이벤트 소싱과 CQRS

최지우

들어가기 앞서..

- 이벤트 소싱과 CQRS 는 아키텍쳐 독립적입니다.
- **서로 상관없지만**, 같이 자주 엮어서 나와요.
  - 이벤트 소싱은 복잡하고 까다롭고, 서비스에서는 필연적으로 CQRS, 메시지 드리븐 아키텍쳐와 MSA와 같이 사용되는 경우가 많습니다.
- 그러다보니, 설명하는 글들을 보면 보통 세가지(이벤트소싱, CQRS, 메시지 드리븐) 를 엮어서 설명하는 경우가 많습니다.
- 발표에서는 각각에 대한 **정의와 예시, 장점과 단점에 대해서** 다루고 전체적인 숲을 보는 **관점에서 세가지 기술이 합쳐져 어떻게 운용(MSA) 되는지 알아보겠습니다**.

DDD Terminology

- 이벤트 소싱, CQRS 모두 DDD를 통해 나온 개념이 많습니다.
- 따라서 그 전에 DDD에서 나오는 용어들이 많이 나옵니다. 관련 용어들에 대해서 미리 알아볼게요.

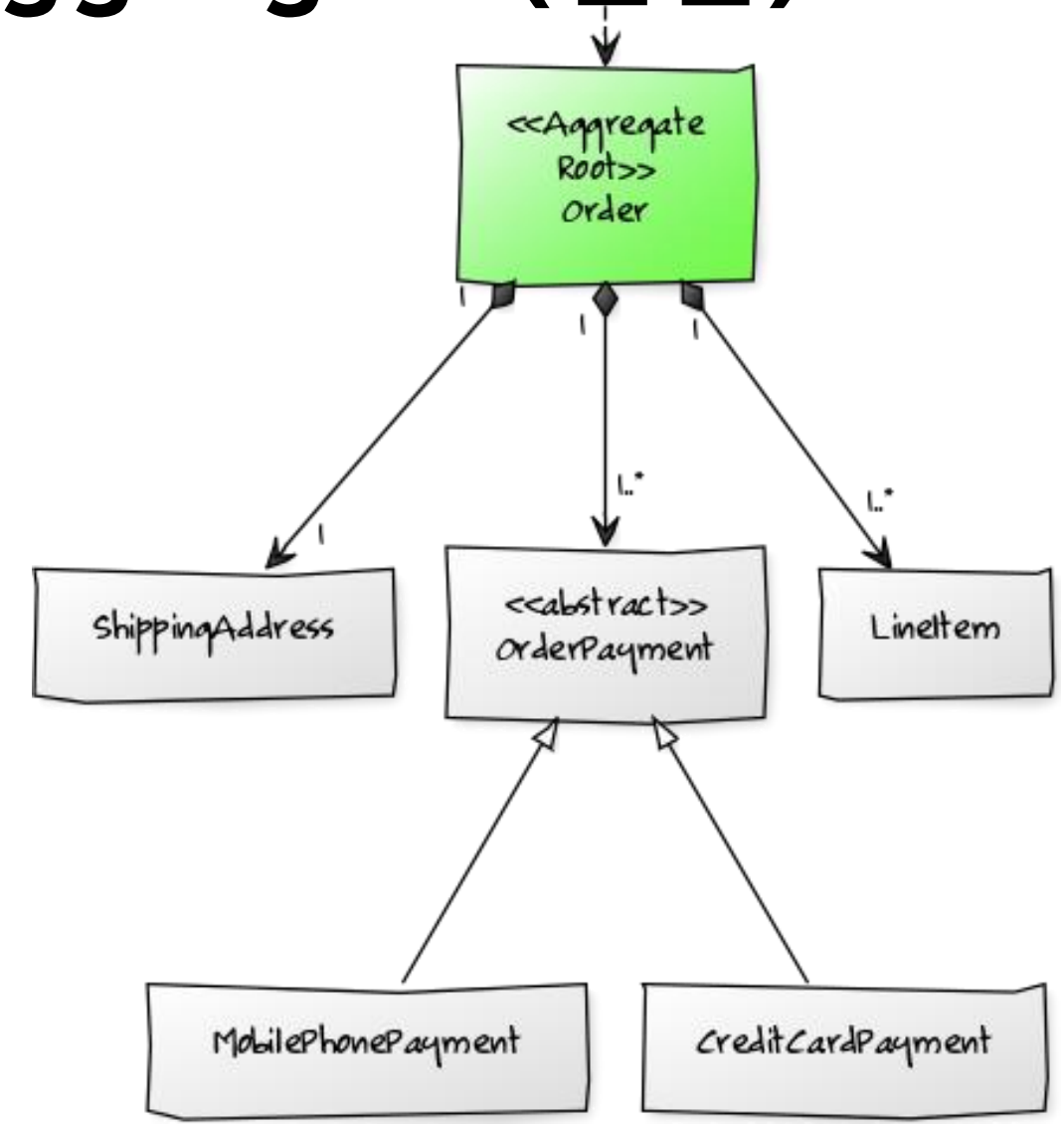
DDD Terminology – **Domain**

- **Domain**은 개발하고자 하는 소프트웨어가 다뤄야 하는 세상의 어떤것을 의미합니다. *(DDD quickly, p 17)*
- **Domain Model Object**는 도메인 영역을 분석하고, 그 분석의 결과로 도출된 객체들입니다. (https://4ngeunlee.tistory.com/223)

DDD Terminology – **Aggregate(집합)**

- **집합**은 객체의 소유권과 경계를 정의하는데 사용되는 패턴입니다 (DDD Quickly, p. 74)
- 쉽게 말해, 데이터를 변경할 때하나의 단위로 간주되는 관련된 객체들의 집합을 의미합니다. (DDD Quickly, p. 77)
- 그런 객체들을 하나의 '단위'로 두고, 그 '단위'를 하나의 집합으로 정의합니다.

DDD Terminology – **Aggregate(집합)**



https://medium.com/@SlackBeck/%EC%95%A0%EA%B7%B8%EB%A6%AC%EA%B2%8C%EC%9E%87-%ED%95%98%EB%82%98%EC%97%90-%EB%A6%AC%ED%8C%8C%EC%A7%80%ED%86%A0%EB%A6%AC-%ED%95%98%EB%82%98-f97a69662f63

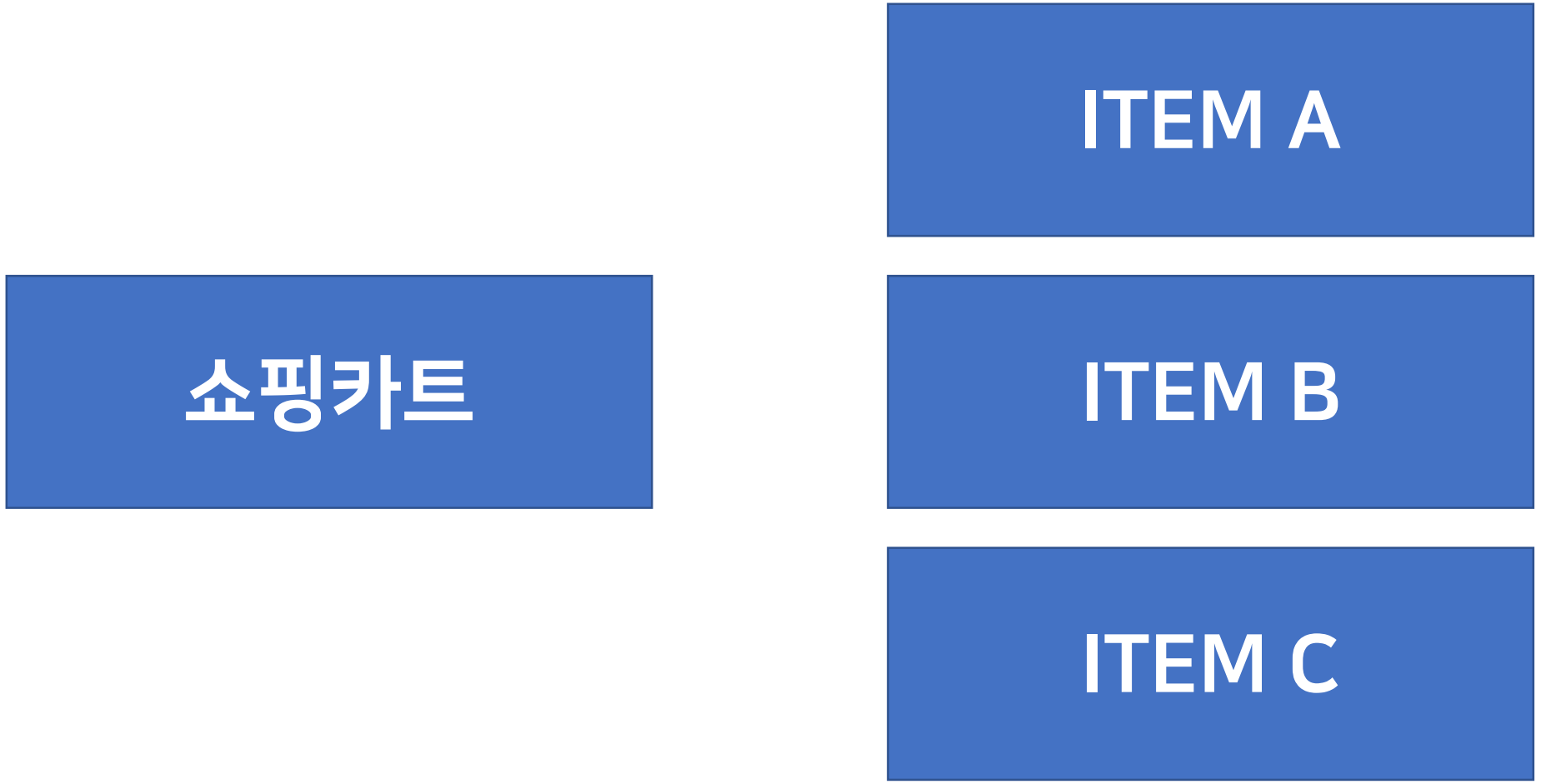
이벤트 소싱

Event Sourcing

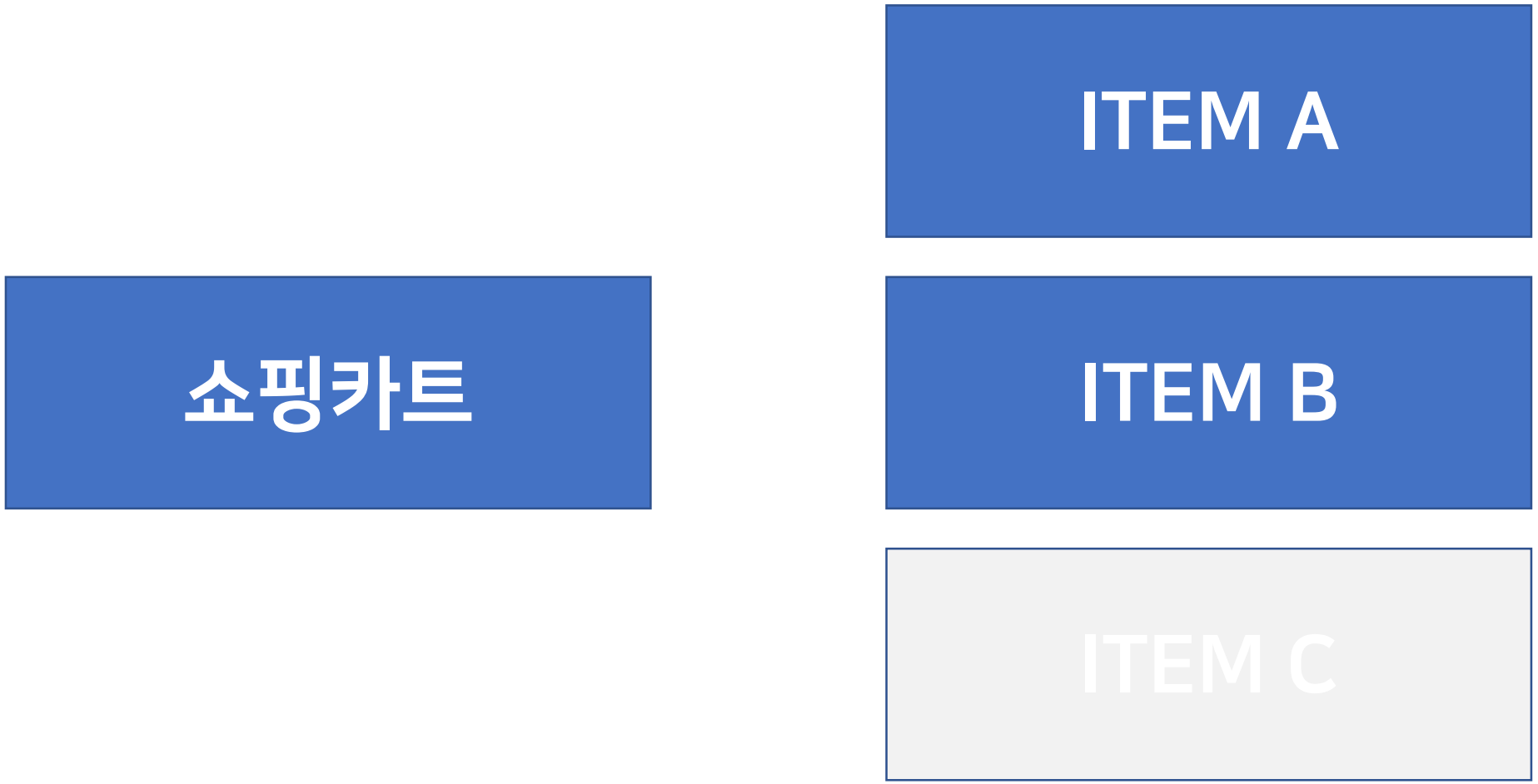
이벤트 소싱의 정의는 뭔가요?

- 여기서의 이벤트는 도메인 모델에서 발생하는 모든 사건(event)를 의미합니다.
- “The core idea of event sourcing is that whenever we make a change to the state of a system, we record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future” - **Martin Fowler**
- 이벤트 소싱은 **어플리케이션의 모든 상태 변화를 순서에 따라 이벤트로 보관한다** 라는 의미입니다.

이벤트 소싱은 어떻게 다른가요?



이벤트 소싱은 어떻게 다른가요?



차이를 알아봅시다.

스테레오타입 저장소.

- Id 52번 유저는 카트에 Item A와 B가 있습니다.

이벤트 소싱 저장소.

- Id 52번 유저가 Item A가 추가 하였습니다.
- Id 52번 유저가 Item B가 추가 하였습니다.
- Id 52번 유저가 Item C가 추가 하였습니다.
- Id 52번 유저가 Item C를 제거하 였습니다.

이벤트 소싱은 저장 형식이 간단해요.

