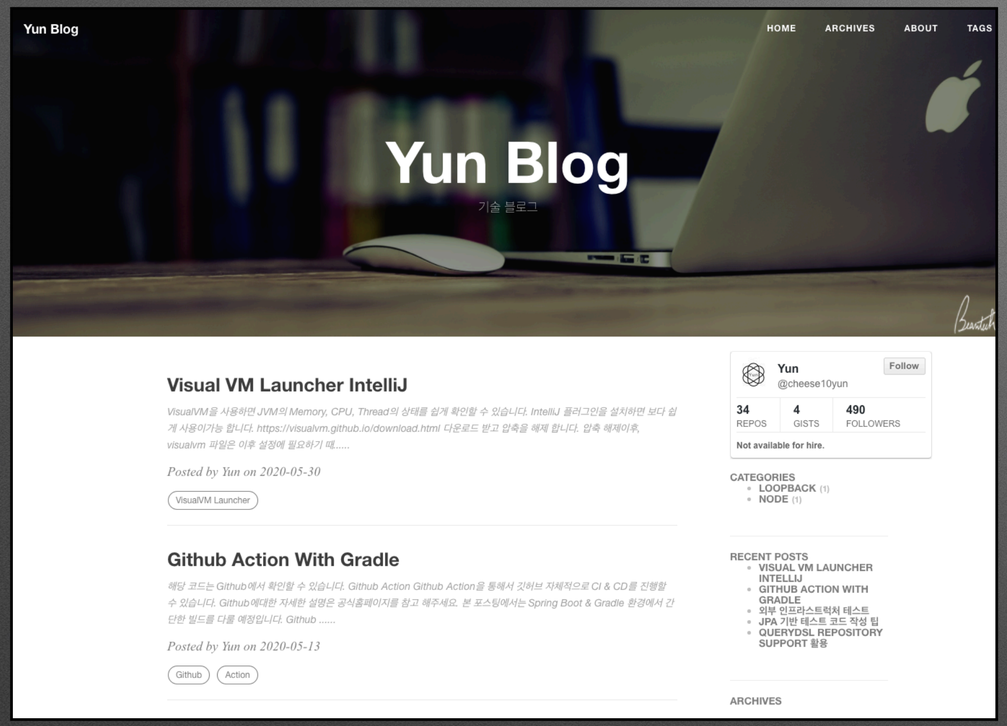
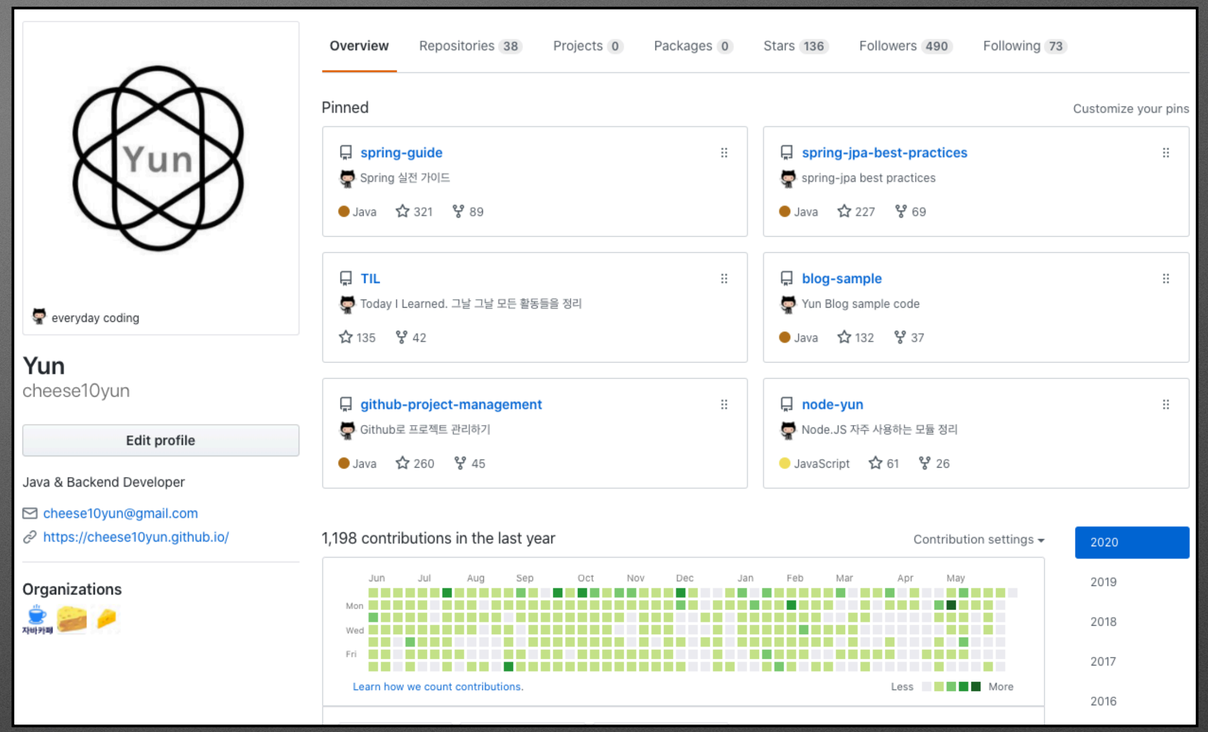
Spring Guide

강의 자료 링크 QR (PDF File)

소개





Yun

- https://cheese10yun.github.io/
- https://github.com/cheese10yun

강의 목적

- 그 동안 서버 개발을 하면선 좋은 코드를 만들기 위해서 노력한 것들을 공유

1. 디렉토리 구조

Layer VS Domain

```
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── demo
    │   │               ├── DemoApplication.java
    │   │               ├── config
    │   │               ├── controller
    │   │               ├── dao
    │   │               ├── domain
    │   │               ├── exception
    │   │               └── service
    │   └── resources
    │       └── application.properties
```

```
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── demo
    │   │               ├── DemoApplication.java
    │   │               ├── coupon
    │   │               │   ├── controller
    │   │               │   ├── domain
    │   │               │   ├── exception
    │   │               │   ├── repository
    │   │               │   └── service
    │   │               ├── member
    │   │               │   ├── controller
    │   │               │   ├── domain
    │   │               │   ├── exception
    │   │               │   ├── repository
    │   │               │   └── service
    │   │               └── order
    │   │                   ├── controller
    │   │                   ├── domain
    │   │                   ├── exception
    │   │                   ├── repository
    │   │                   └── service
    │   └── resources
    │       └── application.properties
```

Layer

```
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── demo
    │   │               ├── DemoApplication.java
    │   │               ├── config
    │   │               ├── controller
    │   │               ├── dao
    │   │               ├── domain
    │   │               ├── exception
    │   │               └── service
    │   └── resources
    │       └── application.properties
```

장점

- 프로젝트에 이해가 낮아도 전체적인 구조를 빠르게 파악 가능
- 작성 하고자하는 계층이 명확할 경우 빠르게 개발 가능

단점

- 각 레이어별로 수십개의 클래스들이 존재 하여 코드 파악이 어려움
- Layer를 기준으로 분리했기 때문에 코드의 응집력이 떨어짐

Domain

```
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── demo
    │   │               ├── DemoApplication.java
    │   │               ├── coupon
    │   │               │   ├── controller
    │   │               │   ├── domain
    │   │               │   ├── exception
    │   │               │   ├── repository
    │   │               │   └── service
    │   │               ├── member
    │   │               │   ├── controller
    │   │               │   ├── domain
    │   │               │   ├── exception
    │   │               │   ├── repository
    │   │               │   └── service
    │   │               └── order
    │   │                   ├── controller
    │   │                   ├── domain
    │   │                   ├── exception
    │   │                   ├── repository
    │   │                   └── service
    │   └── resources
    │       └── application.properties
```

장점

- 관련된 코드들이 응집해 있음
- 디렉토리 구조를 통해 도메인을 이해할 수 있음

단점

- 도메인 지식 없이 이해하기 어려움
- 각 계층을 구분하기 위한 논의가 필요

```
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── spring
    │   │           └── guide
    │   │               ├── ApiApp.java
    │   │               ├── SampleApi.java
    │   │               ├── domain
    │   │               │   ├── coupon
    │   │               │   │   ├── api
    │   │               │   │   ├── application
    │   │               │   │   ├── dao
    │   │               │   │   ├── domain
    │   │               │   │   ├── dto
    │   │               │   │   └── exception
    │   │               │   ├── member
    │   │               │   │   ├── api
    │   │               │   │   ├── application
    │   │               │   │   ├── dao
    │   │               │   │   ├── domain
    │   │               │   │   ├── dto
    │   │               │   │   └── exception
    │   │               │   └── model
    │   │               │       ├── Address.java
    │   │               │       ├── Email.java
    │   │               │       └── Name.java
    │   │               ├── global
    │   │               │   ├── common
    │   │               │   │   ├── request
    │   │               │   │   └── response
    │   │               │   ├── config
    │   │               │   │   ├── SwaggerConfig.java
    │   │               │   │   ├── properties
    │   │               │   │   ├── resttemplate
    │   │               │   │   └── security
    │   │               │   ├── error
    │   │               │   │   ├── ErrorResponse.java
    │   │               │   │   ├── GlobalExceptionHandler.java
    │   │               │   │   └── exception
    │   │               │   └── util
    │   │               └── infra
    │   │                   ├── email
    │   │                   └── sms
    │   │                       ├── AmazonSmsClient.java
    │   │                       ├── SmsClient.java
    │   │                       └── dto
    │   └── resources
    │       ├── application-dev.yml
    │       ├── application-local.yml
    │       ├── application-prod.yml
    │       └── application.yml
```

- domain : 도메인을 담당
- global : 프로젝트의 전체담당
- Infra: 외부 인프라스트럭처 담당

Domain

```
├── domain
│   ├── member
│   │   ├── api
│   │   │   └── MemberApi.java
│   │   ├── application
│   │   │   ├── MemberProfileService.java
│   │   │   ├── MemberSearchService.java
│   │   │   ├── MemberSignUpRestService.java
│   │   │   └── MemberSignUpService.java
│   │   ├── dao
│   │   │   ├── MemberFindDao.java
│   │   │   ├── MemberPredicateExecutor.java
│   │   │   ├── MemberRepository.java
│   │   │   ├── MemberSupportRepository.java
│   │   │   └── MemberSupportRepositoryImpl.java
│   │   ├── domain
│   │   │   ├── Member.java
│   │   │   └── ReferralCode.java
│   │   ├── dto
│   │   │   ├── MemberExistenceType.java
│   │   │   ├── MemberProfileUpdate.java
│   │   │   ├── MemberResponse.java
│   │   │   └── SignUpRequest.java
│   │   └── exception
│   │       ├── EmailDuplicateException.java
│   │       ├── EmailNotFoundException.java
│   │       └── MemberNotFoundException.java
│   └── model
│       ├── Address.java
│       ├── Email.java
│       └── Name.java
```

- api : 컨트롤러 클래스들이 존재합니다. 외부 rest api로 프로젝트를 구성하는 경우가 많으니 api라고 지칭했습니다.
- domain : 도메인 엔티티에 대한 클래스로 구성됩니다. 특정 도메인에만 속하는 Embeddable, Enum 같은 클래스도 구성됩니다.
- dto : 주로 Request, Response 객체들로 구성됩니다.
- exception : 해당 도메인이 발생시키는 Exception으로 구성됩니다.

global

```
├── global
│   ├── common
│   │   ├── request
│   │   └── response
│   │       └── Existence.java
│   ├── config
│   │   ├── SwaggerConfig.java
│   │   ├── properties
│   │   ├── resttemplate
│   │   │   ├── RestTemplateClientHttpRequestInterceptor.java
│   │   │   ├── RestTemplateConfig.java
│   │   │   └── RestTemplateErrorHandler.java
│   │   └── security
│   ├── error
│   │   ├── ErrorResponse.java
│   │   ├── GlobalExceptionHandler.java
│   │   └── exception
│   │       ├── BusinessException.java
│   │       ├── EntityNotFoundException.java
│   │       ├── ErrorCode.java
│   │       └── InvalidValueException.java
│   └── util
```

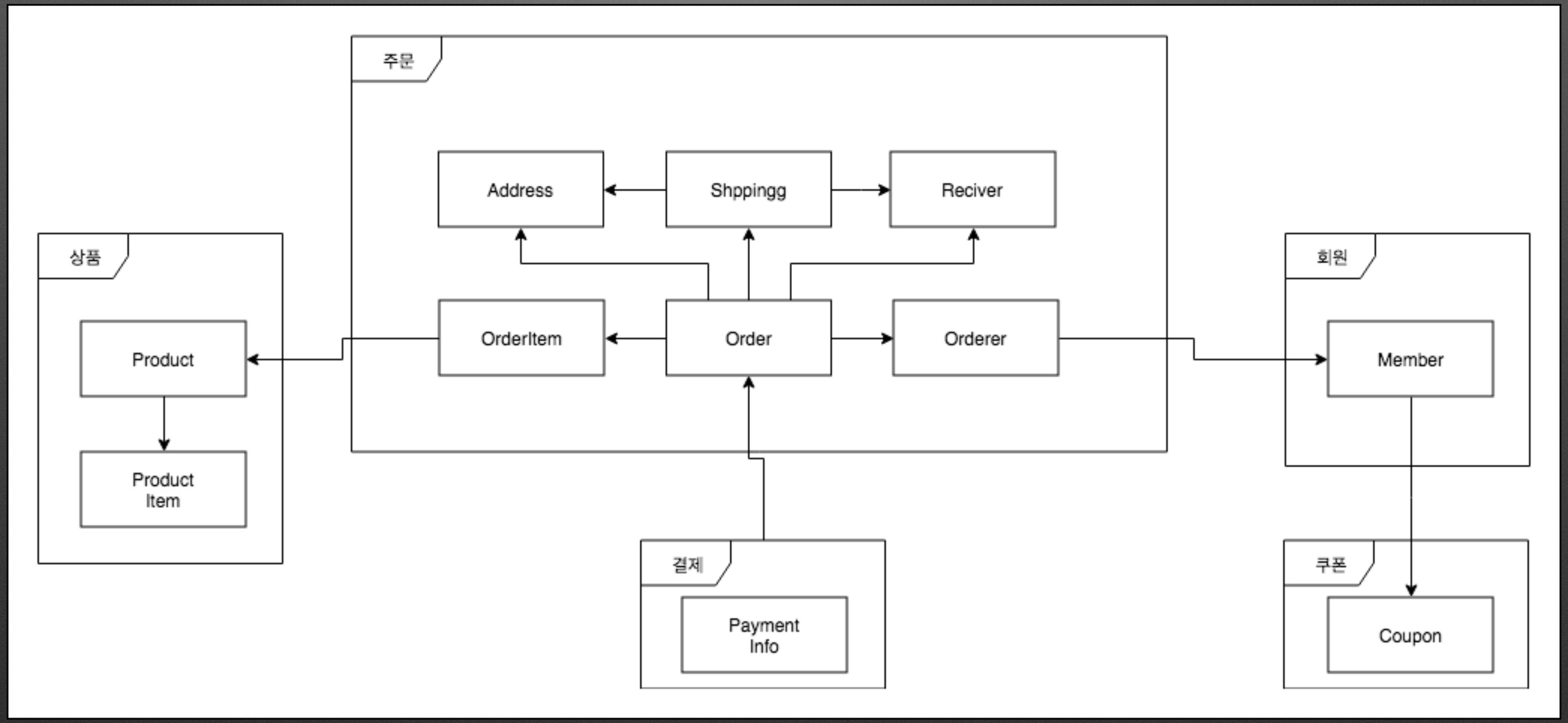
- global은 프로젝트 전방위적으로 시용되는 객체들로 구성됩니다.
- common : 공통으로 사용되는 Value 객체들로 구성됩니다. 페이징 처리를 위한 Request, 공통된 응답을 주는 Response 객체들이 있습니다.
- config : 스프링 각종 설정들로 구성됩니다.
- error : 예외 핸들링을 담당하는 클래스로 구성됩니다. Exception Guide에서 설명했던 코드들이 있습니다.
- util : 유틸성 클래스들이 위치합니다.

infra

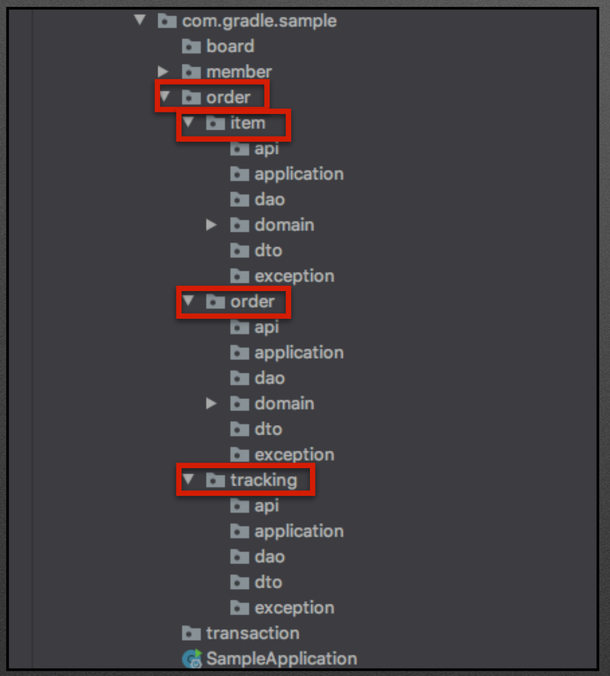
```
└── infra
    ├── email
    └── sms
        ├── AmazonSmsClient.java
        ├── KtSmsClient.java
        ├── SmsClient.java
        └── dto
            └── SmsRequest.java
```

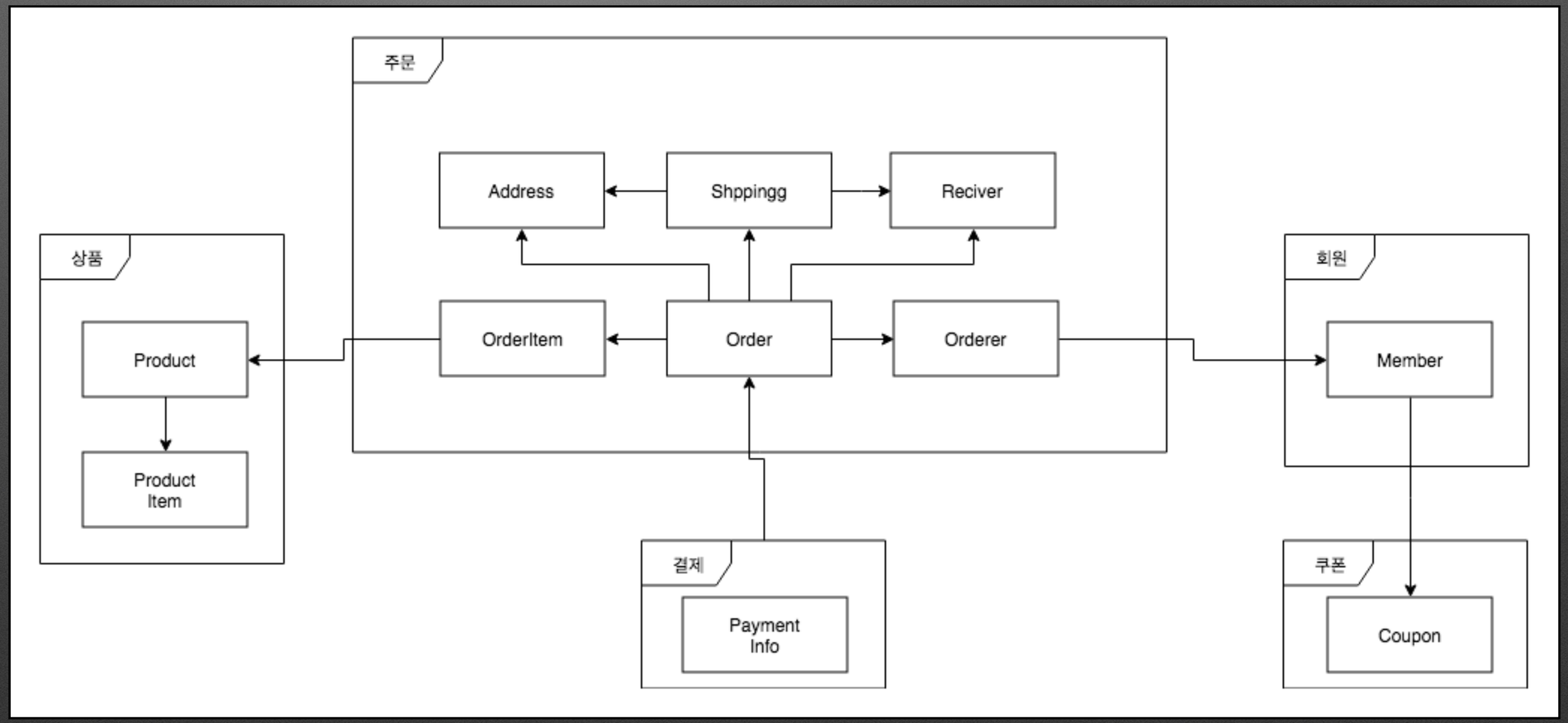
- infra 디렉터리는 인프라스트럭처 관련된 코드들로 구성됩니다.
- common : 공통으로 사용되는 Value 객체들로 구성됩니다. 페이징 처리를 위한 Request, 공통된 응답을 주는 Response 객체들이 있습니다.
- config : 스프링 각종 설정들로 구성됩니다.
- error : 예외 핸들링을 담당하는 클래스로 구성됩니다. Exception Guide에서 설명했던 코드들이 있습니다.
- util : 유틸성 클래스들이 위치합니다.

구조

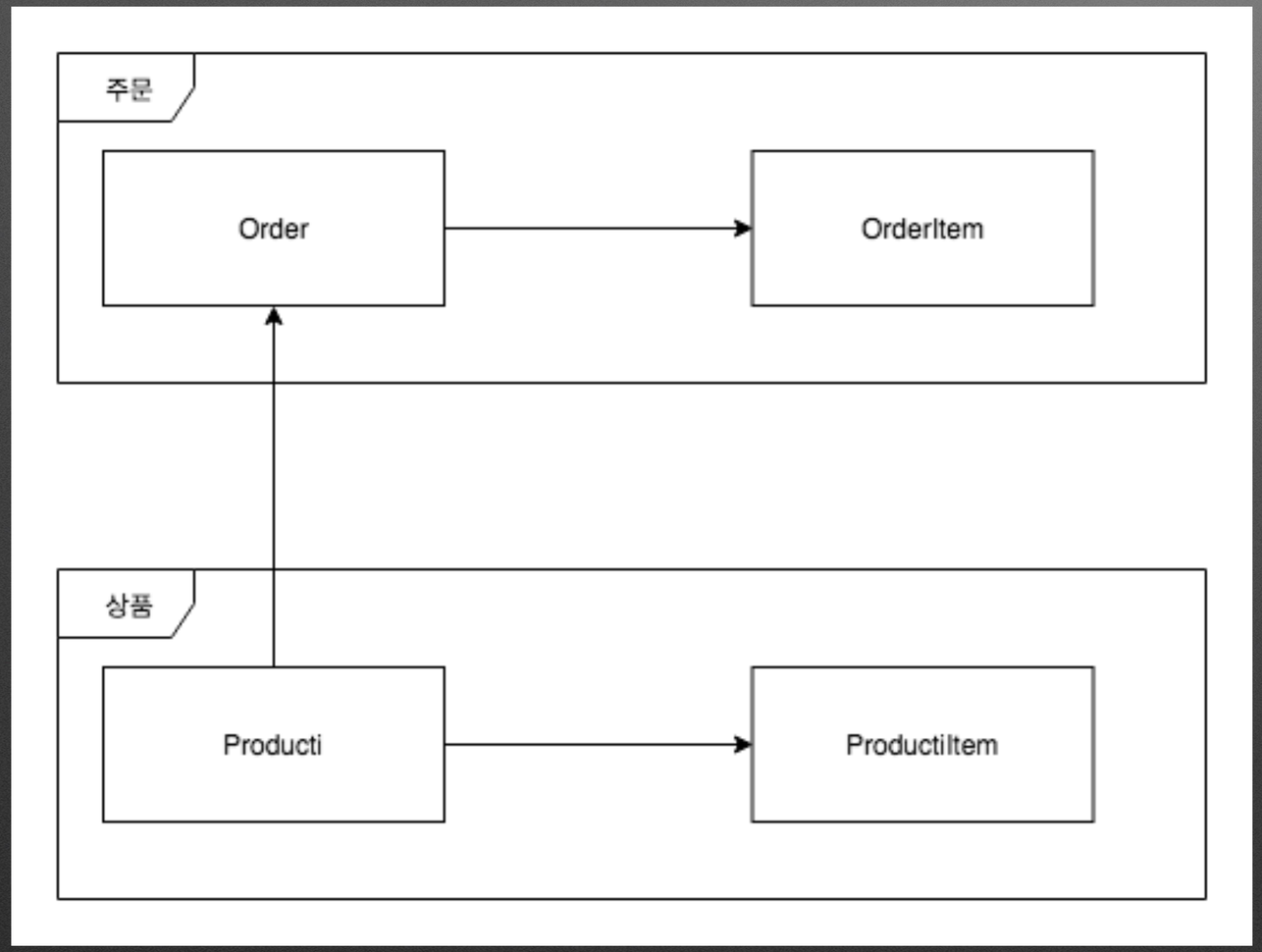

주문
Address
Shppingg
Reciver
OrderItem
Order
Orderer
상품
Product
Product
Item
회원
Member
결제
Payment
Info
쿠폰
Coupon

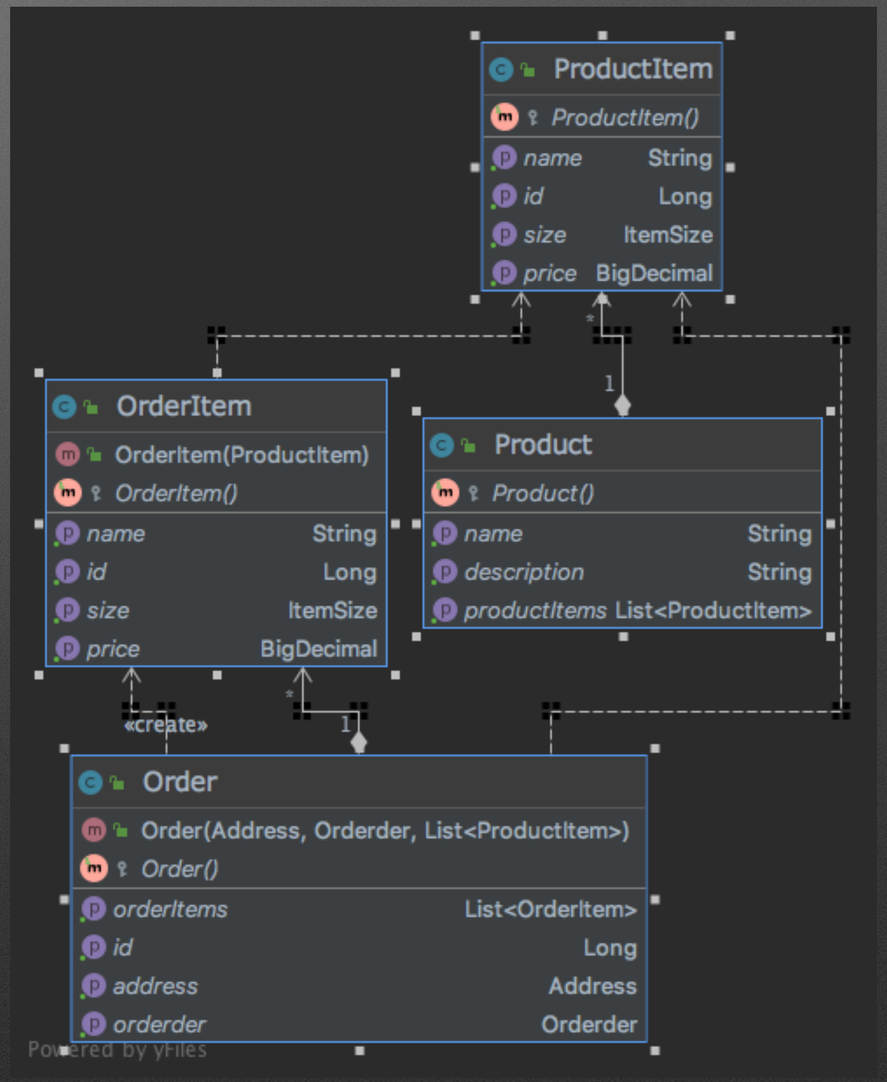
Aggregate


com.gradle.sample
board
member
order
item
api
application
dao
domain
dto
exception
order
api
application
dao
domain
dto
exception
tracking
api
application
dao
dto
exception
transaction
SampleApplication


주문
Address
Shppingg
Reciver
OrderItem
Order
Orderer
상품
Product
Product Item
회원
Member
결제
Payment Info
쿠폰
Coupon

강한 결합 관계


주문
Order
OrderItem
상품
Producti
ProductiItem


ProductItem
ProductItem()
name String
id Long
size ItemSize
price BigDecimal
OrderItem
OrderItem(ProductItem)
OrderItem()
name String
id Long
size ItemSize
price BigDecimal
Product
Product()
name String
description String
productItems List<ProductItem>
«create»
Order
Order(Address, Orderder, List<ProductItem>)
Order()
orderItems List<OrderItem>
id Long
address Address
orderder Orderder
Powered by yFiles

강한 결합 관계

```java
@Entity
@Table(name = "orders")
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Embedded
    private Address address;

    @Embedded
    private Orderder orderder;

    @ElementCollection
    @CollectionTable(name = "orders_item", joinColumns = @JoinColumn(name = "id"))
    private List<OrderItem> orderItems = new ArrayList<>();

    public Order(Address address, Orderder orderder, List<ProductItem> productItems) {
        Assert.notNull(address, message: "address must not be null");
        Assert.notNull(orderder, message: "address must not be null");
        Assert.notEmpty(productItems, message: "address must not be empty");

        this.address = address;
        this.orderder = orderder;
        for (ProductItem productItem : productItems) {
            orderItems.add(new OrderItem(productItem));
        }
    }
}
```

```java
@Embeddable
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class OrderItem {

    @Id
    private Long id;

    @Column(name = "name", nullable = false)
    private String name;

    @Enumerated(EnumType.STRING)
    @Column(name = "size", nullable = false)
    private ItemSize size;

    @Column(name = "price", nullable = false)
    private BigDecimal price;

    public OrderItem(ProductItem productItem) {
        this.name = productItem.getName();
        this.size = productItem.getSize();
    }
}
```

```java
@Embeddable
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class ProductItem {

    @Id
    private Long id;

    @Column(name = "name", nullable = false)
    private String name;

    @Enumerated(EnumType.STRING)
    @Column(name = "size", nullable = false)
    private ItemSize size;

    @Column(name = "price", nullable = false)
    private BigDecimal price;


}
```

- 주문은 상품이 있어야 주문이 가능하다 -> 상품 기반으로 주문을 만들어야 한다

객체 생성

```java
@Embeddable
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Orderder {

    @Column(name = "member_id")
    private Long memberId;

    @Column(name = "orderer_name", nullable = false)
    private String name;

    private Orderder(Long memberId, String name) {
        Assert.notNull(name, message: "name must be not null");
        this.memberId = memberId;
        this.name = name;
    }

    public static Orderder memberOrderer(Long memberId, String name) {
        Assert.notNull(memberId, message: "memberId must be not null");
        return new Orderder(memberId, name);
    }

    public static Orderder nonMemberOrderer(String name) {
        return new Orderder( memberId: null, name);
    }
}
```

- 회원 주문이, 비회원 주문이 있다 -> 해당 요구사항을 최대한 코드로 해석 가능하게

2. Lombok 사용법

Lombok을 잘사용 하는 방법은 단 하나

Lombok을 잘사용 하는 방법은 단 하나

Lombok을 사용하지 않는 것입니다.
코틀린 사용 하세요
하지만... 우린 자바를 써야겠죠...

