkaTemplate;  
  
    @Value("${kafka.bootstrap.servers}")  
    private String kafkaBootstrapServers;  
  
    @Value("${kafka.product.topic}")  
    private String productKafkaTopic;  
  
    @Async  
    @Override  
    public void publish(RawEvent event) {  
        if (event == null) {  
            return;  
        }  
  
        try {  
            final String sendMessage = objectMapper.writeValueAsString(event);  
            kafkaTemplate.send(productKafkaTopic, sendMessage);  
            log.debug("{} 전송 완료 - {}", this.productKafkaTopic, sendMessage);  
        } catch (final JsonProcessingException e) {  
            log.error(e.getMessage(), e);  
        }  
    }  
}
```



EventProjector 구현

- 생성된 Event를 Query Model전용 DB로 Projection

```
public class ProductEventProjector extends AbstractEventProjector {  
    @Autowired  
    private JdbcTemplate jdbcTemplate;  
  
    /** 상품 등록 projection */  
    public void execute(ProductCreated event) {  
        StringBuilder query = new StringBuilder();  
        query.append("INSERT INTO product (");  
        query.append(" productId, name, price, quantity, imagePath, description, created ");  
        query.append(") VALUES ( ");  
        query.append(" ?, ?, ?, ?, ?, ?, ? ) ");  
  
        jdbcTemplate.update(query.toString(),  
            event.getProductId(),  
            event.getName(),  
            event.getPrice(),  
            event.getQuantity(),  
            event.getImagePath(),  
            event.getDescription(),  
            convertLocalDateToString(event.getCreated()));  
    }  
  
    public abstract class AbstractEventProjector implements EventProjector {  
        protected static String APPLY_METHOD_NAME = "execute";  
  