Key (compound primary key)	Value
Object Id Version	Event Type Serialized Payload (ex JSON)

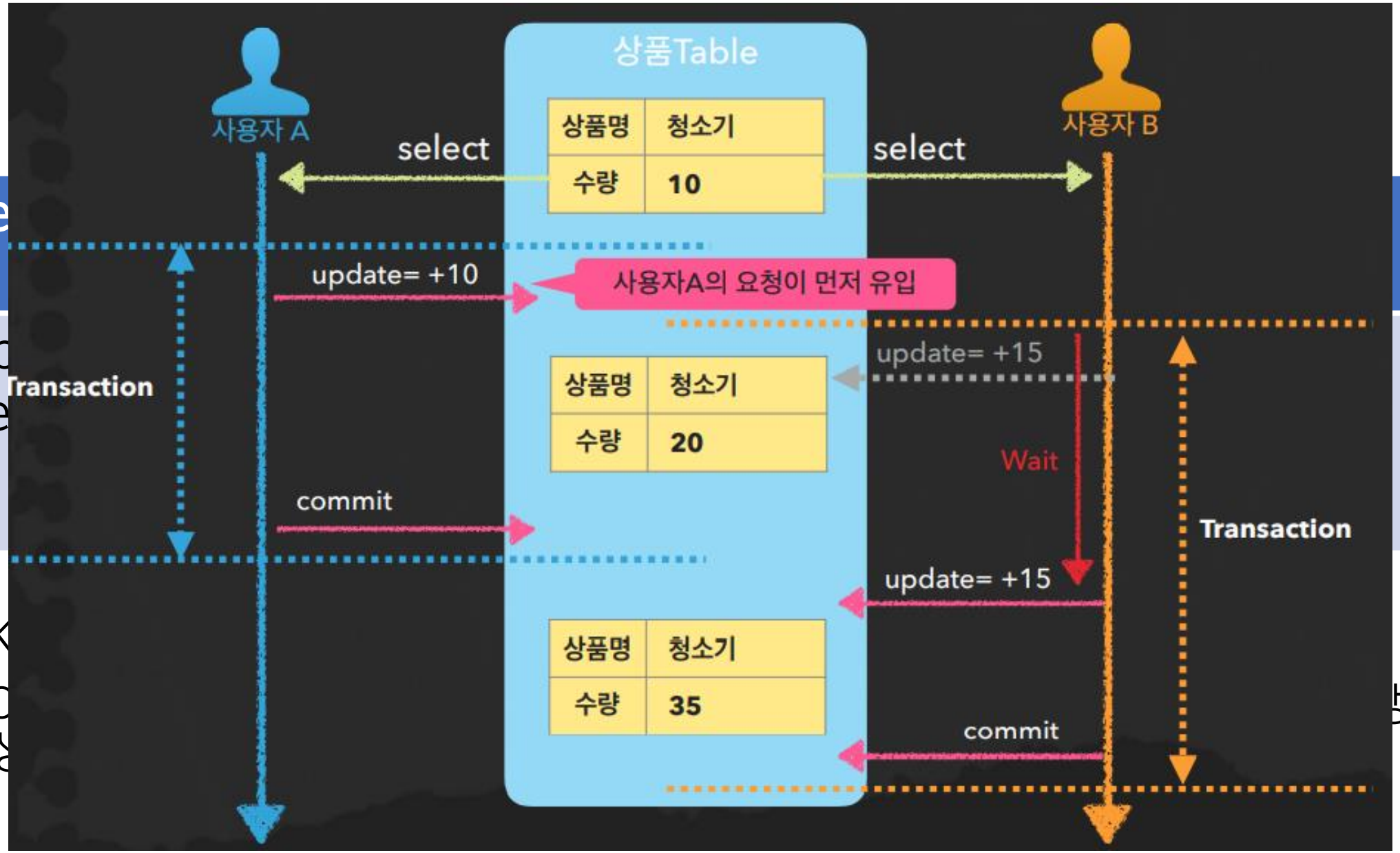
- Key-Value 형태로만 영속적으로 저장한다면 어떤 저장 방식도 괜찮습니다.
- Object id와 Version이 같은 상태로 저장하려고 하면, primary key가 같으니 생성이 안됩니다. 따라서 동시성도 대응됩니다

O

Ke

Ob
Ve

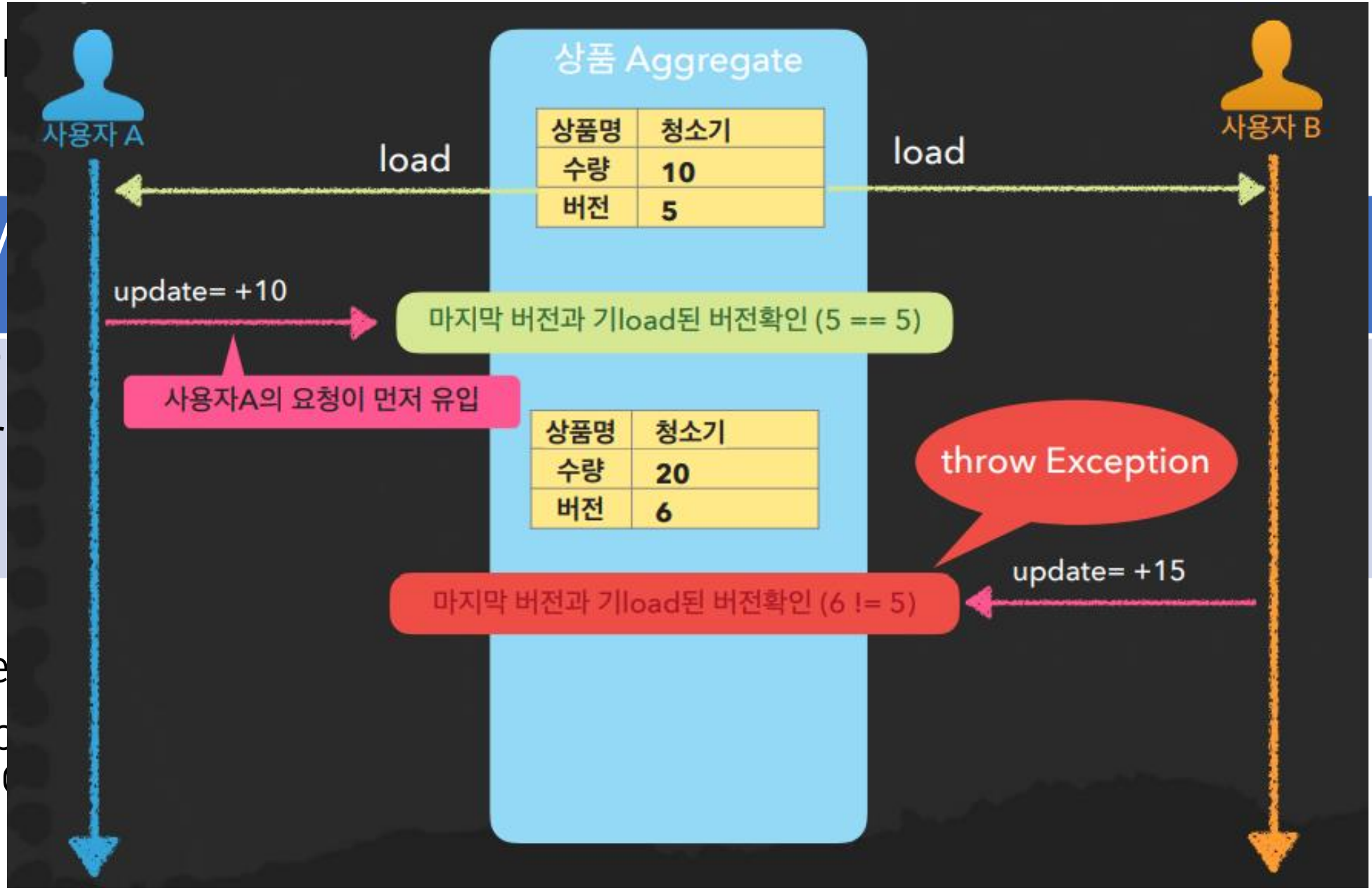
- K
- C
  상


상품Table
사용자 A
사용자 B
select
select
상품명
청소기
수량
10
update= +10
사용자A의 요청이 먼저 유입
update= +15
Transaction
상품명
청소기
수량
20
Wait
commit
Transaction
update= +15
상품명
청소기
수량
35
commit

이

Key

Obj
Ver



- Ke
- Ob
  성

생

저장 예시

스테레오타입 저장소.

데이터베이스

주문 테이블

ID
7dd8

주문항목 테이블

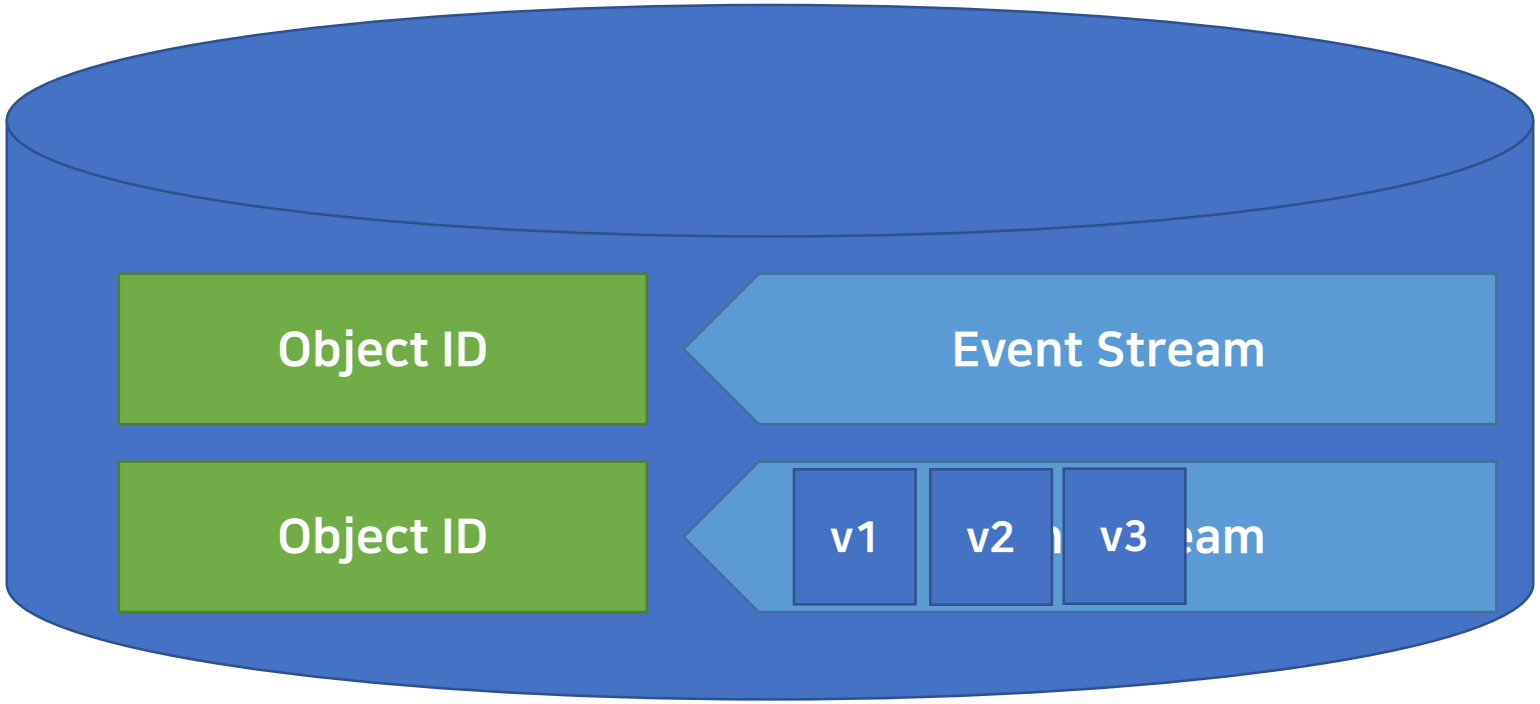
ORDER_ID	ITEM_ID	QUANTITY
7dd8	9a37	5
7dd8	a974	10

이벤트 소싱 저장소.

ORDER_ID	VERSION	EVENT_TYPE	EVENT_DATA
7dd8	1	OrderPlaced	OrderId: 7dd8
7dd8	2	OrderItemAdded	ItemId: 9a37, Quantity: 3
7dd8	3	OrderItemAdded	ItemId: c52a, Quantity: 1
7dd8	4	OrderItemDeleted	ItemId: c52a
7dd8	5	OrderItemAdded	ItemId: a974, Quantity: 10
7dd8	6	OrderItemQuantityChanged	ItemId: 9a37, Quantity: 5

https://justhackem.wordpress.com/2017/02/05/introducing-event-sourcing/

저장형태 - 이벤트스트림



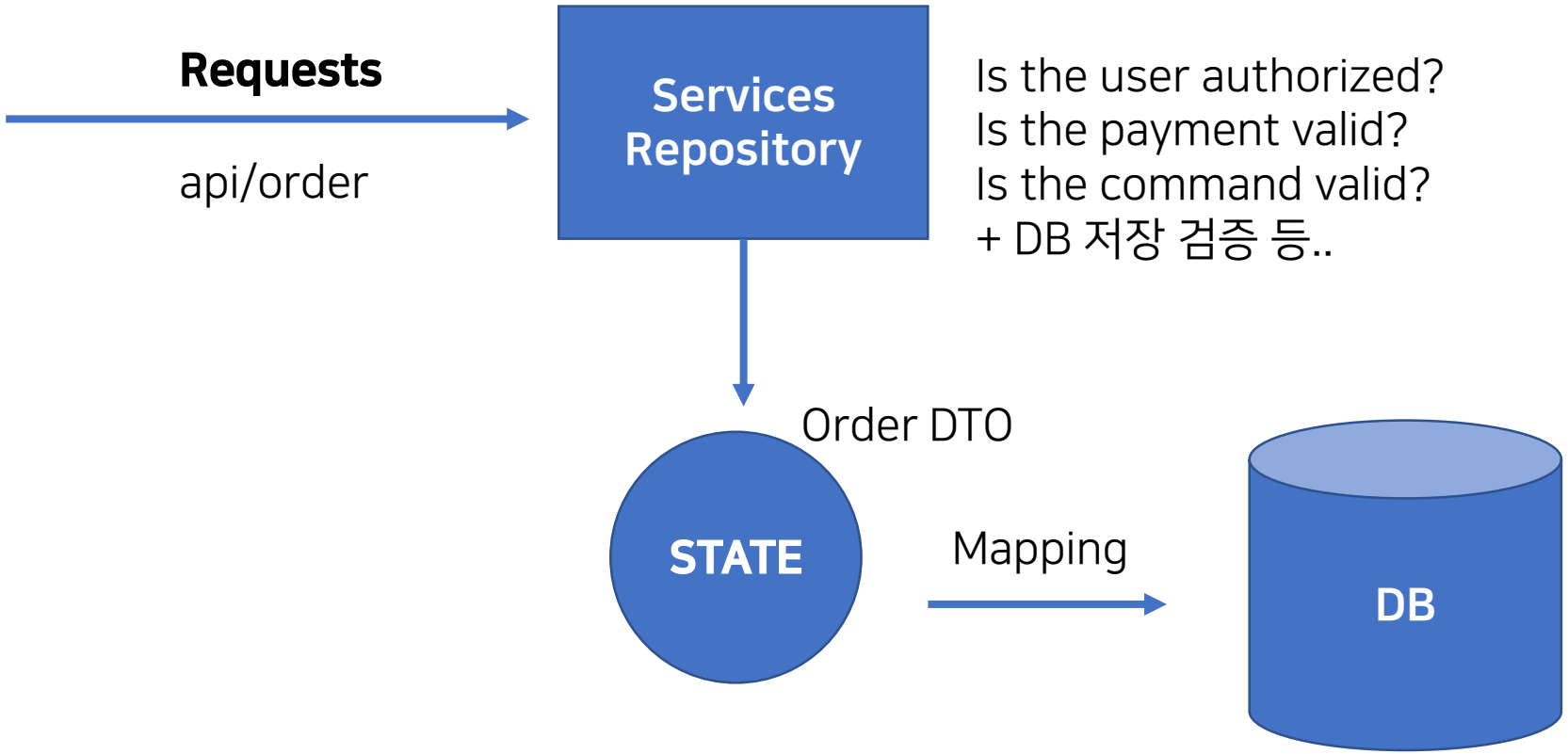
특징 정리

- **삭제와 수정이** 없어요! 데이터 상태 변경은 **이벤트 추가**로만 처리합니다.
- 이벤트는 이미 일어난 사실 그 자체 입니다. 그래서 보통 **과거형 동사**를 사용합니다. (ex OrderPlaced 등..)
- 유일한 키값 (object id) 등 을 사용합니다.
  - 키값을 기준으로 '**이벤트 스트림**' 이 생성됩니다. 이벤트 스트림은 해당 집합 (aggregate) object id 마다 생성되며, 이벤트가 일어난 순서대로 저장된 형태를 의미합니다.
- Object id와 Version를 Primary Key로 사용합니다
- Event Data에 도메인에서 관심있는 데이터를 직렬화 해서 넣어줍니다. 직렬화 방식은 자유롭습니다.