@Data는 지양하자

```java
@Entity
@Table(name = "member")
@Data
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "email", nullable = false)
    private String email;

    @Column(name = "name", nullable = false)
    private String name;

    @CreationTimestamp
    @Column(name = "create_at", nullable = false, updatable = false)
    private LocalDateTime createAt;

    @UpdateTimestamp
    @Column(name = "update_at", nullable = false)
    private LocalDateTime updateAt;
}
```

- @Data는 @ToString, @EqualsAndHashCode, @Getter, @Setter, @RequiredArgsConstructor을 한방에…
- 마법의 코드의 대가는 언젠간 치루게 됩니다…

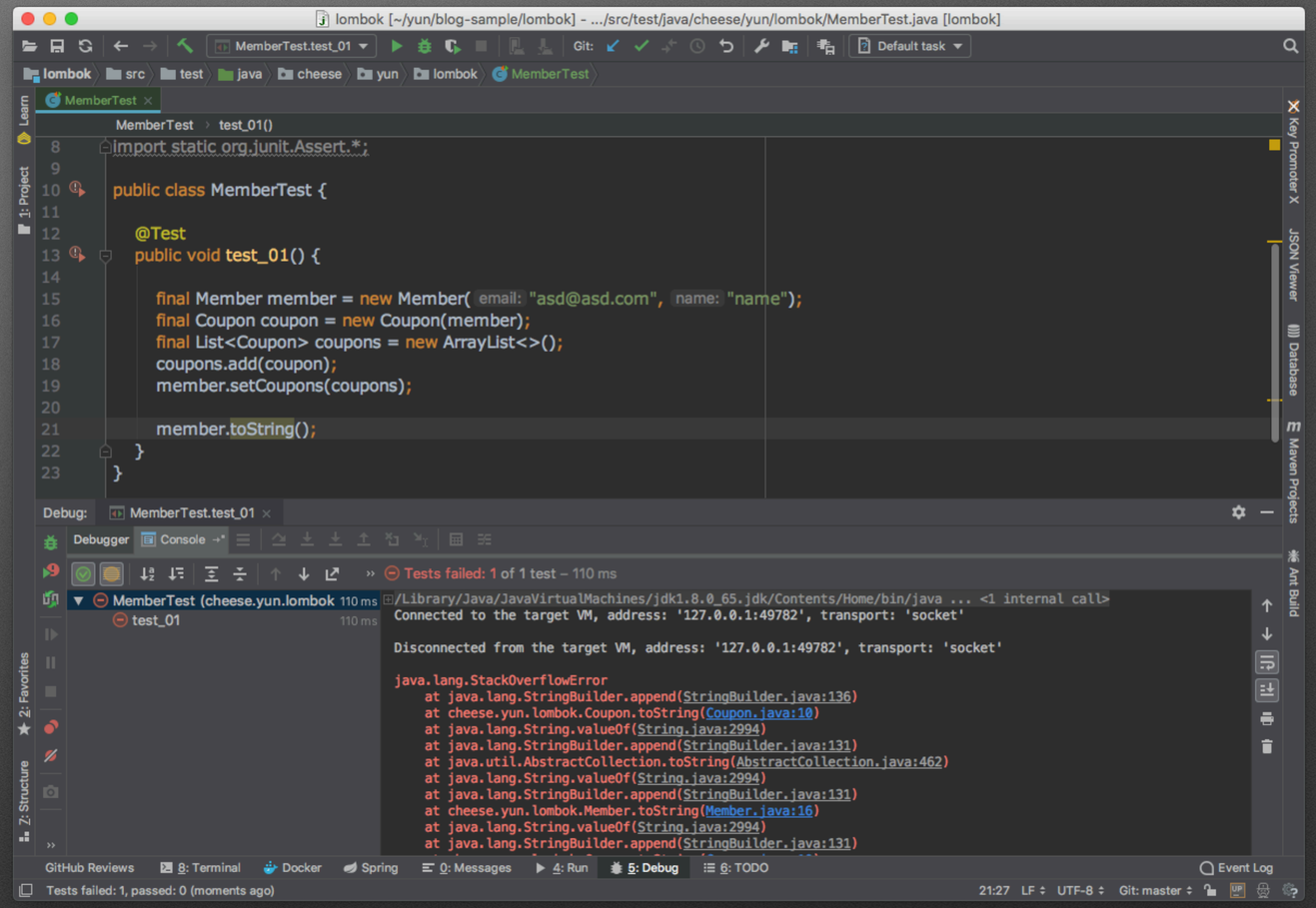
무분별한 @Setter 남용

```java
@Entity
@Table(name = "member")
@Data
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "email", nullable = false)
    private String email;

    @Column(name = "name", nullable = false)
    private String name;

    @CreationTimestamp
    @Column(name = "create_at", nullable = false, updatable = false)
    private LocalDateTime createAt;

    @UpdateTimestamp
    @Column(name = "update_at", nullable = false)
    private LocalDateTime updateAt;
}
```

- Setter는 그 의도가 분명하지 않고 객체를 언제든지 변경할 수 있는 상태가 되어서 객체의 안전성이 보장받기 힘듭니다.

ToString 양방향 순환 참조 문제

```java
@Entity
@Table(name = "member")
@Data
public class Member {
    ....
    @OneToMany
    @JoinColumn(name = "coupon_id")
    private List<Coupon> coupons = new ArrayList<>();
}

@Entity
@Table(name = "coupon")
@Data
public class Coupon {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @ManyToOne
    private Member member;

    public Coupon(Member member) {
        this.member = member;
    }
}
```

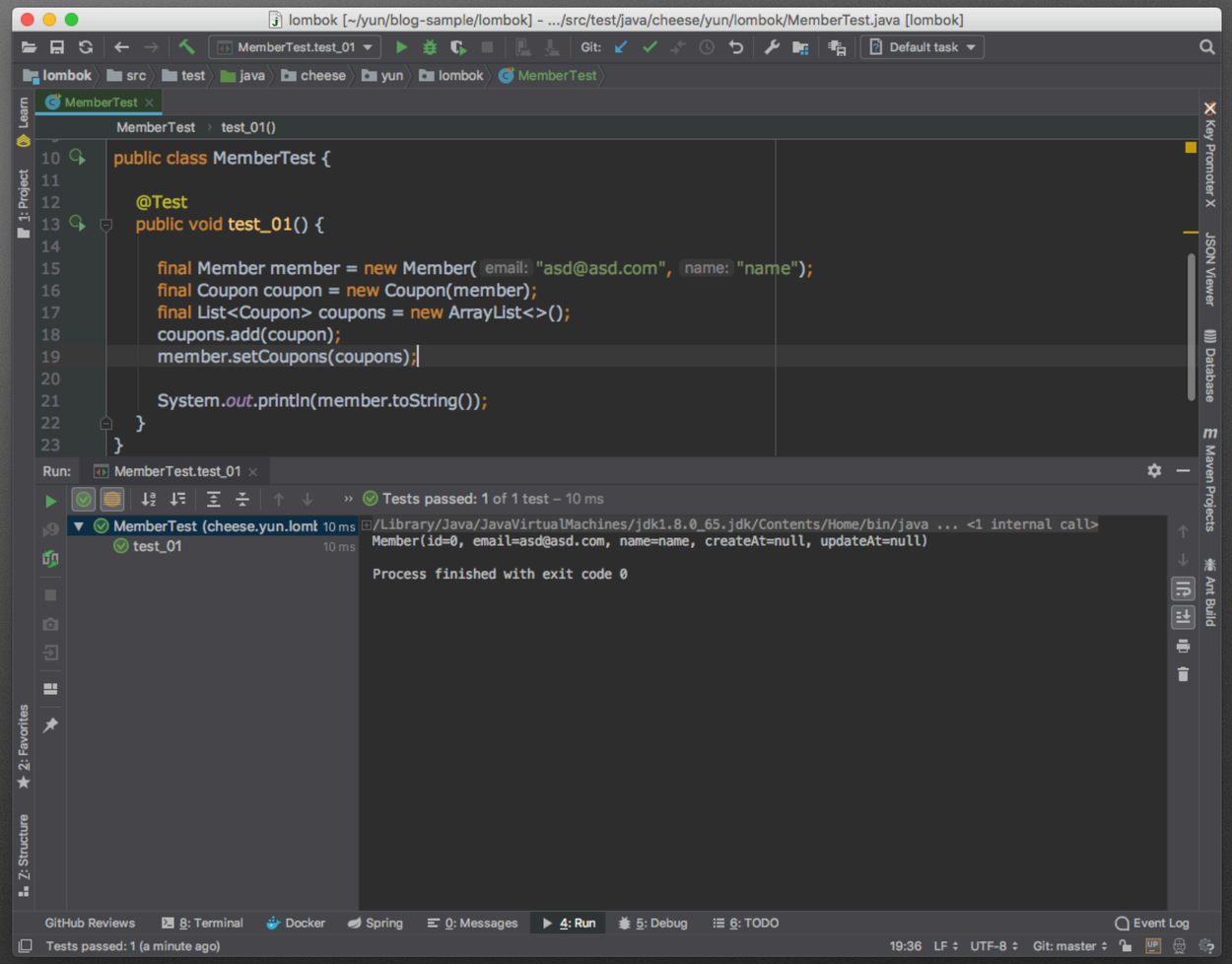


- Member 객체와 Coupon 객체가 양방향 영관관계일 경우 ToString을 호출하면 무한 순환 참조가 발생합니다.
- JPA를 사용하다 보면 객체를 Json으로 직렬화 하는 가정에서 발생하는 문제와 동일한 이유입니다.

ToString 양방향 순환 참조 문제

```java
@Entity
@Table(name = "member")
@Data
public class Member {
    ....
    @OneToMany
    @JoinColumn(name = "coupon_id")
    private List<Coupon> coupons = new ArrayList<>();
}

@Entity
@Table(name = "coupon")
@Data
public class Coupon {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @ManyToOne
    private Member member;

    public Coupon(Member member) {
        this.member = member;
    }
}
```

```java
@ToString(exclude = "coupons")
public class Member {...}
```



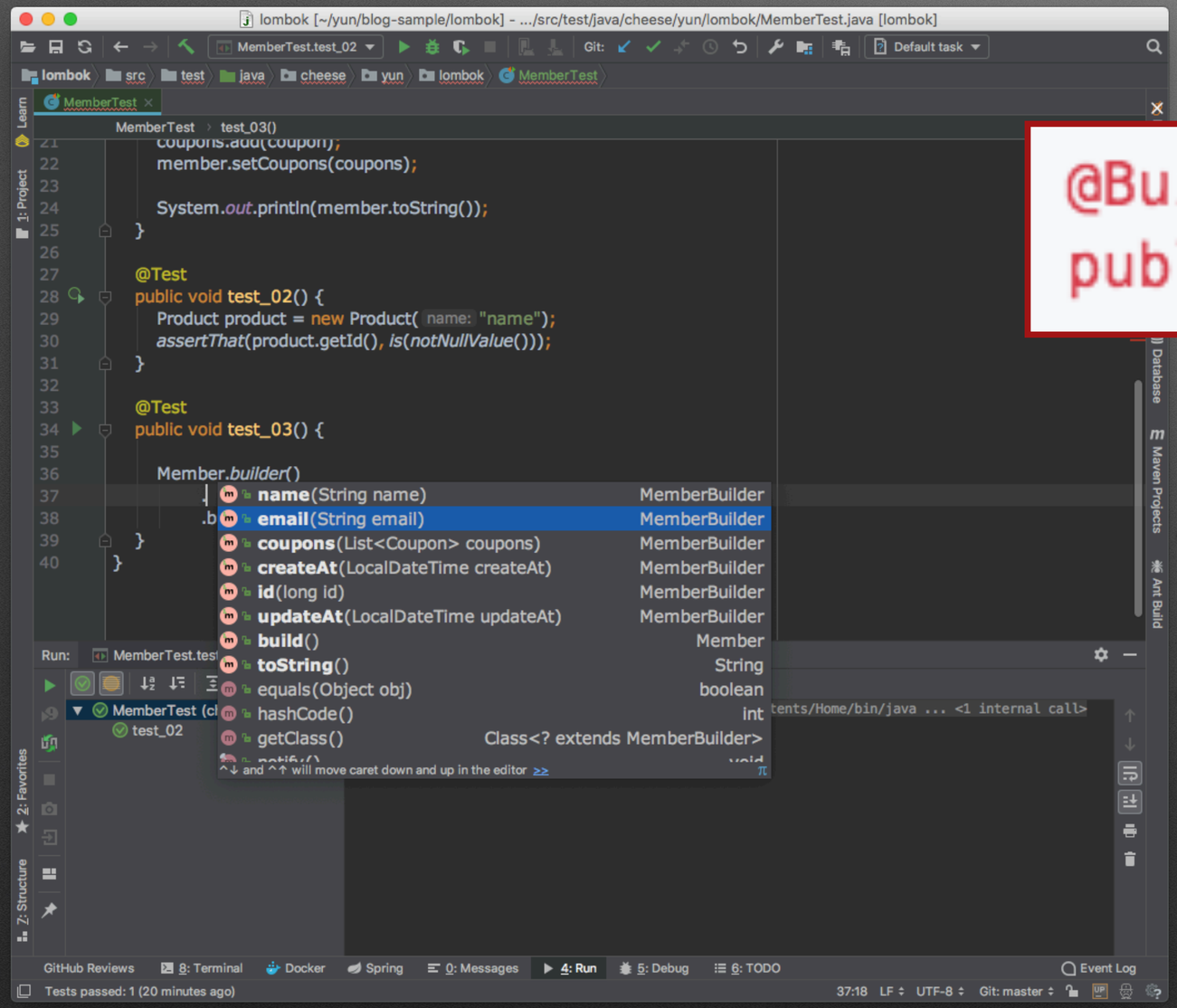
- 해당 어노테이션을 이용해서 ToString 항목에서 제외시키는 것입니다.
- 이 방법보다는 **@ToString(of = {“...”})** 방법을 더 권장합니다.

@EqualsAndHashCode 의 남발..

```java
public boolean equals(final Object o) {
    if (o == this) {
      return true;
    } else if (!(o instanceof Member)) {
      return false;
    } else {
      Member other = (Member)o;
      if (!other.canEqual(this)) {
        return false;
      } else if (this.getId() != other.getId()) {
        return false;
      } else {
        label73: {
          Object this$email = this.getEmail();
          Object other$email = other.getEmail();
          if (this$email == null) {
            if (other$email == null) {
              break label73;
            }
          } else if (this$email.equals(other$email)) {
            break label73;
          }

          return false;
        }

        Object this$name = this.getName();
        Object other$name = other.getName();
        if (this$name == null) {
          if (other$name != null) {
            return false;
          }
        } else if (!this$name.equals(other$name)) {
          return false;
        }
        ....
    }
  }
```

- @EqualsAndHashCode 멤버 필드 전체에 대한 equals 메서드가 진행됨(성능 이슈)
  - 성능 이슈가 Set 자료 구조를 사용할 경우
  - 기타등이슈가 발생할 수 있음 HashCode 생성하는 코드도 동일한 문제
- Lombok에서 자동으로 생성하는 *.class 파일을 한 번 씩 보는 것을 권장

클래스 상단의 @Builder는 지양 하자




```
@Builder
public class Member {...}
```

- 클래스 위에 @Builder를 사용 시 @AllArgsConstructor 효과 발생
- 모든 멤버 필드에 대해서 매개변수를 받는 기본 생성자를 만듭니다.

클래스 상단의 @Builder는 지양 하자

```
public class Member {

    @Builder
    public Member(String email, String name) {
        this.email = email;
        this.name = name;
    }
}
```

- 비지니스 로직에 맞게 필수 값에 대해서만 생성자만 열어둡니다.
- 해당 도메인, 요구사항에 맞게 관련 코드도 만들어야 합니다.

생성자위 Builder에 적절한 책임을 부여하자

```java
@Entity
@Table(name = "refund")
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Refund {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private long id;

  @Embedded
  private Account account;

  @Embedded
  private CreditCard creditCard;

  @OneToOne
  @JoinColumn(name = "order_id", nullable = false, updatable = false)
  private Order order;


  @Builder(builderClassName = "ByAccountBuilder", builderMethodName = "ByAccountBuilder") //
  public Refund(Account account, Order order) {
    Assert.notNull(account, "account must not be null");
    Assert.notNull(order, "order must not be null");

    this.order = order;
    this.account = account;
  }

  @Builder(builderClassName = "ByCreditBuilder", builderMethodName = "ByCreditBuilder")  //
  public Refund(CreditCard creditCard, Order order) {
    Assert.notNull(creditCard, "creditCard must not be null");
    Assert.notNull(order, "order must not be null");

    this.order = order;
    this.creditCard = creditCard;
  }
}
```

```java
public class RefundTest {
    ...
    ...

  @Test
  public void ByAccountBuilder_test() {
    final Refund refund = Refund.ByAccountBuilder() // 빌더 이름으로 명확하게 그 의도를 드러 내고 있습니다.
        .account(account)
        .order(order)
        .build();

    assertThat(refund.getAccount()).isEqualTo(account);
    assertThat(refund.getOrder()).isEqualTo(order);
  }

  @Test
  public void ByCreditBuilder_test() {
    final Refund refund = Refund.ByCreditBuilder() // 빌더 이름으로 명확하게 그 의도를 드러 내고 있습니다.
        .creditCard(creditCard)
        .order(order)
        .build();

    assertThat(refund.getCreditCard()).isEqualTo(creditCard);
    assertThat(refund.getOrder()).isEqualTo(order);
  }
}
```

- 주문에 대한 환불이 있을 경우 환불에 대한 금액을 신용 카드 취소, 계좌 기반 환불이 있을 수 있습니다
- 신용 카드 결제 취소일 경우 신용 카드 정보를 받아야 하고, 계좌 기반 환불에는 계좌 정보를 받아야 합니다.

@Builder.Default는 지양하자

```java
@Getter
@Builder
@ToString
public class Member {

  @Builder.Default
  private String name = "yun";

  @Builder.Default
  private String email = "yun.chee

  private String nickName = "asd";

  private List<Member> members = new ArrayList<>();

}
```

```java
public class MemberTest {

  @Test
  public void test() {

    final Member member = Member.builder().build();
    final Member member1 = Member.builder().email("kakapay").name("kakapay").build();

    System.out.println(member); // Member(name=yun, email=yun.cheese@kakao.com, nickName=null, members=
    System.out.println(member1); // Member(name=kakapay, email=kakapay, nickName=null, members=null)

  }
}
```

```
warning: @Builder will ignore the initializing expression entirely. If you want the initializing express
  private String nickName = "asd";
```

- 코드 예측이 어렵습니다.
- @Builder.Default 어노테이션이 없는 멤버 필드는 Builder로 생성시 할당 받지 못한다.
- POJO 스럽게 객체를 보는 것이 가장 가독성이 좋음

생성자 접근 지시자는 최소한으로 하자

```
@Entity
@Table(name = "refund")
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Refund { ... }
```

```
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public class RefundRequest {...}
```

- 엔티티 객체는 JPA에서 프록시 객체가 필요함으로 기본 생성자 접근지시자가 최소 PROTECTED 이다.
- Request 객체는 HttpMessageConverter에서 리플랙션 기반으로 직렬화 작업을 진행하기 때문에 기본 생성자가 필요(가장 낮은 레벨 지정)

lombok.config 설정을 통해서 제한 하자

lombok

```
1  lombok.Setter.flagUsage = error
2  lombok.AllArgsConstructor.flagUsage = error
3  lombok.data.flagUsage= error
4  lombok.addLombokGeneratedAnnotation = true
```

Messages: Build

```
Information: java: Errors occurred while compiling module 'lombok'
Information: javac 1.8.0_65 was used to compile java sources
Information: 22/12/2018, 3:47 PM - Compilation completed with 2 errors and 2 warnings in 2 s 554 ms
/Users/cheese/yun/blog-sample/lombok/src/main/java/cheese/yun/lombok/Member.java
  Warning:(41, 26) java: @Builder will ignore the initializing expression entirely. If you want the initializing expression to serve as default, add
                   @Builder.Default. If it is not supposed to be settable during building, make the field final.
  Warning:(14, 1) java: Use of @ToString is flagged according to lombok configuration.
/Users/cheese/yun/blog-sample/lombok/src/main/java/cheese/yun/lombok/Coupon.java
  Error:(10, 1) java: Use of @Data is flagged according to lombok configuration.
  Error:(11, 1) java: Use of @Setter is flagged according to lombok configuration.
```

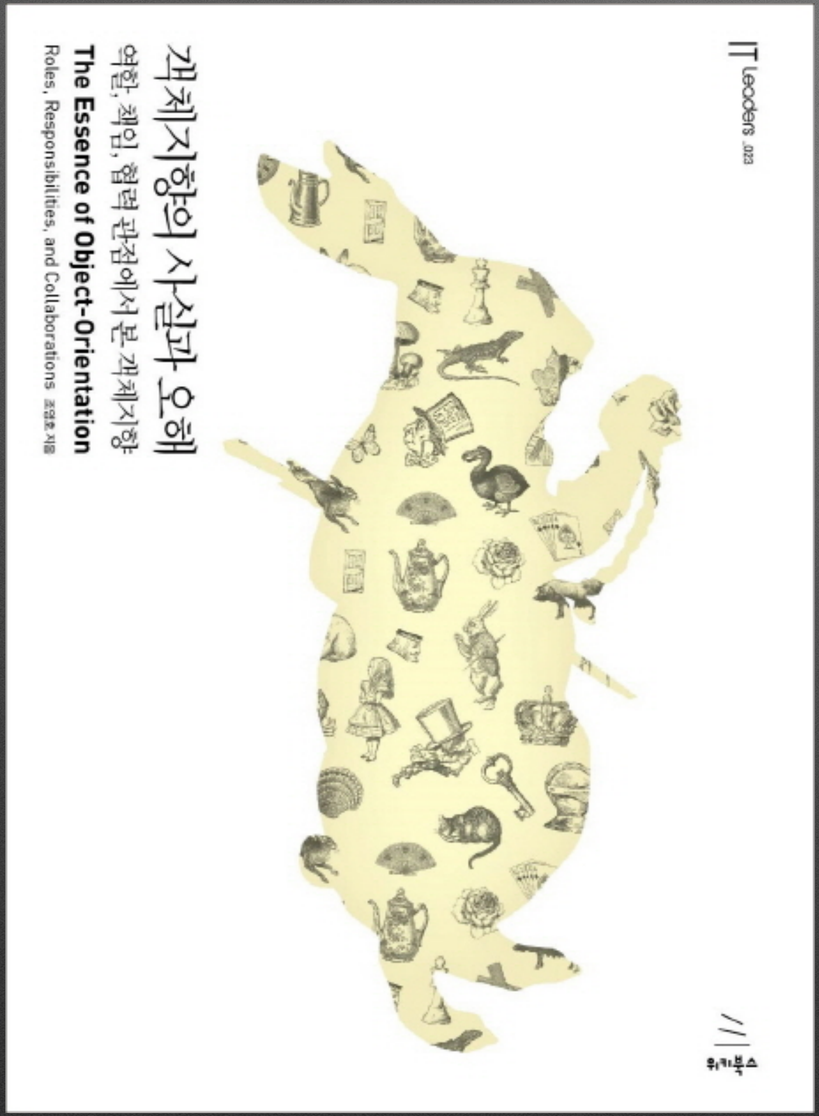
- lombok.config 설정 파일을 통해서 lombok 어노테이션을 제한 할 수 있습니다.
- 위에서 언급한 @Data 등 사용을 했을 경우 위험 부담이 있는 어노테이션들은 해당 설정에서 제한 할 수 있습니다.

본질은...

- 마법(자동)으로 해주는 것들은 지금은 그 비용을 지불하고 있지 않지만 언제가는 그 비용을 지불하게 되는 경우가 있음
- 객체 생성은 매우 중요한 포인트, 최대한 비지니스를 반영하고 안전하게 만들어야함
  - 생성자 최소한, 객체 생성이후 데이터 검증,필수 값만 받기 등등

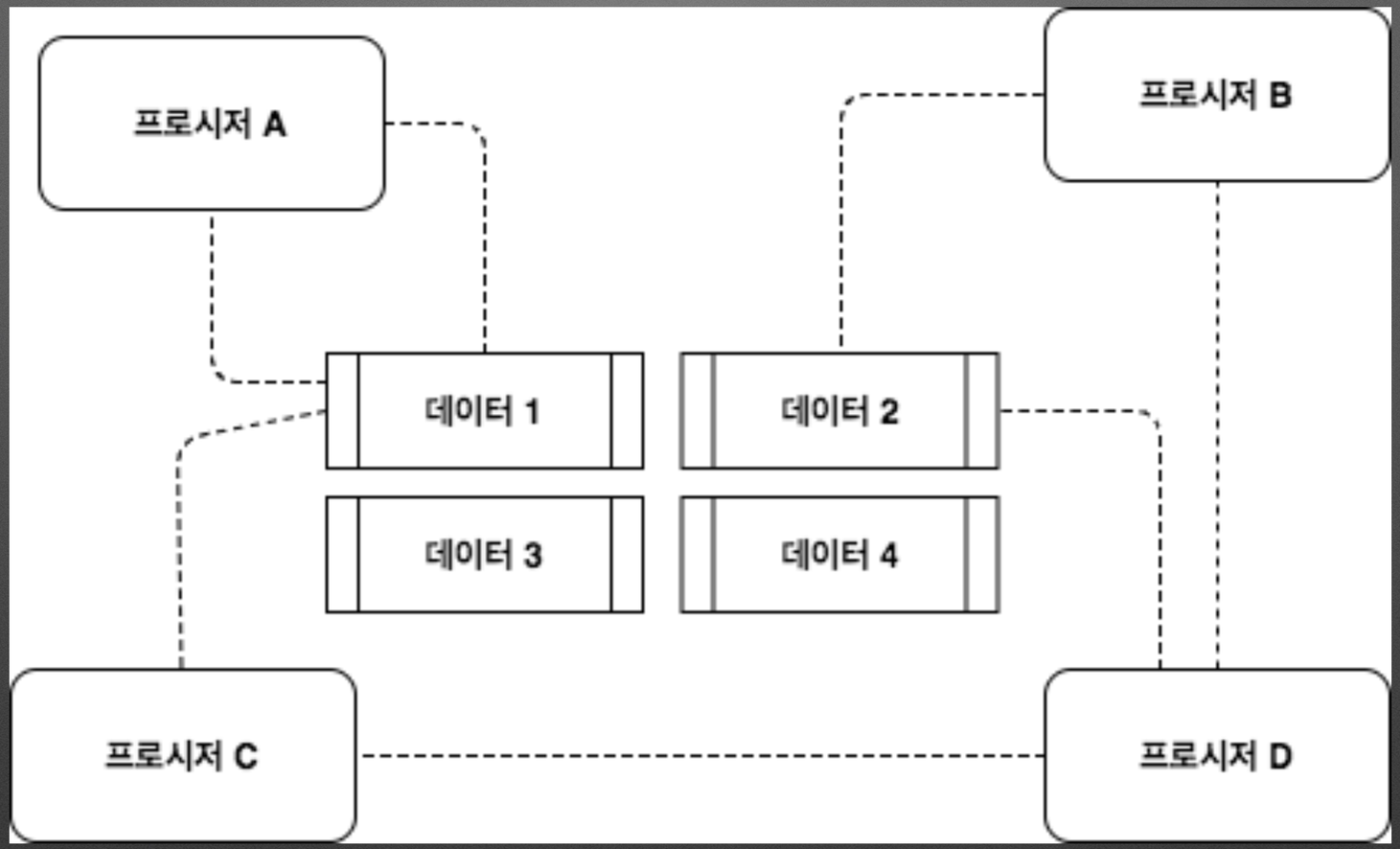
3. 객체지향

참고


IT Leaders 023
객체지향의 사실과 오해
역할, 책임, 협력 관점에서 본 객체지향
The Essence of Object-Orientation
Roles, Responsibilities, and Collaborations 조영호 지음
위키북스

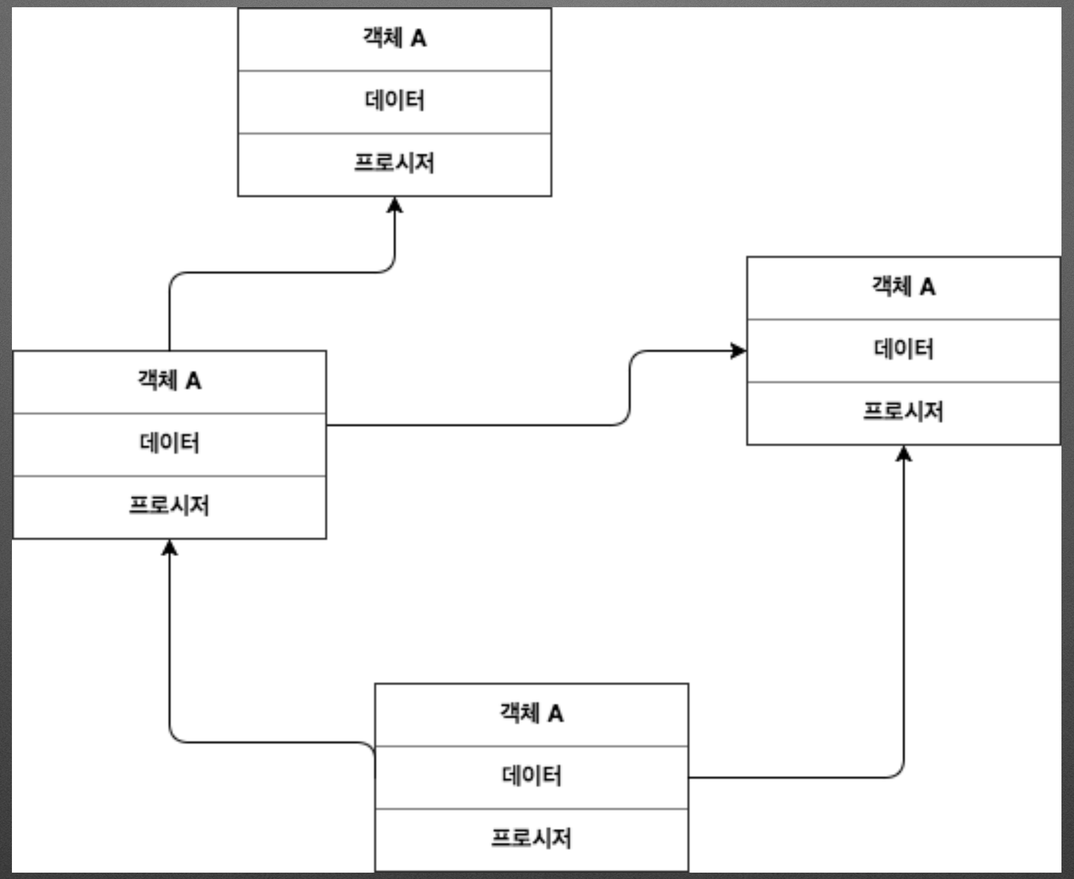
최범균 지음
• 객체 지향
(객체, 책임, 의존, 캡슐화)
• 다형성과 추상화
• 상속보단 조립
• SOLID 설계 원칙
• DI와 서비스 로케이터
• 주요 디자인 패턴
개발자가
반드시 정복해야 할
객체 지향과
디자인 패턴
인투북스

절차 지향



- 프로시저와 데이터간의 접근이 자유롭다
- 개발 초기단계에서는 편의성이 있지만 시간이 지날수록 프로시저가 어느 데이터에든 접근할 수있어 캡슐화가 저하되는 단점이 있다.