        @Override  
        public void handle(Event event) {  
            Method method;  
            try {  
                method = this.getClass().getDeclaredMethod(APPLY_METHOD_NAME, event.getClass());  
                if (method != null) {  
                    method.setAccessible(true);  
                    method.invoke(this, event);  
                }  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    } 중략...
```

Event Sourcing 장점

- Bypass 객체/관계 불일치
 - Event는 관계가 없는 매우 단순한 형태
- 모든 변경사항에 대한 완벽한 이력 저장 (영구적)
- 탁월한 주작성과 디버그 용이성
 - 이벤트를 차례대로 검사하면서 버그 원인 추적 가능
- 탁월한 쓰기 성능
 - 이벤트는 저장 operation만 존재하기에 성능이 좋음

Event Sourcing 단점

- 익숙하지 않음
 - SQL위주의 개발성향인 경우 적응하기 힘듬
- 단순 모델(복잡하지 않는 Domain) 적합하지 않음
 - 배 보다 배꼽이 더 클 수 있음 - 구현 함께 많아요….
- 도구부족 & 성숙되지 않은 기술
 - 이벤트 소싱과 CQRS 지원 프레임워크 아직은 부족
 - 최근엔 Axon/Eventuate 등의 (Based Java) Framework이 있음
- 운영시 불편함?
 - 이벤트 데이터는 일반적인 쿼리 방식으로 조회 불가.
 - **CQRS** Read Model 필수

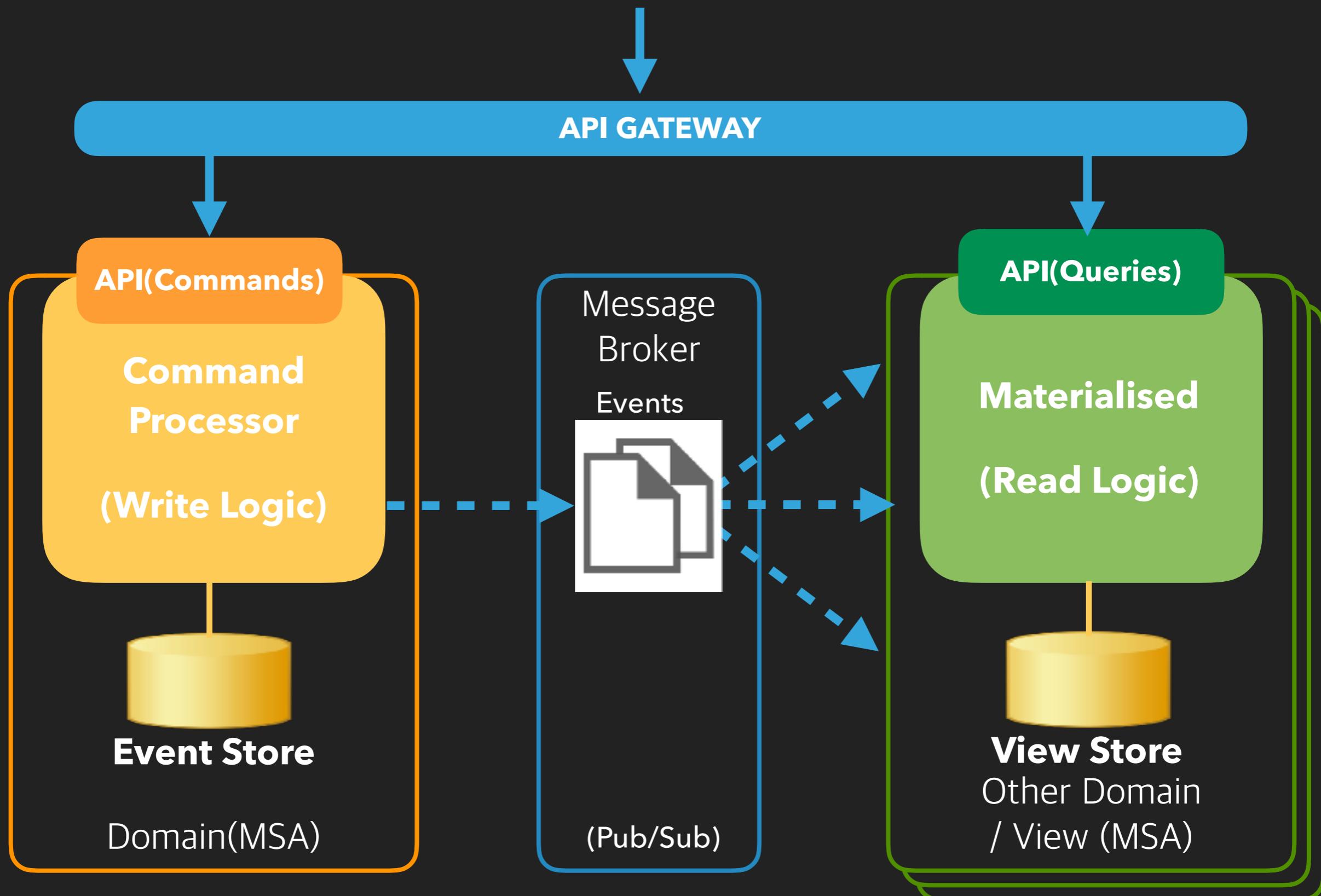
사시ز...

이벤트 소식은

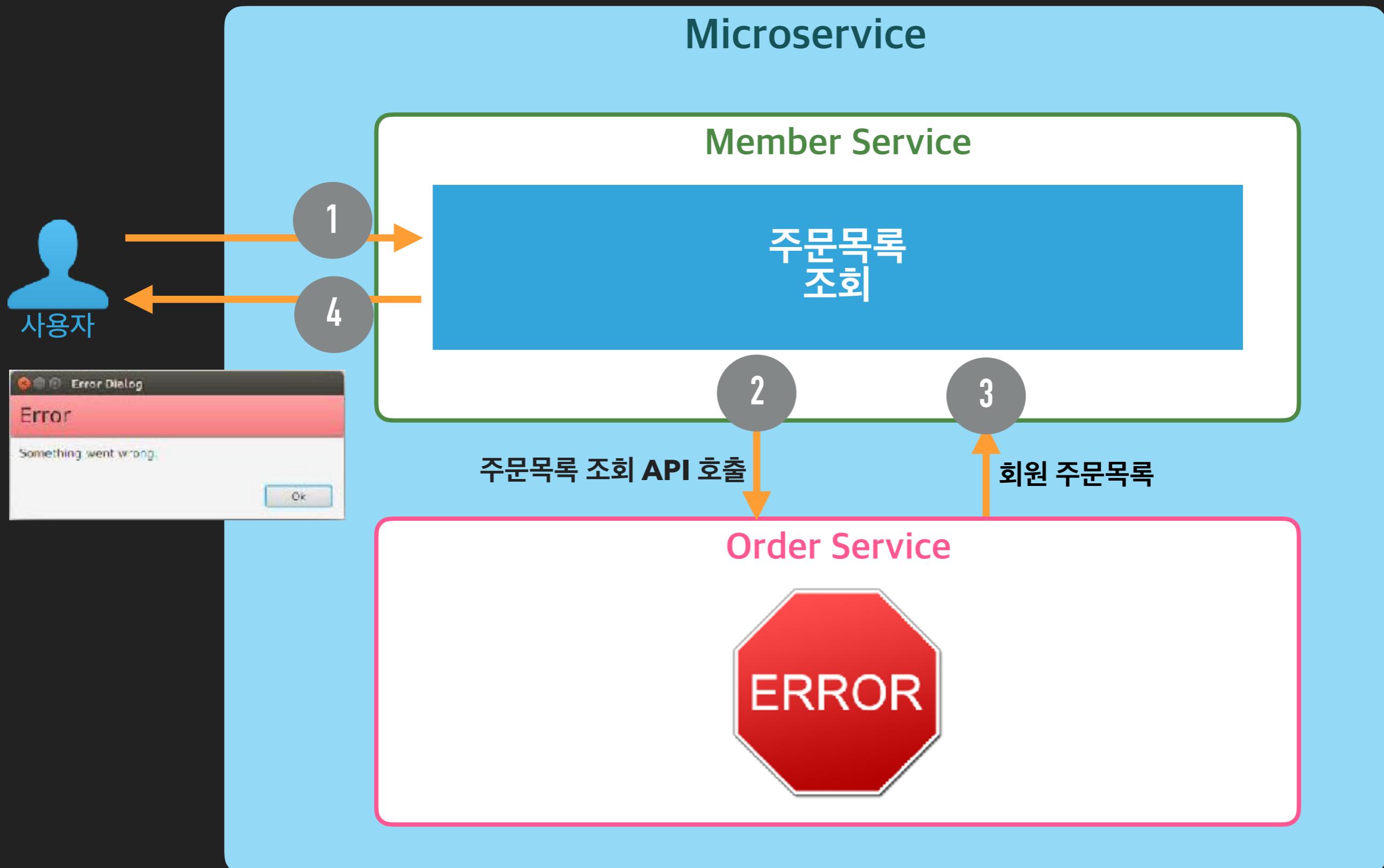
MSA 한경이 아니라면...

적용할 이유가 없어요...

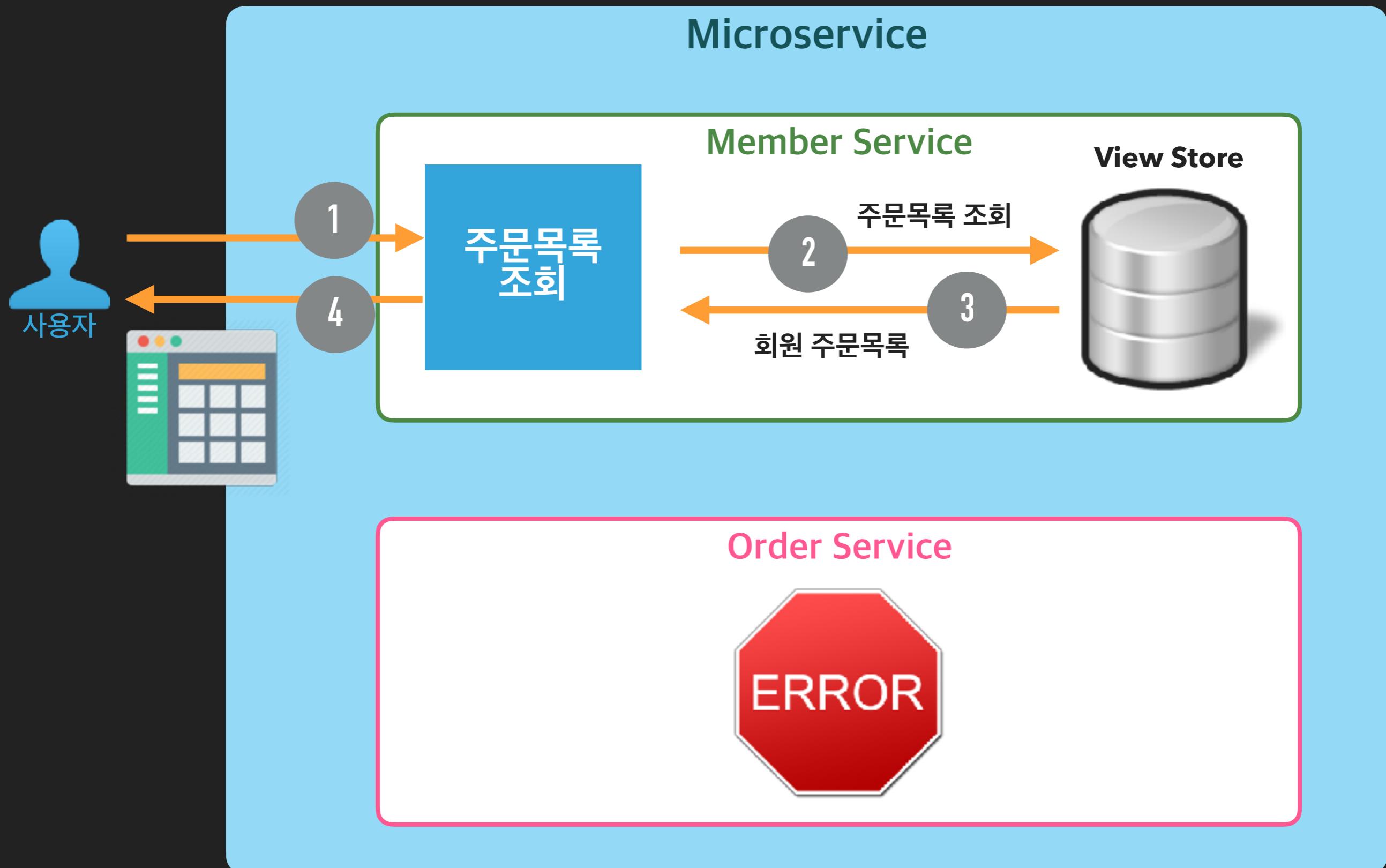
이벤트 소싱 적용된 MSA 구조



MSA 장애전파

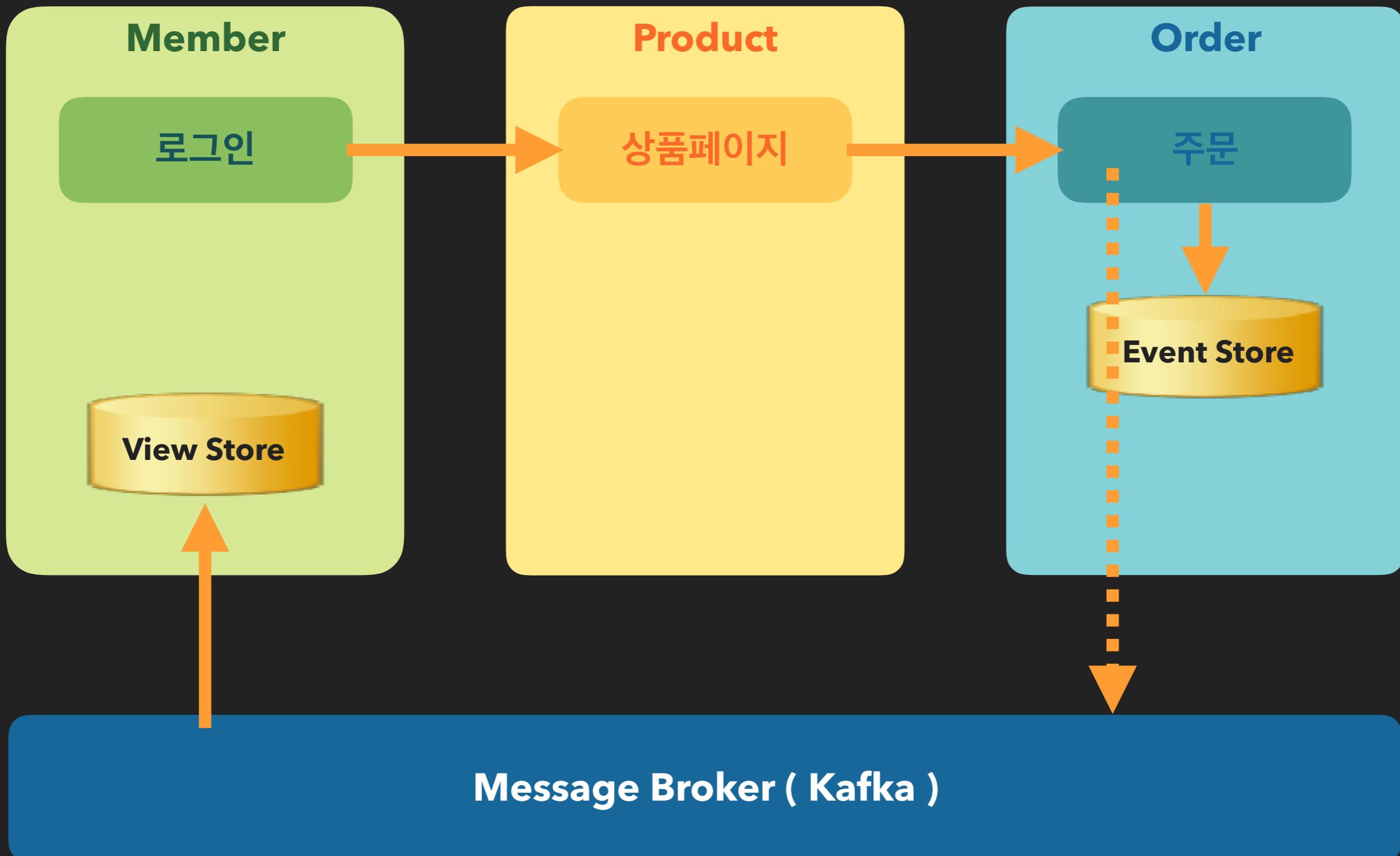


MSA의 Event Projection 활용



Demo

Demo - 구성



No!
Silver Bullet



Q & A