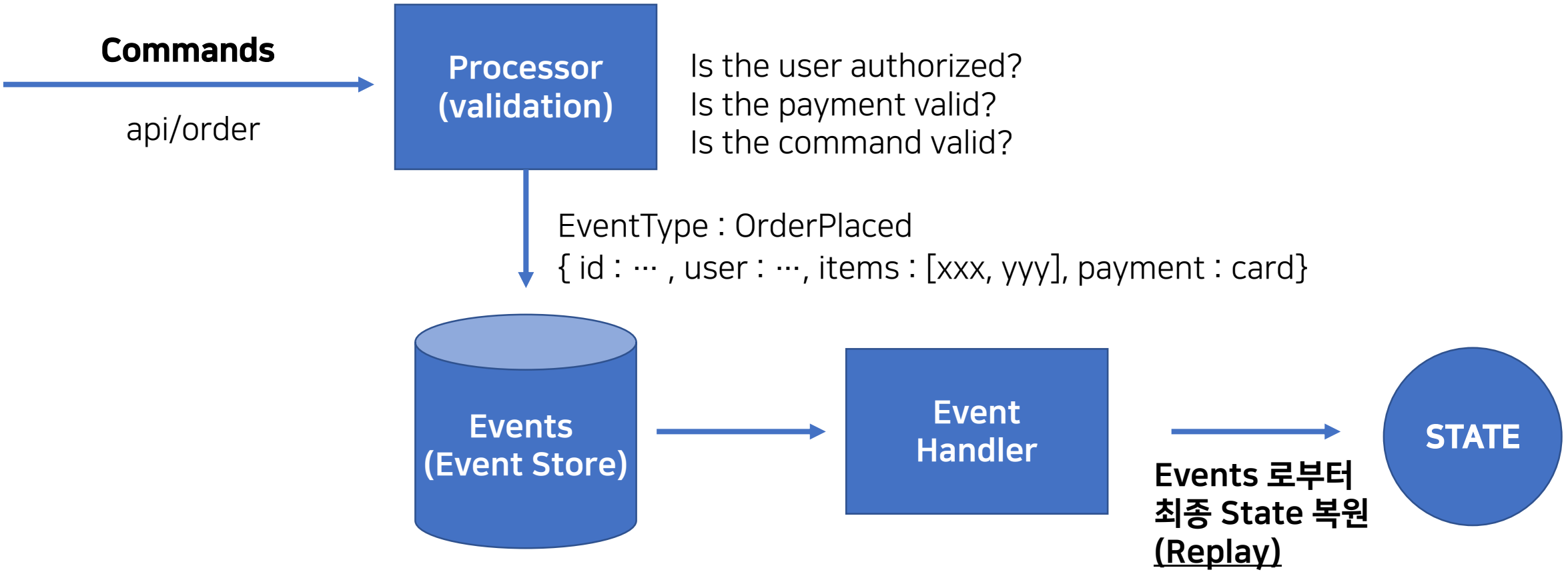
기존 도메인 처리 방식은 어떤가요?

- 기존 전통적 방식을 관찰해봅시다.

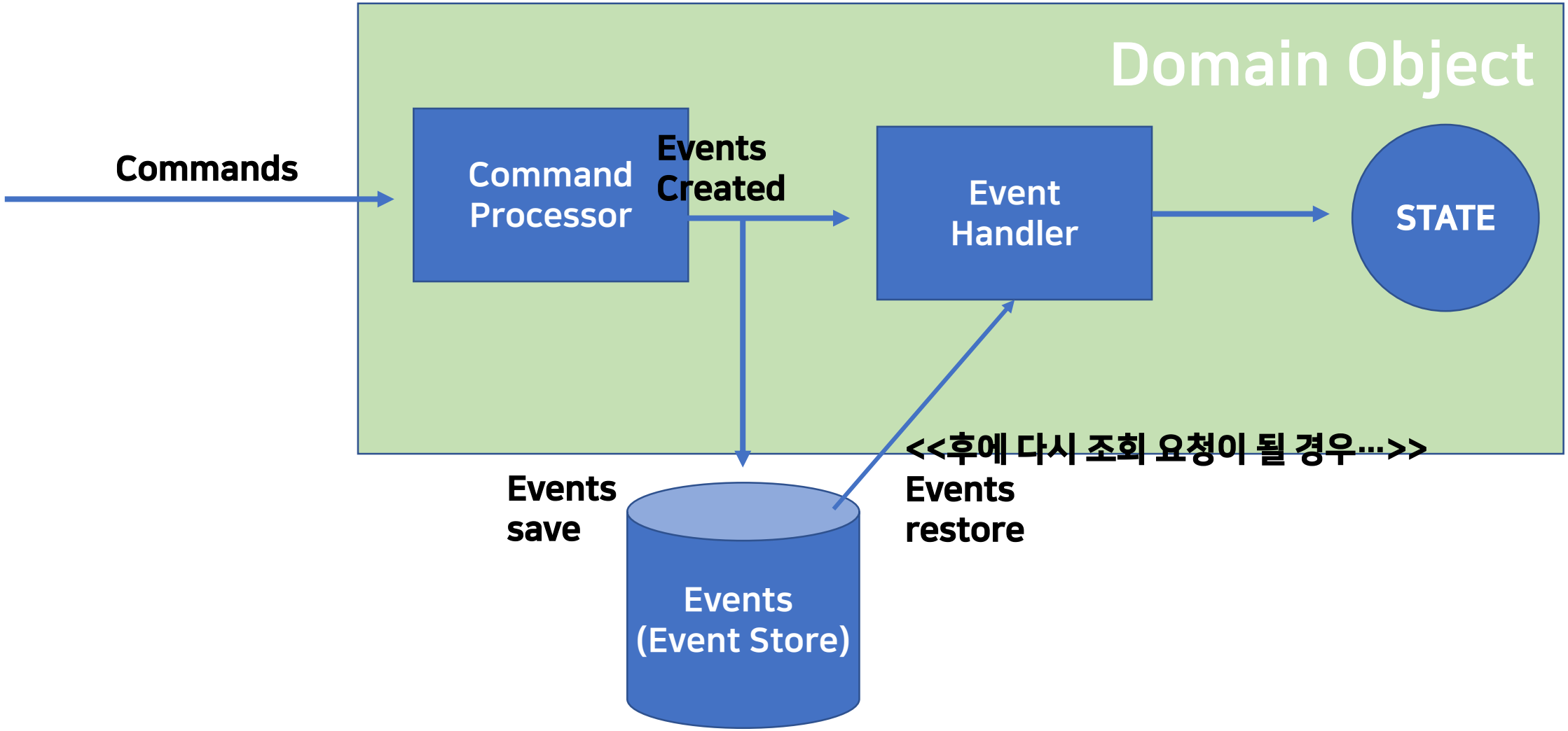
기존 기록 방식은 어떤가요?



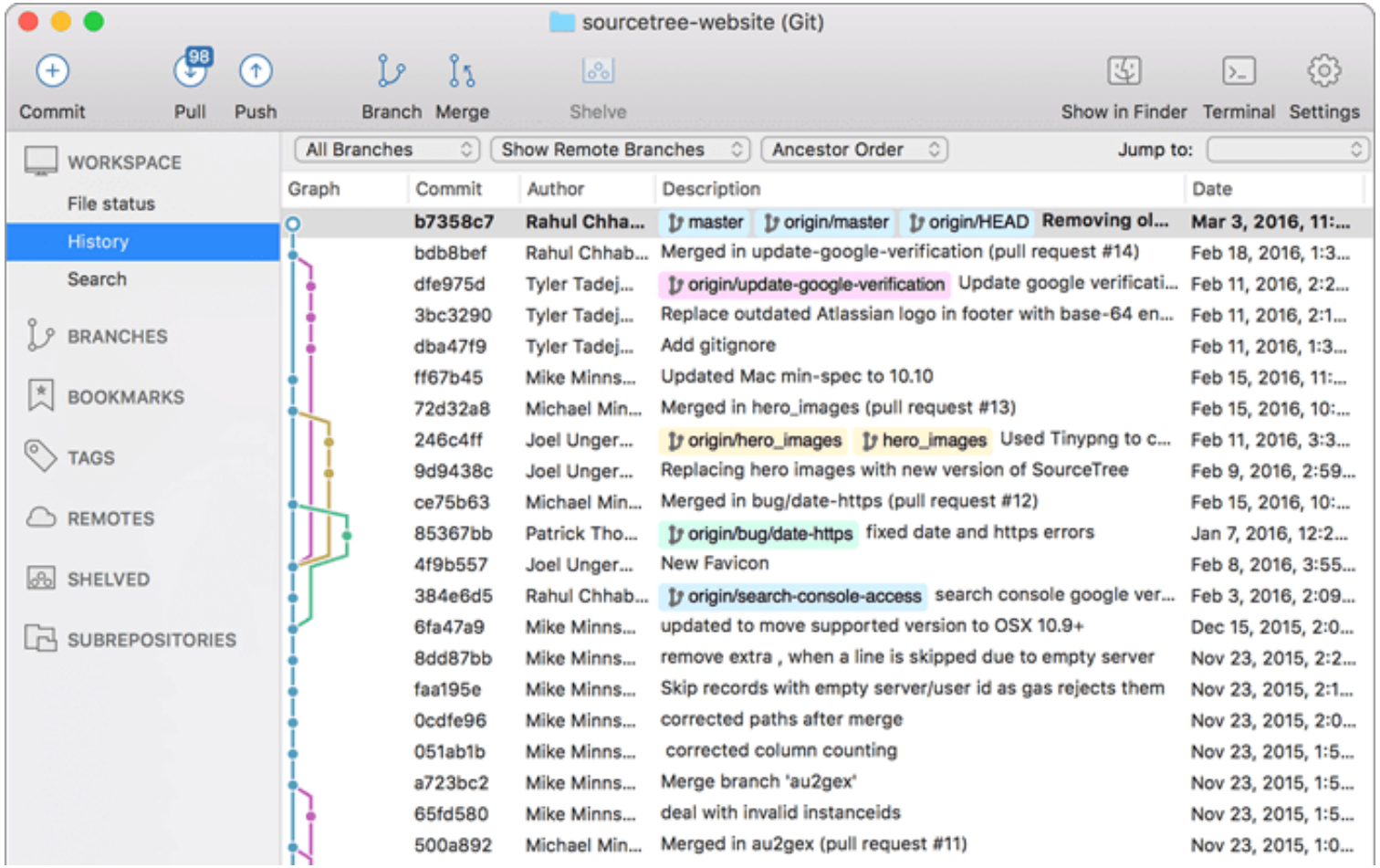
이벤트 소싱은 어떨까요?


Commands
api/order
Processor
(validation)
Is the user authorized?
Is the payment valid?
Is the command valid?
EventType : OrderPlaced
{ id : ··· , user : ···, items : [xxx, yyy], payment : card}
Events
(Event Store)
Event
Handler
Events 로부터
최종 State 복원
(Replay)
STATE

구현부 입장에서..


Domain Object
Commands
Command Processor
Events Created
Event Handler
STATE
Events save
Events (Event Store)
<<후에 다시 조회 요청이 될 경우…>>
Events restore

생소한가요? 사실 우리 가까이에 있었습니다



이벤트소싱은 왜 사용할까요?

- 비즈니스에서 도메인에 핏한 데이터를 가져올 수 있어요.
- 분산시스템, 메세징 아키텍쳐 등에 잘 활용될 수 있는 특징을 가지고 있어요.
- 도메인의 복잡성을 줄여주는데 한몫합니다.

비즈니스에서 의미있는 데이터 수집

- 이벤트는 높은 설명력을 가진 데이터입니다.
- 오류가 발생할 경우 오류가 발생한 지점의 이벤트를 복원하여 문제를 해결할 수 있다.