객체 지향



- 처음에는 어려울 수 있음, 데이터 프로시저를 알맞게 묶는 것이 어려움
- 하지만 시간이 흘러갈 수록 캡슐화 이점 때문에 큰 장점이 있음

* 캡슐화가 안좋은 코드는 자신의 상태를 외부에서 물어보게 됨

어떠한 객체도 섬이 아니다.

- 객체는 공동체안에 살고 있는 성실한 객체 시민은 자신에게 주어진 역할과 책임을 다하는 동시에 시스템의 더 큰목적을 이루기 위해 다른 객체와도 적극적으로 협력한다.
- 애플리케이션의 기능은 더 작은 책임으로 분할되고 책임은 적절한 역할을 수행할 수 있는 객체의 의해 수행된다. 시스템의 기능은 객체 간의 연쇄적인 요청과 응답의 흐름으로 구성된 협력으로 구현된다.

역할은 관련성 높은 책임의 집합이다

- 여러 객체가 **동일한 역할**을 수행할 수 있다.
- 역할은 대체 **가능성**을 의미한다.
- 각 객체는 책임을 수행하는 방법을 **자율적**으로 선택할 수 있다.
- 하나의 객체가 **동시에 여러 역할**을 수행할 수 있다.
- 역할은 유연하고 **재사용 가능한 협력관계**를 구축하는데 중요한 설계 요소이다

상태와 행동을 함께 지닌 자율적인 객체

- 객체를 **상태(state)**와 **행동(behavior)**을 함께 지닌 실체라고 정의한다.
- 객체가 협력에 참여하기 위해 어떤 행동을 해야한다면 그 행동을 하는데 필요한 상태도 함께 지니고 있어야한다는 것을 의미한다.
- 객체가 협력에 참여하는 과정 속에서 **스스로 판단**하고 스스로 결정하는 **자율적인 존재**로 남기 위해서는 필요한 행동과 상태를 함께 지니고 있어야한다.
- 행동이 상태를 결정한다.

상태와 행동을 함께 지닌 자율적인 객체

- 객체는 다른 객체가 '무엇(what)'을 수행하는지는 알수 있지만 '어떻게(how)'수행하는지에 대해서는 알 수없다.
- 객체는 상태와 행위를 하나의 단위로 묶는 자율적인 존재다. 과거 전통적인 개발 방법은 데이터와 프로세스를 엄격하게 구분한다.
- 이에 반해 객체지향에서는 데이터와 프로세스를 객체라는 하나의 틀 안에서 함께 묶어 놓음으로써 객체의 자율성을 보장한다. 이것이 전통적인 개발 방법과 객체지향을 구분짓는 가장 핵심적인 차이이다.

Sample Code

협력 속에 사는 객체

- 객체는 충분히 '협력적'이어야한다. 객체는 다른 객체의 요청에 충실히 귀 기울이고 다른 객체에게 적극적으로 도움을 요청할 정도로 열린 마음을 지녀야한다. 객체는 다른 객체의 명령에 복종하는 것이 아니라 요청에 응답할 뿐이다. 어떤 방식으로 응답할지는 객체 스스로 판단하고 결장한다. 심지어 요청에 응할지 여부도 객체 스스로 결정할 수 있다.
- 객체가 충분히 '자율적'이어야한다. 요청에 따른 행동은 객체 스스로가 판단해야한다.

객체지향의 본질

- 객체지향이란 시스템을 상호작용하는 **자율적인 객체들의 공동체**로 바라보고 객체를 이용해서 시스템을 분할하는 방법이다.
- 자율적인 객체란 상태,행위를 함께 지니며 스스로 자기 자신을 책임지는 객체를 의미한다.
- 객체는 시스템의 행위를 구현하기 위해 다른 객체와 **협력**한다. 각 객체는 협력 내에서 정해진 **역할**을 수행하며 역할은 관련된 책임의 집합이다.
- 객체는 다른 객체와 협력하기 위해서 메시지를 전송하고, 메세지를 수신한 객체는 메시지를 처리하는데 접합한 메시지를 **자율적**으로 선택한다.

메시지를 따라라 - What/Who 사이클

- 책임-주도 설계의 핵심은 어떤 **행위가 필요한지**를 먼저 결정한 후에 이 행위를 **수행할 객체**를 결정하는 것이다. 이 과정을 What/Who 사이클 이라고 한다.
- What/Who 사이클 이 의미하는 것은 객체 사이의 협력 관계를 설계하기 위해서 먼저 '**어떤 행위 What**'를 수행할 것인지를 결정한 후에 누가 '누가 Who'그 행위를수행할 것인가를 결정해야 한다는 것이다. 여기서 '어떤 행위'가 메시지다.
- 객체가 어떤 메시지를 수신하고 처리할 수 있냐가 객체의 책임을 결정한다.

메시지를 따라라 - 묻지 말고 시켜라

- 메시지를 먼저 결정하고 객체가 메시지를 따르게 하는 설계 방식은 객체가 외부에 제공하는 인터페이스가 독특한 스타일을 따르게 한다. 이 스타일을 묻지 말고 시켜라 **Tell, Don't**
- 송신자는 수신자가 어떤 객체 인지 모르기 때문에 객체에 관해 **꼬치꼬치 캐물을 수 없다**. 단지 송신자는 수신자가 어떤 객체인지는 모르지만 자신이 전송한 **메시지를 잘 처리할 것이라는 것을 믿고** 메시지를 전송할 수 밖에 없다.
- 이런 스타일의 협력 패턴은 묻지 말고 시켜라라는 이름으로 널리 알려져 있다. 이 스타일은 객체지향 애플리케이션이 **자율적인 객체들의 공동체** 라는 사실을 강조한다.

메시지를 믿어라

- 메시지를 전송하는 객체의 관점에서 자신이 전송하는 메시지를 수신할 수 있다면 협력 하는 객체의 종류가 무엇인지는 중요하지 않는다.
- 중요한 것은 메시지를 수신한 객체가 메시지의 의미를 이해하고 메시지를 전송한 객체가 의도한 대로 요청을 처리할 수 있는 지 여부다.
- 메시지를 이해할 수 있다면 다양한 타입의 객체로 협력 대상을 자유롭게 교체할 수 있기 때문에 설계가 좀 더 유연해진다.

Sample Code

책임

- 객체지향의 세계에서는 어떤 객체가 어떤 요청에 대해 대답해 줄 수 있거나, 적절한 행동을 할 의무가 있는 경우 해당 객체가 책임을 가진다고 말한다.
- 객체지향에서 가장 중요한 능력은 책임을 능숙하게 소프트웨어 객체에게 할당하는 것이라고 도 말한다.

책임의 분류

하는 것

- 객체를 생성하거나 계산하는 등의 **스스로하는 것**
- 다른 객체의 **행동을 시작 시키는 것**
- 다른 객체의 활동을 **제어하고 조절**하는 것

아는 것 (knowing)

* 개인적인 정보에 관해 아는것
* 관련된 객체에 관해 아는 것
* 자신이 유도하거나 계산할 수 있는 것에 관해 아는 것

역할

- 역할은 협력 내에서 다른 객체로 **대체**할 수 있음을 나타내는 일종의 표식이다.
- 협력안에서 역할은 "이 자리는 해당 역할을 수행할 수 있는 어떤 객체라도" **대신**할 수 있습니다. 라고 말하는 것과 같다.
- 역할의 가장 큰 가치는 하나의 협력 안에서 여러 종류의 객체가 참여할 수 있게 함으로써 협력을 **추상화**할 수 있다는 것이다. 역할을 이용하면 **협력을 추상화함으로써 단순화**할 수 있다.

역할 - 대체 가능성

- 역할은 협력 안에서 구체적인 객체로 대체될 수 있는 추상적인 협력자이다. 따라서 본질적으로 역할은 다른 객체에 의해 대체 가능함을 의미한다.
- 객체가 역할을 대체하기 위해서는 **행동이 호환돼야** 한다는 점에 주목하라
- 객체가 역할에 주어진 책임 이외에 **다른 책임을 수행**할 수도 있다는 사실에 주목하라
- 역할은 대체 가능성은 **행위 호환성**을 의미하고, **행위 호환성은 동일한 책임**의 수행을 의미한다.

다형성

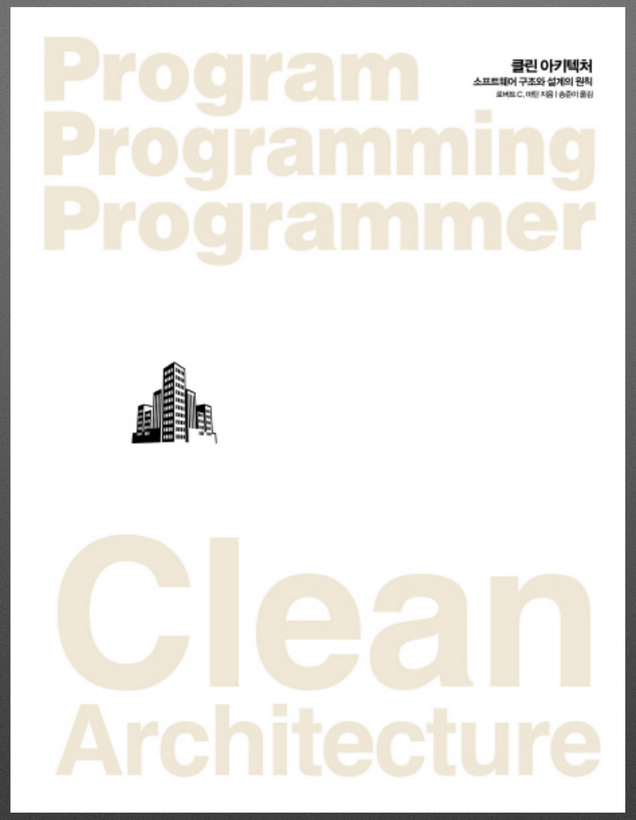
- 다형성이란 서로 다른 유형의 객체가 동일한 메시지에 대해서 서로 다르게 반응하는 것을 의미한다.
- 서로 다른 타입에 속하는 객체들이 동일한 메시지를 수신할 경우 서로 다른 메서드를 이용해 메시지를 처리 할 수 있다는 메커니즘을 가리킨다.
- 다형성은 역할, 책임, 협력과 같은 관련이 있다. 서로 다른 객체들이 다형성을 만족시킨다는 것은 객체들이 동일한 책임을 공유한다는 것을 의미한다.
- 기본적으로 다형성은 동일한 역할을 수행할 수 있는 객체들 사이의 대체 가능성을 의미한다. 다형성은 객체들의 대체 가능성을 이용해 설계를 유연하고 재사용 가능하게 만든다.

Sample Code

Sample Code

참고 서적

Design Patterns
Elements of Reusable Object-Oriented Software
GoF의 디자인 패턴
개정판
재사용성을 지닌 객체지향 소프트웨어의 핵심 요소
에릭 갬마 · 리처드 헬름 · 랄프 존슨 · 존 블리시디스 지음
김정아 옮김
Protec Media
Jolt Awards Productivity Award
대한민국학술원 우수학술도서
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Program
Programming
Programmer
클린 아키텍처
소프트웨어 구조와 설계의 원칙
로버트 C. 마틴 지음 | 송준이 옮김
Clean
Architecture

최범균 지음
• 객체 지향
(객체, 책임, 의존, 캡슐화)
• 다형성과 추상화
• 상속보단 조립
• SOLID 설계 원칙
• DI와 서비스 로케이터
• 주요 디자인 패턴
개발자가
반드시 정복해야 할
객체 지향과
디자인 패턴
인투북스

4. SOLID 패턴

SOLID 디자인패턴

- SRP: 단일 책임 원칙
- OCP: 개방 폐쇄 원칙
- LSP: 리스코프 치환 원칙
- ISP: 인터페이스 분리 원칙
- DIP: 의존성 역전 원칙

왜 SOLID를 공부해야할까?

* 수정 구술은 없다... 우리가 할 수 있는건...

개방 폐쇄의 원칙: Open Close Principle

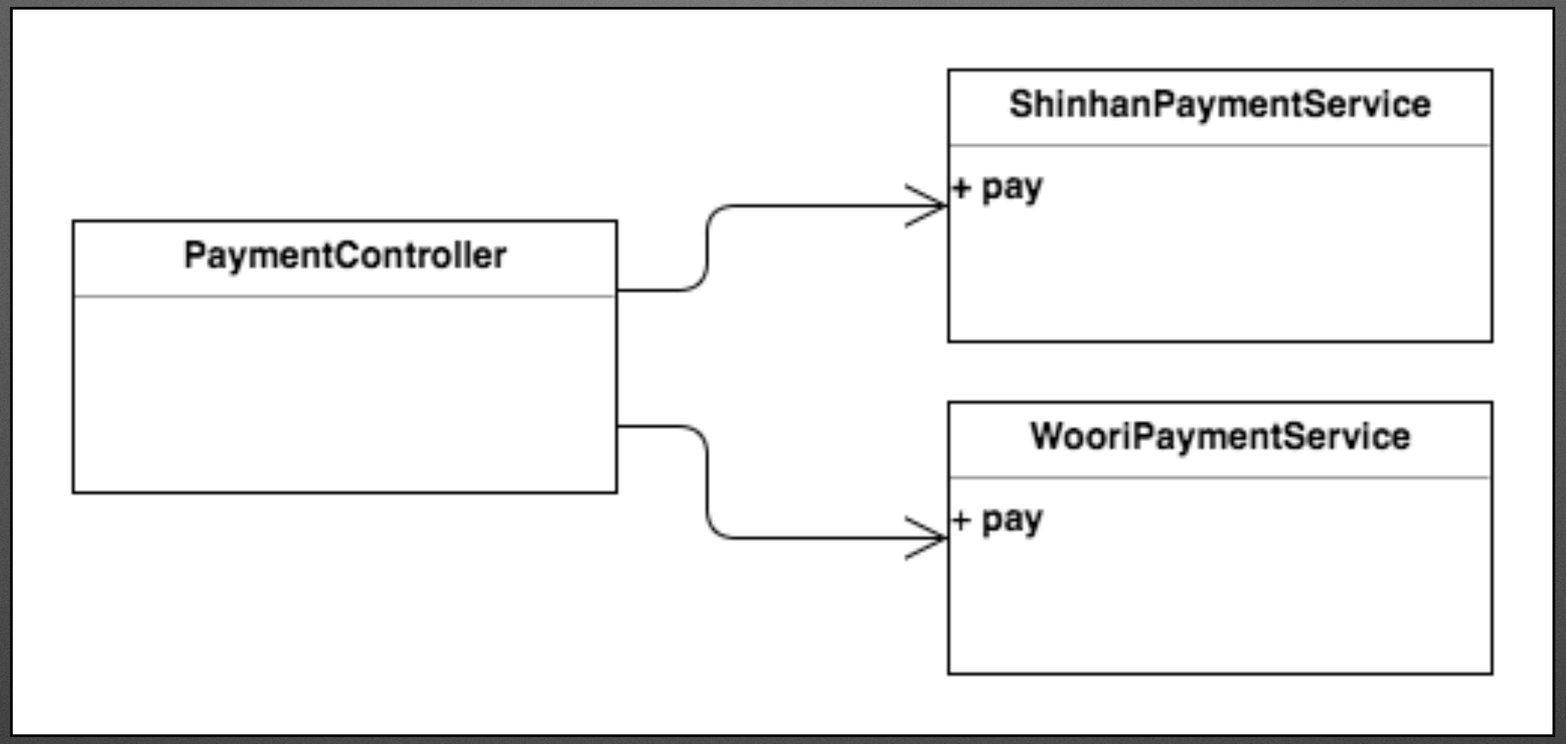
- 확장에는 열려 있고, 변경에는 닫혀 있다.
- 기능을 추가하거나 변경하면서 그것을 사용하는 코드는 수정하지 않는다.

용어 풀이

> **확장 이란 ?**
>
> 새로운 타입을 추가함으로써 새로운 기능을 추가할 수 있다. 즉 확장이란 새로운 타입을 추가함으로써 새로운 기능(요구사항)을 구현한다. **확장에는 열려 있다는 것은 새로운 타입(클래스)을 추가함으로써 기능(요구사항)을 확장하는 것이다..**

> **변경 이란 ?**
>
> 확장이 발생했을 때 상위 레벨이 영향을 미치지 않아야 한다. **확장(새로운 클래스)이 발생했을 때 해당 코드를 호출하는 쪽에서 변경이 발생하지 않다면 변경에 닫혀 있다는 것이다.**

OCP를 준수하기 위한 실패 과정들



- 기존 PamentController에서 ShinhanCardPaymentService를 의존해서 신한 카드 결제를 진행 했습니다. (OCP 위반)
- 이제 우리 은행 카드결제가 추가되었으니 아래 컨트롤러처럼 기존 신한 카드를 건드리지(변경에 닫혀 있다) 않고 우리 카드를 추가하는 가장 쉬운 방법은 아래처럼 컨트롤러를 각각 구성하는 방법입니다.

카드사 마다 API

```
@RequestMapping(value = "/ocp/anti/payment/shinhan", method = RequestMethod.POST)
public void pay(@RequestBody ShinhanCardDto.PaymentRequest req){
    shinhanCardPaymentService.pay(req);
}

@RequestMapping(value = "/ocp/anti/payment/woori", method = RequestMethod.POST)
public void pay(@RequestBody WooriCardDto.PaymentRequest req){
    wooriCardPaymentService.pay(req);
}
```

- 해당 구조는 매우 좋지 않습니다. **카드사가 추가될 때 마다 API를 추가**적으로 만들어야 합니다.
- 이는 확장에 좋지 않은 코드 뿐만 아니라 올바른 카드사의 API를 찾기위한 코드가 필요하게 됩니다.

공통 Request로 API 통일

```
public static class PaymentRequest {
    private String cardNumber;
    private String csv;
    private CardType type;
}
// 컨트롤러
@RequestMapping(value = "/ocp/anti/payment", method = RequestMethod.POST)
public void pay(@RequestBody CardPaymentDto.PaymentRequest req){
    if(req.getType() == CardType.SHINHAN){
        shinhanCardPaymentService.pay(req);
    }else if(req.getType() == CardType.WOORI){
        wooriCardPaymentService.pay(req);
    }
    // 그 해당 카드 이외의 타입이 들어오면 예외처리는 어떻게??...
}
```

- 넘겨받는 RequestBody 값의 통일로 하나의 API에서 신한 카드, 우리 카드 결제를 진행할 수 있게 되었습니다.
- 예를 들어 신한 카드는 카드 번호를 shinhanCardNumber, 우리 카드는 number로 사용했을 경우 매번 미세하게 다른 값들을 공통 Request로 통일하게 됩니다.

공통 Request로 API 통일

```
public static class PaymentRequest {
    private String cardNumber;
    private String csv;
    private CardType type;
}
// 컨트롤러
@RequestMapping(value = "/ocp/anti/payment", method = RequestMethod.POST)
public void pay(@RequestBody CardPaymentDto.PaymentRequest req){
    if(req.getType() == CardType.SHINHAN){
        shinhanCardPaymentService.pay(req);
    }else if(req.getType() == CardType.WOORI){
        wooriCardPaymentService.pay(req);
    }
    // 그 해당 카드 이외의 타입이 들어오면 예외처리는 어떻게??...
}
```

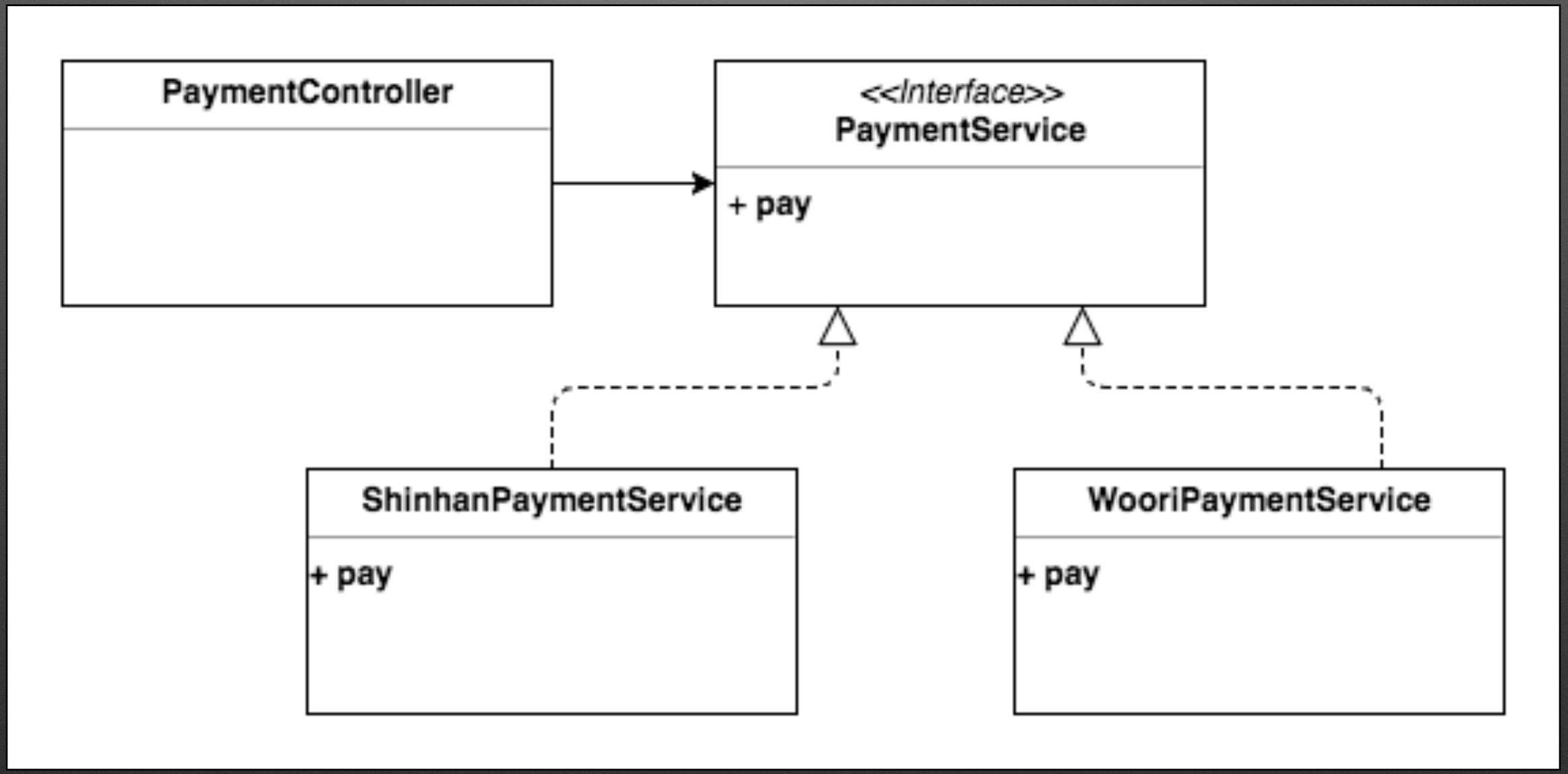
- 기존 코드는 카드 파트너사의 API Request에 의존 관계를 분리할 수 있습니다.
- DIP의 '상위 레벨의 정책은 하위 레벨의 상세 합에 의존하면 안 된다.'를 준수하게 됩니다. (해당 개념은 DIP에 자세하게 포스팅하겠습니다.)
- Request 값을 통일 함으로써 추상화의 기초가 됩니다.

공통 Request로 API 통일 -> OCP 준수하지 않음

> 확장(새로운 클래스)이 발생했을 때 해당 코드를 호출하는 쪽에서 **변경이 발생하지 않다면** 변경에 닫혀 있다는 것입니다.

- 우리 카드가 추가되었을 경우 **해당 컨트롤러의 코드에 if**이 추가되었습니다. 이는 위에서 정리한 변경이 발생하는 경우입니다. 또 한 카드가 지속해서 추가될 때 마다 해당 카드결제를 위한 if 문을 지속해서 추가해야 합니다. 이런 **반복적인 if는 해당 리팩토링 대상**일 확률이 매우 높습니다.
- 그 밖에도 추가될 카드의 결제를 담당하는 XXXPaymentService 클래스들이 지속해서 의존성 이루어집니다. 그 결과 **PaymentController는 컨트롤러 계층임에도 너무 많은 책임을 갖게** 되며, 확장에 어렵고, 변경에 취약한 구조가 됩니다.

OCP 준수


PaymentController
<<Interface>>
PaymentService
+ pay
ShinhanPaymentService
+ pay
WooriPaymentService
+ pay

OCP 준수

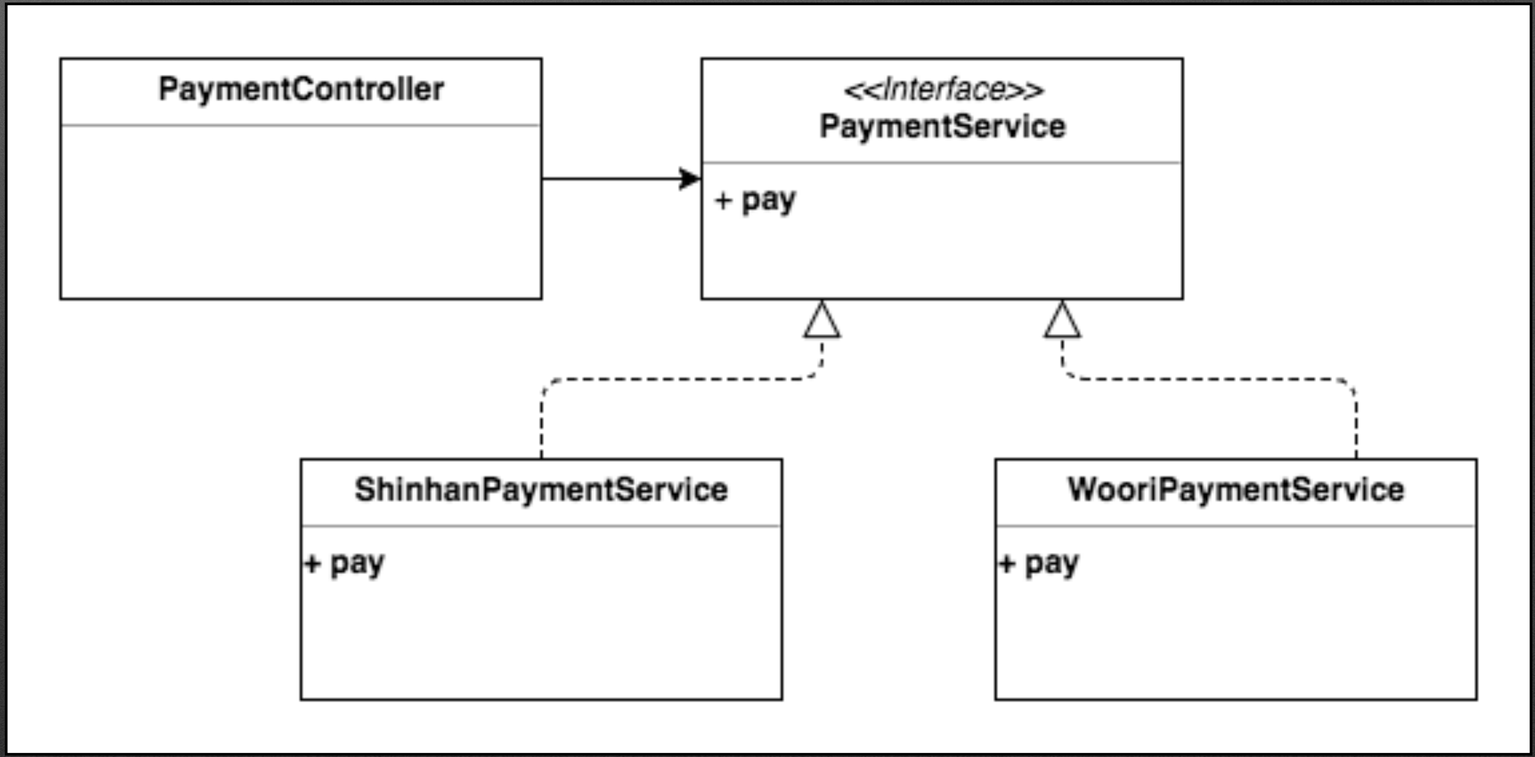
```
public class PaymentController {
    @RequestMapping(value = "/payment", method = RequestMethod.POST)
    public void pay(@RequestBody CardPaymentDto.PaymentRequest req) {
        final CardPaymentService cardPaymentService = cardPaymentFactory.getType(req.getType());
        cardPaymentService.pay(req);
    }
}
public class ShinhanCardPaymentService implements CardPaymentService {
    @Override
    public void pay(CardPaymentDto.PaymentRequest req) {
        final ShinhanCardDto.PaymentRequest paymentRequest = buildPayment(req);
        shinhanCardApi.pay(paymentRequest);
    }
}
public class WooriCardPaymentService implements CardPaymentService {
    @Override
    public void pay(CardPaymentDto.PaymentRequest req) {
        final WooriCardDto.PaymentRequest paymentRequest = buildPayment(req);
        wooriCardApi.pay(paymentRequest);
    }
}
```

OCP 준수

- 의존관계를 인터페이스를 통해서 역전시키는 것입니다. 새로운 카드 결제가 추가된다고 하더라도 PaymentController 수정 없이 CardPayment를 확장하고 있습니다.
- 다시 한번 정리하면 새로운 카드 결제가 추가(새로운 결제 타입 추가) 되더라도(확장에 열려있음) 카드 결제를 호출하는 코드 쪽에서 변경이(변경에는 닫혀 있음) 변경이 발생하지 않습니다. 물론 cardPaymentFactory 클래스의 의존성 주입 코드가 한번은 추가됩니다.
- 즉 우리 카드 결제라는 새로운 기능이 추가되었을 때 이것을 새로운 타입으로 추가해서 런타임시 교체하여 확장에는 열려 있고, 또 그 코드를 호출하는 쪽은 인터페이스를 바라보고 있으므로 변경에는 닫혀 있는 구조가 됩니다.