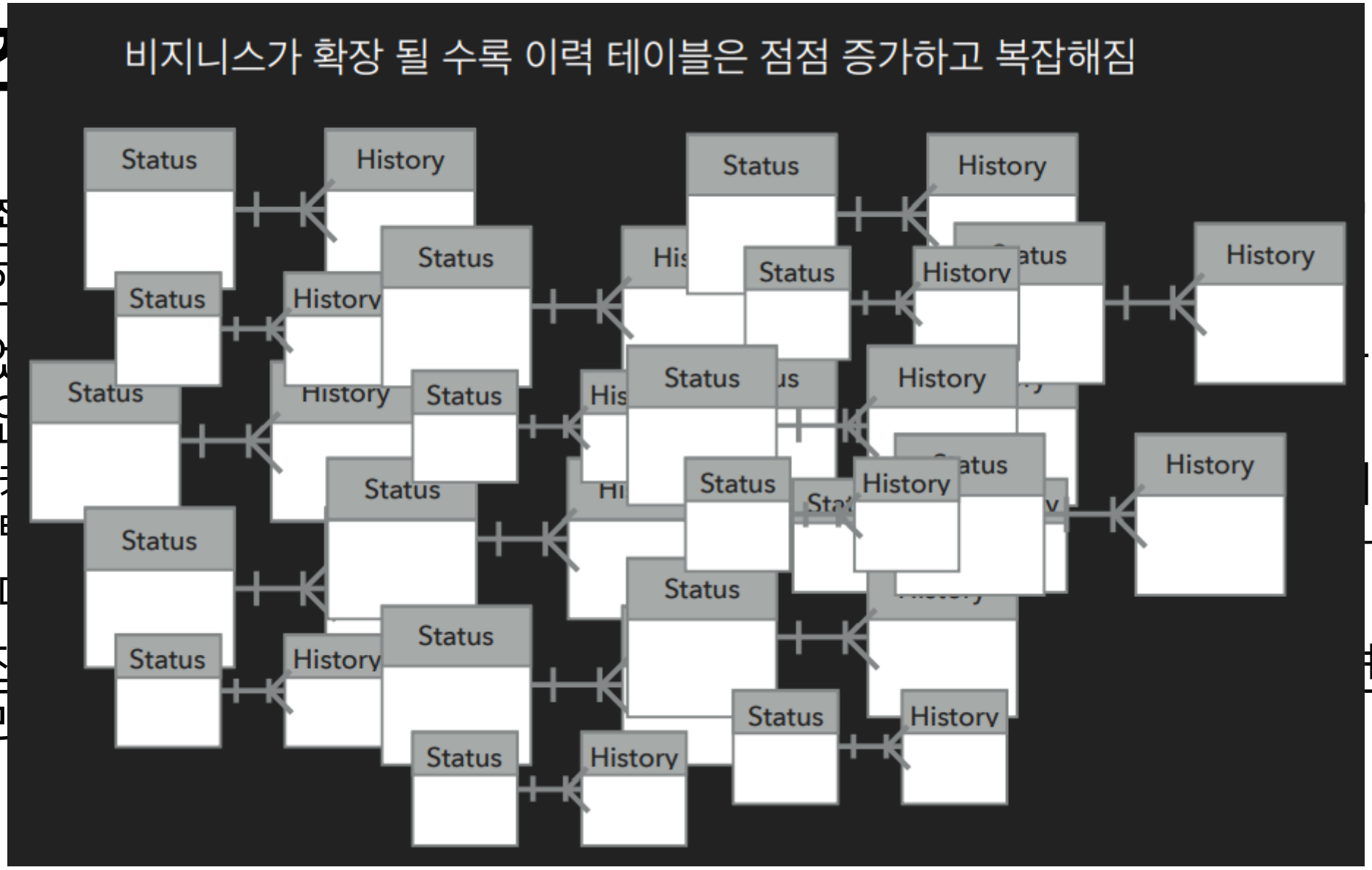
비즈니스에서 의미있는 데이터 수집

- 기존에 로그데이터는 도메인에 핏한 데이터라기보다 프로그래머가 필요한 정보에 대해서 가져오는 경우가 있어요.
- 무엇보다도 상태저장 + 로그 데이터는 한번에 atomic하게 처리가 되진 않습니다.
  - 카트에 아이템을 추가하는것을 로그 데이터로 쌓는다고 하면 보통 카트에 아이템을 추가하는 액션 이후 로그가 쌓일 때 까지 사용자가 기다리지는 않습니다.
  - 따라서 데이터가 유실될 수 있는 리스크가 있죠.
- 복잡한 시스템의 각 서비스마다 로그 데이터를 각각 따로 관리하면 더 관리가 힘들어지고 복잡해진다는 문제가 있습니다.

비즈니스에서 의미있는 데이터 수집

- 기존에 로그데이터는 도메인에 핏한 데이터라기보다 프로그래머가 필요한 정보에 대해서 가져오는 경우가 있어요.
- 무엇보다도 상태저장 + 로그 데이터는 한번에 atomic하게 처리가 되진 않습니다.
  - 카트에 아이템을 추가하는것을 로그 데이터로 쌓는다고 하면 보통 카트에 아이템을 추가하는 액션 이후 로그가 쌓일 때 까지 사용자가 기다리지는 않습니다.
  - 따라서 데이터가 유실될 수 있는 리스크가 있죠.
- 복잡한 시스템의 각 서비스마다 로그 데이터를 각각 따로 관리하면 더 관리가 힘들어지고 복잡해진다는 문제가 있습니다.

비즈



- 기존
- 무엇
  -
  -
- 복잡

가 필요한

되 진 엔

아이 B

니다.

변 더 관리

분산시스템, 메시지 중심 아키텍처에 유리

- 테이블의 형태가 매우 간단해요.
- 그래서 key-value형태만 지원이 가능하면 어떤 저장소도 사용할 수 있어요.
- 이런 형태는 분산시스템에서 다양한 저장 플랫폼을 선택하는데 유리합니다.
- 또한 이런 이벤트를 읽고 쓰는것을 (일급 객체) 도메인이 이해하고 있다면, 메세징 아키텍쳐를 사용하는것에도 유리합니다.

도메인의 복잡성을 줄여줍니다 - 1

- 저장된것에 대한 수정과 삭제가 없어요.
- 데이터에 대한 접근과 수정에 대한 경쟁이 없고, 그만큼 동시성 처리도 수월해진다고 볼 수 있어요.

도메인의 복잡성을 줄여줍니다 - 2

- 저장하는 데이터가 도메인 관심사에 집중되어 있어요. 관계가 없습니다.
- 기존의 CRUD 형태는 데이터베이스의 형태에 의존적인 형태가 있습니다.
  - 비즈니스가 복잡해질수록 데이터베이스와 비즈니스 로직에 1:1로 맵핑이 까다로운 경우가 있어요. (개체-관계형 임피던스 불일치)
  - 물론 ORM 등의 솔루션들이 있지만, 근본적인 문제를 해결해주진 않아요. 성능 문제도 고려해야합니다.
  - 테이블 데이터가 생성될 때 마다 DTO를 만들어서 맵핑까지 신경써줘야해요.
  - 데이터베이스의 형태를 기준을 로직을 억지로 녹여내려고 하면 힘들어요.. (대표적으로 리소스를 기준으로 한 명사형 URI Path 형태들…)

그럼에도 불구하고..

- 이벤트 소싱은 필연적으로 **분산시스템**을 사용할 수 밖에 없습니다.
- 분산시스템을 사용한다는것은 그 자체로 고통이고 해야할 일들이 많습니다.
- 따라서 그만큼의 비즈니스에서의 가치가 있다면 도입을 고려해야하는게 좋습니다. 그렇지 않으면 **오버엔지니어링**이 될 가능성이 큽니다.

- **왜 분산시스템이 필연적으로 필요한지는 후반부에 다룹니다!**

Hmm..

실현 가능한 솔루션인가요?

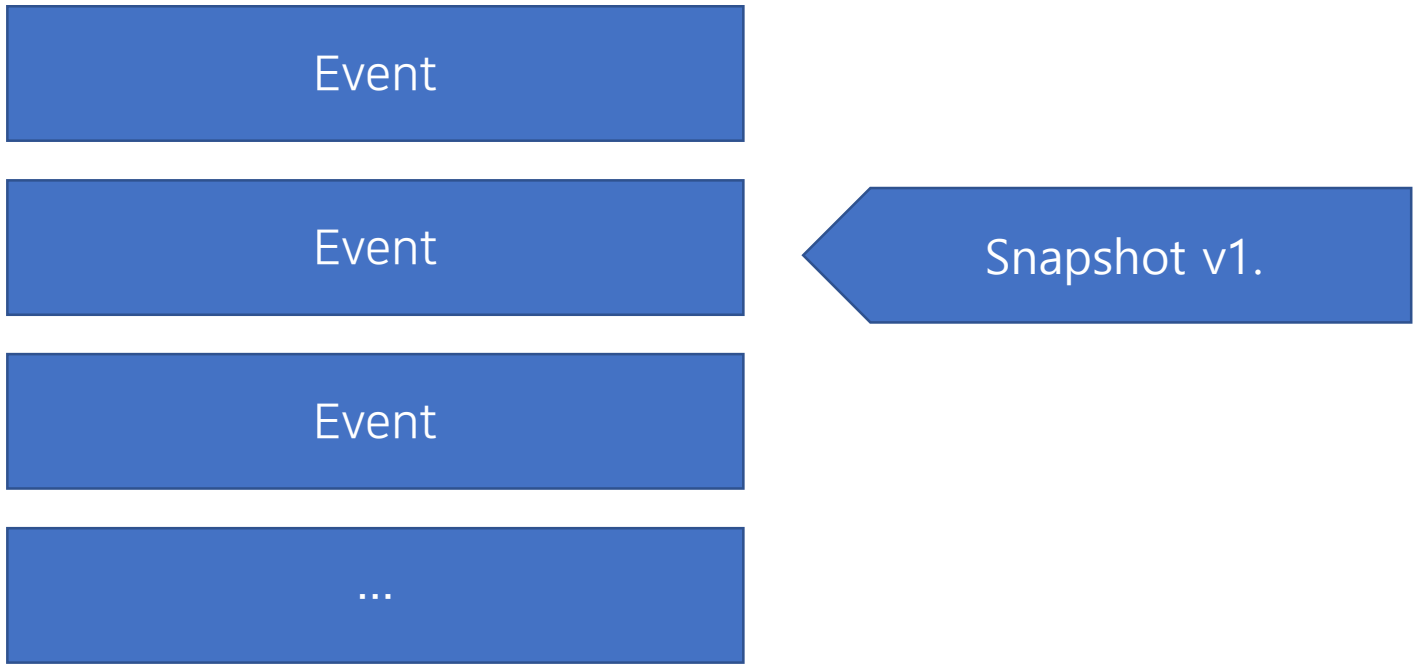
- 만약 도메인 id가 살아남아서 이벤트 스트림안에 이벤트가 100만개가 있다고 하면.. 매번 100만개를 replay해서 state를 만들어야 하나요?
- RDMBS가 수행하던 복잡한 쿼리는 어떻게 처리할 수 있나요?

매번 100만개의 Replay?

- 상황 : 유저 A 의 카트 이벤트가 100만개가 쌓여있는 상황. 이벤트 스트림에 100만개의 이벤트가 쌓여있는데, 그것을 조회할 때, 100만건의 이벤트를 모두 Replay를 하면 느려지지 않을까?

매번 100만개의 Replay?

- 일정 규칙에 의해서 스냅샷을 찍어놓습니다.
- 스냅샷을 참고하여 Replay를 하게 되면, 적은수의 이벤트를 가지고 최종 상태를 복원할 수 있습니다.



복잡한 쿼리는 어떻게 수행할 수 있나요?

- 만약 **재고가 10개 미만인 상품들에 대해서 모두 조회를** 해야 한다면 어떻게 해야할까요?
- 기존 RDBMS솔루션에서는 풀스캔 혹은 인덱싱을 통해서 빠르게 처리하려 합니다.
- 이벤트 소싱은 그게 불가능합니다. 모든 관련된 이벤트를 조사해야합니다. 이는 RDBMS의 레코드 풀스캔보다 상황이 심각할 수 있습니다. 저장소에 있는 모든 이벤트를 메모리로 로드해서 상태를 만들어야하는 상황이 올 수 있습니다.

복잡한 쿼리는 어떻게 수행할 수 있나요?

- 그렇기에.. 이벤트소싱에는 현실적으로 **CQRS**가 필수입니다.

CQRS

Command Query Responsibility Segregation

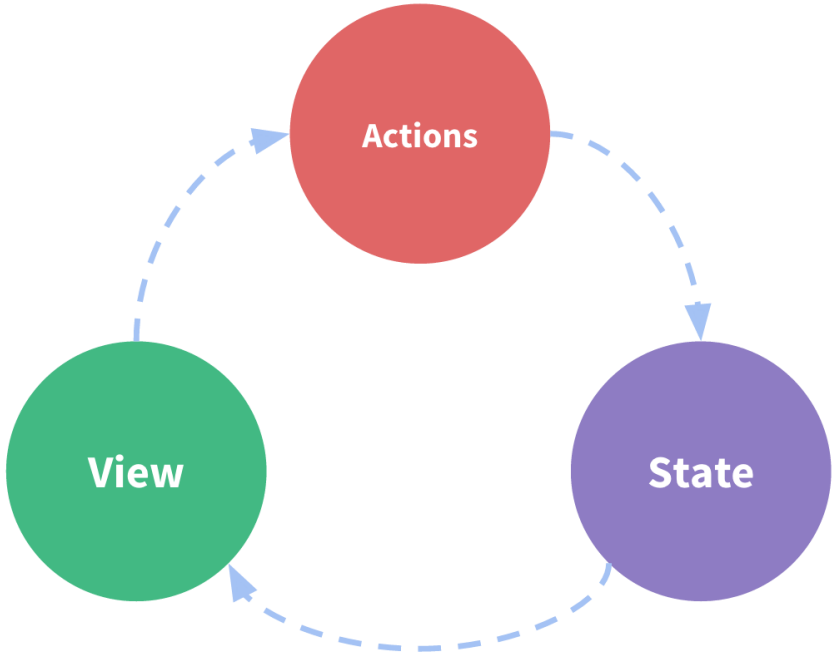
Command–query separation (CQS)

- **Bertrand Meyer**
- 대부분의 메소드는 **명령**과 **조회**로 나눠진다.

Command-query separation (CQS)

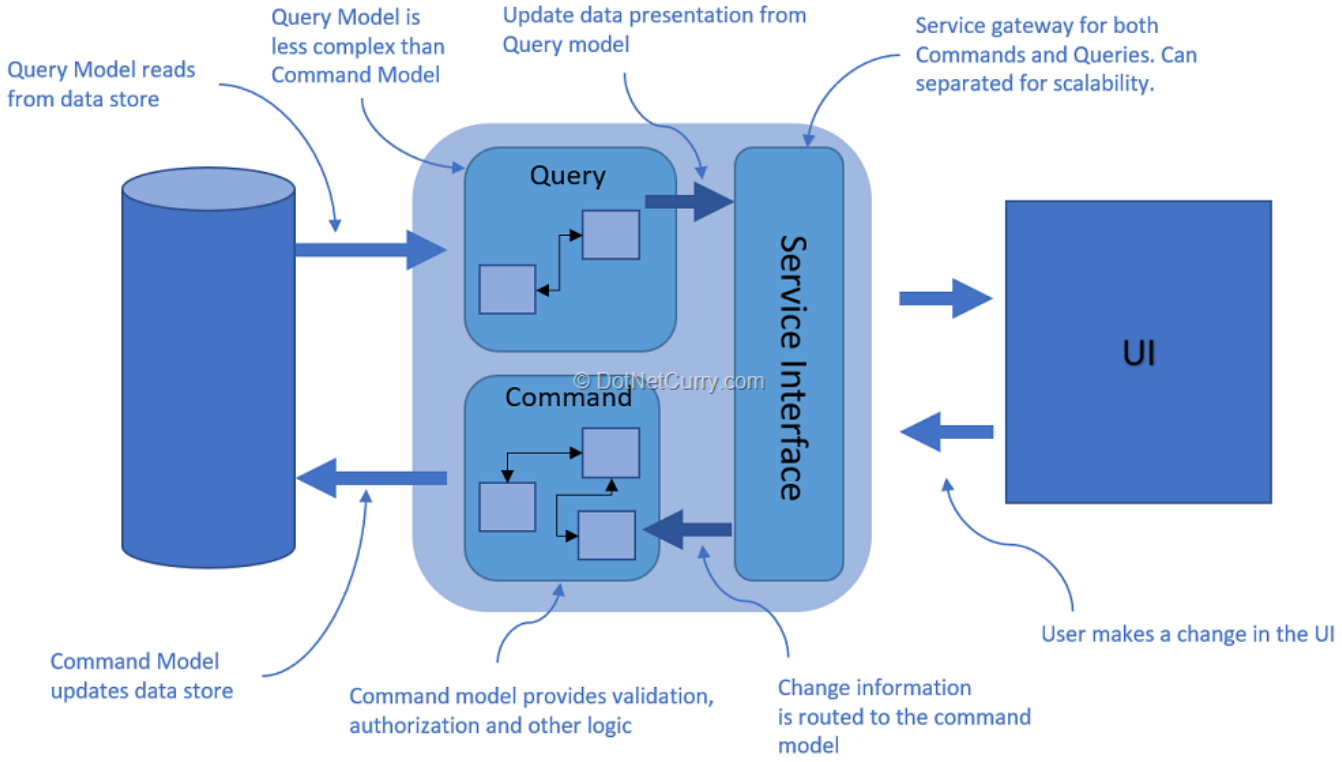
- **명령** : 어떠한 시스템의 상태를 변경하는 행위. Return 값이 없다.
- **조회** : 어떠한 시스템의 상태를 반환하는 행위. 상태를 변경하지 않는다.
- 물론 항상 나눠지는것은 아님..
  - stack의 pop()연산은 명령과 조회를 동시에!

Command–query separation (CQS)



High-level CQS

- 패턴이기 때문에 어떻게 구현하는지 각자 나름이에요.
- 점차 시스템/어플리케이션 레벨로 CQS 를 확장하기 시작했어요.



High-level CQS ≈ CQRS

- **CQS** : Command Query Separation
- **CQRS** :Command Query Responsibility Segregation

High-level CQS ≈ CQRS

- **CQS** : 연산, 메소드, 컴포넌트 수준에서 명령과 조회를 나눠요.
- **CQRS** : 오브젝트(도메인 객체) 이상에서, 시스템 수준에서 분리를 합니다.

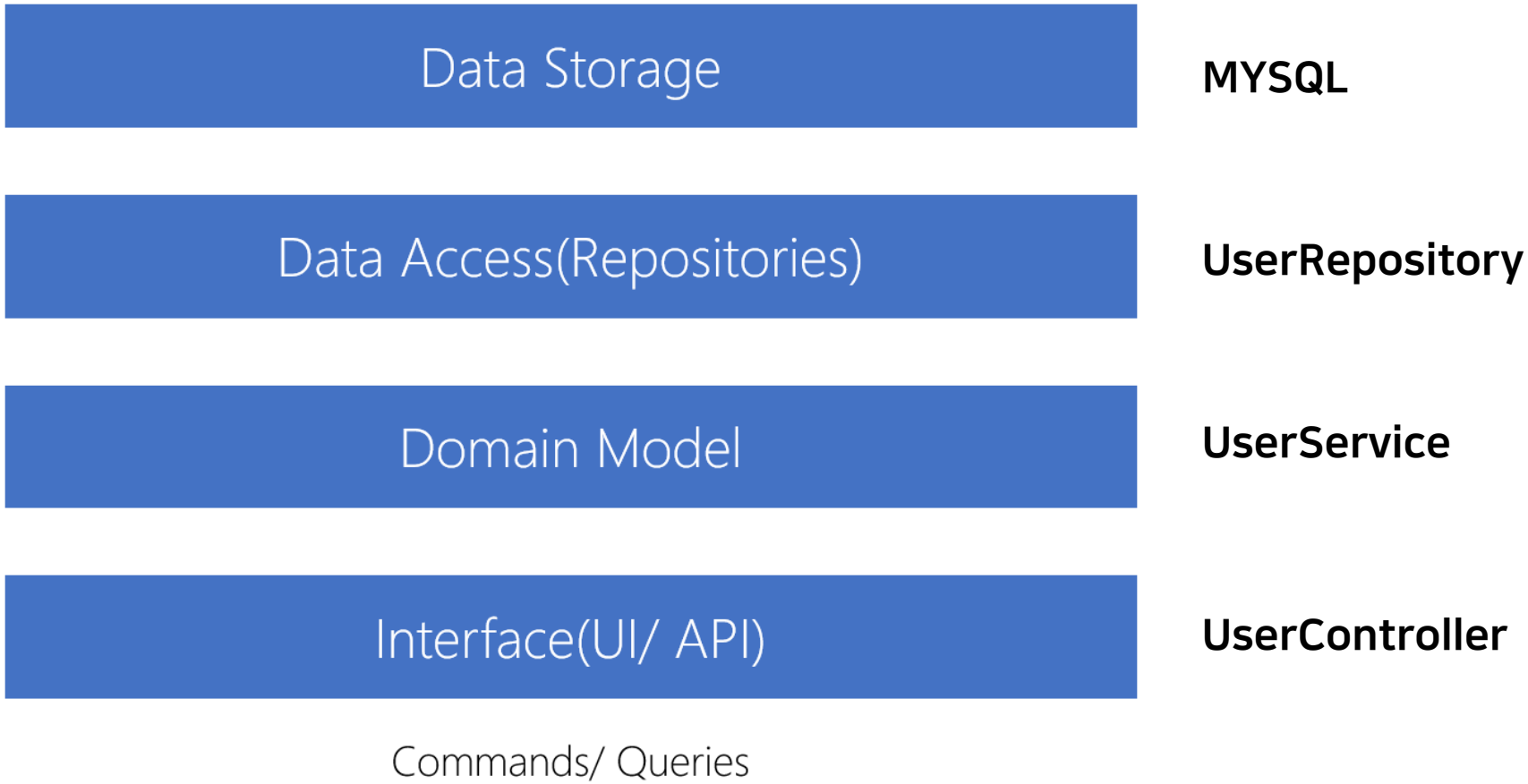
High-level CQS ≈ CQRS

- CQRS 는 매우 간단한 패턴이며, 앞서 말했던 정의가 기본입니다.
- 간단한 패턴이기 때문에 다양한 구현 방식이나 practice가 존재해요.
- 보통 예제들은 다계층 아키텍쳐, 이벤트 소싱이나 DDD 에서의 사용 예제가 나오지만, 이는 하나의 구현 예시일 뿐, CQRS 가 이렇게 복잡한 패턴은 아닙니다. (기본/디럭스 등..)

HOW : CQRS 구현 예시

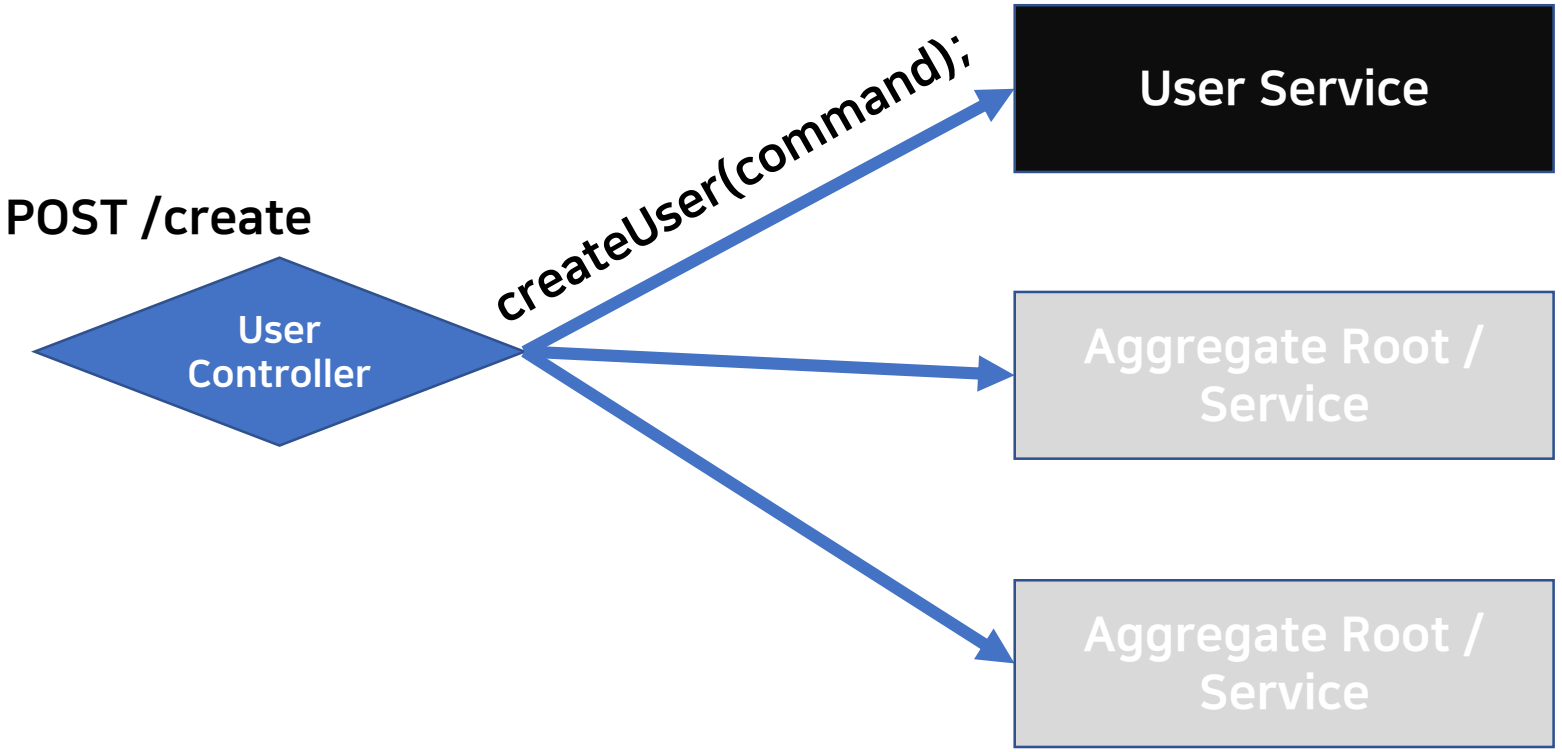
- CQRS를 구현해본 예제를 한번 톺아보도록 하겠습니다.

기존 전통적 방식의 구현을 볼게요.

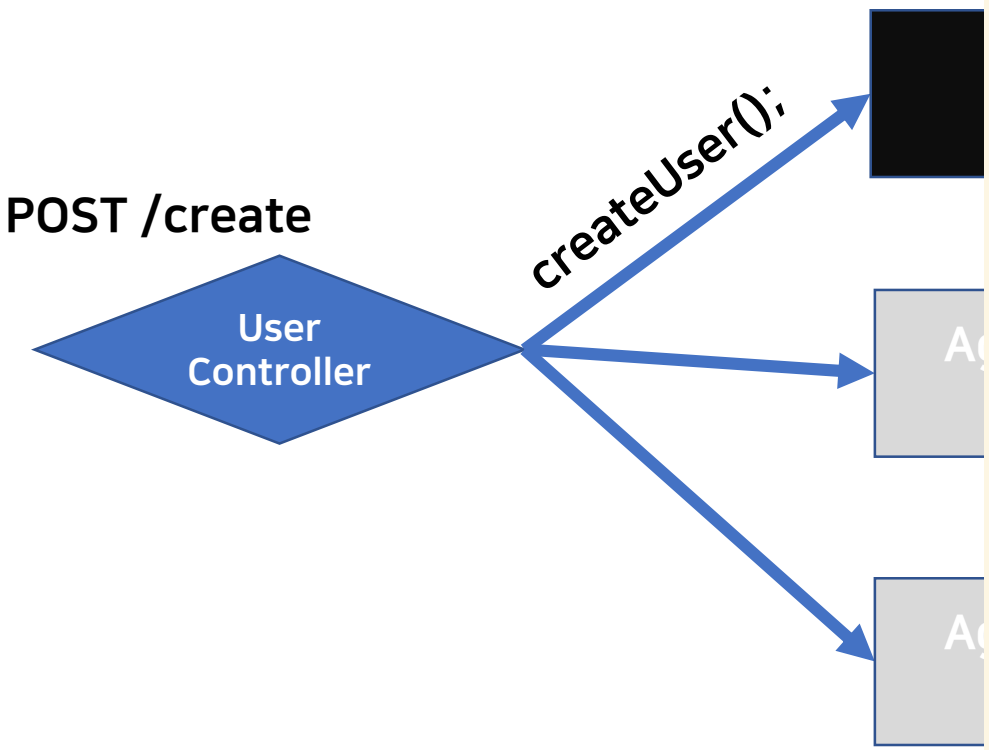


https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/

전통적 방식의 서비스


POST /create
User Controller
createUser(command);
User Service
Aggregate Root / Service
Aggregate Root / Service

전통적 방식의 서비스




```java
public createUser(String username,  String password){

  // Insert UserInfo
    int userId = repostiroy.insert(
        new User(
            username,
            _passwordHasher.hash(password)
        )
    );

    // find user by userId.
    User user = repostiroy.find(userId);

    // return user info for external services.
    return new UserPresentation(
        userId,
        username
    );
}
```

원래 소스에서 로직을 일부 수정하였습니다.
원본소스는 https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/ 참고!

전통적 방식의 서비스

```
1  public createUser(String username,  String password){
2
3      // Insert UserInfo
4      repostiroy.insert(
5          new User(
6              username,
7              _passwordHasher.hash(password)
8          )
9      );
10
11     // find user by userId.
12     User user = repostiroy.find(username);
13
14     // return user info for external services.
15     return ; UserPresentation(
16         user.Id,
17         username
18     );
19 }
20
```

- 새 사용자 엔터티를 추가합니다.

전통적 방식의 서비스

```
1 public createUser(String username,  String password){
2
3   // Insert UserInfo
4     repostiroy.insert(
5         new User(
6             username,
7             _passwordHasher.hash(password)
8         )
9     );
10
11    // find user by userId.
12    User user = repostiroy.find(username);
13
14    // return user info for external services.
15    return ; UserPresentation(
16        user.Id,
17        username
18    );
19 }
20
```

- 유저 정보를 얻기 위해서 유저를 조회합니다.

전통적 방식의 서비스

```
1 public createUser(String username,  String password){
2
3   // Insert UserInfo
4     repostiroy.insert(
5         new User(
6             username,
7             _passwordHasher.hash(password)
8         )
9     );
10
11    // find user by userId.
12    User user = repostiroy.find(username);
13
14    // return user info for external services.
15    return ; UserPresentation(
16        user.Id,
17        username
18    );
19 }
20
```

- 유저 정보에 대한 정보를 반환합니다. User객체는 **passwordHasher**라는 내부 서비스를 사용하기 때문에, 따로 외부 공개용 객체 (**UserPresentation**)를 만들어 전달합니다.