> **확장 이란 ?**
>
> 새로운 타입을 추가함으로써 새로운 기능을 추가할 수 있다. 즉 확장이란 새로운 타입을 추가함으로써 새로운 기능(요구사항)을 구현한다. **확장에는 열려 있다는 것은 새로운 타입(클래스)을 추가함으로써 기능(요구사항)을 확장하는 것이다..**

> **변경 이란 ?**
>
> 확장이 발생했을 때 상위 레벨이 영향을 미치지 않아야 한다. **확장(새로운 클래스)이 발생했을 때 해당 코드를 호출하는 쪽에서 변경이 발생하지 않다면 변경에 닫혀 있다는 것이다.**



의존성역전의 원칙: Dependency Inversion Principle

- 상위 수준 정책은 하위 수준 세부 정보에 의존해서는 안됩니다.
- 상위 모듈은 하위 모듈의 구현에 의존해서는 안 된다.
- 하위의 모듈이 상위 모듈에 정의한 추상 타입에 의존 해야한다.

요구사항

- 카드 결제 시스템을 만들어야 한다.
- **현재 지원하는** 카드는 신한 카드 **하나뿐이다.**
- 이제 우리 **카드 결제가 추가**되어 구현해야 한다.
- 앞으로도 카드는 **지속해서 추가**될 예정이다.

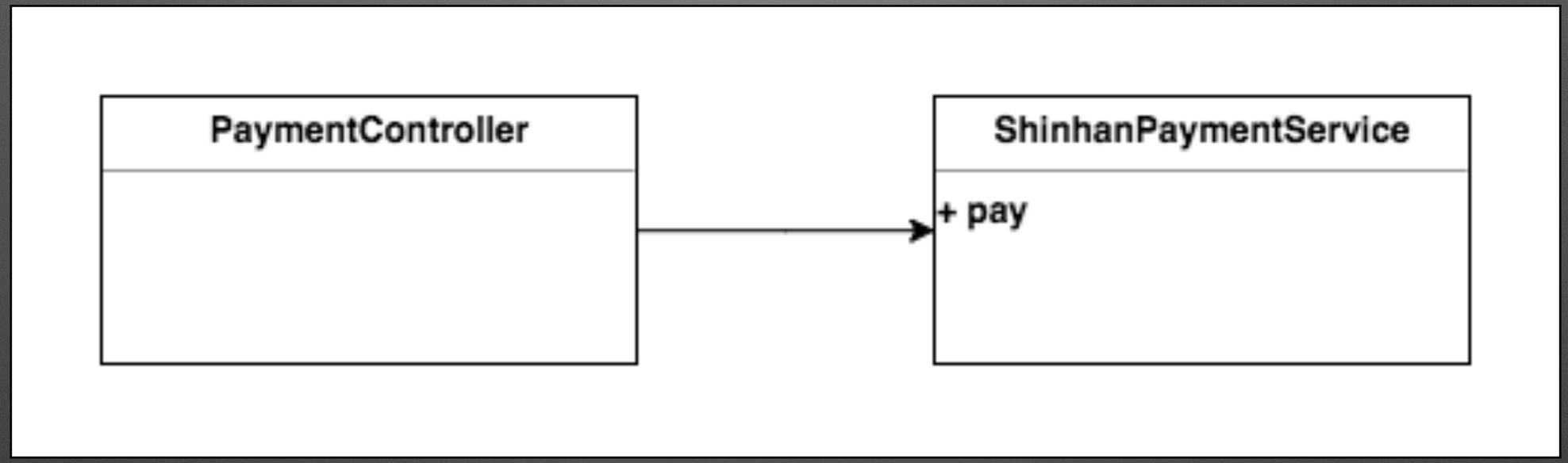
용어 풀이

> ***상위 모듈 이란?***
>
> 상위 모듈이란 상위 정책을 의미합니다. 위의 요구사항에서는 카드 결제라는 것이 상위 정책을 뜻합니다.

> ***하위 모듈 이란?***
>
> 하위 모듈이란 상위 정책에 따른 하위 정책을 말합니다. 위의 요구사항에서 카드 결제라는 상위 정책이 있으면 신한 카드 결제라는 하위 (세부) 정책이 있습니다.

> ***추상 타입 이란?***
>
> 추상 타입은 인터페이스, 추상화 클래스를 의미합니다. 상위 정책이 하위 정책에 의존하지 않고 추상 타입에 의존하라는 것은 **카드 결제라는 상위 정책이 신한 카드 결제라는 하위 정책을 의존하지 말고 추상 클래스, 인터페이스를 의존하라는 것입니다.**

DIP 준수하지 않은 코드



- 카드 결제(상위 수준 정책)가 신한 카드 결제(하위 수준)에 의존하고 있습니다

지나친 의존 관계

```java
class PaymentController {
    @RequestMapping(value = "/dip/anti/payment", method = RequestMethod.POST)
    public void pay(@RequestBody ShinhanCardDto.PaymentRequest req){
        shinhanCardPaymentService.pay(req);
    }
}
class ShinhanCardPaymentService {
    public void pay(ShinhanCardDto.PaymentRequest req) {
        shinhanCardApi.pay(req);
    }
}
```

```json
// RequestBody JSON 포멧
{
  "shinhanCardNumber":"4845-9005-9423-4452",  //만약 shinhanCardNumber -> cardNumber 으로 변경된다면 ?
  "cvc":"233"
}
```

- 카드 결제 기능(상위 정책)이 신한 카드 결제(하위 정책)에 지나치게 의존적입니다.
- 결제의 JSON의 키값만 변경 시 컨트롤러, 프론트엔드 까지 변경까지 영향을 미치게 됩니다.
- 의존 관계는 많은 변경 포인트를 유발합니다.

확장에 유연하지 못함

```
@RequestMapping(value = "/anti/payment", method = RequestMethod.POST)
public void pay(@RequestBody CardPaymentDto.PaymentRequest req){
    if(req.getType() == CardType.SHINHAN){
        shinhanCardPaymentService.pay(req);
    }else if(req.getType() == CardType.WOORI){
        wooriCardPaymentService.pay(req);
    }
}
```

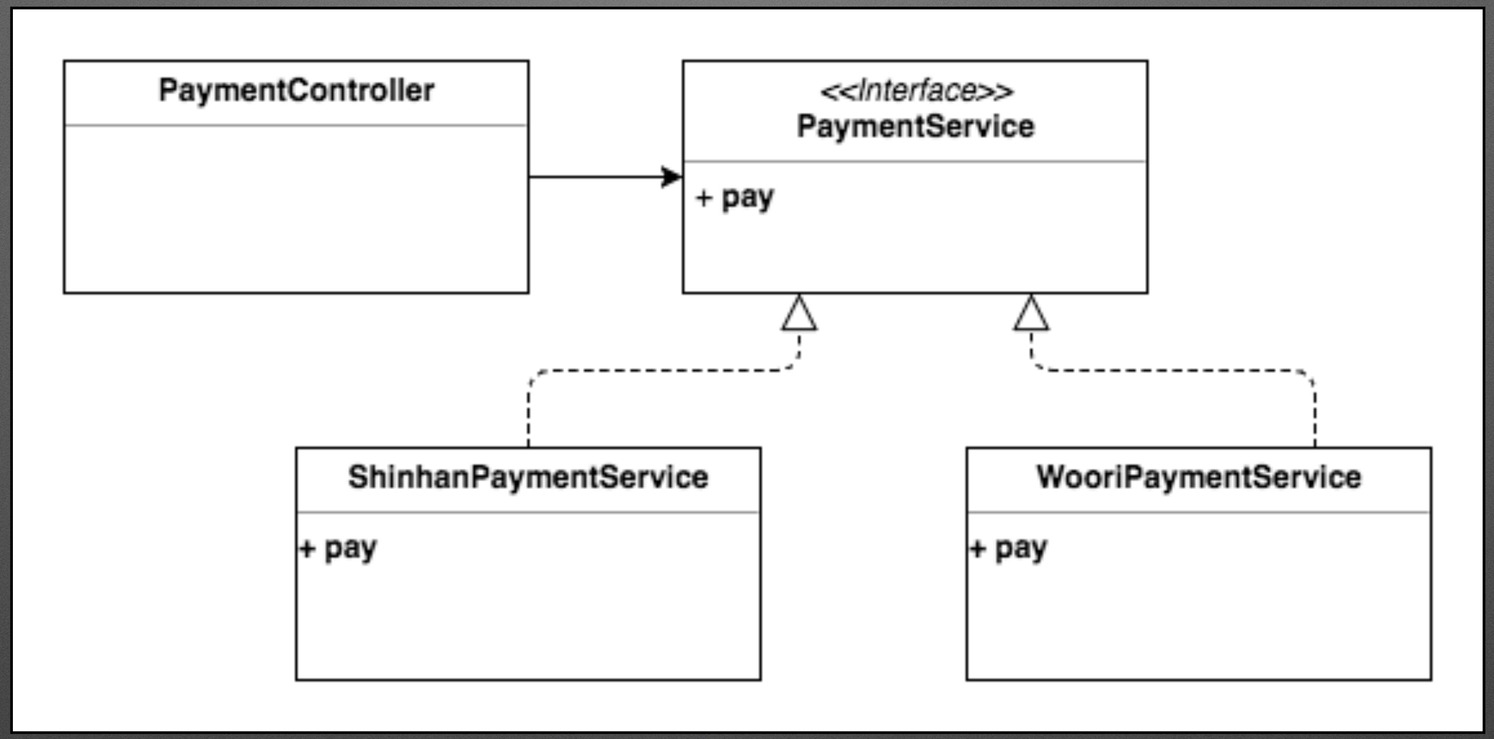
- 우리 카드, 신한 카드 결제 요청을 받을 PaymentRequest Dto 클래스를 생성하였고 CardType으로 해당 맞는 카드 타입에 맞는 서비스를 호출하는 구조입니다.
- 가장 쉽게 생각할 수 있는 구조이지만 결코 좋은 구조가 아닙니다. 카드가 지속해서 추가될 때마다 해당 카드결제를 위한 if 문을 지속해서 추가해야 합니다. 이런 반복적인 if는 해당 리팩토링 대상일 확률이 매우 높습니다.

확장에 유연하지 못함

```
@RequestMapping(value = "/anti/payment", method = RequestMethod.POST)
public void pay(@RequestBody CardPaymentDto.PaymentRequest req){
    if(req.getType() == CardType.SHINHAN){
        shinhanCardPaymentService.pay(req);
    }else if(req.getType() == CardType.WOORI){
        wooriCardPaymentService.pay(req);
    }
}
```

- 그 밖에도 추가될 카드의 결제를 담당하는 XXXPaymentService 클래스들이 지속해서 의존성 이루어집니다. 그 결과 PaymentController는 컨트롤러 계층임에도 너무 많은 책임을 갖게 되며, 확장에 어렵고, 변경에 취약한 구조가 됩니다.
- 이 설명은 개방 폐쇄의 원칙에도 위배 됨. DIP와 OCP는 연관이 크며 SOLID도 각기 다른 메커니즘이 아니라 서로 유기적으로 연결되어 있습니다.

DIP 준수



- 상위 모듈은 하위 모듈의 구현에 의존해서는 안 된다. 하위의 모듈이 상위 모듈에 정의한 추상 타입에 의존 해야한다.
- 카드 결제(상위 모듈은)는 신한 카드 결제(하위 모듈)에 의존해서는 안된다. 신한 카드 결제(하위 모듈)은 카드 결제(상위 모듈)에 정의한 카드 결제 인터페이스(추상 타입)에 의존 해야한다.

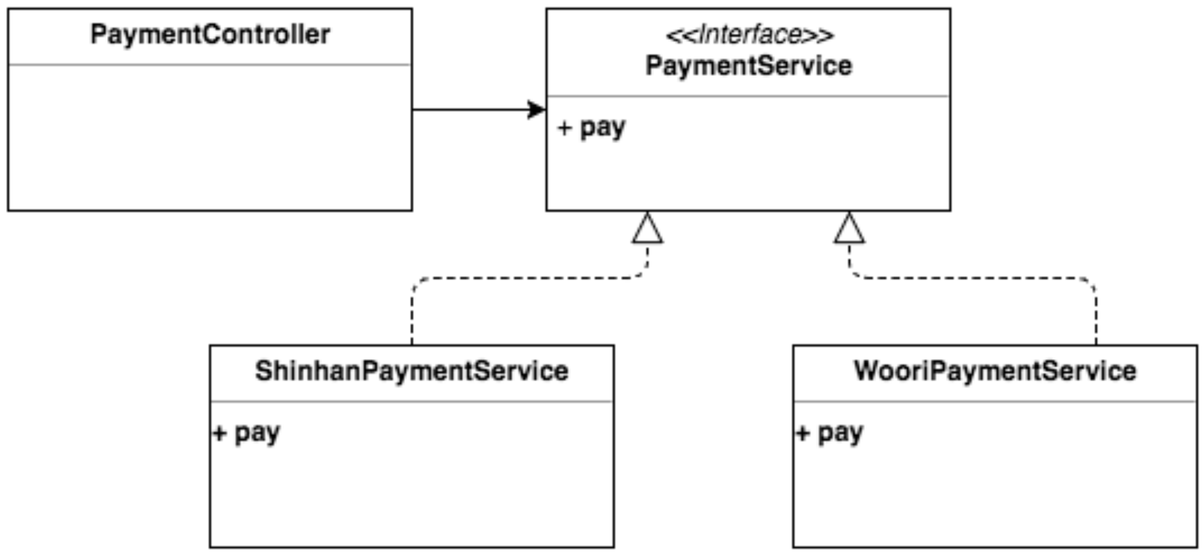
```java
class PaymentController {
    @RequestMapping(value = "/payment", method = RequestMethod.POST)
    public void pay(@RequestBody CardPaymentDto.PaymentRequest req) {
        final CardPaymentService cardPaymentService = cardPaymentFactory.getType(req.getType());
        cardPaymentService.pay(req);
    }
}

public interface CardPaymentService {
    void pay(CardPaymentDto.PaymentRequest req);
}

public class ShinhanCardPaymentService implements CardPaymentService {
    @Override
    public void pay(CardPaymentDto.PaymentRequest req) {
        shinhanCardApi.pay(req);
    }
}
```

- 의존관계를 인터페이스를 통해서 의존성을 역전 시켰습니다
- 컴파일 단계에서는 PaymentController는 PaymentService 인터페이스를 바라보지만 런타임에서는 cardPaymentFactory 통해서 ShinhanCardPaymentService를 바라보게 됩니다.
- 하위 정책의 신한 카드, 우리 카드가 변경되더라도 PaymenrService 인터페이스를 의존하고 있으므로 확장에 열려 있고 변경에는 닫혀 있는 OCP를 준수하게 됩니다.

> ***상위 모듈 이란 ?***
>
> 상위 모듈이란 상위 정책을 의미합니다. 위의 요구사항에서는 카드 결제라는 것이 상위 정책을 뜻합니다.

> ***하위 모듈 이란 ?***
>
> 하위 모듈이란 상위 정책에 따른 하위 정책을 말합니다. 위의 요구사항에서 카드 결제라는 상위 정책이 있으면 신한 카드 결제라는 하위 (세부) 정책이 있습니다.

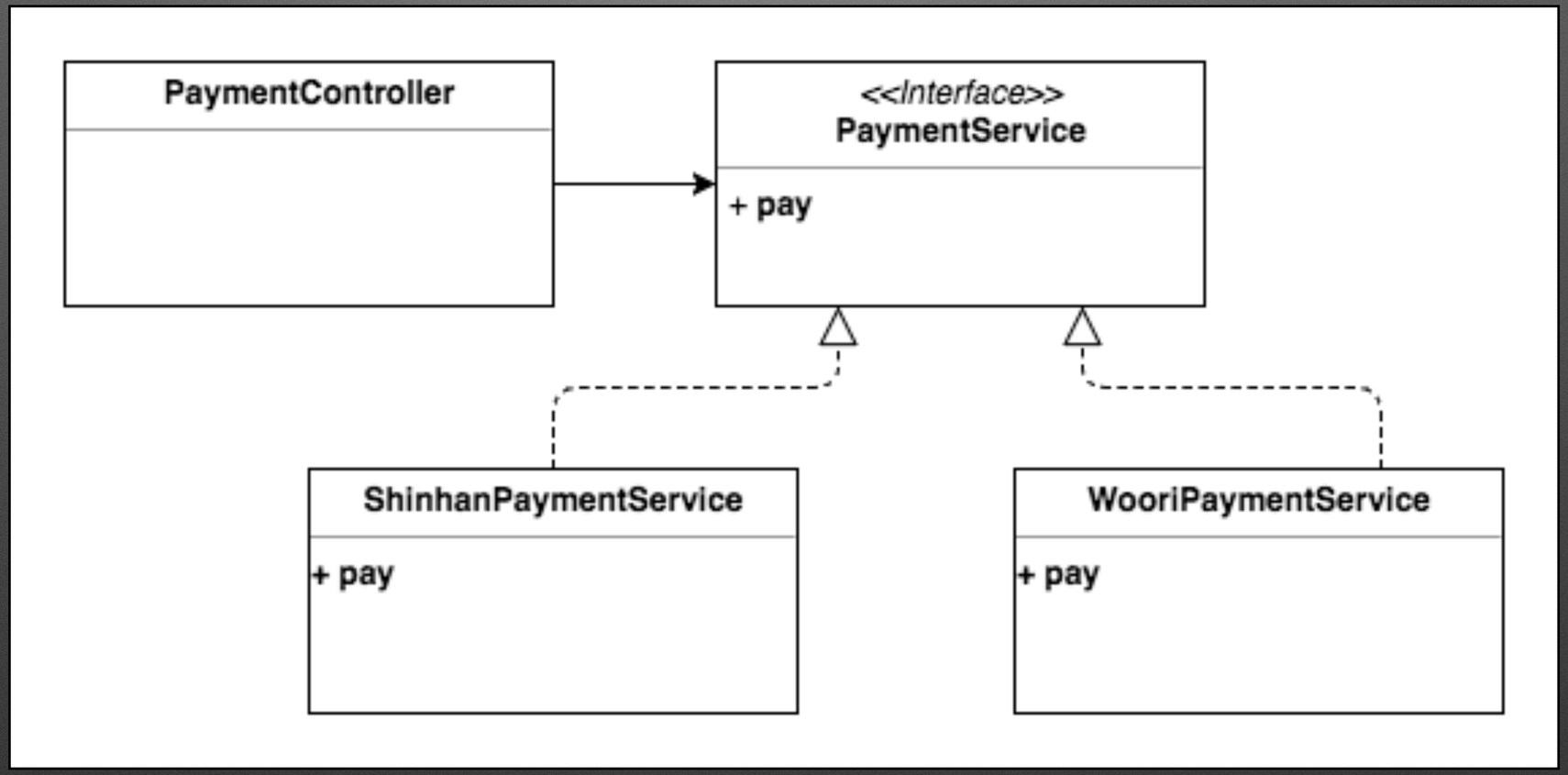
> ***추상 타입 이란 ?***
>
> 추상 타입은 인터페이스, 추상화 클래스를 의미합니다. 상위 정책이 하위 정책에 의존하지 않고 추상 타입에 의존하라는 것은 **카드 결제라는 상위 정책이 신한 카드 결제라는 하위 정책을 의존하지 말고 추상 클래스, 인터페이스를 의존하라는 것입니다.**



단일 책임의 원칙: Single Responsibility Principle

- 솔직히 이 친구는 빡쎕니다... 책임이란 단어가 ....
- 클래스는 **단 한 개의 책임**을 가져야 한다.
- 클래스의 **변경하는 이유는 단 한 개**여야 한다.
- 누가 해당 메소드의 **변경을 유발하는 사용자(Actor)** 인가?

기존 국내 카드 결제의 SRP



기존 국내 카드 결제의 SRP

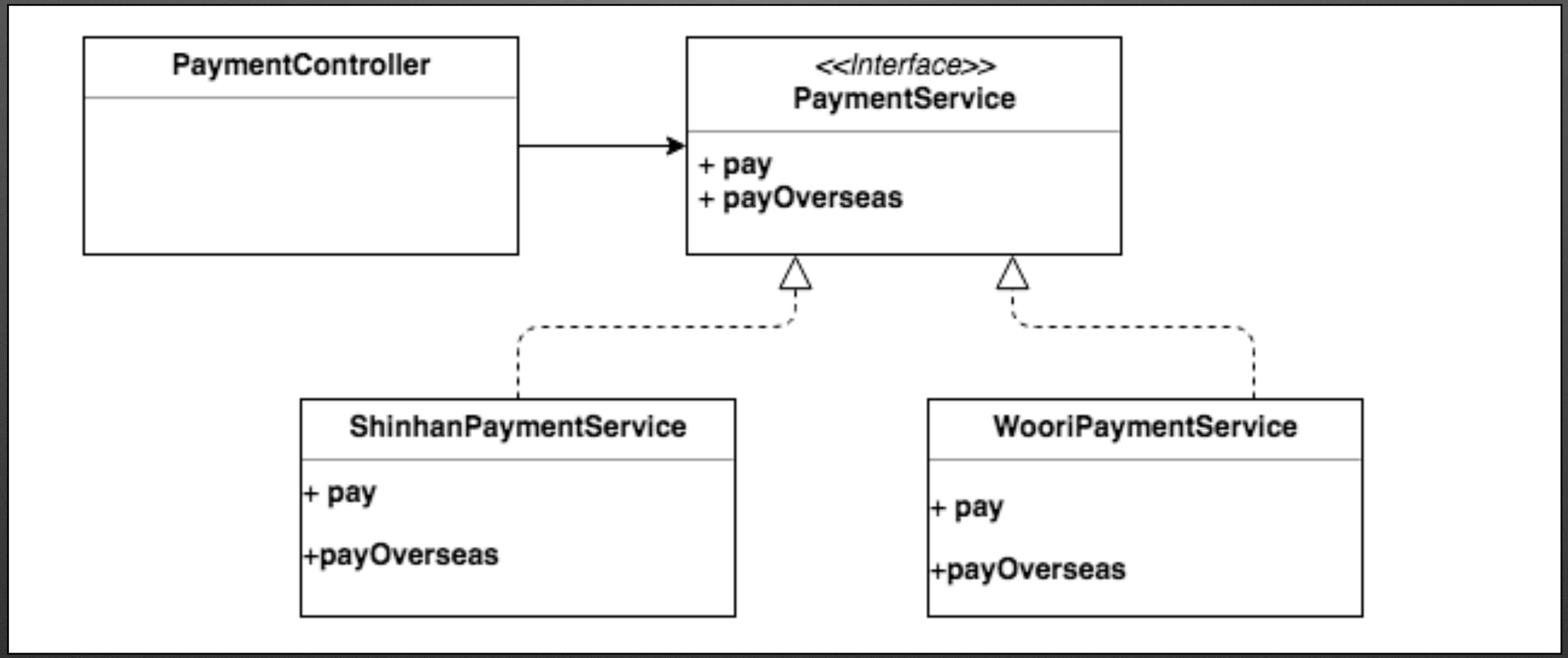
```java
public interface CardPaymentService {
    void pay(CardPaymentDto.PaymentRequest req);
}

public class ShinhanCardPaymentService implements CardPaymentService {
    @Override
    public void pay(CardPaymentDto.PaymentRequest req) {
        // .. 신한 카드 국내 결제 로직..
        shinhanCardApi.pay(paymentRequest);
    }
}

public class WooriCardPaymentService implements CardPaymentService {
    @Override
    public void pay(CardPaymentDto.PaymentRequest req) {
        // .. 우리 카드 국내 결제 로직..
        wooriCardApi.pay(paymentRequest);
    }
}
```

- 클래스의 책임 : 해당 **카드사의 결제**를 진행
- 변경의 근원 : **카드 결제를 하는 Actor**
- Actor : **카드결제를 행하는 행위자**

기존 국내 카드 결제의 SRP

- 클래스의 책임을 수행시키는 **Actor의 변경 시에만 클래스의 변경**이 가해져야 한다
- 만약 Actor가 결제 완료 시간 등 결제 정보를 받기를 원하게 된다면 pay 메서드의 리턴 타입이 변경이 발생합니다. 즉 카드 결제의 **변경은 Actor의 요구**에서부터 발생하게 됩니다.
- 여기서 **Actor를 단순히 사용자로 바라보면 안 되고 Actor는 그 행위(국내 결제)를 하는 행위자**로 봐야 한다고 생각합니다.
- **단일 책임이라는 것은 단일 Actor**를 뜻한다고 생각합니다

추가될 해외 카드 결제의 SRP(미준수)



추가될 해외 카드 결제의 SRP(미준수)

```
public interface CardPaymentService {
    void pay(CardPaymentDto.PaymentRequest req);
    void payOverseas(CardPaymentDto.PaymentRequest req);
}

public class ShinhanCardPaymentService implements CardPaymentService {
    @Override
    public void payOverseas(CardPaymentDto.PaymentRequest req) {
        // .. 신한 카드 해외 결제 로직..
        shinhanCardApi.pay(paymentRequest);
    }
}

public class WooriCardPaymentService implements CardPaymentService {
    @Override
    public void payOverseas(CardPaymentDto.PaymentRequest req) {
        // 우리 카드 결제는 해외 결제 기능이 없음...
    }
}
```

- 신한 카드는 해외 결제를 할 수 있지만 우리 카드는 해외 결제 기능을 제공하고 있지 않습니다. 각 구현 클래스들은 CardPaymentService 인터페이스를 구현하고 있으므로 payOverseas 기능이 추가되면 우리 카드 결제는 반드시 해당 메서드를 구현 해야 합니다.
- 해외 결제만 되고 국내 결제가 안 되는 카드 파트너가 추가되면 어떻게 될까요? 그렇게 되면 위와 반대로 payOverseas 구현 메소드는 구현하고 pay는 구현하지 못하게 됩니다.

책임이란 변화에 대한 것

- 국내 결제에서 해외 결제라는 **책임이 하나 더 생긴 것입니다**.
- 그렇게 두 **개의 책임이 생겼고 그 결과 두 개의 Actor**가 생긴 것이라고 생각합니다.
- (위에서 언급한 단일 책임 = 단일 Actor) 이로써 **클래스의 책임을 나누는 작업이 필요해집니다**.

우리카드가 해외 결제를 제공하고, 추가 파트너들도 해외 결제를 제공한다면 ?

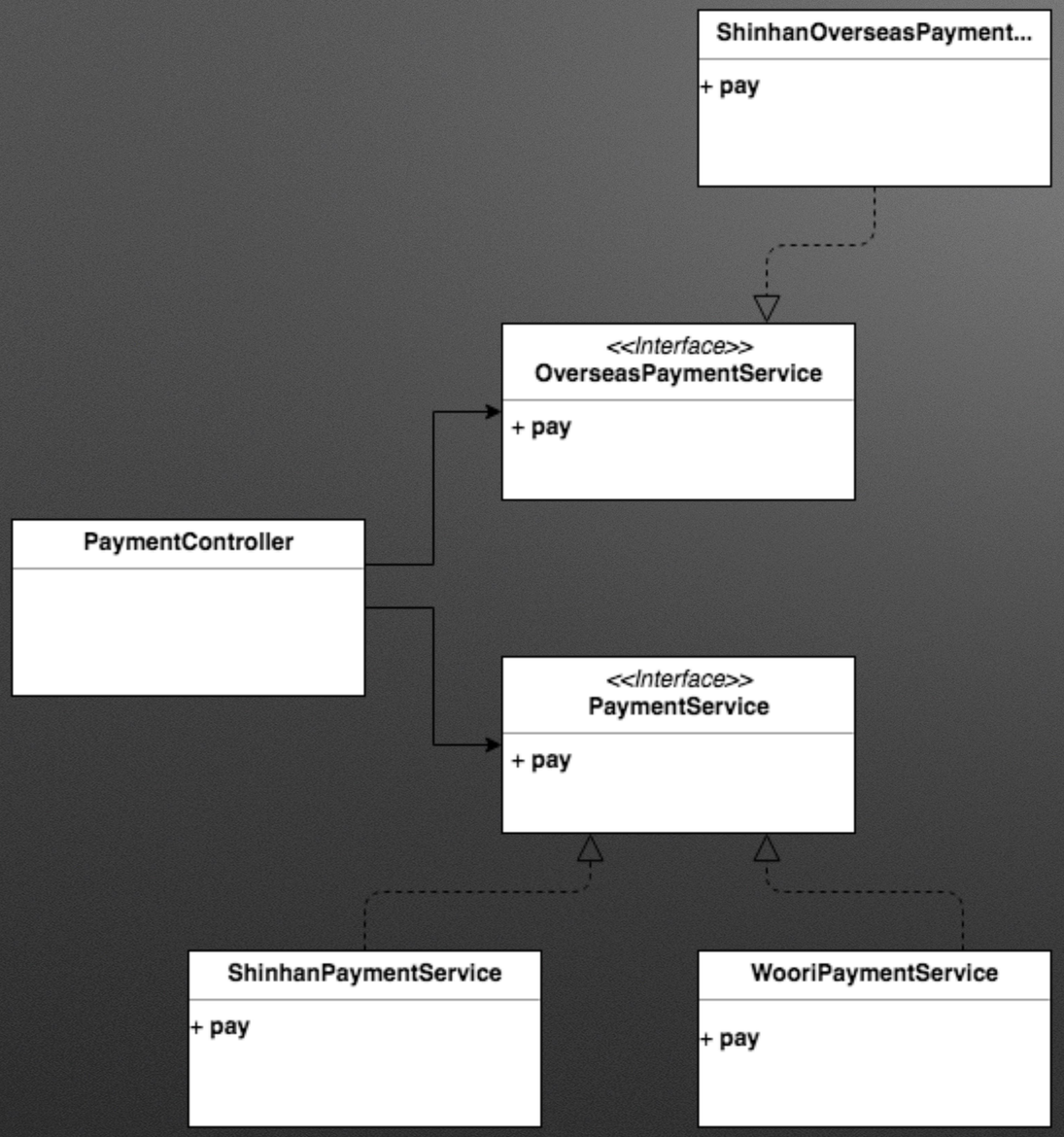
- 해외 결제를 할 수 있는 Actor는 한 개가 됩니다. Actor가 하나라는 것은 책임이 하나라는 뜻도 됩니다. 이런 경우 단일 책임의 원칙을 지켰다고 저는 개인적으로 생각합니다.
- 하지만 우리는 파트너사들이 어떤 기능을 제공할지, 또 어떤 파트너사들이 추가될지, 어떻게 변경될지 이런 부분들을 예측하기가 어려우므로 SRP를 지속적으로 준수하는 것은 정말 어렵다고 생각합니다.
- 수정 구술은 없습니다.