혹시 문제를 발견하셨나요?

```
1 public createUser(String username,  String password){
2
3   // Insert UserInfo
4     repostiroy.insert(
5         new User(
6             username,
7             _passwordHasher.hash(password)
8         )
9     );
10
11    // find user by userId.
12    User user = repostiroy.find(username);
13
14    // return user info for external services.
15    return ; UserPresentation(
16        user.Id,
17        username
18    );
19 }
20
```

테스트 코드를 보겠습니다.

```
public createUser_insert_to_repository(){

      // config Data.
      String random = new Random();
      String fakeUserName = "foo";
      String password = random.Next().ToString();
      String passwordHash = random.Next().ToString();

      // mocking
      UserRepository repository = Mock.Of<UserRepository>();
      passwordHasher = Mock.Of<PasswordHasher>(passwordHash);

      // DI
      UserService testService = new UsersDomainModel(repository, passwordHasher);

      // Mocking 'find()' methods.
      Mock.get(repository)
          .Setup(x => x.find(It.IsAny<int>()))
          .ReturnsAsync(new User());

     // create user id!
     testService.createUser(
         fakeUserName,
```

C#기반의 소스에서 Java스타일로 일부 변경되었습니다.
정상 동작을 기대하지 않습니다.
원본소스는 https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/

테스트 코드를 보겠습니다.

```
1 public createUser_insert_to_repository(){
2
3       // config Data.
4       String random = new Random();
5       String fakeUserName = "foo";
6       String password = random.Next().ToString();
7       String passwordHash = random.Next().ToString();
8
9       // mocking
10      UserRepository repository = Mock.Of<UserRepository>();
11      passwordHasher = Mock.Of<PasswordHasher>(passwordHash);
12
13      // DI
14      UserService testService = new UsersDomainModel(repository, passwordHasher);
15
16      // Mocking 'find()' methods.
17      Mock.get(repository)
```

유저 정보를 저장소에 저장할 수 있는가?

테스트 코드를 보겠습니다.

```
1 public createUser_insert_to_repository(){
2
3       // config Data.
4       String random = new Random();
5       String fakeUserName = "foo";
6       String password = random.Next().ToString();
7       String passwordHash = random.Next().ToString();
8
9       // mocking
10      UserRepository repository = Mock.Of<UserRepository>();
11      passwordHasher = Mock.Of<PasswordHasher>(passwordHash);
12
13      // DI
14      UserService testService = new UsersDomainModel(repository, passwordHasher);
15
16      // Mocking 'find()' methods.
17      Mock.get(repository)
```

실행에 필요한 command(혹은 입력) 데이터 생성

테스트 코드를 보겠습니다.

```
1 public createUser_insert_to_repository(){
2
3       // config Data.
4       String random = new Random();
5       String fakeUserName = "foo";
6       String password = random.Next().ToString();
7       String passwordHash = random.Next().ToString();
8
9       // mocking
10      UserRepository repository = Mock.Of<UserRepository>();
11      passwordHasher = Mock.Of<PasswordHasher>(passwordHash);
12
13      // DI
14      UserService testService = new UsersDomainModel(repository, passwordHasher);
15
16      // Mocking 'find()' methods.
17      Mock.get(repository)
```

필요 객체들 mocking.

```
1  public createUser_insert_to_repository(){
2
3        // config Data.
4        String random = new Random();
5        String fakeUserName = "foo";
6        String password = random.Next().ToString();
7        String passwordHash = random.Next().ToString();
8
9        // mocking
10       UserRepository repository = Mock.Of<UserRepository>();
11       passwordHasher = Mock.Of<PasswordHasher>(passwordHash);
12
13       // DI
14       UserService testService = new UsersDomainModel(repository, passwordHasher);
15
16       // Mocking 'find()' methods.
17       Mock.get(repository)
18           .Setup(x => x.find(It.IsAny<int>()))
19           .ReturnsAsync(new User());
20
21      // create user id!
22      testService.createUser(
23          fakeUserName,
24          password
```

Service에 주입

```
 7        String passwordHash = random.Next().ToString();
 8
 9        // mocking
10        UserRepository repository = Mock.Of<UserRepository>();
11        passwordHasher = Mock.Of<PasswordHasher>(passwordHash);
12
13        // DI
14        UserService testService = new UsersDomainModel(repository, passwordHasher);
15
16        // Mocking 'find()' methods.
17        Mock.get(repository)
18            .Setup(x => x.find(It.IsAny<int>()))
19            .ReturnsAsync(new User());
20
21    // create user id!
22    testService.createUser(
23        fakeUserName,
24        password
25    );
26
27        // assert
28        Mock<UserRepository> mock = Mock.get(repository);
29        mock.verify(repo =>
30            repo.insert(It.Is<User>(user =>
31                user.UserName == userName &&
```

Repostiory.find() mocking

```
 7        String passwordHash = random.Next().ToString();
 8
 9        // mocking
10        UserRepository repository = Mock.Of<UserRepository>();
11        passwordHasher = Mock.Of<PasswordHasher>(passwordHash);
12
13        // DI
14        UserService testService = new UsersDomainModel(repository, passwordHasher);
15
16        // Mocking 'find()' methods.
17        Mock.get(repository)
18            .Setup(x => x.find(It.IsAny<int>()))
19            .ReturnsAsync(new User());
20
21        // create user id!
22        testService.createUser(
23            fakeUserName,
24            password
25        );
26
27        // assert
28        Mock<UserRepository> mock = Mock.get(repository);
29        mock.verify(repo =>
30            repo.insert(It.Is<User>(user =>
31                user.UserName == userName &&
```

서비스 로직 코드 실행

```
12
13          // DI
14          UserService testService = new UsersDomainModel(repository, passwordHasher);
15
16          // Mocking 'find()' methods.
17          Mock.get(repository)
18              .Setup(x => x.find(It.IsAny<int>()))
19              .ReturnsAsync(new User());
20
21      // create user id!
22      testService.createUser(
23          fakeUserName,
24          password
25      );
26
27      // assert
28      Mock<UserRepository> mock = Mock.get(repository);
29      mock.verify(repo =>
30          repo.insert(It.Is<User>(user =>
31              user.UserName == userName &&
32              user.PasswordHash == passwordHash)));
33  }
```

Assert

테스트 코드를 보겠습니다.

- Repository에 insert명령을 통해 유저 데이터를 넣는것을 기대하는 테스트인데… 테스트의 의도와는 상관없는 코드가 들어가있네요.

```
15
16      // Mocking 'find()' methods.
17      Mock.get(repository)
18          .Setup(x => x.find(It.IsAny<int>()))
19          .ReturnsAsync(new User());
```

테스트 코드를 보겠습니다.

- Repository에 insert명령을 통해 유저 데이터를 넣는것을 기대하는 테스트인데… **테스트의 의도와는 상관없는 코드가** 들어가있네요.
- 우리가 이 테스트를 통해 검증하고자 하는것은
  - 실제로 repository에 값이 insert되는지이지
  - 메소드가 어떤 값을 반환하는지에 대한 내용이 아닙니다.

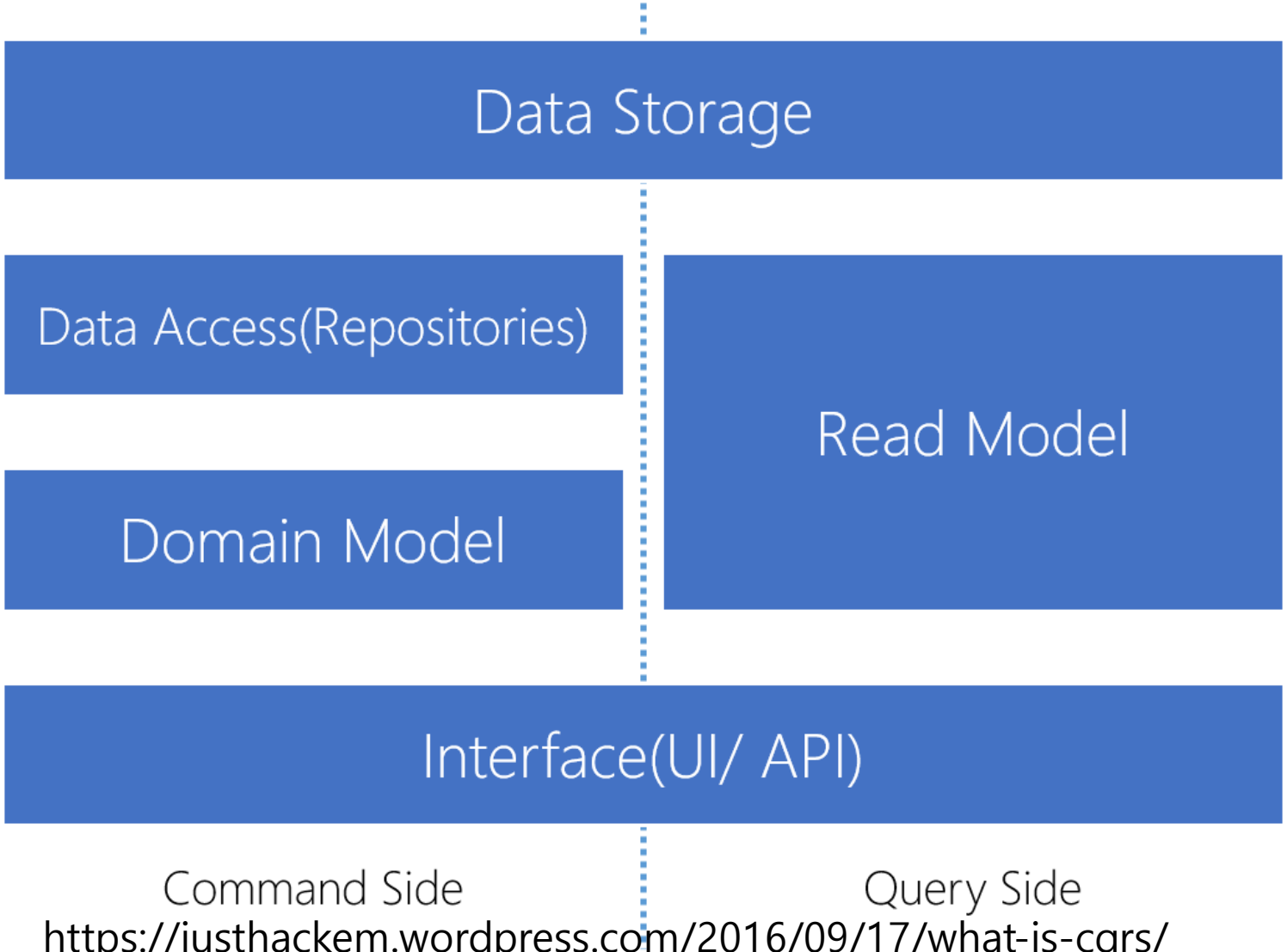
```
15
16        // Mocking 'find()' methods.
17        Mock.get(repository)
18            .Setup(x => x.find(It.IsAny<int>()))
19            .ReturnsAsync(new User());
```

문제 분석

- **테스트의 의도와는 상관없는 코드가 들어가 있습니다.**
  - 반환값에 대한 내용은 테스트에서 기대하지 않은 내용입니다.
  - 만약 하고싶다면, 별도 테스트케이스가 필요할것입니다.
  - 하지만 저 코드가 빠지면 NullpointException이 발생합니다.
  - 저장소에 대한 테스트 대역을 제대로 맞춰주지 않았기 때문입니다.
- **다시말해, Command 함수에 Query 소스가 껴있습니다.**
  - 각자의 역할을 가지고 있는 행위들이 한가지 소스에 가지고 있습니다.
  - 테스트와 리팩토링이 힘들어지는 원인이 될 수 있습니다.

https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/

CQRS 패턴을 활용한 개선방향



https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/

CQRS 패턴을 활용한 개선방향

- Insert함수에는 Command 명령만 처리하게 합니다. (책임 분리)
- 그리고 별도의 **ReadModel**을 만듭니다. 즉, Query명령 전용 Model 입니다.
- ReadModel은 Data Storage를 조회에만 집중합니다.

domainLayer.createUser();
한줄에서

domainLayer.createUser()
readModel.find();

두가지로 분리.

```
1 // UserController
2 @Post
3 public createUser(UserCommand command){
4     _domainLayer.createuser(command);
5     UserPresentation user = _readModel.find(command.userId);
6     return "/user";
7 }
```

자세한 구현은 https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/ 를 참조해주세요

이벤트 소싱과 CQRS

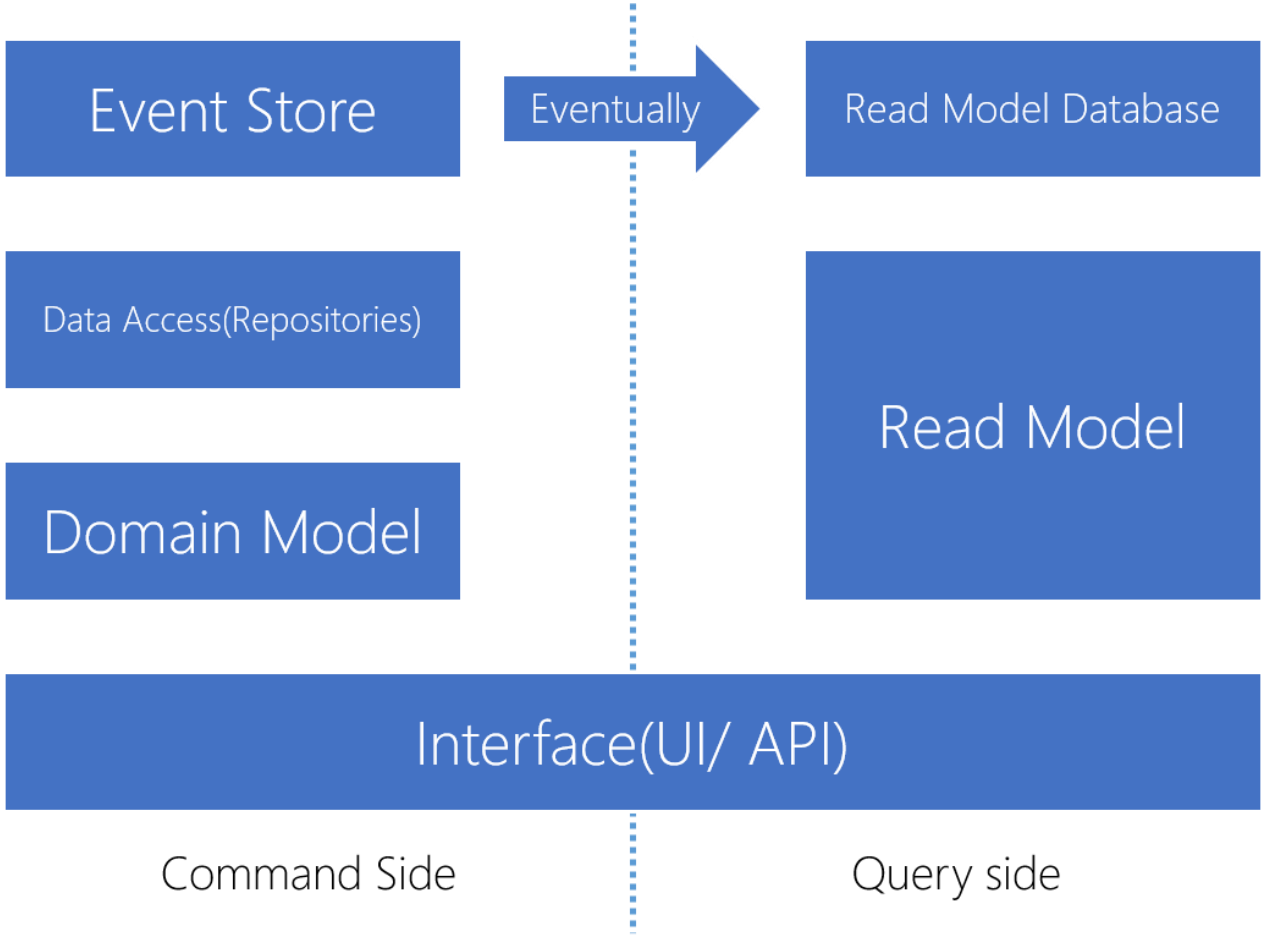
- 이벤트 소싱의 문제..?
- **“재고가 10개미만의 상품을 조회해주세요”**

이벤트 소싱과 CQRS

- 스냅샷도 소용없다..
- 전부 로드해서 상태를 복원
  - 조회… 너무느리다..!

이벤트 소싱과 CQRS

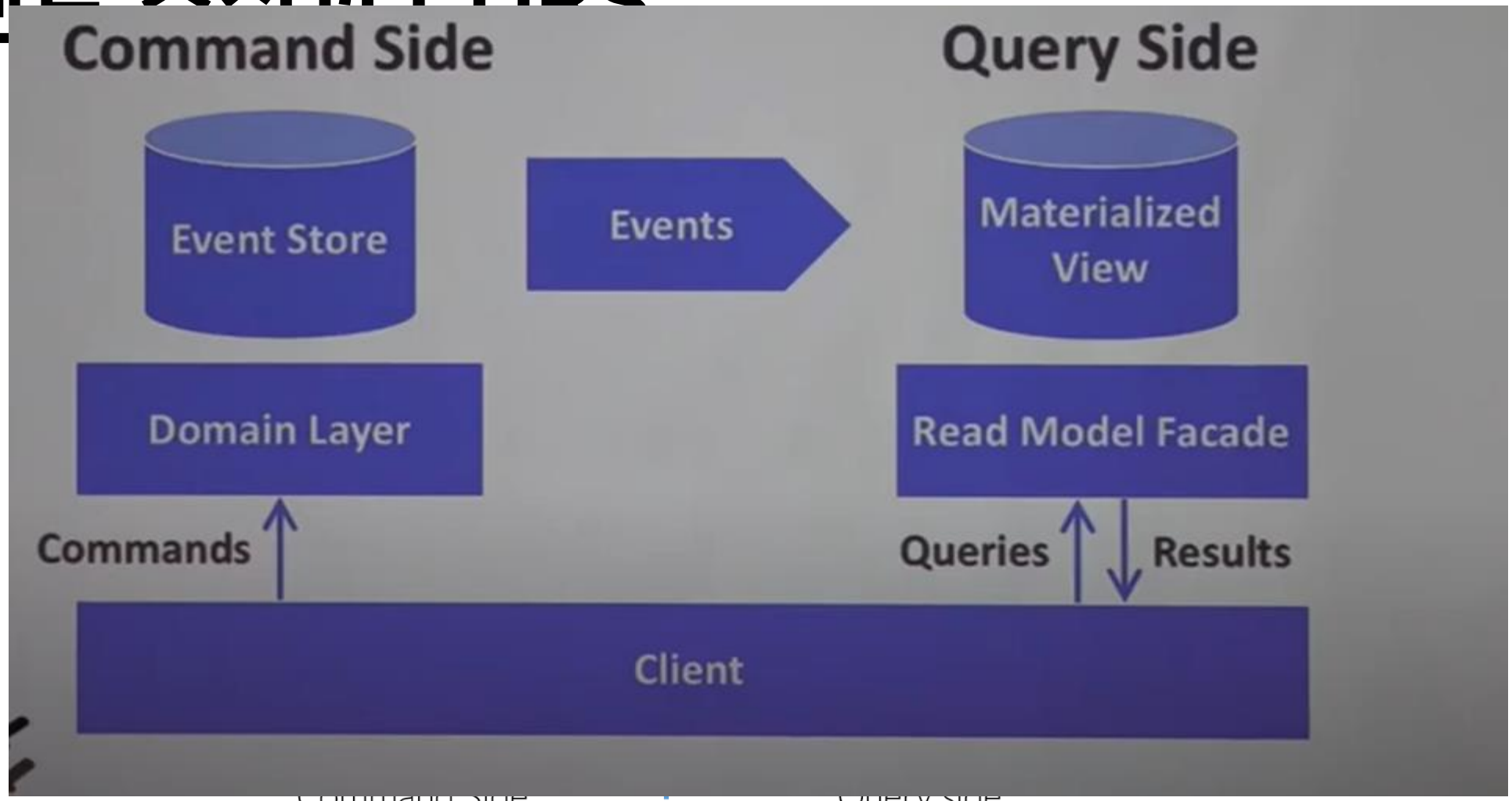
- 별도의 쿼리 전용 DB를 둔다!



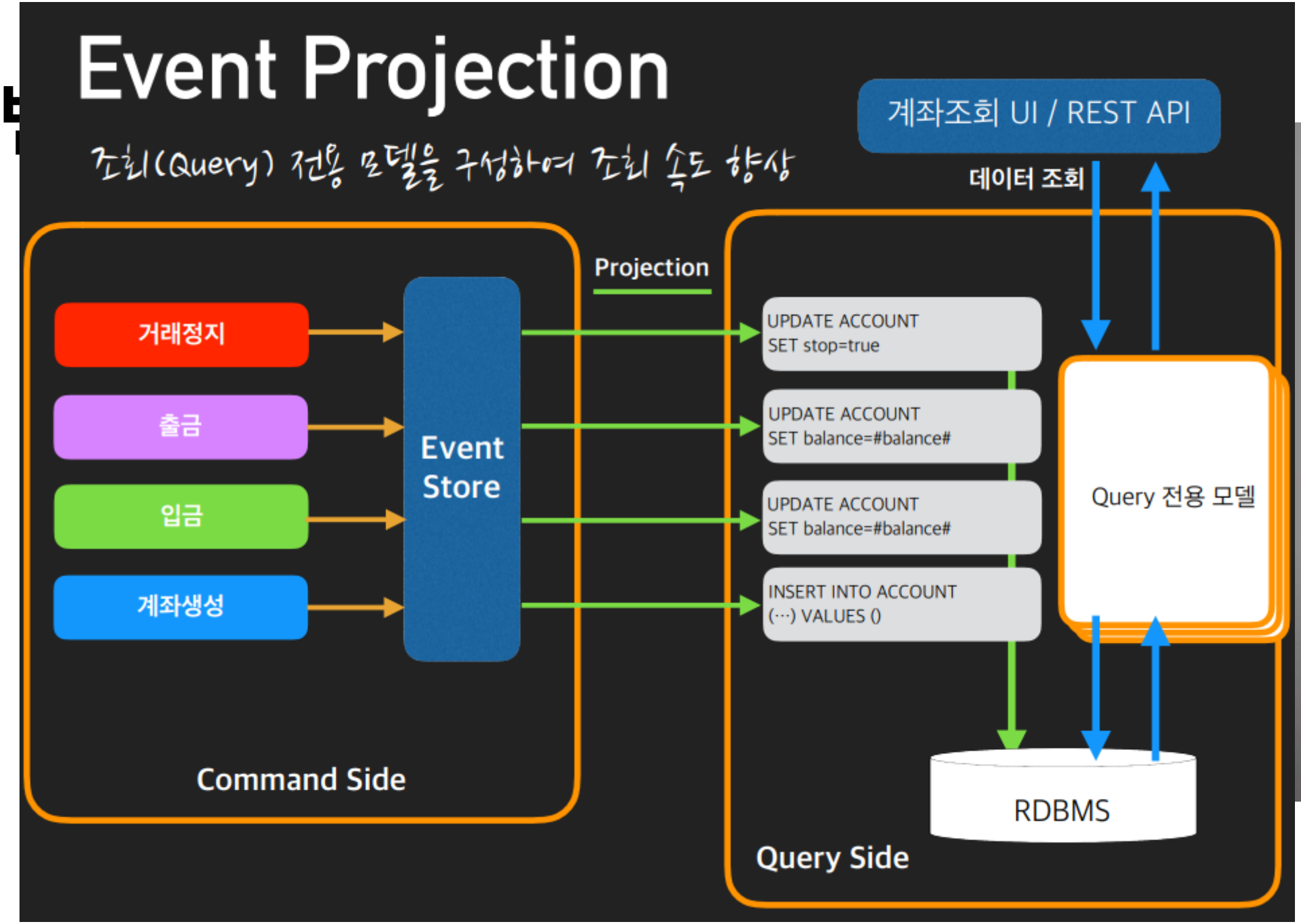
이벤트 소싱과 CQRS

- 별도의 쿼리 전용 DB를 둔다!
- 별도의 쿼리 전용 DB는 조회전용으로, 이벤트 저장소와는 성격이 다르다. (기존 RDBMS라고 생각해도 될것같다.)
  - **Query**는 Read Model & ReadModel Database에 위임한다 (CQRS의 장점)
  - **Command**는 도메인 이벤트를 중점적으로 이벤트 저장소에 영속적으로 저장한다. (이벤트 소싱의 장점)

이벤트 소싱과 CQRS

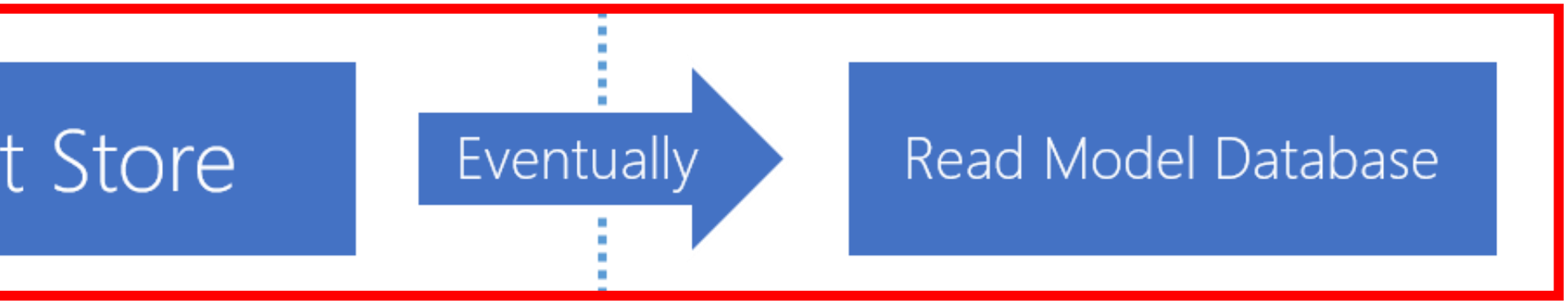

Command Side
Query Side
Event Store
Events
Materialized View
Domain Layer
Read Model Facade
Commands
Queries
Results
Client

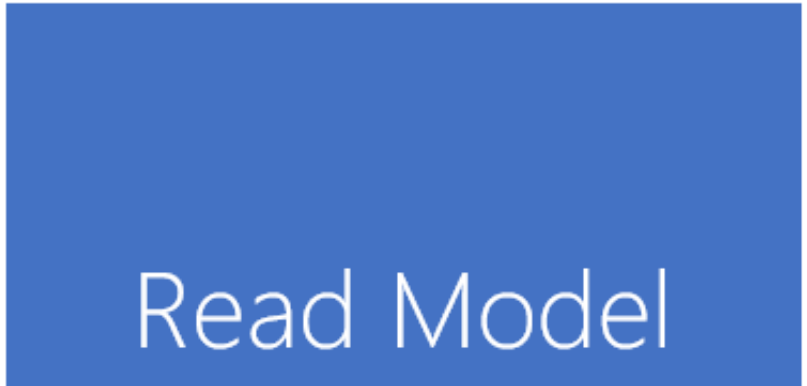
이


Event Projection
조회(Query) 전용 모델을 구성하여 조회 속도 향상
계좌조회 UI / REST API
데이터 조회
Projection
거래정지
출금
입금
계좌생성
Event Store
UPDATE ACCOUNT
SET stop=true
UPDATE ACCOUNT
SET balance=#balance#
UPDATE ACCOUNT
SET balance=#balance#
INSERT INTO ACCOUNT
(…) VALUES ()
Query 전용 모델
RDBMS
Command Side
Query Side

Hmm..

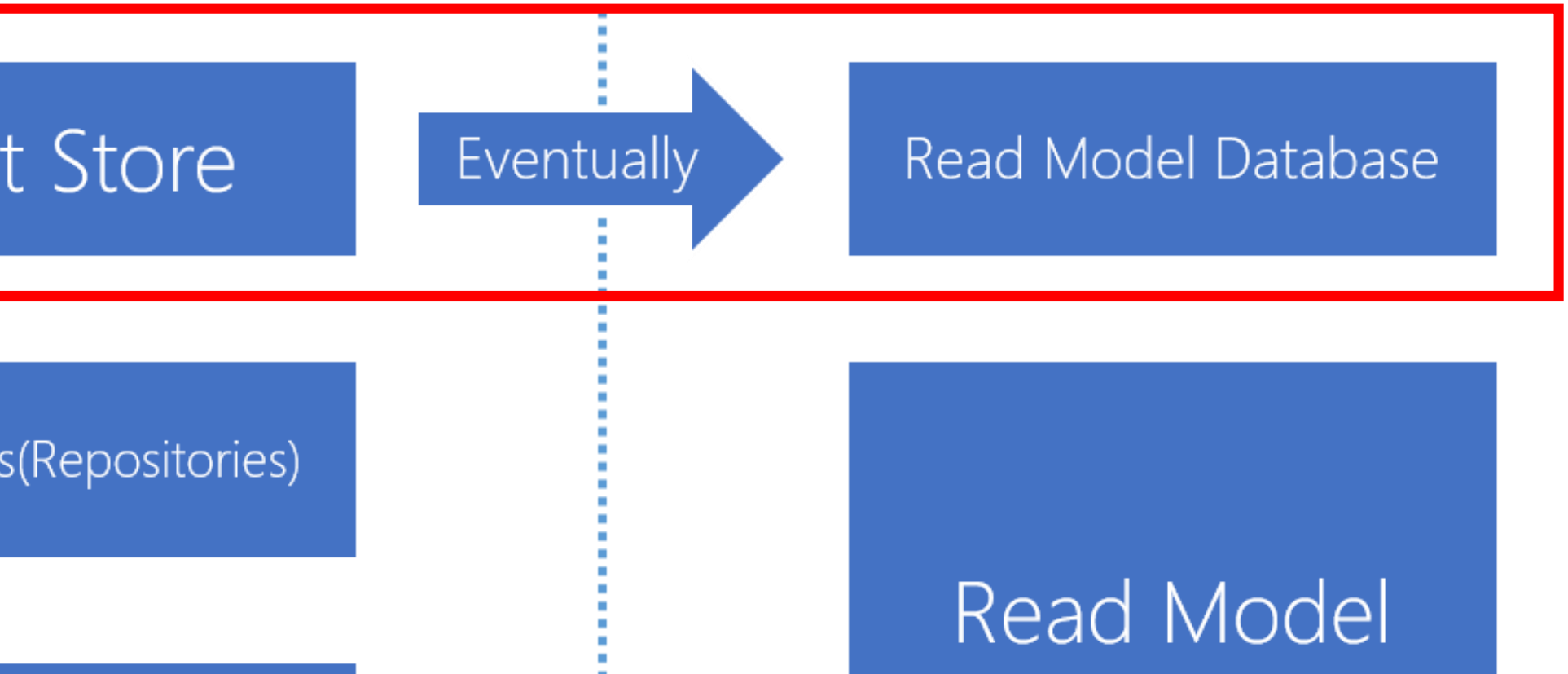
- 음..? 이게 어떻게 이루어지는건가요???