추가될 해외 카드 결제의 SRP(미준수)

```
public interface CardPaymentService {
    void pay(CardPaymentDto.PaymentRequest req);
    void payOverseas(CardPaymentDto.PaymentRequest req);
}

public class ShinhanCardPaymentService implements CardPaymentService {
    @Override
    public void payOverseas(CardPaymentDto.PaymentRequest req) {
        // .. 신한 카드 해외 결제 로직..
        shinhanCardApi.pay(paymentRequest);
    }
}

public class WooriCardPaymentService implements CardPaymentService {
    @Override
    public void payOverseas(CardPaymentDto.PaymentRequest req) {
        // 우리 카드 결제는 해외 결제 기능이 없음...
    }
}
```

- 신한 카드는 해외 결제를 할 수 있지만 우리 카드는 해외 결제 기능을 제공하고 있지 않습니다. 각 구현 클래스들은 CardPaymentService 인터페이스를 구현하고 있으므로 payOverseas 기능이 추가되면 우리 카드 결제는 반드시 해당 메서드를 구현 해야 합니다.
- 해외 결제만 되고 국내 결제가 안 되는 카드 파트너가 추가되면 어떻게 될까요? 그렇게 되면 위와 반대로 payOverseas 구현 메소드는 구현하고 pay는 구현하지 못하게 됩니다.

추가될 해외 카드 결제의 SRP(준수)



- 카드 파트너사의 해외 결제 여부로 더이상 PaymentService에서 **국내 결제와, 해외 결제를 처리를 못하게 되었습니다.**
- 그렇다면 **책임을 분리**시키고 그것을 인터페이스로 바라보게 하여 앞으로 **해외 결제 카드추가시 확장에 열려**있게 할 수 있습니다.

리스코프 치환의 원칙: The Liskov Substitution Principle

- 리스코브 치환 원칙은 계약과 확장에 관한것
- 계약위반 사례
  - 명시된 명세에서 벗어난 값을 리턴한다.
  - 명시된 명세에서 벗어난 Exception을 발생한다.
  - 명시된 명세에서 벗어난 기능을 수행한다.
- 리스코프 치환 원칙을 어기면 개방 폐쇄 원칙을 어길 가능성이 높아진다.
- 타입 의존성은 지독한 의존성이다

리스코프 치환의 원칙: 확장에 관한것

```java
public class LSP {
    static  P p = new P();

    static class T {
        public void  doSomething() {
            System.out.println("T#doSometing called");
        }
    }

    static class S extends T {
        public void doSomething(){ // 5. S는 T의 서브타입이다.
            System.out.println("S#doSometing called");
        }
    }

    static class P {
        public void doSomething(T p){ // 3. T 타입을 사용하는 모든 프로그램 p에서
            p.doSomething();
        }
    }

    public static void main(String[] args){
        T p = new T(); // 1. T 타입의 객체 p
        S c = new S(); // 2. S 타입의 객체 c

        LSP.p.doSomething(p); // 4. c가 p를 대체해도 P에는 어떤 변경도 없다면
        LSP.p.doSomething(c);
    }
}
```

1. T 타입의 객체 p
2. S 타입의 객체 c
3. T 타입을 사용하는 모든 프로그램 p에서
4. c가 p를 대체해도 P에는 어떤 변경도 없다면
5. S는 T의 서브타입이다.

리스코프 치환의 원칙: Sample Code는 아래 도서 참고

리스코프 치환의 원칙: 확장에 관한것

```
public class Coupon {
    public int calculateDiscountAmount(Item item) {
        return item.getPrice() * discountRate;
    }
}
```

- Coupon 클래스에 calculateDiscountAmount() 메서는 item 클래스의 getPrice() 할인된 값을 구하고 있음
- 특정 Item은 무조건 할인 해주지 않은 정책이 추가되면, 이를 위해 Item 클래스를 상속받는 SpecialItem 클래스를 추가했다고 했을 경우
- Coupon 클래스의 calculateDiscountAmount 메서드는 item 객체의 실제 타입의 SpecialItem인 경우 할인 액수를 0으로 처리 해야함

리스코프 치환의 원칙: 확장에 관한것

```
public class Coupon {
    public int calculateDiscountAmount(Item item) {
        if(item instanceof SpecialItem) return 0; // LSP 위반 발생

        return item.getPrice() * discountRate;
    }
}
```

- instanceof 코드는 대표적인 LSP 위반 사례이다.
- Item 타입을 사용하는 코드는 SpecialItem 타입이 존재하는지 알 필요 없이 오직 Item 타입에만 사용해야 한다.
- 위 코드는 instanceof 연산자를 통해서 SpecialItem 타입인지의 여부를 확인하고 있다. 즉 하위 타입인 SpecialItem이 상위 타입인 Item을 완벽하게 대체하지 못하는 상황이 발생하고 있다.

리스코프 치환의 원칙: 확장에 관한것

```
public class Coupon {
    public int calculateDiscountAmount(Item item) {
        if(item instanceof SpecialItem) return 0; // LSP 위반 발생

        return item.getPrice() * discountRate;
    }
}
```

- instanceof 연산자를 사용한다는 것은 상위 타입 Item으로만 사용해서 프로그래밍할 수 없다는 것을 뜻한다. 이는 **SpecialItem과 같은 새로운 종류의 하위 타입이 생길 때마다 상위 타입인지를 사용하는 코드를 수정**해야 한다. 이것은 **OCP 원칙을 지킬수 없게**된다
- **LSP를 어기게 된 이유는 Item에 대한 추상화가 덜 되었기 때문이다**. 할인되지 않은 상품 타입이 추가되었다는 것은 이후에 비슷한 요구가 발생할 수 있는 가능성이 높음을 뜻한다.

리스코프 치환의 원칙: 확장에 관한것

```
public class Item {
    // 변화가되는 기능을 상위 타입에 추가
    public boolean isDiscountAvaliable() {
        return true;
    }
    ...
}

public class SpecialItem extends Item {
    @Override
    public boolean isDiscountAvailabe() {
        return false;
    }
}

public class Coupon {
    public int calculateDiscountAmount(Item item) {
        if (! item.isDiscountAvailabe()) return 0; // instanceof 연산자 사용 제거
    return item.getPrice() * discountRate;
    }
}
```

리스코프 치환의 원칙: 확장에 관한것

- Item 객체에 각격 할인 기능을 여부를 판단하는 기능을 추가하고 SpecialItem 에서는 기능을 재정의 했다.
- 상위 타입에 할인 여부를 확인할 수 있는 메서드를 추가해서 instanceof를 제거할 수 있게 되었다.
- 상위 타입 Item 으로만 하위 타입의 구체적인 타입을 알필요 없이 코드구현이 가능하다.
- OCP를 준수할수 있게 되었다. LSP를 지키지 않으면 기능을 확장하기가 어렵게된다.

5. 테스팅 방법과 전략

테스트를 왜 해야할까요?

테스트를 왜 해야할까요?

- 가오가 중요합니다...

Spring Boot Test 종류

어노테이션	설명	Bean
@SpringBootTest	통합 테스트, 전체	Bean 전체
@WebMvcTest	단위 테스트, Mvc 테스트	MVC 관련된 Bean
@DataJpaTest	단위 테스트, Jpa 테스트	JPA 관련 Bean
@RestClientTest	단위 테스트, Rest API 테스트	일부 Bean
@JsonTest	단위 테스트, Json 테스트	일부 Bean

- **Bean Context를 최소한**으로 올려서 테스트를 빠르게한다.
- 원하는 구간 테스팅에 집중한다.

Spring Boot Test

- @SpringBootTest는 통합 테스트를 제공하는 기본적인 스프링 부트 테스트 어노테이션입니다.
- 애플리케이션이 실행 될 때의 설정을 임의로 바꾸어 테스트를 진행할 수 있으며 여러 단위 테스트를 하나의 통합된 테스트로 수행할 때 접합합니다.
- @SpringBootTest는 만능입니다. 실제 구동되는 애플리케이션과 똑같이 애플리케이션 컨텍스트를 도르하여 테스트하기 때문에 하고 싶은 테스트를 모두수행할 수 있습니다.
- 애플리케이션의 설정을 모두 로드하기 때문에 애플리케이션 규모가 클수록 느려집니다. 이렇게되면 단위 테스트라는 의미가 희석됩니다.
- @RunWith 어노태이션을 사용하면 JUnit에 내장된 러너를 사용하는 대신 어노테이션으로 정의돈 러너 클래스를 사용합니다.

Spring Boot Test

```
@RunWith(SpringRunner.class)
@SpringBootTest(
        properties = {
                "property.value=propertyTest",
                "value=test"
        },
        classes = {TestApplication.class},
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT
)
public class TestApplicationTests {

    @Value("${value}")
    private String value;

    @Value("${property.value}")
    private String propertyValue;

    @Test
    public void contextLoads() {
        assertThat(value, is("test"));
        assertThat(propertyValue, is("propertyTest"));
    }

}
```

Spring Boot Test

- properties: 테스트가 실행되기전에 {key=value} 형식으로 프로퍼티를 추가할 수 있습니다.
- classes: 애플리케이션 컨텍스트에 로드할 클래스를 지정합니다. 따로 지정하지 않으면 @SpringBootConfiguration을 찾아서 로드합니다.
- webEnvironment: 애플리케이션이 실행될 때의 웹 환경을 설정할 수 있습니다. 기본값은 Mock 서블릿을 로드하여 구동되며 예제에서는 랜덤 포트값을 지정
- 프로파일 환경을 갖는다면 @ActiveProfiles("test")과 같은 방식으로 원하는 프로파일 환경값을 부여 가능
- 테스트에서 @Transactional을 사용하면 테스트를 마치고 나서 수정된 데이터가 롤백됩니다. 다만 테스트가 서버의 다른 스레드에서 실행 중이라면 WebEnvironment.RANDOM_PORT, DEFINED_PORT를 사용하여 테스트를 수행하면 트랜잭션이 롤백되지 않습니다.
- @Transactional 어노테이션을 붙여주면 자동으로 rollback 처리됩니다.

@WebMvcTest

```java
@RunWith(SpringRunner.class)
@WebMvcTest(BookApi.class)
public class BookApiTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private BookService bookService;

    @Test
    public void getBook_test() throws Exception {
        //given
        final Book book = new Book(1L, "title", 1000D);

        given(bookService.getBook()).willReturn(book);

        //when
        final ResultActions actions = mvc.perform(get("/books/{id}", 1L)
                .contentType(MediaType.APPLICATION_JSON_UTF8))
                .andDo(print());

        //then
        actions
                .andExpect(status().isOk())
                .andExpect(jsonPath("id").value(1L))
                .andExpect(jsonPath("title").value("title"))
                .andExpect(jsonPath("price").value(1000D))
        ;

    }
}
```

@DataJpaTest

- @DataJpaTest 어노테이션은 **JPA 관련된 설정만 로드**합니다.
- 데이터소스의 설정이 정상적인지, JPA를 사용하서 데이터를 제대로 생성, 수정, 삭제하는지 등의 테스트가 가능합니다.
- 기본적으로 인메모리 데이터베이스를 이용합니다.
- @Entity 클래스를 스캔하여 스프링 데이터 JPA 저장소를 구성합니다.

@DataJpaTest

```
@RunWith(SpringRunner.class)
@DataJpaTest
@ActiveProfiles("test")
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class BookJpaTest {


    @Autowired
    private BookRepository bookRepository;

    @Test
    public void book_save_test() {
        final Book book = new Book("title", 1000D);
        final Book saveBook = bookRepository.save(book);
        assertThat(saveBook.getId(), is(notNullValue()));
    }

    @Test
    public void book_save_and_find() {
        final Book book = new Book("title", 1000D);
        final Book saveBook = bookRepository.save(book);
        final Book findBook = bookRepository.findById(saveBook.getId()).get();
        assertThat(findBook.getId(), is(notNullValue()));
    }
}
```

@DataJpaTest

- @AutoConfigureTestDatabase 어노테이션의 기본 설정값인 Replace.Any를 사용하면 기본적으로 내장된 데이터소스를 사용합니다.
- 위와 같이 @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)을 지정하면 @ActiveProfiles("test") 기준으로 프로파일이 설정됩니다.
- @DataJpaTest는 테스트가 끝날 때마다 자동으로 테스트에 사용한 데이터를 롤백합니다.
- 데이터 베이스의 종류도 선택할 수 있습니다.

@RestClientTest

```
@RunWith(SpringRunner.class)
@RestClientTest(BookRestService.class)
public class BookRestServiceTest {

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Autowired
    private BookRestService bookRestService;

    @Autowired
    private MockRestServiceServer server;

    @Test
    public void rest_test() {

        server.expect(requestTo("/rest/test"))
                .andRespond(
                        withSuccess(new ClassPathResource("/test.json", getClass()), MediaType.APPLICATI

        Book book = bookRestService.getRestBook();

        assertThat(book.getId(), is(notNullValue()));
        assertThat(book.getTitle(), is("title"));
        assertThat(book.getPrice(), is(1000D));

    }
}
```

- @RestClientTest는 REST 관련 테스트를 도와주는 어노테이션입니다.
- REST 통신의 JSON 형식이 예상대로 응답을 반환하는지 등을 테스트 합니다.

@RestClientTest

- @RestClientTest는 테스트 대상이 되는 Bean을 주입받습니다.
- @Rule로 지정한 필드 값은 @Before, @After 어노테이션에 상관없이 하나의 테스트가 메서드가 끝날 때마다 정의한 값으로 초기화 시켜줍니다. 테스트에서 자체적으로 자체적으로 규칙을 정의하여 재사용할 때 유용합니다.
- MockRestServiceServer는 클라이언트와 서버 사이의 REST 테스트를 위한 객체입니다. 내부에서 RestTemplate를 바인딩하여 실제로 통신이 이루어지게끔 구성할 수도 있습니다. 이 코드에서는 목 객체와 같이 실제로 통신이 이루어지지는 않지만 지정한 결로에 예상되는 반환값을 명시합니다.
- rest_test() 메서드는 요청에 대한 응답과 기댓값이 같은지 테스트합니다. 요청을 보내면 현지 리소스 폴더에 있는 test.json 데이터로 응답을 주도록 설정합니다(목 객체와 비슷한 개념입니다.).

@RestTemplate Mock 기반 테스트 하기

왜 Mock 기반 테스트를 진행 해야하나 ?

- 외부 인프라스트럭처를 테스트 코드를 작성할 때는 mock 기반으로 테스트를 진행해야 합니다. 외부 인프라스트럭처에는 대표적으로 외부 API가 있습니다.
- 그런데 왜 Mock 기반으로 테스트해야 할까요? 외부 환경이기 때문에 제어권이 우리 밖에 있습니다. 그래서 우리 코드의 테스트를 위해서 Request, Response가 우리가 주어진 값인 경우 정상 동작하는지 테스트하기 위함입니다.
- Mockito 기반으로 데이터베이스의 의존성을 Mock으로 처리하고 실제 비즈니스 코드를 검증하는 거와 같은 결이라고 생각하면 됩니다.

@RestTemplate Mock 기반 테스트 하기

```
@Configuration
class Configuration {

    @Bean
    fun restTemplate(objectMapper: ObjectMapper): RestTemplate {
        return RestTemplateBuilder()
            .rootUri("http://localhost:8080")
            .setConnectTimeout(Duration.ofSeconds(10))
            .additionalMessageConverters()
            .build()
    }
}
@RestController
@RequestMapping("/teams")
class TeamApi(
    private val query: JPAQueryFactory
) {

    @GetMapping
    fun get(@RequestParam(required = false) name: String?): List<Team> {
        return query.selectFrom(qTeam)
            .from(qTeam)
            .fetch()
    }
}

@Service
class ApiService(
    private val restTemplate: RestTemplate
) {

    fun getTeam(name: String): List<Team> {
        return restTemplate.getForObject("/teams?name=$name", Array<Team>::class.java)!!.toList()
    }
}
```

@RestTemplate Mock 기반 테스트 하기

```
@SpringBootTest
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
internal class ApiServiceTest(
    private val restTemplate: RestTemplate, // (1)
    private val apiService: ApiService // (2)

) {
    private var server: MockRestServiceServer by notNull()

    @BeforeEach
    internal fun setUp() {
        server = MockRestServiceServer.createServer(restTemplate) // (3)
    }


    @Test
    internal fun `api test`() {
        //given
        val path = "/team-api-response.json"
        val name = "team"
        server
            .expect(
                requestTo("http://localhost:8080/teams?name=$name")
            )// (4)
            .andExpect(method(HttpMethod.GET)) // (5)
            .andRespond(
                withStatus(HttpStatus.OK)
                    .contentType(MediaType.APPLICATION_JSON)
                    .body(ClassPathResource(path, javaClass))
            )// (6)

        //when
        // (7)
        val teams = apiService.getTeam("team")

        //then
        // (8)
        for (team in teams) {
            println(team)
        }

        // (9)
        then(teams).hasSize(10)
        then(teams).anySatisfy {
            then(it.name).startsWith("team")
        }
    }
}
```

- (1) : 해당 서비스에서 사용하는 RestTemplate를 주입받습니다. 실제 실무에서는 RestTemplate를 vendor마다 등록하기 때문에 테스트하고자 하는 RestTeamplte를 주입받습니다.
- (2) : 해당 API 통신하는 서비스를 주입받습니다.
- (3) : 주입받은 RestTemplate 기반으로 MockRestServiceServer 생성합니다. MockRestServiceServer는 클라이언트와 서버 사이의 REST 테스트를 위한 객체입니다. 내부에서 RestTemplate를 바인딩 하여 실제로 통신이 이루어지게끔 구성할 수도 있습니다. 이 코드에서는 목 객체와 같이 실제로 통신이 이루어지지는 않지만 지정한 결로에 예상되는 반환값을 명시합니다.
- (4) : 검증을 시도합니다. 실제 restTeamplte의 Request가 저 URL으로 해당하는지 검증합니다.
- (5) : 실제 Request의 HTTP Method가 GET을 이용하는지 검증합니다.
- (6) : Response에 대한 검증을 시작합니다. HTTP Status Code, Body, MediaType 등을 검증합니다. path 변수에는 해당 API의 예상되는 json 값을 작성해 놓습니다. Givne 절이라고 생각하면 됩니다.
- (7) : 실제 서비스 코드에서 호출하는 코드를 호출합니다.
- (8) : 테스트 동작 여부를 확인하기 위해서 print 하는 코드입니다.
- (9) : 실제 when 절의 코드가 예상하는 것처럼 동작했는지 검증합니다.

무엇을 테스트할 것인가?

Request 테스트

```
...

val name = "team"
    server
        .expect(
            requestTo("http://localhost:8080/teams?name=$name")
        )// (4)

val teams = apiService.getTeam("kotlin") // team -> kotlin 으로 이름 변경
```

```
2020-02-01 17:45:31.201 TRACE 13435 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding paramet
2020-02-01 17:45:31.201 TRACE 13435 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding paramet

Request URI expected:<http://localhost:8080/teams?name=team> but was:<http://localhost:8080/teams?name=kotlin>
Expected :http://localhost:8080/teams?name=team
Actual   :http://localhost:8080/teams?name=kotlin
<Click to see difference>
```

- 일단 Mock 기반 테스트는 단위 테스트 성격이 높습니다. 그렇다면 무엇을 테스트할 것인지 명확하게 테스트해야 합니다. 해당 테스트에서는 원하는 곳으로 Request가 가는지, 우리가 원하는 Reponse가 예상되었을 경우 우리 로직이 제대로 동작하는지 테스트를 진행하겠습니다.
- MockRestServiceServer에서는 요청을 teams?=name=team으로 했고 실제 apiService에서는 kotlin 을 넘겼기 때문에 두 요청은 일치하지 않습니다. 이 상태에서 테스트 코드를 실행하면 아래와 같은 오류를 발견합니다.
- 즉 실제 apiService 를 통한 실제 요청과 mock 기반 요청이 일치하는지 확인을 검증하는 것입니다. 파라미터가 name 하나여서 간단하지 조회용 condition이 상당히 복잡합니다. 이런 것들을 케이스 별로 테스트하는 것이 바람직합니다.

무엇을 테스트할 것인가?

```
server
        .expect(
            requestTo("http://localhost:8080/teams?name=$name")
        )// (4)
        .andExpect(method(HttpMethod.POST)) // (5) GET -> POST 변경
```

```
Unexpected HttpMethod expected:<POST> but was:<GET>
Expected :POST
Actual   :GET
<Click to see difference>
```

무엇을 테스트할 것인가?

Response 테스트

```
server
        ...
        .andRespond(
            withStatus(HttpStatus.OK)
                .contentType(MediaType.APPLICATION_JSON)
                .body(ClassPathResource(path, javaClass))
        )// (6)

val teams = apiService.getTeam("team")
```

```
[
  {
    "name": null,
    "id": 1,
    "createdAt": "2020-02-01T16:58:34.377",
    "updatedAt": "2020-02-01T16:58:34.377",
    "members": []
  }
  ...
]
```

```
Error while extracting response for type [class [Lcom.example.querydsl.domain.Team;] and content type [application/json]; nested exception is org.springframework.http.converter
.HttpMessageNotReadableException: JSON parse error: Instantiation of [simple type, class com.example.querydsl.domain.Team] value failed for JSON property name due to missing (therefore
NULL) value for creator parameter name which is a non-nullable type; nested exception is com.fasterxml.jackson.module.kotlin.MissingKotlinParameterException: Instantiation of [simple
type, class com.example.querydsl.domain.Team] value failed for JSON property name due to missing (therefore NULL) value for creator parameter name which is a non-nullable type
at [Source: (BufferedInputStream); line: 4, column: 5] (through reference chain: java.lang.Object[][0]->com.example.querydsl.domain.Team["name"])
org.springframework.web.client.RestClientException: Error while extracting response for type [class [Lcom.example.querydsl.domain.Team;] and content type [application/json]; nested
exception is org.springframework.http.converter.HttpMessageNotReadableException: JSON parse error: Instantiation of [simple type, class com.example.querydsl.domain.Team] value failed
for JSON property name due to missing (therefore NULL) value for creator parameter name which is a non-nullable type; nested exception is com.fasterxml.jackson.module.kotlin
.MissingKotlinParameterException: Instantiation of [simple type, class com.example.querydsl.domain.Team] value failed for JSON property name due to missing (therefore NULL) value for
creator parameter name which is a non-nullable type
```

- .body(ClassPathResource(path, javaClass))는 /src/test/resources/ path에 있는 해당 json 파일 기반으로 body를 mock 해서 돌려줍니다.
- 역 직렬화를 못하게 name을 null으로 변경해보겠습니다. (team name 객체는 notnull입니다.)
- 역 직렬화를 못하기 때문에 테스트가 실패하는 것을 확인할 수 있습니다.

무엇을 테스트할 것인가?

- RestTemplate 기반으로 테스트하는 방법을 포스팅하긴 했지만 더 강조하고 싶은 부분은 외부 디펜던시 어떻게 끊고 어떤 부분을 어떤 관점에서 테스트할지 고민해보고 테스트를 작성하는 것이 좋다는 것입니다.

테스트 전략

어노테이션	설명	부모 클래스	Bean
@SpringBootTest	통합 테스트, 전체	IntegrationTest	Bean 전체
@WebMvcTest	단위 테스트, Mvc 테스트	MockApiTest	MVC 관련된 Bean
@DataJpaTest	단위 테스트, Jpa 테스트	RepositoryTest	JPA 관련 Bean
None	단위 테스트, Service 테스트	MockTest	None
None	POJO, 도메인 테스트	None	None

1. Mock 기반 Service 테스트 = POJO 도메인테스트
2. WebMvc Test
3. Spring Boot Test

통합 테스트

장점

- 모든 Bean을 올리고 테스트를 진행하기 때문에 쉽게 테스트 진행 가능
- 모든 Bean을 올리고 테스트를 진행하기 때문에 운영환경과 가장 유사하게 테스트 가능
- API를 테스트할 경우 요청부터 응답까지 전체적인 테스트 진행 가능

단점

- 모든 Bean을 올리고 테스트를 진행하기 때문에 테스트 시간이 오래 걸림
- 테스트의 단위가 크기 때문에 테스트 실패시 디버깅이 어려움
- 외부 API 콜같은 Rollback 처리가 안되는 테스트 진행을 하기 어려움

Code

서비스 테스트

장점

- 진행하고자 하는 테스트에만 집중할 수 있습니다.
- 테스트 진행시 중요 관점이 아닌 것들은 Mocking 처리해서 외부 의존성들을 줄일 수 있습니다.
  - 예를 들어 주문 할인 로직이 제대로 동작하는지에 대한 테스트만 진행하지 이게 실제로 데이터베이스에 insert되는지는 해당 테스트의 관심사가 아닙니다.
- 테스트 속도가 빠릅니다.

단점

- 의존성 있는 객체를 Mocking 하기 때문에 문제가 완결된 테스트는 아닙니다.
- Mocking 하기가 귀찮습니다.
- Mocking 라이브러리에 대한 학습 비용이 발생합니다.

Code

Repository Test

장점

- `Repository` 관련된 Bean들만 등록하기 때문에 통합 테스트에 비해서 빠릅니다.
- `Repository` 에 대한 관심사만 갖기 때문에 테스트 범위가 작습니다.

단점

- 테스트 범위가 작기 때문에 실제 환경과 차이가 발생합니다.

Code

POJO 테스트

장점

- POJO 객체이므로 테스트하기 편합니다. 외부에서 주입 받을 의존성도 없고 Mocking할 대상도 없습니다.
- 엔티티 객체는 사용하는 계층이 많으므로 테스트의 효율성이 높습니다.

단점

- 단점은 없다고 생각합니다. POJO를 테스트 하므로 테스트 속도 및 난도가 낮지만 높은 안전성을 갖게 됩니다.

Code

외부 인프라스트럭처 테스트

외부 인프라스트럭처 테스트

- 대부분의 애플리케이션은 외부 인프라스트럭처와 통신하면서 진행됩니다. 대표적인 외부 스트럭처는 외부 API들이 있습니다. 이런 외부 인프라스트럭처는 Mocking 해서 원하는 응답 값을 지정하고 검증하고 싶은 부분을 검증을 진행하는 것이 흔한 패턴입니다.
- 대표적으로 Mockito 프레임워크가 있으며 Mock 테스트는 Spring Guide - 테스트 전략 : Service 테스트, RestTemplate Mock 기반 테스트 하기에서 포스팅한 적 있습니다. 그런데 이런 식의 Mock 테스트 코드는 문제없지만, 실제 구동 환경(Local, Sandbox, Beta)에서는 문제가 있을 수 있습니다.

요구사항

- 파트너 등록을 진행한다.
- 파트너 등록 시 계좌 주명, 계좌번호 일치 여부를 검증한다.
  - 계좌 주명 검증은 신한 API를 사용한다.
    - API는 허가된 서버만 호출할 수 있다.
    - API Call 1건당 비용이 발생한다.
- 계좌 주명, 계좌번호 일치하는 경우 저장한다.
- 계좌 주명, 계좌번호 일치하지 않은 경우 예외를 발생시킨다.

Code

```
@RestController
@RequestMapping("/partners")
class PartnerApi(
    private val partnerRegistrationService: PartnerRegistrationService
) {

    @PostMapping
    fun register(@RequestBody @Valid dto: PartnerRegistrationRequest) {
        partnerRegistrationService.register(dto)
    }
}

data class PartnerRegistrationRequest(
    @field:NotEmpty
    val name: String,
    @field:NotEmpty
    val accountHolder: String,
    @field:NotEmpty
    val accountNumber: String
)

@Service
class PartnerRegistrationService(
    private val partnerRepository: PartnerRepository,
    private val bankClient: ShinhanBankClient
) {

    fun register(dto: PartnerRegistrationRequest): Partner {

        // 은행 코드 검증일 진행한다
        bankClient.verifyAccountHolder(
            accountHolder = dto.accountHolder,
            accountNumber = dto.accountHolder
        )

        return partnerRepository.save(Partner(
            accountNumber = dto.accountNumber,
            accountHolder = dto.accountHolder,
            name = dto.name
        ))
    }
}
```

```
@Service
class ShinhanBankClient(
    private val shinChanBankApi: ShinChanBankApi
) {

    // 계좌주명, 계좝번호가 일치하지 않으면 예외 발생
    fun verifyAccountHolder(accountNumber: String, accountHolder: String) {
        val response = shinChanBankApi.checkAccountHolder(accountHolder, accountNumber)
        require(!response.matched.not()) { "계좌주명이 일치하지 않습니다." }
    }
}

@Service
class ShinChanBankApi {

    // 계좌주명, 계좌번호가 하드 코딩된 값과 일치여불르 확인한다.
    fun checkAccountHolder(accountHolder: String, accountNumber: String): AccountHolderVerificationResponse {
        return when {
            accountHolder == "yun" && accountNumber == "110-2222-2222" -> AccountHolderVerificationResponse(tru
            else -> AccountHolderVerificationResponse(false)
        }
    }
}

data class AccountHolderVerificationResponse(
    val matched: Boolean
)
```

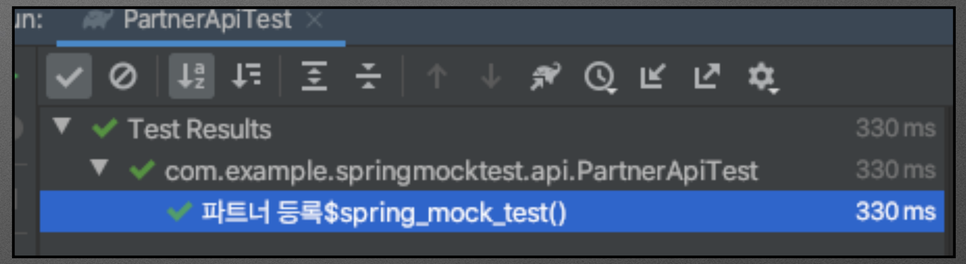
- 예제 코드가 길지만 흐름은 간단합니다. 파트너 등록을 진행할 때 ShinhanBankClient를 통해서 계좌 주명 일치 여부를 검증하고, 일치하지 않은 경우 예외가 발생시켜 파트너 등록을 진행하지 않습니다.

- 본 포스팅과는 관련 없는 내용이지만 실제 API Call은 ShinChanBankApi 클래스에서 진행하고, 일치하지 않은 경우 예외가 발생하는 비즈니스 코드는 ShinhanBankClient 클래스에서 진행합니다.

- 단순하게 API 통신만 담당하는 객체, API 통신을 담당하는 객체를 이용해서 비즈니스 코드를 만드는 객체 이렇게 객체의 책임과 역할을 명확하게 나누고 그 크기를 작게 유지하는 것이 좋은 설계 좋은 코드라고 생각합니다.

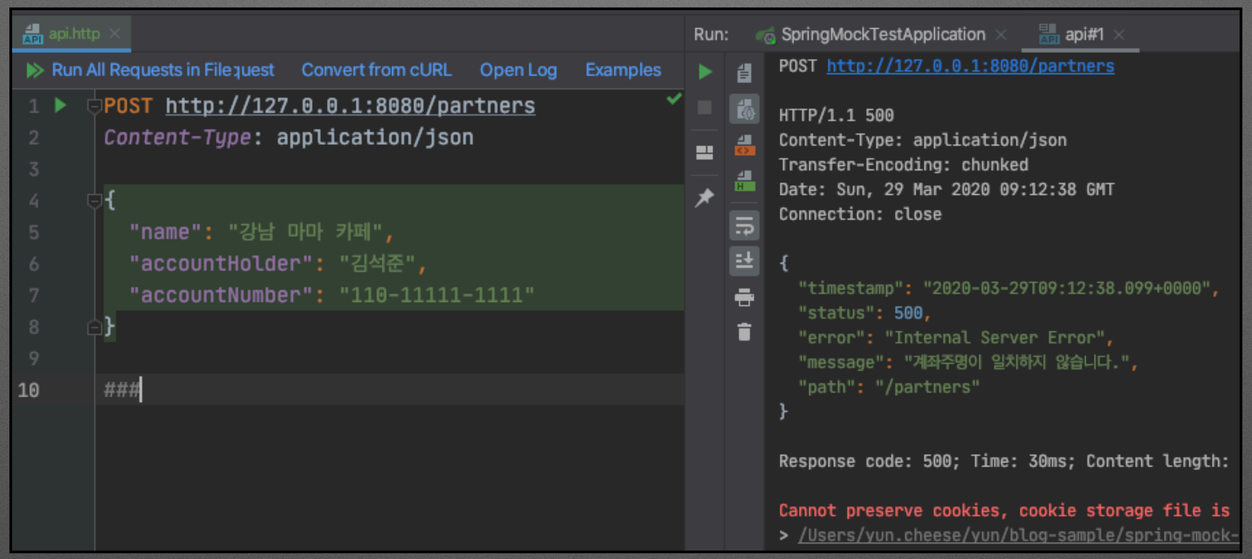
Test Code

```
internal class PartnerApiTest : SpringApiTestSupport() {

    @MockBean
    private lateinit var shinChanBankApi: ShinChanBankApi

    @Test
    internal fun `파트너 등록`() {

        given(shinChanBankApi.checkAccountHolder(anyString(), anyString()))
            .willReturn(AccountHolderVerificationResponse(true))

        mockMvc.post("/partners") {
            contentType = MediaType.APPLICATION_JSON
            content = """
                {
                  "name" : "123",
                  "accountHolder" : "123",
                  "accountNumber" :  "123"
                }
            """.trimIndent()
        }.andExpect {
            status { isOk }
        }
    }
}
```



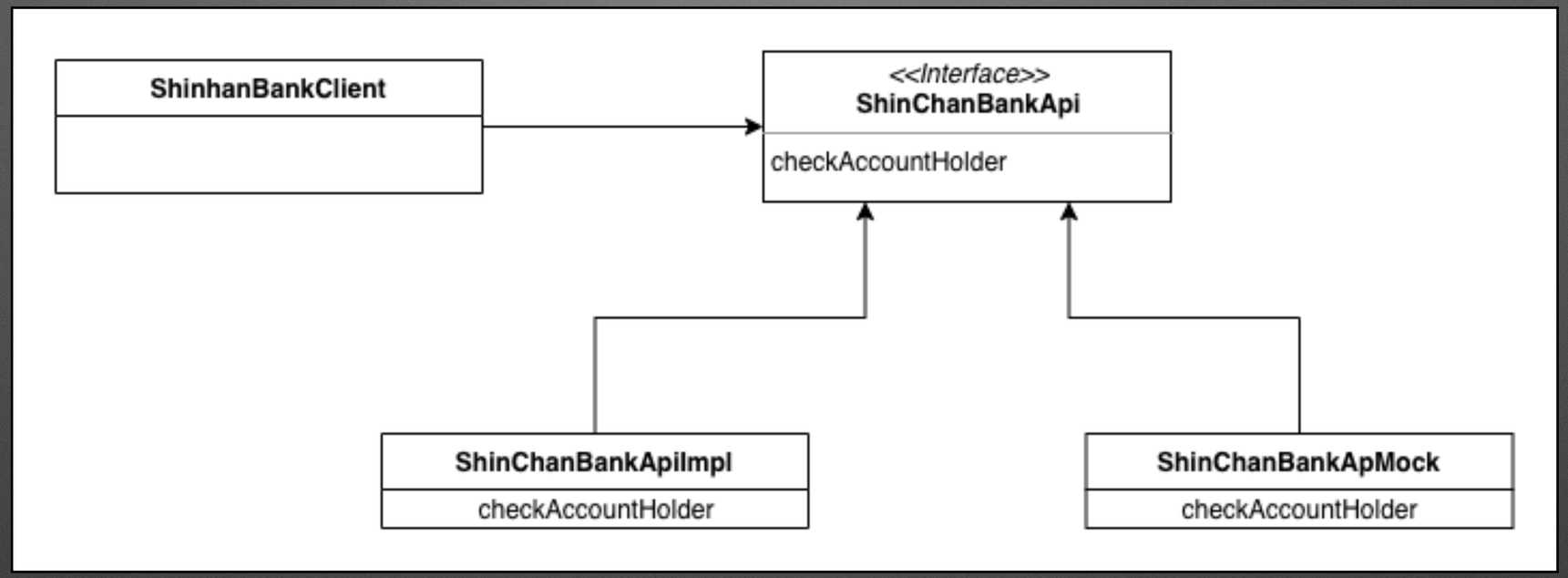
- MockBean을 통해서 가짜 객체를 주입받고 given() 메서드를 통해서 일치한다는 가정을 하고 테스트를 진행하게 됩니다.

하지만 문제는 ?



- 하지만 문제가 있습니다. 실제 구동 환경(Local, Sandbox, Beta...)에서는 해당 API Call을 진행하면 예외가 발생합니다.
- 허가된 Production 서버 이외의 조건들은 호출은 가능하지만 API 비용 발생 문제 등이 있어 Production 이외의 환경에서는 막는 것이 적절할 수 있습니다. 그 밖에 다양한 이유들이 있습니다.
  - 실제 Email을 호출하는 경우, 실제 SMS으로 문자가 나가는 경우, 실제 송금을 진행하는 경우
- 상대 서버에 sandbox, beta 등 production 환경 이외의 테스트 서버가 없는 경우
- 해당 이유들은 비용상의 문제, 서버 권한 문제 외 적으로 막아야 합니다. 그리고 무엇보다 Production 환경 외에서 해당 API를 사용(테스트) 할 수 없게 되는 것이 가장 큰 문제입니다.

해결 방법은 ?



- 해결 방법은 간단합니다. ShinhanBankClient가 ShinhanBankApi 인터페이스를 의존하게 함으로써 의존관계를 인터페이스를 통해서 역전시키는 것입니다. 이것도 개방 폐쇄의 원칙: Open Close Principle에서 포스팅한 적 이 있습니다.

해결 방법은 ?

```
interface ShinChanBankApi {
    fun checkAccountHolder(accountHolder: String, accountNumber: String): AccountHolderVerificationResponse
}

@Service("shinChanBankApi")
@Profile("production")
class ShinChanBankApiImpl : ShinChanBankApi {
    // 계좌주명, 계좌번호가 하드 코딩된 값과 일치여불르 확인한다.
    override fun checkAccountHolder(accountHolder: String, accountNumber: String): AccountHolderVerificationR
        return when {
            accountHolder == "yun" && accountNumber == "110-2222-2222" -> AccountHolderVerificationResponse(t
            else -> AccountHolderVerificationResponse(false)
        }
    }
}

@Service("shinChanBankApi")
@Profile("sandbox", "beta", "local", "test")
private class ShinChanBankApMock : ShinChanBankApi {
    // 어떤 값이 들어 와도 일치 한다고 가정한다
    override fun checkAccountHolder(accountHolder: String, accountNumber: String): Account
        return AccountHolderVerificationResponse(true)
    }
}
```

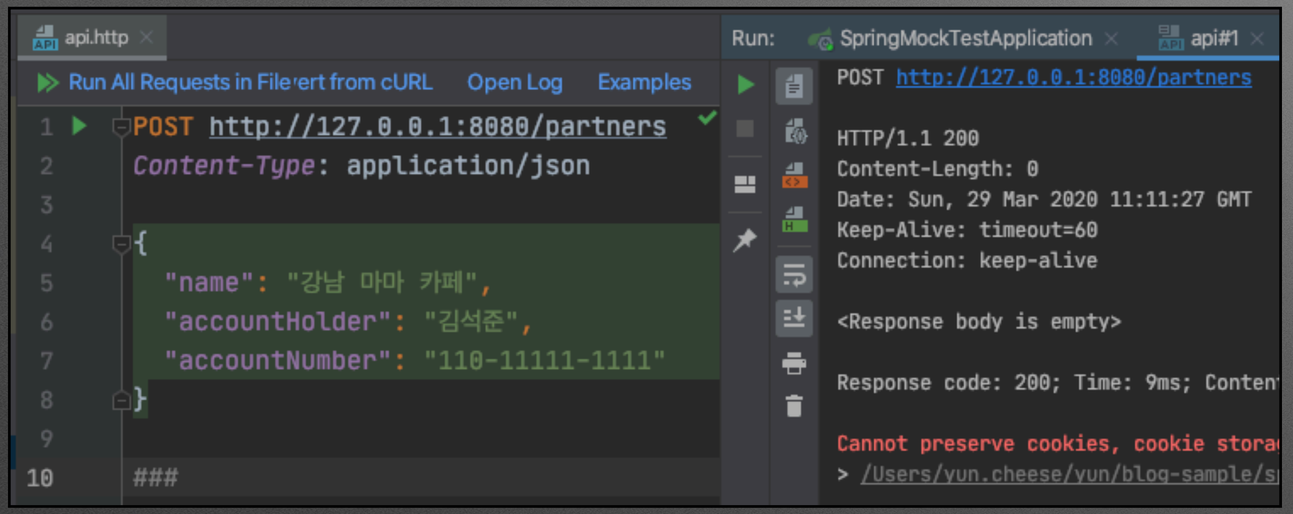
```
@Service
class PartnerRegistrationService(
    private val partnerRepository: PartnerRepository,
    private val bankClient: ShinhanBankClient
) {

    fun register(dto: PartnerRegistrationRequest): Partner {
        // 은행 코드 검증일 진행한다
        bankClient.verifyAccountHolder(
            accountHolder = dto.accountHolder,
            accountNumber = dto.accountHolder
        )

        return partnerRepository.save(Partner(
            accountNumber = dto.accountNumber,
            accountHolder = dto.accountHolder,
            name = dto.name
        ))
    }
}
```

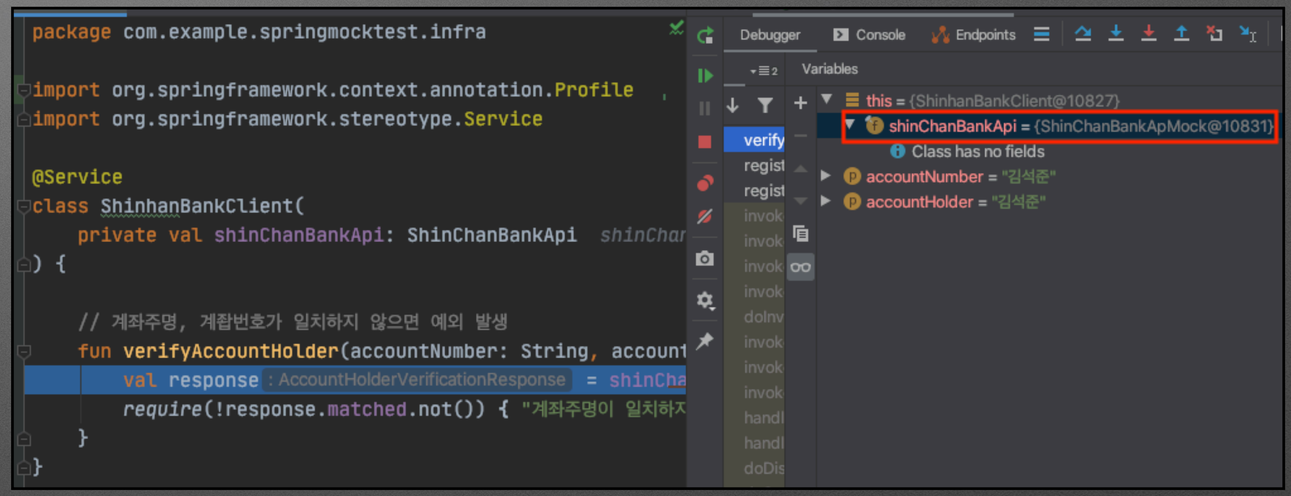
해결 방법은 ?

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.6.RELEASE)

2020-03-29 20:02:04.384  INFO 64521 --- [           main] c.e.s.SpringMockTestApplicationKt        : Starting SpringMockTestApplicationKt on
 yuncheeseui-MacBookPro.local with PID 64521 (/Users/yun.cheese/yun/blog-sample/spring-mock-test/build/classes/kotlin/main started by yun.cheese in
 /Users/yun.cheese/yun/blog-sample/spring-mock-test)
2020-03-29 20:02:04.391  INFO 64521 --- [           main] c.e.s.SpringMockTestApplicationKt        : The following profiles are active: local
2020-03-29 20:02:04.953  WARN 64521 --- [           main] ConfigServletWebServerApplicationContext : Exception encountered during context
 initialization - cancelling refresh attempt: org.springframework.beans.factory.BeanDefinitionStoreException: Failed to parse configuration class
 [com.example.springmocktest.SpringMockTestApplication]; nested exception is org.springframework.context.annotation
 .ConflictingBeanDefinitionException: Annotation-specified bean name 'shinChanBankApi' for bean class [com.example.springmocktest.infra
 .ShinChanBankApiImpl] conflicts with existing, non-compatible bean definition of same name and class [com.example.springmocktest.infra
 .ShinChanBankApMock]
2020-03-29 20:02:05.374 ERROR 64521 --- [           main] o.s.boot.SpringApplication               : Application run failed

org.springframework.beans.factory.BeanDefinitionStoreException: Failed to parse configuration class [com.example.springmocktest
 .SpringMockTestApplication]; nested exception is org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean
  name 'shinChanBankApi' for bean class [com.example.springmocktest.infra.ShinChanBankApiImpl] conflicts with existing, non-compatible bean definition
  of same name and class [com.example.springmocktest.infra.ShinChanBankApMock]
  at org.springframework.context.annotation.ConfigurationClassParser.parse(ConfigurationClassParser.java:188) ~[spring-context-5.2.5.RELEASE.jar:5.2.5
  RELEASE]
```

- @Service("beanName") beanName을 지정합니다.
- @Profile 어노테이션을 통해서 각 환경에 맞는 bean이 등록되게 진행합니다.
- PartnerRegistrationService에서 ShinChanBankApi 인터페이스를 의존하게 합니다.

해결 방법은 ?





- spring.profiles.active=local 으로 지정하고 해당 API를 호출하면 Status 200을 응답받은 것을 확인할 수 있습니다.
- spring.profiles.active=local환경에서는 shinhanBankApi객체에 ShinhanBankApiMock이 등록된것을 확인할 수 있습니다.

JPA 기반 테스트 코드 작성 팁

JPA 기반 테스트 불편한 점

```
internal class JpaTest(
    private val paymentRepository: PaymentRepository,
    private val memberRepository: MemberRepository,
    private val orderRepository: OrderRepository
) : SpringBootTestSupport() {

    @Test
    internal fun `특정 테스트를 하기위해서는 외부 dependency가 필요하다`() {
        // 특정 테스트를 하기 위해서 많은 디펜던시가 필요하다.
        paymentRepository.save(Payment(BigDecimal.TEN))
        memberRepository.save(Member("username", 10, Team("team-name")))
        orderRepository.save(Order(BigDecimal.TEN))
    }
}
```

- 많은 디펜던시가 필요한 부분의 테스트 코드를 작성할 때는 많은 Repository를 주입 받아서 테스트를 진헹 해야합니다. 위 코드처럼 특정 구간의 서비스를 테스트하기 위해서는 Given절을 작성할 때 흔하게 발생합니다.
- 물론 테스트 코드이기 때문에 DI 받아야하는 항목들이 많아지는것이 상대적으로 문제가 크게 발생하지는 않지만 코드 양이 많이 발생하는 문제가 있습니다.

검증이 필요한데 ?...

```
@Test
internal fun `특정 테스트를 하기위해서는 외부 dependency가 필요하다`() {
    // 특정 테스트를 하기 위해서 많은 디펜던시가 필요하다.
    paymentRepository.save(Payment(BigDecimal.TEN))
    memberRepository.save(Member("username", 10, Team("team-name")))
    orderRepository.save(Order(BigDecimal.TEN))


    // 특정 서비스가 여러 entity rows를 변경할때 아래와 같은 조회로 Then 이어가야 합니다.
    // paymentRepository.findBy... epository 메서드는 없는데??...
    // memberRepository.findBy...
    // orderRepository.findBy...
}
```

- 해당 서비스의 코드가 여러 enttiy의 여러 row의 변경을 가할 때 Then절에서 검증을 진행할 때 문제가 발생합니다.
- 해당 조회 코드가 없는 경우 오직 테스트를 위해서만 조회용 코드를 일반 코드에 작성해야 합니다. 그렇지 않다면 Test Scope에서 사용할 Repository를 따로 작성해야 합니다.

해결 방법

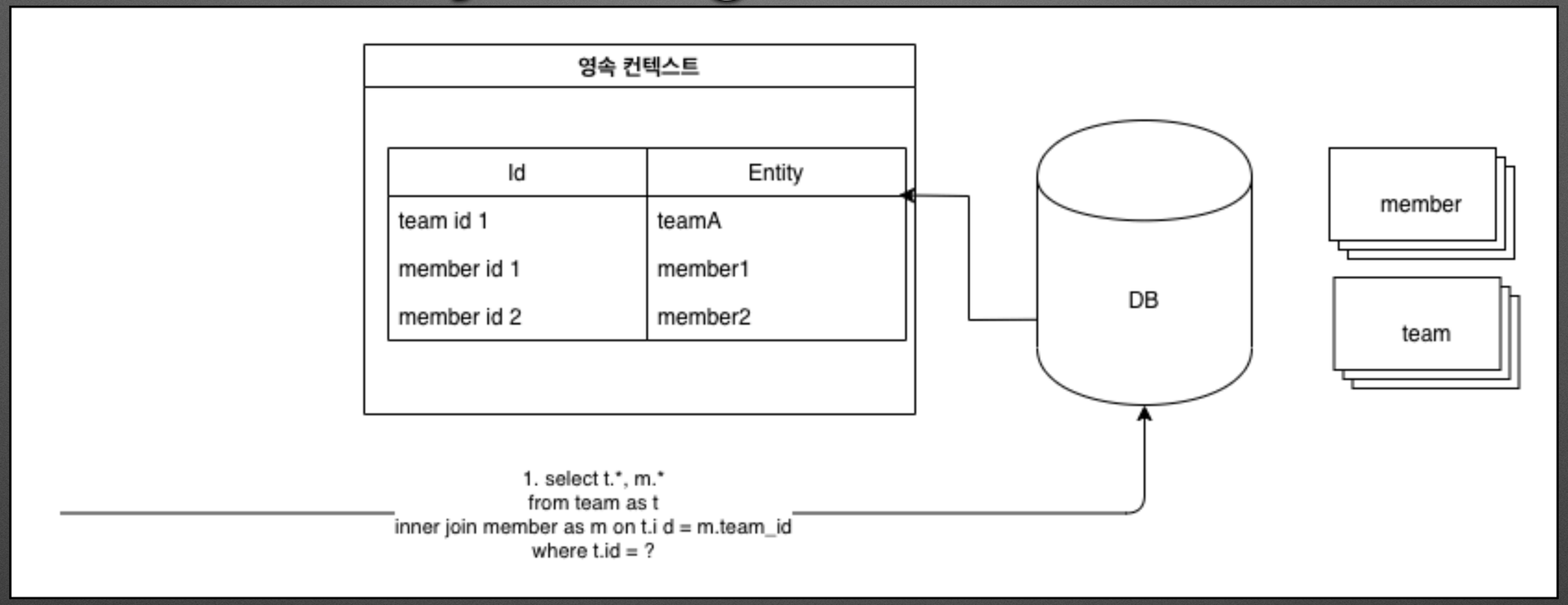
```
@SpringBootTest
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
@ActiveProfiles("test")
abstract class SpringBootTestSupport {

    @Autowired
    protected lateinit var entityManagerFactory: EntityManagerFactory

    @Autowired
    protected lateinit var query: JPAQueryFactory

    protected val entityManager: EntityManager by lazy {
        entityManagerFactory.createEntityManager()
    }

    protected val transaction: EntityTransaction by lazy {
        entityManager.transaction
    }

    protected fun <T> save(entity: T): T {
        transaction.begin()

        try {
            entityManager.persist(entity)
            entityManager.flush() // transaction commit시 자동으로 flush 발생시키나 명시적으로 선언
            transaction.commit()
            entityManager.clear()

        } catch (e: Exception) {
            transaction.rollback()
        }

        return entity
    }


    protected fun <T> saveAll(entities: Iterable<T>): Iterable<T> {
        transaction.begin()

        for (entity in entities) {
            try {
                entityManager.persist(entity)
                entityManager.flush() // transaction commit시 자동으로 flush 발생시키나 명시적으로 선언
                transaction.commit()
                entityManager.clear()

            } catch (e: Exception) {
                transaction.rollback()
            }
        }
        return entities
    }
}
```

과도한 Dependency 해결 방법

```
protected fun <T> save(entity: T): T {
    transaction.begin()

    try {
        entityManager.persist(entity)
        entityManager.flush() // transaction commit시 자동으로 flush 발생시키나 명시적으로 선언
        transaction.commit()
        entityManager.clear()
    } catch (e: Exception) {
        transaction.rollback()
    }
    return entity
}


protected fun <T> saveAll(entities: Iterable<T>): Iterable<T> {
    transaction.begin()

    for (entity in entities) {
        try {
            entityManager.persist(entity)
            entityManager.flush() // transaction commit시 자동으로 flush 발생시키나 명시적으로 선언
            transaction.commit()
            entityManager.clear()

        } catch (e: Exception) {
            transaction.rollback()
        }
    }
    return entities
}
```

- Given절에서 작성하는 데이터 Set up이기 때문에 트랜잭션을 완전히 분리하기 위해서 transactionn을 commit을 직접 수행 시킵니다.
- transaction commit 시에 flush가 동작하지만 명시적으로 코드를 작성했습니다.

EntityManager를 이용해서 clear



- 영속성 컨텍스트는 1차 캐시를 우선시 합니다. 즉 추가적인 JPQL 쿼리로 조회한 값이 영속성 컨텍스트에 존재하는 경우(식별자 값으로 동일성을 판단함) JPQL로 조회한 데이터를 버립니다. 즉 1차 캐시된것을 우선시합니다.
- 그러기 때문에 EntityManager를 이용해서 영속성 컨텍스트를 초기화하는 것입니다. 자세한 내용은 JPA 벌크 작업 주의점 - 영속성 컨텍스트는 1차 캐시된 것이 우선이다.
- 그리고 Give절에 작성하는 데이터는 이미 데이터베이스에 영속화되어 있다는 개념으로 테스트하는 것이니 이런 문제가 없더라도 최대한 동일한 환경을 구성해주는 것이 좋다고 생각합니다.

검증이 필요한데 ?... 해결방법

```
@Configuration
class Configuration {

    @Bean
    fun query(entityManager: EntityManager): JPAQueryFactory {
        return JPAQueryFactory(entityManager)
    }
}

...
abstract class SpringBootTestSupport {
    ...
    @Autowired
    protected lateinit var query: JPAQueryFactory
}
```

- JPAQueryFactory을 query으로 Bean 등록을 진행합니다. Test Scope에서만 사용하는 것을 원하는 경우에는 Test directory에 @TestConfiguration으로 지정해도 됩니다.
- query을 이용해서 테스트 코드 검증을 진행할 수 있습니다.

검증이 필요한데 ?... 해결방법

```kotlin
@Transactional
internal class JpaTestSupport : SpringBootTestSupport() {

    @Test
    internal fun `entityManager를 이용해서 dependency가 최소화 `() {
        // 특정 테스트를 하기 위해서 많은 디펜던시가 필요하다.
        save(Payment(BigDecimal.TEN))
        save(Member("username", 10, save(Team("team-ename"))))
        save(Coupon(BigDecimal.TEN))

        // 특정 서비스가 여러 entity rows를 변경할때 아래와 같은 조회로 Then 이어가야 합니다.
        // paymentRepository.findBy... epository 메서드는 없는데??...
        // memberRepository.findBy...
        // couponRepository.findBy...

        val payments = query.selectFrom(QPayment.payment)
            .where(QPayment.payment.amount.gt(BigDecimal.TEN))
            .fetch()

        val members = query.select(QMember.member.age)
            .from(QMember.member)
            .where(QMember.member.age.gt(20))
            .fetch()

        val coupons = query.selectFrom(QCoupon.coupon)
            .where(QCoupon.coupon.amount.eq(123.toBigDecimal()))
            .fetch()
    }
}
```

```kotlin
internal class PaymentServiceTest(
    private val paymentService: PaymentService
) : SpringBootTestSupport() {

    @Test
    internal fun `paymentZero test`() {
        //given
        val targetAmount = 105.toBigDecimal()
        saveAll((1..100).map {
            Payment(it.toBigDecimal().plus(BigDecimal.TEN).setScale(0))
        })

        //when
        paymentService.paymentZero(targetAmount)

        //then
        val count = query
            .selectFrom(qPayment)
            .where(qPayment.amount.gt(targetAmount))
            .fetchCount()

        then(count).isEqualTo(0)
    }
```

- 여러 엔티티의 여러 row의 수정을 진행했을 경우 해당 엔티티의 변경을 확인하기 위한 검증이 필요합니다.
- 이때 조회용 메서드를 단순히 테스트 코드에서만 사용하기 위해서 작성하기 위해서 작성하거나 Test Scope에 별도의 Repository를 구성해야 했지만 이제는 위에서 등록한 query을 이용해서 해당 비즈니스에 맞는 쿼리를 작성할 수 있습니다.

주의 할점

- SpringBootTestSupport 클래스에서 정의한 save(), saveAll() 메서드를 사용하는 경우 테스트에서 사용하는 트랜잭션과 완전하게 분리가 됩니다.

- 스프링 테스트에서는 @Transactional이 있는 경우 해당 테스트 메서드가 종료되면 트랜잭션으로 같이 Rollback 이 진행되게 됩니다. 하지만 SpringBootTestSupport에서 작성한 save(), saveAll()메서드 같은 경우에는 완전히 다른 트랜잭션이므로 given절에서 작성한 트랜잭션이 롤백 되지 않습니다.

- 트랜잭션 Rollbak을 원하시면 EnttiyManager를 주입받아 save() 하는 코드로 변경하면 됩니다.

- 테스트 코드의 중요성의 강조는 의미 없을 정도로 현업에서 자리 잡았다고 생각합니다. 이렇게 중요성이 있는 부분이니 테스트 코드를 작성하기 편한 방법도 많이 연구되었으면 좋겠다는 생각이 있습니다.

테스트 코드 작성에 대한 나름의 고찰

POJO 테스트는 언제나 옳다

- 이전에 포스팅한 Test 전략 가이드, Spring Boot 테스트 종류 등 스프링에서는 매우 다양한 테스트 방법을 제공해주고 있습니다. 이런 Slice 테스트 지원으로 테스트 하고자 하는 레이어를 부분적으로 빠르게 테스트할 수 있습니다.
- 이런 강력한 기능들 때문에 POJO에 대한 테스팅은 상대적으로 관심도가 낮다 보니 POJO에 대한 테스트 코드 작성에도 관심이 부족한 거 같습니다.
- POJO 테스트의 장점
  - POJO 객체를 테스트하는 것이라서 빠르다.
  - 디펜더시가 상대적으로 적어서 테스트 코드 작성이 편하다.
  - 단위 테스트하기 가장 이상적이다.

POJO 테스트는 언제나 옳다

```java
public class CouponTest {

    @Test(expected = CouponExpireException.class)
    public void 만료일이지난_쿠폰_사용_불가() {
        final Coupon coupon = CouponBuilder.build(LocalDate.now().minusDays(1));
        coupon.use();
    }

    @Test(expected = CouponAlreadyUseException.class)
    public void 이미_사용한_쿠폰은_쿠폰_사용_불가() {
        final Coupon coupon = CouponBuilder.build(LocalDate.now().plusDays(3));
        coupon.use();

        // 이미 사용한 쿠폰을 다시 한번 사용했을 다시 사용
        coupon.use();
    }

    @Test
    public void 사용가능_쿠폰_사용시_used_is_true() {
        final Coupon coupon = CouponBuilder.build(LocalDate.now().plusDays(1));
        coupon.use();
        assertThat(coupon.isUsed()).isTrue();
    }
}
```

- 해당 도메인을 설명해주는 Document 역할도 해준다

Matchers는 AssertJ가 좋다.