Hmm..

- 비동기적인 처리?? 데이터 일관성에 대한 처리??
- 에러발생시 이벤트 로그 유실?



메시지 드리븐 아키텍처

Message-driven architecture

는

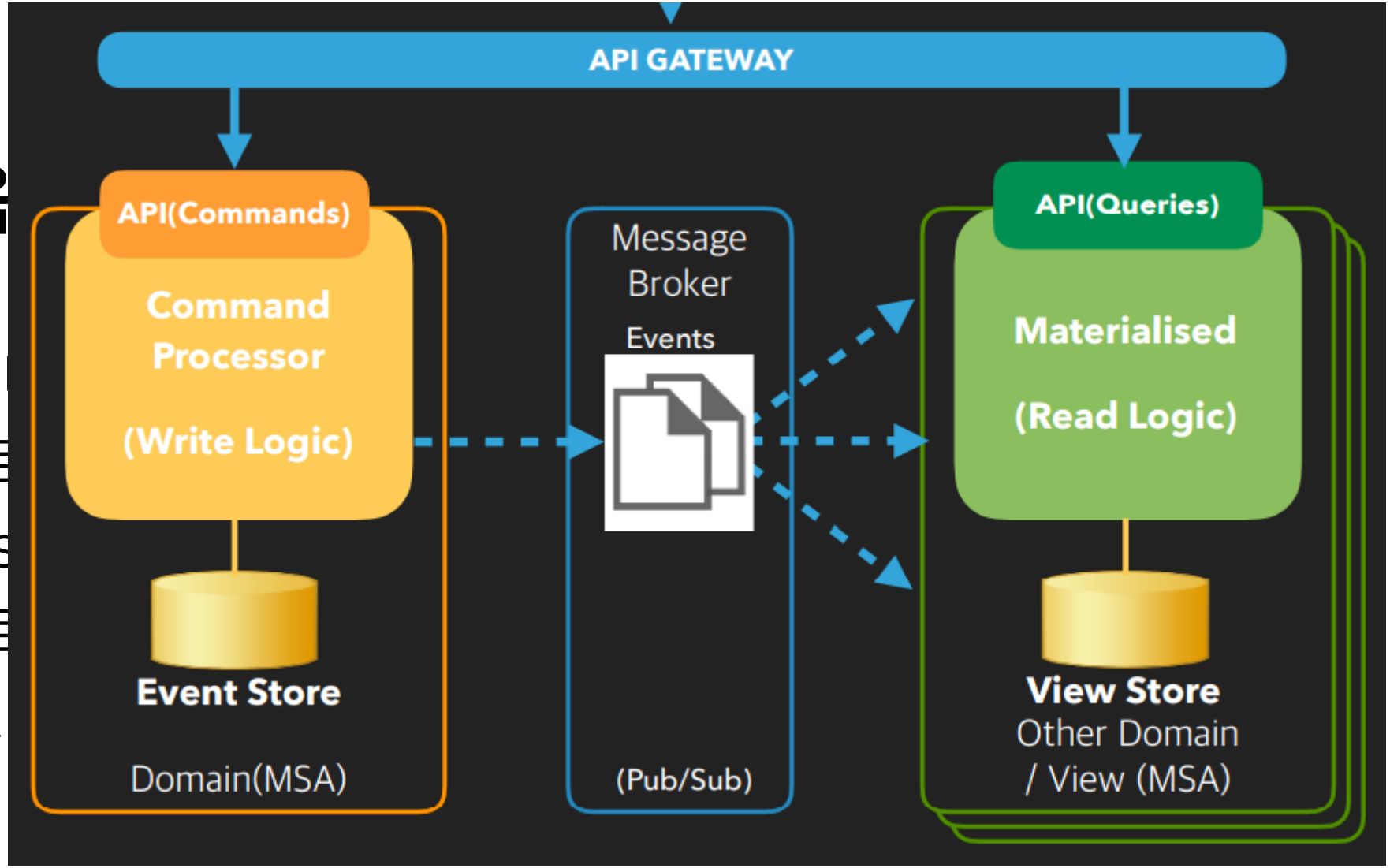
• 다음주

- 조금 더 디테일한 개념.
- 이벤트 소싱 FAQ.
- Message-driven architecture.
- 이벤트 소싱 Java 구현 예제 분석.
- MSA 적용 예시 및 장애 전파 방지.

는

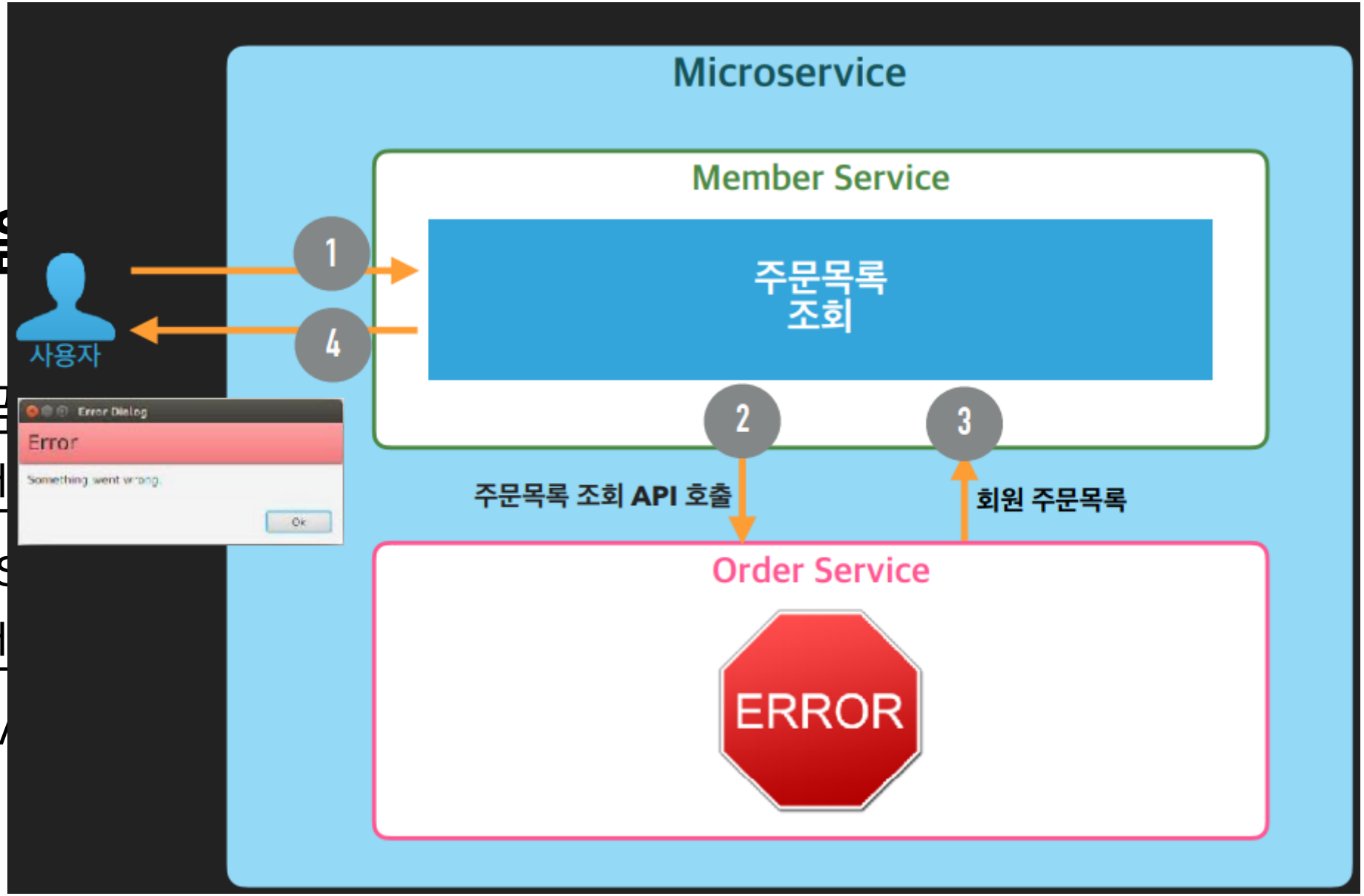
- 다음

- 조금
- 이벤트
- Mess
- 이벤트
- MSA



는

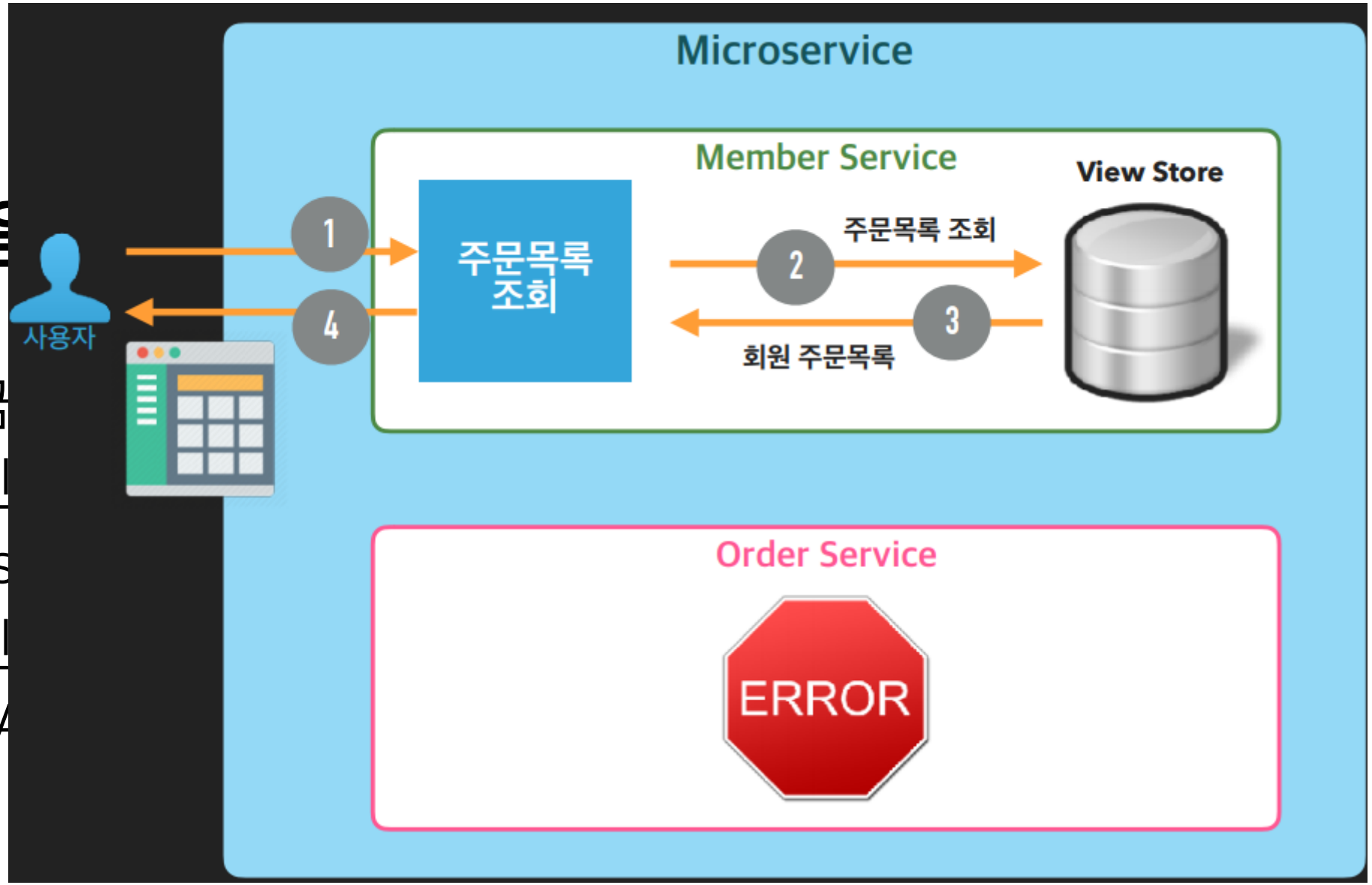
- 다음
- 조금
- 이번
- Mes
- 이번
- MSA



는

- 다음

- 조금
- 이벤
- Mes
- 이벤
- MSA



이벤트소싱 TRADE-OFF

- 분산시스템이 필요한가?
- 비즈니스의 이득이 충분히 생기는가?
- 비동기적인 서비스가 필요한가?
- 비즈니스 모델이 많이 충분히 복잡한가?
- 이해하고 구현할 수 있는 사람들이 충분히 있는가? (러닝커브 및 READ MODEL, 이벤트 핸들러 구현량)
- 관련 인프라를 이해할 수 있고, 에러에 대응할 수 있는가?
- 변경사항에 대해 도메인 객체에 대한 이력이 필요한가?
- MSA환경인가?

정리

이벤트 소싱	조회모델의 필요성	CQRS
분산시스템	이벤트 전달 (projection)	메시지
비동기	MSA	MSA 장애 전파 방지

Reference

- https://www.dotnetcurry.com/patterns-practices/1461/command-query-separation-cqs
- https://martinfowler.com/bliki/CommandQuerySeparation.html
- https://martinfowler.com/articles/201701-event-driven.html
- https://justhackem.wordpress.com/2017/02/05/introducing-event-sourcing/
- https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/
- https://martinfowler.com/bliki/CQRS.html
- https://www.facebook.com/groups/1919342698298205

- 유튜브 강의
- https://www.youtube.com/watch?v=Yd7TXUdcaUQ
- https://www.youtube.com/watch?v=TDhknOIYvw4
- https://www.youtube.com/watch?v=wJqHFpRmkmw
- https://www.youtube.com/watch?v=12EGxMB8SR8

- 구현 코드
- https://github.com/jaceshim/springcamp2017