```
@Test
public void 다른_Matchers는_무슨함수가_있는지_알기_어렵다() {
    CouponCode actual, @NotNull Matcher<? super CouponCode> matcher
    String reason, T actual, @NotNull Matcher<? super T> matcher
    assertThat(coupon.getCode(), );
    coupon
    new
    null
}
}
```

```
@Test
public void 사용가능_쿠폰_사용시_used_is_true() {
    final Coupon coupon = CouponBuilder.build(LocalDate.now().plusDays(1));
    coupon.use();
    assertThat(coupon.isUsed()).isTrue();

    assertThat(coupon.isUsed()).;
}
```

isTrue()	capture of ?
is(Condition<? super Boolean> condition)	capture of ?
isFalse()	capture of ?
info	WritableAssertionInfo
isEqualTo(boolean expected)	capture of ?
isNotEqualTo(boolean other)	capture of ?
isNot(Condition<? super Boolean> condition)	capture of ?
toString()	String
as(Description description)	capture of ?
as(String description, Object... args)	capture of ?
asList	AbstractListAssert<?, List<?>, Object, ObjectAsser...

Did you know that Quick Documentation View (F1) works in completion lookups as well? >> π

- assertThat(coupon.isUsed()) 코드 뒤에 .을 붙여 사용하는 방식이라서 해당 함수가 무엇이 있는지 명확하게 알려줍니다.

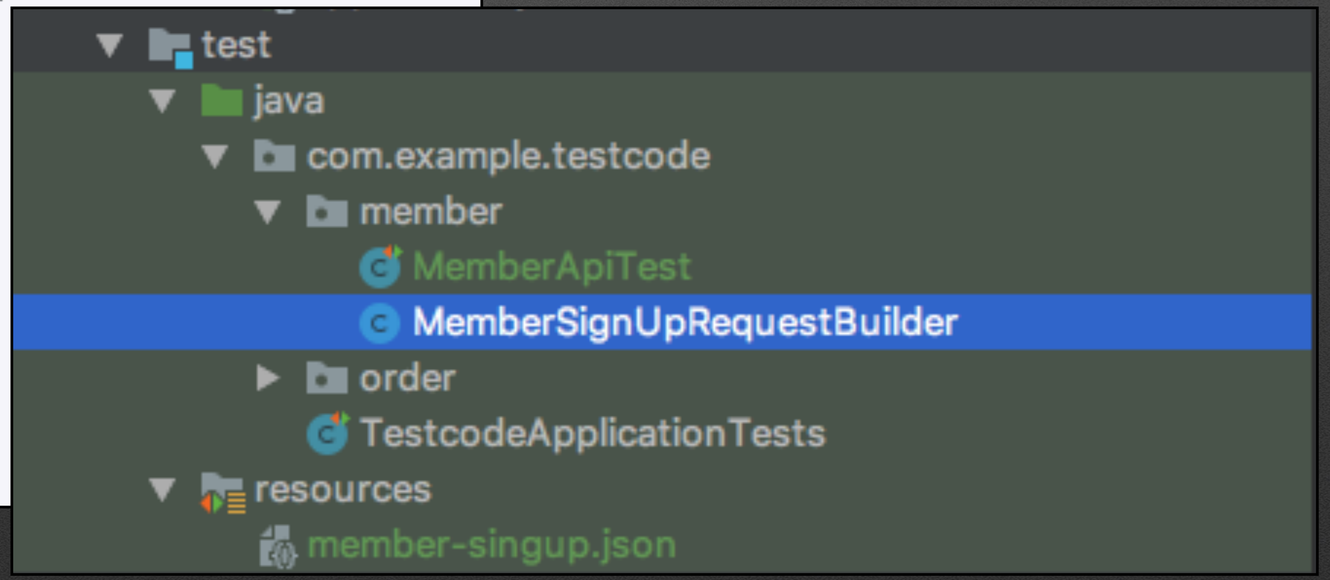
Controller 테스트의 어려움

```
{
  "orderer": {
    "name": "yun",
    "email": "yun@asd.com"
  },
  "product": [
    {
      "name": "양말",
      "price": 123233
    },
    {
      "name": "바자",
      "price": 1000
    },
    {
      "name": "치마",
      "price": 1120
    }
  ],
  "coupon": {
    "code": "xx2sd292kd"
  },
  "address": {
    "address1": "서울블라브라..."
    "address2": "어디 동...",
    "zip": "123-2332"
  }
}
```

- 특히 위처럼 json이 복잡한 경우에는 객체 기반으로 생성하는 것이 많이 번거롭습니다.
- 이런 경우에는 json으로 관리하는 것이 효율적일 수 있습니다.

Controller 테스트의 어려움

```java
...
public class MemberApiTest {

    ...

    @Test
  public void json_파일로테스트() throws Exception {
    //given
    final String requestBody = readJson("classpath:member-singup.json");

    //when
    final ResultActions resultActions = mvc.perform(post("/members")
        .contentType(MediaType.APPLICATION_JSON_UTF8)
        .content(requestBody))
        .andDo(print());

    //then
    resultActions
        .andExpect(status().isOk());

  }
}
```



통합 테스트의 어려움

통합 테스트에서 어려운 점들이 Given 작성하는 것입니다. 페이징 및 필터 관련된 API를 테스트한다고 했을 경우 데이터를 Set up 해주기가 어렵습니다.

만약 주문 생성 API를 테스트하기 위해서는

- 회원
- 상품
- 상품 카테고리
- 해당 상품의 입점사 정보
- 쿠폰 (만약 쿠폰을 사용했을 때 필요)

아무리 간단하게 생각해도 최소한 이 정도의 데이터를 Set up 해야 합니다. 실제 오픈 커머스 같은 경우에는 주문 한 번을 하기 위해서 수십 개의 테이블들을 Set up 해야 합니다. 테스트를 위해서 객체를 만들고 JPA로 데이터를 Set up 한다면 차라리 저는 테스트를 포기하겠습니다.

Import를 이용한 데이터 Set up

```sql
# data.sql
INSERT INTO
    `member` (`id`, `email`, `name`)
VALUES
        (1, 'yun@asd.com', 'yun'),
        (2, 'wan@asd.com', 'wan'),
        (3, 'jin@asd.com', 'jin'),
        (4, 'ck@asd.com', 'ck');
```

```java
@Test
public void member_page_test() throws Exception {
  //given

  //when
  final ResultActions resultActions = mvc.perform(get("/members")
      .contentType(MediaType.APPLICATION_JSON_UTF8))
      .andDo(print());

  //then
  resultActions
      .andExpect(status().isOk())
      .andExpect(jsonPath("content").exists())
      .andExpect(jsonPath("pageable").exists())
      .andExpect(jsonPath("pageable").exists())
      .andExpect(jsonPath("numberOfElements").value("4"));

}
```

- SQL로 given을 구성하게 되면 테스트하고자 하는 부분의 when, then 구절은 어렵지 않게 작성할 수 있습니다. 또 비즈니스 로직과 상관없이 데이터를 특정 시점으로 만들 수 있습니다.
- 하지만 단점도 있습니다. 칼럼의 변경 및 확장 시 테스트 코드의 지속적인 관리가 어렵다는 점이 있습니다. 또 누군가의 member insert를 추가적으로 하면 특정 테스트 코드는 실패하게 됩니다. 모든 작업자들이 data.sql을 추가 및 변경하다 보면 문제가 계속 생기게 됩니다.

Import를 이용한 데이터 Set up -> 해결법

```java
@Test
public void member_page_test() throws Exception {
  //given
  dataSetUp("classpath:member_page_test.sql"); // 로직은 구현하지 않았습니다.

  //when
  final ResultActions resultActions = mvc.perform(get("/members")
      .contentType(MediaType.APPLICATION_JSON_UTF8))
      .andDo(print());

  //then
  resultActions
      .andExpect(status().isOk())
      .andExpect(jsonPath("content").exists())
      .andExpect(jsonPath("pageable").exists())
      .andExpect(jsonPath("pageable").exists())
      .andExpect(jsonPath("numberOfElements").value("4"));

}
```

Junit5 With Kotlin & Spring Boot 2.2

@ValueSource

`@ValueSource` 어노테이션을 사용하면 배열을 값을 테스트 메서드로 손쉽게 전달할 수 있습니다.

```
@ParameterizedTest
@ValueSource(strings = ["", " "])
internal fun `isBlank `(value: String) {
    print("value: $value ") // value:  value:
    assertThat(value.isBlank()).isTrue()
}

@ParameterizedTest
@ValueSource(ints = [1, 2, 3, 4])
internal fun `ints values`(value: Int) {
    print("value: $value ") // value: 1 value: 2 value: 3 value: 4
}
```

EnumSource

`@EnumSource` 어노테이션을 통해서 Enum을 효율적으로 테스트 할 수 있습니다.

```
enum class Quarter(val value: Int, val description: String) {
    Q1(1, "1분기"),
    Q2(2, "2분기"),
    Q3(3, "3분기"),
    Q4(4, "4분기")
}
```

각 분기를 뜻하는 Enum을 위와 같이 정리했습니다.

```
@ParameterizedTest
@EnumSource(Quarter::class)
internal fun `분기의  value 값은 1 ~ 4 값이다`(quarter: Quarter) {
    println(quarter.name) // quarter: Q1 quarter: Q2 quarter: Q3 quarter: Q4
    assertThat(quarter.value in 1..4).isTrue()
}
```

enum에 정의된 모든 값들을 출력하는 것을 확인할 수 있습니다. `@EnumSource` 을 사용하면 모든 enum을 iterator 하기 편리합니다.

```
@ParameterizedTest
@EnumSource(value = Quarter::class, names = ["Q1", "Q2"])
internal fun `names을 통해서 특정 enum 값만 가져올 수 있다`(quarter: Quarter) {
    print("${quarter.name} ") // quarter: Q1 quarter: Q2
    assertThat(quarter.value in 1..2).isTrue()
}
```

특정 enum을 지정해서 가져오고 싶은 경우 `names = ["Q1", "Q2"]` 을 사용하면 됩니다.

@CsvSource

`@CsvSource` 어노테이션을 통해서 CSV 포멧으로 테스팅을 편리하게 진행 할 수 있습니다.

```
@ParameterizedTest
@CsvSource(
        "010-1234-1234,01012341234",
        "010-2333-2333,01023332333",
        "02-223-1232,022231232"
)
internal fun `전화번호는 '-'를 제거한다`(value: String, expected: String) {
    val valueReplace = value.replace("-", "")
    assertThat(valueReplace).isEqualTo(expected)
}
```

`,` 단위로 테스트 메서드의 매개변수로 값을 넘길 수 있습니다.

```kotlin
data class Amount(
        val price: Int,
        val ea: Int) {

    val totalPrice: Int
        get() = price * ea
}
```

```kotlin
@ParameterizedTest
@MethodSource("providerAmount")
internal fun `amount total price 테스트 `(amount: Amount, expectedTotalPrice: Int) {
    assertThat(amount.totalPrice).isEqualTo(expectedTotalPrice)
}

companion object {
    @JvmStatic
    fun providerAmount() = listOf(
            Arguments.of(Amount(1000, 2), 2000),
            Arguments.of(Amount(2000, 5), 10000),
            Arguments.of(Amount(4000, 5), 20000),
            Arguments.of(Amount(5000, 3), 15000)
    )
}
```

Junit5 With Kotlin & Spring Boot 2.2

Junit5 관련된 내용은 아니지만 Junit5와 Spring Boot 관련 테스트할때 좋은 패턴을 정리했습니다.

생성자 주입

`@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)` 어노테이션을 통해서 테스트 코드에서도 생성자 주입이 가능해 졌습니다.

```
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
@ActiveProfiles("test")
@DataJpaTest
internal class MemberRepositoryTest(val memberRepository: MemberRepository) {

    @Test
    internal fun `members 조회 테스트`() {
        //given
        val email = "asd@asd.com"
        val name = "name"

        //when
        val member = memberRepository.save(Member(email, name))

        //then
        assertThat(member.email).isEqualTo(email)
        assertThat(member.name).isEqualTo(name)
    }
}
```

Junit5 With Kotlin & Spring Boot 2.2

```kotlin
@SpringBootTest
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
@ActiveProfiles("test")
@AutoConfigureMockMvc
internal class MemberApiTest(
        val memberRepository: MemberRepository,
        val mockMvc: MockMvc
) {

    @Test
    internal fun `test`() {
        memberRepository.saveAll(listOf(
                Member("email1@asd.com", "jin"),
                Member("email2@asd.com", "yun"),
                Member("email3@asd.com", "wan"),
                Member("email4@asd.com", "kong"),
                Member("email5@asd.com", "joo")
        ))

        mockMvc.get("/members") {
            accept = MediaType.APPLICATION_JSON
        }.andExpect {
            content { contentType(MediaType.APPLICATION_JSON) }
            jsonPath("$[0].name") { value("jin") }
            jsonPath("$[1].name") { value("yun") }
            jsonPath("$[2].name") { value("wan") }
            jsonPath("$[3].name") { value("kong") }
            jsonPath("$[4].name") { value("joo") }
        }.andDo {
            print()
        }
    }
}
```

Junit5 With Kotlin & Spring Boot 2.2

@Sql 손쉽게 Data Set up

`*.sql` 파일로 손쉽게 데이터를 Set up할 수 있습니다. JPA를 사용중이라면 Given절로 JPA를 만들기가 어렵고 불편한 경우 사용하면 좋을거 같습니다.

```sql
# member-data-setup.sql
insert into member (`email`, `name`, `created_at`, `updated_at`)
values
('sample1@asd.com', 'name', now(), now()),
('sample2@asd.com', 'name', now(), now()),
('sample3@asd.com', 'name', now(), now()),
('sample4@asd.com', 'name', now(), now()),
('sample5@asd.com', 'name', now(), now()),
('sample6@asd.com', 'name', now(), now()),
('sample15@asd.com', 'name', now(), now());
```

```
└── test
    ├── kotlin
    │   └── com
    └── resources
        └── member-data-setup.sql
```

위에서 작성한 `*.sql` 파일을 `test/resources` 디렉토리에 위치시킵니다

```kotlin
@Test
@Sql("/member-data-setup.sql")
internal fun name() {
    val members = memberRepository.findAll()

    then(members).anySatisfy {
        then(it.name).isEqualTo("name")
        then(it.email).contains("@")
                .startsWith("sample")
                .endsWith("com")
    }
}
```

Junit5 With Kotlin & Spring Boot 2.2

```kotlin
@Test
internal fun `member save test`() {
    //given
    val email = "asd@asd.com"
    val name = "name"

    //when
    val member = memberRepository.save(Member(email, name))

    //then

    // 기존 사용법 assertThat
    assertThat(member.email).isEqualTo(email)
    assertThat(member.name).isEqualTo(name)
    assertThat(member.createdAt).isBeforeOrEqualTo(LocalDateTime.now())
    assertThat(member.updatedAt).isBeforeOrEqualTo(LocalDateTime.now())

    // BDD 사용법
    then(member.email).isEqualTo(email)
    then(member.name).isEqualTo(name)
    then(member.createdAt).isBeforeOrEqualTo(LocalDateTime.now())
    then(member.updatedAt).isBeforeOrEqualTo(LocalDateTime.now())
}
```

* AssertJ에서는 BDD 스타일의 BDDAssertion을 제공해주고 있습니다.

```kotlin
@Test
internal fun `문장 검사`() {
    then("AssertJ is best matcher").isNotNull()
            .startsWith("Assert
            .contains(" ")
            .endsWith("matcher"
}
```

```kotlin
@Test
internal fun `findByName test`() {
    //given
    memberRepository.saveAll(listOf(
            Member("email1@asd.com", "kim"),
            Member("email2@asd.com", "kim"),
            Member("email3@asd.com", "kim"),
            Member("email4@asd.com", "name"),
            Member("email5@asd.com", "name")
    ))

    //when
    val members = memberRepository.findByName("kim")

    //then
    then(members).anySatisfy {
        then(it.name).isEqualTo("kim")
    }
}
```

```kotlin
class Member private constructor() {

    fun validateBeforeSave() {
        fun validate(value: String, fieldName: String) {
            if (value.isEmpty()) {
                throw IllegalArgumentException("$id empty $fieldName")
            }
        }
        validate(email, "email")
        validate(name, "name")
    }
}
```

```kotlin
@Test
internal fun `member 실패`() {
    val member = Member("", "")
    thenThrownBy { member.validateBeforeSave() }
            .isExactlyInstanceOf(IllegalArgumentException::class.java)
            .hasMessageContaining("empty")
}
```

6. Exception 처리 방법

Exception 처리를 왜 해야할까요?

> 클린 코드 : 오류 코드 보다 예외를 사용하라 리팩토링

```java
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // 디바이스 상태를 점검한다.
        if (handle != DeviceHandle.INVALID) {
            // 레코드 필드에 디바이스 상태를 저장한다.
            retrieveDeviceRecord(handle);
            // 디바이스가 일시정지 상태가 아니라면 종료한다.
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log("Device suspended. Unable to shut down");
            }
        } else {
            logger.log("Invalid handle for: " + DEV1.toString());
        }
    }
    ...
}
```

- 코드가 복잡하다
- 코드를 작성하다 보면 무슨 오류를 처리 했는지 기억도 안난다.
- 나중에 내가 봐도 모른다.

```java
public class DeviceController {
    ...
    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }

    private void tryToShutDown() throws DeviceShutDownError {
        DeviceHandle handle = getHandle(DEV1);
        DeviceRecord record = retrieveDeviceRecord(handle);
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
    }

    private DeviceHandle getHandle(DeviceID id) {
        ...
        throw new DeviceShutDownError("Invalid handle for: " + id.toString());
        ...
    }
    ...
}
```

- 논리 코드와 오류 처리 코드가 분리되있어 가독성이 좋다.
- 정확히 무슨 예외를 발생시키고 있는지 명확하다.

우리가 현실에서 만나는 Exception 코드

먹고 죽는 놈

```
try {
    // 비지니스 로직 수행...
}catch (Exception e){
    e.printStackTrace();
}
```

- 잡아 놓고 무슨 처리를 하지도 않는다.
- 예외가 발생 하면 그냥 로그만 찍고 넘어가 게된다.
- 로그 라도 남기면 양반이다. 로그도 없으면 그냥 예외가 발생하는지도 잘 모른다.
- 예외 처리하지 않는다면 예외를 잡지도 마라

먹고 죽는 놈 : 개선

```
try {
    // 비지니스 로직 수행...
}catch (Exception e){
    e.printStackTrace();
    throw new XXX비지니스로직예외(e);
}
```

- 일단 로그 찍는다.
- 예외를 다시 발생 시킨다.
- 다시 발생 시키는 경우 더 구체적인 예외를 만든다
- 구체적인 예외를 발생시킬 때 되도록 UnCheckedException을 발생 시킨다.

예외 처리 유형

- 먹고 죽고 로그 안찍는다. (극악무도)
- 먹고 죽고 로그라도 찍는다. (노양심)
- Catch 이후 아무 예외나 던진다. (여기부터 인간계)
- Catch 이후 적절한 예외를 던져 예외의 정보를 잘 전달한다. (믿을 수 있는 전우형)

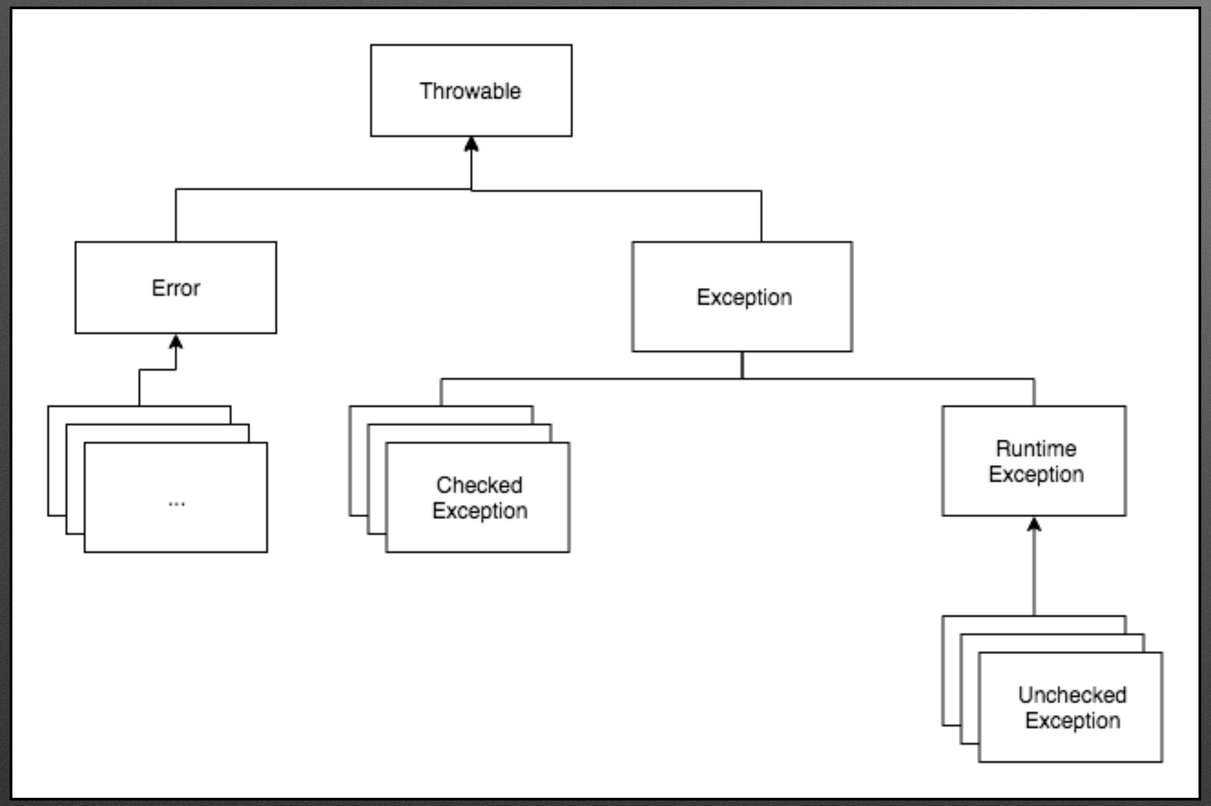
Try Catch 전략

```
try {
    // 비지니스 로직 수행...
}catch (Exception e){
    e.printStackTrace();
    throw new XXX비지니스로직예외(e);
}
```

`try catch` 를 사용해야 하는 경우라면 더 구체적인 예외로 Exception을 발생시키는 것이 좋습니다. 간단하게 정리하면

1. `try catch` 를 최대한 지양해라
2. `try catch` 로 에러를 먹고 주는 코드는 지양해라(이런 코드가 있다면 로그라도 추가해주세요...)
3. `try catch` 를 사용하게 된다면 더 구체적인 Exception을 발생시키는 것이 좋다.

Check Exception VS UnChecked Exception



- Error는 시스템이 비정상적인 상황에서 발생하다.
- 이 수준의 Error는 처리 노답...
- Checked, Unchecked Exception는 처리가 가능
- Checked = 반드시 처리해야하는 거. (상위로 던지든, 내가 처리 하든) -> 모든 비극의 시작
- Unchecked = 반드시 처리하지는 않아도 됨
- Runtime Exception을 상속 기준으로 나뉨

Check Exception VS UnChecked Exception

.	Checked Exception	Unchecked Exception
처리 여부	반드시 예외 처리 해야함	예외 처리 하지 않아도됨
트랜잭션 Rollback 여부	Rollback 안됨	Rollback 진행
대표 Exception	IOException, SQLException	NullPointerException, IllegalArgumentException

- Error는 시스템이 비정상적인 상황에서 발생하다.
- 이 수준의 Error는 처리 노답...
- Checked, Unchecked Exception는 처리가 가능
- Checked = 반드시 처리해야하는 거. (상위로 던지든, 내가 처리 하든) -> 모든 비극의 시작
- Unchecked = 반드시 처리하지는 않아도 됨

Check Exception

```java
@Test
public void throws_던지기() throws JsonProcessingException {
  final ObjectMapper objectMapper = new ObjectMapper();
  final Member member = new Member("yun");
  final String valueAsString = objectMapper.writeValueAsString(member);

}

@Test
public void try_catch_감싸기() {
  final ObjectMapper objectMapper = new ObjectMapper();
  final Member member = new Member("yun");
  final String valueAsString;
  try {
    valueAsString = objectMapper.writeValueAsString(member);
  } catch (JsonProcessingException e) {
    e.printStackTrace();
    throw new RuntimeException();
  }
}
```

- 위 JsonProcessingException는 IOException Exception을 상속하는 Checked Exception이다.
- throws로 상위 메서드로 넘기든 자신이 try catch 해서 throw를 던지든 해야 한다.
- 이것은 문법적인 강제 선택이다. 그에 반해 Unchecked Exception은 명시적인 예외 처리를 하지 않아도 된다.

Rollback 여부

```java
@Service
@RequiredArgsConstructor
@Transactional
public class MemberService {

  private final MemberRepository memberRepository;

  // (1) RuntimeException 예외 발생
  public Member createUncheckedException() {
    final Member member = memberRepository.save(new Member("yun"));
    if (true) {
      throw new RuntimeException();
    }
    return member;
  }

  // (2) IOException 예외 발생
  public Member createCheckedException() throws IOException {
    final Member member = memberRepository.save(new Member("wan"));
    if (true) {
      throw new IOException();
    }
    return member;
  }
}
```

Rollback 여부

```
--- Ynu Log
Hibernate:
    /* insert yun.blog.exception.member.Member
        */ insert
        into
            member
            (id, name)
        values
            (null, ?)
2019-05-16 00:55:16.117 TRACE 49422 --- [nio-8080-exec-2] o.h.type.descriptor.sql.BasicBinder
2019-05-16 00:55:16.120 ERROR 49422 --- [nio-8080-exec-2] o.a.c.c.C.[.[.[/].[dispatcherServlet]

java.lang.RuntimeException: null

--- Wan Log
Hibernate:
    /* insert yun.blog.exception.member.Member
        */ insert
        into
            member
            (id, name)
        values
            (null, ?)
2019-05-16 00:55:43.931 TRACE 49422 --- [nio-8080-exec-4] o.h.type.descriptor.sql.BasicBinder
2019-05-16 00:55:43.935 ERROR 49422 --- [nio-8080-exec-4] o.a.c.c.C.[.[.[/].[dispatcherServlet]

java.io.IOException: null
        at yun.blog.exception.member.MemberService.createCheckedException(MemberService.java:27)
```

- (1) RuntimeException 예외 발생 발생 시키면 **yun이라는 member는 rollback이 진행됩니다.**
- (2) IOException 예외 발생이 되더라도 **wan은 rollback이 되지 않고 트랜잭션이 commit까지 완료됩니다.**
- 로그 메시지를 보면 member yun, wan 모두 insert 쿼리는 보이지만 yun rollback이 진행되고 wan은 rollback이 되지 않고 commit까지 됩니다.

왜 Checked Exception은 Rollback되지 않는 것일까?

왜 Checked Exception은 Rollback되지 않는 것일까?

```
public void sendFile(String fileName){

    File file;
    try {
        file = FileFindService.find(fileName);
    } catch (FileNotFoundException e){ // FileNotFou
        // 파일을 못찾았으니 기본 파일을 찾아서 전송 한다
        file = FileFindService.find("default.png");
    }

    send(file);
}
```

- 기본적으로 Checked Exception는 복구가 가능하다는 메커니즘을 가지고 있다.
- 이미지 파일을 찾아서 전송해주는 함수에서 이미지를 찾지 못했을 경우 기본 이미지를 전송한다. 복구 전략을 가질 수 있게 된다.
- 기본적으로 복구가 가능하니 네가 복구를 작업을 진행했을 수 있으니까 Rollback은 진행하지 않을게라는 의미가 있다고 생각합니다 (뇌피셜 100%)

왜 Checked Exception은 Rollback되지 않는 것일까?

```
public void sendFile(String fileName){
    if(FileFindService.existed(filename)){
        // 파일이 있는 경우 해당 파일을 찾아서 전송
        send(FileFindService.find(fileName));
    }else{
        // 파일이 있는 없는 경우 기본 이미지 전송
        send(FileFindService.find("default.png"));
    }
}
```

- 이런 식의 예외는 복구하는 것이 아니라 **일반적인 코드의 흐름**으로 제어해야 한다.

하지만 우리 현실은...

하지만 현실은 (1)...

- 하지만 우리가 일반적으로 Checked Exception 예외가 발생했을 경우 복구 전략을 갖고 그것을 **복구할 수 있는 경우는 그렇게 많지 않습니다.**

- 유니크해야 하는 이메일 값이 중복돼서 SQLException이 발생하는 경우 어떻게 복구 전략을 가질 수 있을까요? 유저가 입력했던 이메일 + 난수를 입력해서 insert 시키면 가능은 하겠지만 현실에서는 그냥 **RuntimeException을 발생시키고 입력을 다시 유도하는 것이 현실적입니다.**

하지만 현실은 (1)...

- 여기서 중요한 것은 해당 **Exception을 발생시킬 때 명확하**게 어떤 예외가 발생해서 Exception이 발생했는지 정보를 전달해주는 것입니다. 위 같은 경우에는 DuplicateEmailException (Unchecked Exception)을 발생 시는 것이 바람직합니다.
- Checked Exception을 만나면 더 구체적인 Unchecked Exception을 발생시켜 정확한 정보를 전달하고 로직의 흐름을 끊어야 합니다. **우리는 JPA에 구현체를 가져다 사용하더라도 Checked Exception을 직접 처리하지 않고** 있는 이유도 다 적절한 RuntimeException으로 예외를 던져주고 있기 때문입니다.

Checked Exception 처리 전략

```java
public class ObjectMapperUtil {

  private final ObjectMapper objectMapper = new ObjectMapper();

  // 예외처리를 throws를 통해서 위임하고 있습니다.
  public String writeValueAsString(Object object) th
    return objectMapper.writeValueAsString(object);
  }

  // 예외처리를 throws를 통해서 위임하고 있습니다.
  public <T> T readValue(String json, Class<T> clazz
    return objectMapper.readValue(json, clazz);
  }
}
```

```java
@Test
public void writeValueAsString_test() {
  final Member member = new Member( name: "yun");
  objectMapperUtil.writeValueAsString(member);
}

@Test
public void readValue_test() {
  final String json = "{\"name\": \"yun\"}";
  final Member member = objectMapperUtil.readValue(json, Member.class);
}
}
```

- 해당 메서드의 테스트 코드입니다. 예외 처리를 상위로 던져버리기 때문에 메서드를 상용하는 곳에서 다시 throw를 하던지 예외를 try catch 하든지 해야 합니다.
- 이렇게 무의하고 반복적인 예외를 던지는 것은 좋지 않습니다.

더 구체적인 Unckecked Exception 발생 시켜라

```
public String writeValueAsString(Object object) {
    try {
      return objectMapper.writeValueAsString(object);
    } catch (JsonProcessingException e) {
      throw new JsonSerializeFailed(e.getMessage());
    }
  }

  public <T> T readValue(String json, Class<T> clazz) {
    try {
      return objectMapper.readValue(json, clazz);
    } catch (IOException e) {
      throw new JsonDeserializeFailed(e.getMessage());
    }
  }
```

- Checked Exception을 Unckecekd Exception으로 던지고 있기 때문에 메서드를 사용하는 곳에서는 아무런 예외처리를 진행하지 않아도 됩니다.
- 저것보다 더 구체적으로 작성하는 편이 좋습니다.

API Server에서 Exception 처리 방법

통일된 Error Response

```
{
  "message": " Invalid Input Value",
  "status": 400,
  // "errors":[], 비어있을 경우 null 이 아닌 빈 배열을 응답한다.
  "errors": [
    {
      "field": "name.last",
      "value": "",
      "reason": "must not be empty"
    },
    {
      "field": "name.first",
      "value": "",
      "reason": "must not be empty"
    }
  ],
  "code": "C001"
}
```

- Error Response 객체는 항상 동일한 Error Response를 가져야 합니다.
- 그렇지 않으면 클라이언트에서 예외 처리를 항상 동일한 로직으로 처리하기 어렵습니다.
- Map<Key, Value> 형식은 좋지 않습니다. 런타입시에 정확한 형태를 갖추기 때문에 객체를 처리하는 개발자들도 정확히 무슨 키에 무슨 데이터가 있는지 확인하기 어렵습니다.

통일된 Error Response

```
{
  "message": " Invalid Input Value",
  "status": 400,
  // "errors":[], 비어있을 경우 null 이 아닌 빈 배열을 응답한다.
  "errors": [
    {
      "field": "name.last",
      "value": "",
      "reason": "must not be empty"
    },
    {
      "field": "name.first",
      "value": "",
      "reason": "must not be empty"
    }
  ],
  "code": "C001"
}
```

- message : 에러에 대한 message를 작성합니다.
- status : http status code를 작성합니다. header 정보에도 포함된 정보이니 굳이 추가하지 않아도 됩니다.
- errors : 요청 값에 대한 field, value, reason 작성합니다. 일반적으로 @Valid 어노테이션으로 JSR 303: Bean Validation에 대한 검증을 진행 합니다.
- 만약 errors에 바인인된 결과가 없을 경우 null이 아니라 빈 배열 []을 응답해줍니다. null 객체는 절대 리턴하지 않습니다. null이 의미하는 것이 애매합니다.
- code : 에러에 할당되는 유니크한 코드값입니다

Error Response 객체

```java
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class ErrorResponse {

    private String message;
    private int status;
    private List<FieldError> errors;
    private String code;
    ...

    @Getter
    @NoArgsConstructor(access = AccessLevel.PROTECTED)
    public static class FieldError {
        private String field;
        private String value;
        private String reason;
        ...
    }
}
```

- ErrorResponse 객체 입니다. POJO 객체로 관리하면 errorResponse.getXXX(); 명확하게 객체에 있는 값을 가져올 수 있습니다.
- 특정 Exception에 대해서 ErrorResponse 객체를 어떻게 만들 것인가에 대한 책임을 명확하게 갖는 구조로 설계할 수 있습니다. 세부적인 것은 코드를 확인해주세요.

@ControllerAdvice로 모든 예외를 핸들링

```
@ControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    /**
     *  javax.validation.Valid or @Validated 으로 binding error 발생시 발생한다.
     *  HttpMessageConverter 에서 등록한 HttpMessageConverter binding 못할경우 발생
     *  주로 @RequestBody, @RequestPart 어노테이션에서 발생
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    protected ResponseEntity<ErrorResponse> handleMethodArgumentNotValidException(
        log.error("handleMethodArgumentNotValidException", e);
        final ErrorResponse response = ErrorResponse.of(ErrorCode.INVALID_INPUT_VA
        return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
    }
```

- 추가로 스프링 및 라이브러리 등 자체적으로 발생하는 예외는 @ExceptionHandler 으로 추가해서 적절한 Error Response를 만듭니다.
- 비지니스 요구사항에 예외일 경우 BusinessException 으로 통일성 있게 처리하는 것을 목표로 한다. 추가로 늘어날 수는 있겠지만 그 개수를 최소한으로 하는 노력이 필요합니다.

Error Code 정의

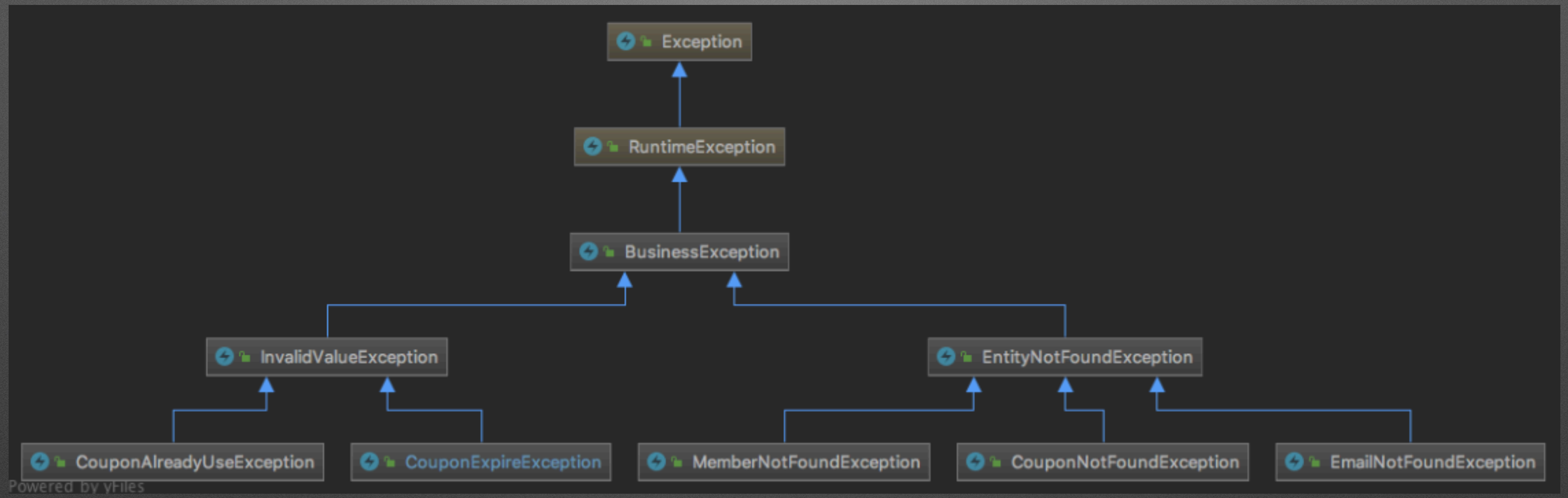
```java
public enum ErrorCode {

    // Common
    INVALID_INPUT_VALUE(400, "C001", " Invalid Input Value"),
    METHOD_NOT_ALLOWED(405, "C002", " Invalid Input Value"),
    ....
    HANDLE_ACCESS_DENIED(403, "C006", "Access is Denied"),

    // Member
    EMAIL_DUPLICATION(400, "M001", "Email is Duplication"),
    LOGIN_INPUT_INVALID(400, "M002", "Login input is invalid"),

    ;
    private final String code;
    private final String message;
    private int status;

    ErrorCode(final int status, final String code, final String message) {
        this.status = status;
        this.message = message;
        this.code = code;
    }
}
```

- 에러 코드는 **enum 타입으로 한 곳에서 관리**합니다.
- 에러 코드가 전체적으로 흩어져있을 경우 코드, **메시지의 중복**을 방지하기 어렵고 전체적으로 관리하는 것이 매우 어렵습니다.
- C001 같은 코드도 동일하게 enum으로 관리 하는 것도 좋습니다. 에러 메시지는 Common과 각 도메인별로 관리하는 것이 효율적일 거 같습니다.

Business Exception 처리

- 여기서 말하는 **Business Exception은 요구사항에 맞지 않을 경우 발생시키는 Exception**을 말합니다. 만약 쿠폰을 사용 하려고 하는데 이미 사용한 쿠폰인 경우에는 더 이상 정상적인 흐름을 이어갈수가 없게 됩니다. 이런 경우에는 적절한 Exception을 발생시키고 로직을 종료 시켜야합니다.
- 더 쉽게 정리하면 **요구사항에 맞게 개발자가 직접 Exception을 발생시키는 것들이 Business Exception** 이라고 할수 있습니다.
- **유지 보수하기 좋은 코드를 만들기 위해서는 Exception을 발생**시켜야 합니다. 쿠폰을 입력해서 상품을 주문했을 경우 상품 계산 로직에서 이미 사용해 버린 쿠폰이면 로직을 이어나가기는 어렵습니다.
- 단순히 어려운 것이 아니라 해당 **계산 로직의 책임이 증가**하게 됩니다. 계산 로직은 특정 공식에 의해서 제품의 가격을 계산하는 것이 책임이지 쿠폰이 이미 사용 해 경우, 쿠폰이 만료되었을 경우, 제품이 매진 됐을 경우 등등의 책임을 갖게 되는 순간 유지 보수하기 어려운 코드가 됩니다. **객체의 적절한 책임을 주기 위해서라도 본인이 처리 못 하는 상황일 경우 적절한 Exception을 발생**시켜야 합니다.

비지니스 예외를 위한 최상위 BusinessException 클래스



- 최상위 BusinessException을 상속 받는 InvalidValueException, EntityNotFoundExceptuon 등이 있습니다.
- InvalidValueException : 유효하지 않은 값일 경우 예외를 던지는 Excetion
- 쿠폰 만료, 이미 사용한 쿠폰 등의 이유로 더이상 진행이 못할경우
- EntityNotFoundException : 각 엔티티들을 못찾았을 경우
- findById, findByCode 메서드에서 조회가 안되었을 경우

비지니스 예외를 위한 최상위 BusinessException 클래스

```
@ExceptionHandler(BusinessException.class)
protected ResponseEntity<ErrorResponse> handleBusinessException(final BusinessException e) {
    log.error("handleEntityNotFoundException", e);
    final ErrorCode errorCode = e.getErrorCode();
    final ErrorResponse response = ErrorResponse.of(errorCode);
    return new ResponseEntity<>(response, HttpStatus.valueOf(errorCode.getStatus()));
}
```

- 최상위 BusinessException을 기준으로 예외를 발생시키면 통일감 있는 예외 처리를 가질 수 있습니다. 비니지스 로직을 수행하는 코드 흐름에서 로직의 흐름을 진행할 수 없는 상태인 경우에는 적절한 BusinessException 중에 하나를 예외를 발생 시키거나 직접 정의하게 됩니다.
- 이렇게 발생하는 모든 예외는 handleBusinessException 에서 동일하게 핸들링 됩니다. 예외 발생시 알람을 받는 등의 추가적인 행위도 손쉽게 가능합니다. 또 BusinessException 클래스의 하위 클래스 중에서 특정 예외에 대해서 다른 알람을 받는 등의 더 디테일한 핸들링도 가능해집니다.

컨트롤러 예외 처리

```java
@RestController
@RequestMapping("/members")
public class MemberApi {

    private final MemberSignUpService memberSignUpService;

    @PostMapping
    public MemberResponse create(@RequestBody @Valid final SignUpRequest dto) {
        final Member member = memberSignUpService.doSignUp(dto);
        return new MemberResponse(member);
    }
}
```

- 컨틀롤러에서 모든 요청에 대한 값 검증을 진행하고 이상이 없을 시에 서비스 레이어를 호출해야 합니다. 위에서도 언급했듯이 잘못된 값이 있으면 서비스 레이어에서 정상적인 작업을 진행하기 어렵습니다. 무엇보다 컨틀롤러의 책임을 다하고 있지 않으면 그 책임은 자연스럽게 다른 레이어로 전해지게 되며 이렇게 넘겨받은 책임을 처리하는데 큰 비용과 유지보수 하기 어려워질 수밖에 없습니다.
- 컨트롤러의 중요한 책임 중의 하나는 요청에 대한 값 검증이 있습니다. 스프링은 Bean Validation 검증을 쉽고 일관성 있게 처리할 수 있도록 도와줍니다. 모든 예외는 @ControllerAdvice 선언된 객체에서 핸들링 됩니다. 컨트롤러로 본인이 직접 예외까지 처리하지 않고 예외가 발생하면 그냥 던져버리는 패턴으로 일관성 있게 개발할 수 있습니다.

Custom Validation 어노테이션 만들기 (1)

```
@Documented
@Constraint(validatedBy = EmailDuplicationValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface EmailUnique {

  String message() default "Email is Duplication";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};
}
```

- 스프링은 컨트롤러에서 클라이언트에서 넘겨받은 값에 대한 검증을 Bean Validation 기반으로 쉽고 강력하게 할 수 있습니다
- 또 한 커스텀 한 어노테이션을 확장도 쉽게 구현할 수 있습니다.
- 커스텀한 이메일이 유니크한지 검증하는 어노테이션을 만들어 사용하면 반복적인 코드를 쉽게 줄일 수 있습니다.

Custom Validation 어노테이션 만들기 (1)

```java
@Component
@RequiredArgsConstructor
public class EmailDuplicationValidator implements ConstraintValidator<EmailUnique, String> {

  private final MemberRepository memberRepository;

  @Override
  public void initialize(EmailUnique emailUnique) {

  }

  @Override
  public boolean isValid(String email, ConstraintValidatorContext cxt) {

    boolean isExistEmail = memberRepository.existsByEmail(email);

    if (isExistEmail) {
      cxt.disableDefaultConstraintViolation();
      cxt.buildConstraintViolationWithTemplate(
          MessageFormat.format("Email {0} already exists!", email))
          .addConstraintViolation();
    }
    return !isExistEmail;
  }
}
```

```java
public class SignUpRequest {
    @EmailUnique @Email
    private String email;
}

public class MemberApi {

  private final MemberRepository memberRepository;

  @PostMapping
  public Member create(@RequestBody @Valid final SignUpRequest dto) {

    return memberRepository.save(Member.builder()
        .email(dto.getEmail())
        .build());
  }

}
```

Custom Validation 어노테이션 만들기 (1)

```java
public class MemberApiTest {

  @Test
  public void signUp_test_이메일이_중복된_경우() throws Exception {
    //given
    final SignUpRequest dto = new SignUpRequest("yun@test.com");

    //when
    final ResultActions resultActions = requestSignUp(dto);

    //then
    resultActions
        .andExpect(status().isBadRequest());
  }

  private ResultActions requestSignUp(SignUpRequest dto) throws E
    return mockMvc.perform(post("/members")
        .contentType(MediaType.APPLICATION_JSON_UTF8)
        .content(objectMapper.writeValueAsString(dto)))
        .andDo(print());
  }

}
```


```json
{
    "timestamp": "2019-03-19T17:11:26.919+0000",
    "status": 400,
    "error": "Bad Request",
    "errors": [
        {
            "codes": [
                "EmailUnique.signUpRequest.email",
                "EmailUnique.email",
                "EmailUnique.java.lang.String",
                "EmailUnique"
            ],
            "arguments": [
                {
                    "codes": [
                        "signUpRequest.email",
                        "email"
                    ],
                    "arguments": null,
                    "defaultMessage": "email",
                    "code": "email"
                }
            ],
            "defaultMessage": "Email yun@test.com already exists!",
            "objectName": "signUpRequest",
            "field": "email",
            "rejectedValue": "yun@test.com",
            "bindingFailure": false,
            "code": "EmailUnique"
        }
    ],
    "message": "Validation failed for object='signUpRequest'. Error count: 1",
    "path": "/members"
}
```

Custom Validation 어노테이션 만들기 (2)

카드 결제

```
{
  "price": 100.00,
  "payment": {
    "paymentMethod": "CARD",
    "card": {
      "number": "25523-22394",
      "brand": "323",
      "csv" : "삼성카드"
    }
  },
  "address": {
    "city": "NOWON-GU",
    "state": "SEOUL",
    "zipCode": "09876?"
  }
}
```

무통장 결제

```
{
  "price": 100.00,
  "payment": {
    "paymentMethod": "BANK_TRANSFER",
    "account": {
      "number": "110-202034-2234",
      "bankCode": "2003",
      "holder" : "홍길동"
    }
  },
  "address": {
    "city": "NOWON-GU",
    "state": "SEOUL",
    "zipCode": "09876?"
  }
}
```

- 스프링에서는 JSR 303 기반 어노테이션 기반으로 일관성 있는 Validation을 진행할 수 있습니다. 하지만 @NotNull, @NotEmpty, @Email과 같은 검증은 가능하지만 비즈니스적에 맞는 Validation은 별도로 진행해야 합니다.
- 예를 들어 주문에 대한 결제 정보를 받는 경우 카드 결제 시에는 카드 정보, 무통장 결제에는 계좌 정보를 입력받아야 합니다.
- 위 JSON 같이 account, card 값은 paymentMethod에 따라 선택적으로 필수 값이 변경되니 단순하게 @NotNull, @NotEmpty 어노테이션으로 검증하기가 어렵습니다. 이러한 경우 ConstraintValidator을 직접 구현해야 합니다.

ConstraintValidator의 장점

```
@PostMapping
    public OrderSheetRequest order(@RequestBody @Valid final OrderSheetRequest dto) {

        if (dto.getPayment().getPaymentMethod() == PaymentMethod.BANK_TRANSFER)) {
            // 계좌정보가 제대로 넘어 왔는지 검증
        }

        if((dto.getPayment().getPaymentMethod() == PaymentMethod.CARD)){
            // 카드 정보 제대로 넘어 왔는지 검증
        }

        return dto;
    }
```

- ConstraintValidator을 직접 구현하지 않고도 다양한 방법으로 해당 Validation을 진행할 수 있지만 ConstraintValidator 기반으로 검증하는게 통일성 있게 관리할 수 있는 장점이 있습니다.
- 검증을 진행하는 방법과 시점이 매우 다양하다 보니 많은 개발자들이 만들어가는 프로젝트인 경우에 전체적인 코드의 통일성이 떨어지게 됩니다. Validation을 사용하면 검증 방법과, 검증 시점(어느 레이어에서 진행되는지)을 통일성 있게 가져갈 수 있습니다.

ConstraintValidator의 장점

```
@PostMapping
    public OrderSheetRequest order(@RequestBody @Valid final OrderSheetRequest dto) {

        if (dto.getPayment().getPaymentMethod() == PaymentMethod.BANK_TRANSFER)) {
            // 계좌정보가 제대로 넘어 왔는지 검증
        }

        if((dto.getPayment().getPaymentMethod() == PaymentMethod.CARD)){
            // 카드 정보 제대로 넘어 왔는지 검증
        }

        return dto;
    }
```

- 다른 컨트롤러에서 위와 같은 중복 코드가 발생할 수 있고 이러한 중복 코드로 코드의 응집력이 떨어지게 되면 변경이 발생했을 때 해당 코드를 찾아가서 직접 수정해야 합니다.
- 이러한 컨트롤러에서의 중복 코드를 제거하기 위해서는 컨트롤러 진입 직전 Interceptor에서 검증하는 것이 바람직합니다. ConstraintValidator 기반으로 유효성 검증을 진행하면 검증 시점의(Interceptor에서 진행) 통일성을 갖습니다.

일관성 있는 ErrorResponse

```
POST http://127.0.0.1:8080/order
Content-Type: application/json

{
  "price": 100.00,
  "payment": {
    "paymentMethod": "BANK_TRANSFER",
    "account": {
      "holder": "",
      "number": "",
      "bankCode": ""
    }
  },
  "address": {
    "city": "NOWON-GU",
    "state": "SEOUL",
    "zipCode": "09876?"
  }
}
```

- 스프링 가이드 - 통일된 Error Response 객체에서 이야기했듯이 ErrorReponse는 항상 동일한 포맷으로 응답해야 합니다.
- ConstraintValidator을 사용해서 @Valid을 기반으로 검증을 하면 MethodArgumentNotValidException이 발생하고 이 예외를 @ControllerAdvice로 모든 예외를 핸들링에서 정리한 것처럼 통일성 있는 ErrorResponse 포맷으로 처리할 수 있습니다.

일관성 있는 ErrorResponse

```
POST http://127.0.0.1:8080/order
Content-Type: application/json

{
  "price": 100.00,
  "payment": {
    "paymentMethod": "BANK_TRANSFER",
    "account": {
      "holder": "",
      "number": "",
      "bankCode": ""
    }
  },
  "address": {
    "city": "NOWON-GU",
    "state": "SEOUL",
    "zipCode": "09876?"
  }
}
```

```
{
  "message": "입력값이 올바르지 않습니다.",
  "status": 400,
  "errors": [
    {
      "field": "payment.account.bankCode",
      "value": "",
      "reason": "은행코드는 필수입니다."
    },
    {
      "field": "payment.account.number",
      "value": "",
      "reason": "계좌번호는 필수값입니다."
    },
    {
      "field": "payment.account.holder",
      "value": "",
      "reason": "계좌주는 값은 필수 입니다."
    }
  ],
  "code": "C001",
  "timestamp": "2019-09-21T23:21:26.006"
}
```

ConstraintValidator 사용법

Annotation 생성

```java
@Documented
@Constraint(validatedBy = OrderSheetFormValidato
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface OrderSheetForm {

    String message() default "Order sheet form i

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default

}
```

```java
public class OrderSheetFormValidator implements ConstraintValidator<OrderSheetForm, OrderSheetRequest> {

    @Override
    public void initialize(OrderSheetForm constraintAnnotation) {

    }

    @Override
    public boolean isValid(OrderSheetRequest value, ConstraintValidatorContext context) {
        int invalidCount = 0; // (2)

        if (value.getPayment().hasPaymentInfo()) {
            addConstraintViolation(context, "카드 정보 혹은 계좌정보는 필수입니다.", "payment"); // (3)
            invalidCount += 1;
        }

        if (value.getPayment().getPaymentMethod() == PaymentMethod.CARD) {
            final Card card = value.getPayment().getCard();

            if (card == null) {
                addConstraintViolation(context, "카드 필수입니다.", "payment", "card");
            } else {
                if (StringUtils.isEmpty(card.getBrand())) {
                    addConstraintViolation(context, "카드 브렌드는 필수입니다.", "payment", "card", "brand");
                    invalidCount += 1;
                }
                if (StringUtils.isEmpty(card.getCsv())) {
                    addConstraintViolation(context, "CSV 값은 필수 입니다.", "payment", "card", "csv");
                    invalidCount += 1;
                }
                if (StringUtils.isEmpty(card.getNumber())) {
                    addConstraintViolation(context, "카드 번호는 필수 입니다.", "payment", "card", "number");
                    invalidCount += 1;
                }
            }
        }

        ...
        return invalidCount == 0; // (6)
    }
```

ConstraintValidator 사용법

```java
@NoArgsConstructor(access = AccessLevel.PRIVATE)
@Getter
@OrderSheetForm // (1)
public class OrderSheetRequest {

    @Min(1)
    private BigDecimal price;

    @NotNull
    @Valid // (2)
    private Payment payment;

    @NotNull
    @Valid
    private Address address;

    @NoArgsConstructor(access = AccessLevel.PRIVATE)
    @Getter
    @ToString // (3)
    public static class Payment {

        @NotNull
        private PaymentMethod paymentMethod;
        private Account account;
        private Card card;

        @JsonIgnore
        public boolean hasPaymentInfo() {
            return account != null && card != null;
        }

    }
    ...
}
```

- (1) `@OrderSheetForm` 을 추가해서 `OrderSheetFormValidator` 가 동작하게 합니다.
- (2) `@Valid` 을 추가해서 각 클래스의 `JSR-303` 기반 어노테이션이 동작하게 합니다. `@Valid` 이 없는 경우 payment.PaymentMethod의 `@NotNull` 동작하지 않습니다.
- (3) Error[].value 값이 객체인 경우에 해당 객체의 정보를 출력하기 위해서 `@ToString` 을 추가합니다.

Contoller

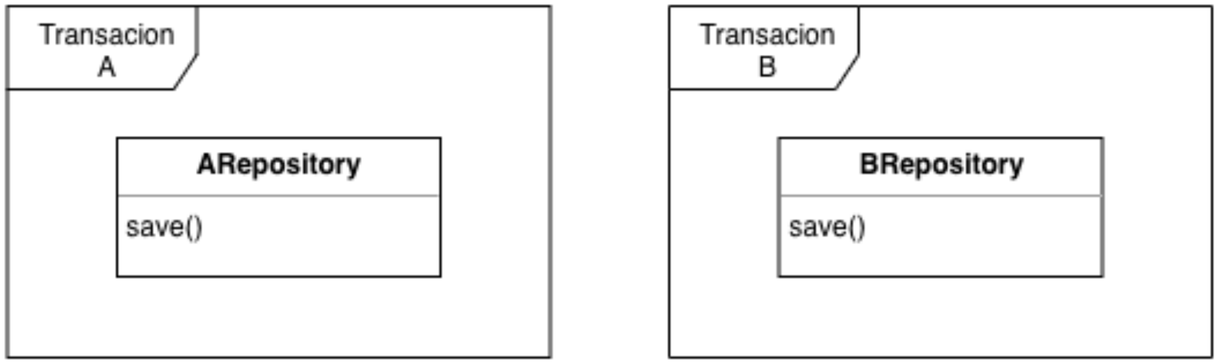
```java
@RestController
@RequestMapping("/order")
public class OrderApi {

    @PostMapping
    public OrderSheetRequest order(@RequestBody @Valid final OrderSheetRequest dto) {
        return dto;
    }
}
```

- `@Valid` 어노테이션으로 검증을 진행합니다.

7. 트랜잭션

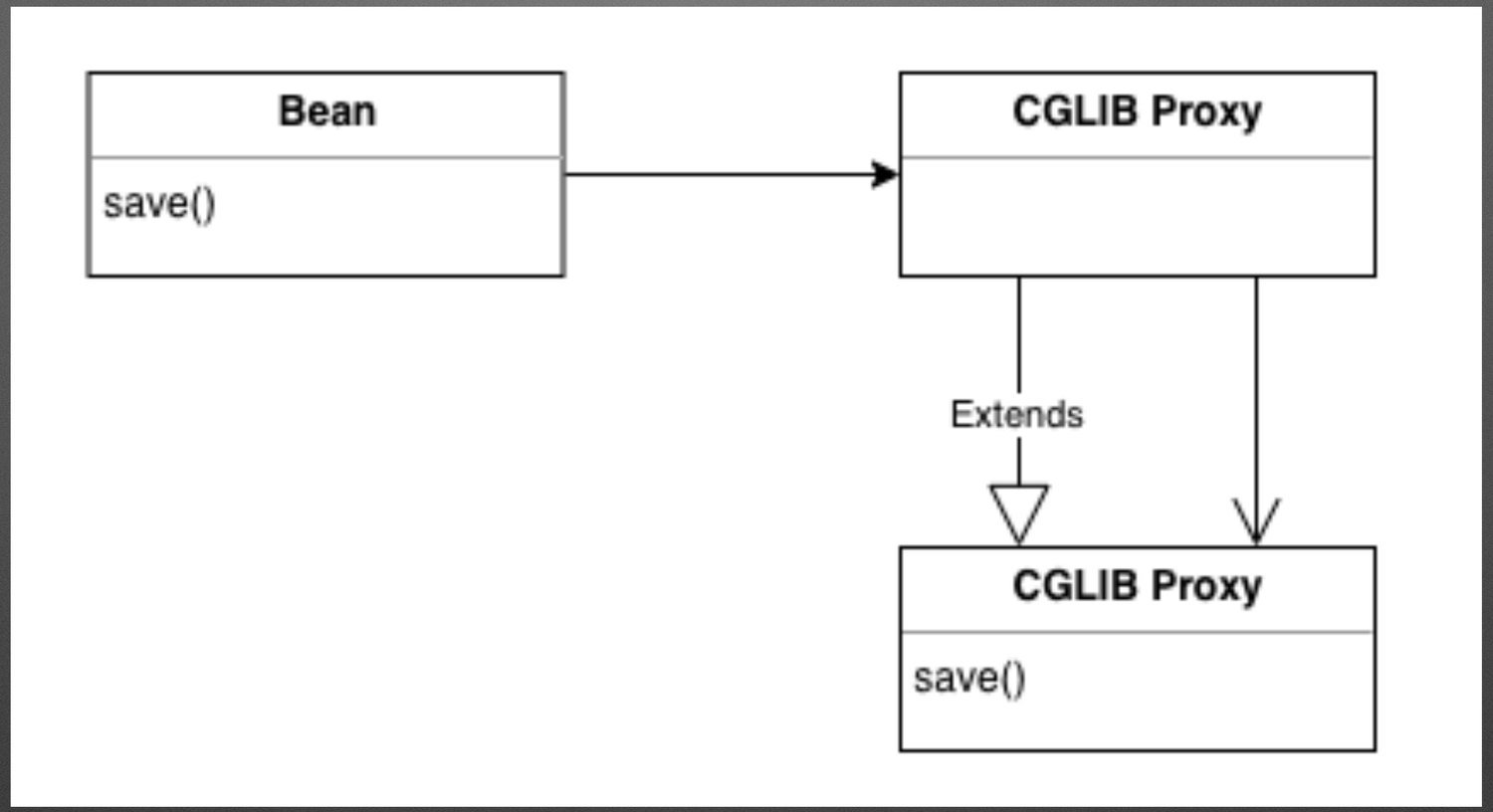
트랜잭션 단위

```
@RestController
@RequestMapping("/sample")
class SampleController(
        private val aRepository: ARepository,
        private val bRepository: BRepository
) {
    @GetMapping
    fun transaction() {
        aRepository.save(A("A"))
        bRepository.save(B("B")) // 여기에서 exception 발생시 위 save는 Rollback 되지 않는다.
    }
}
```



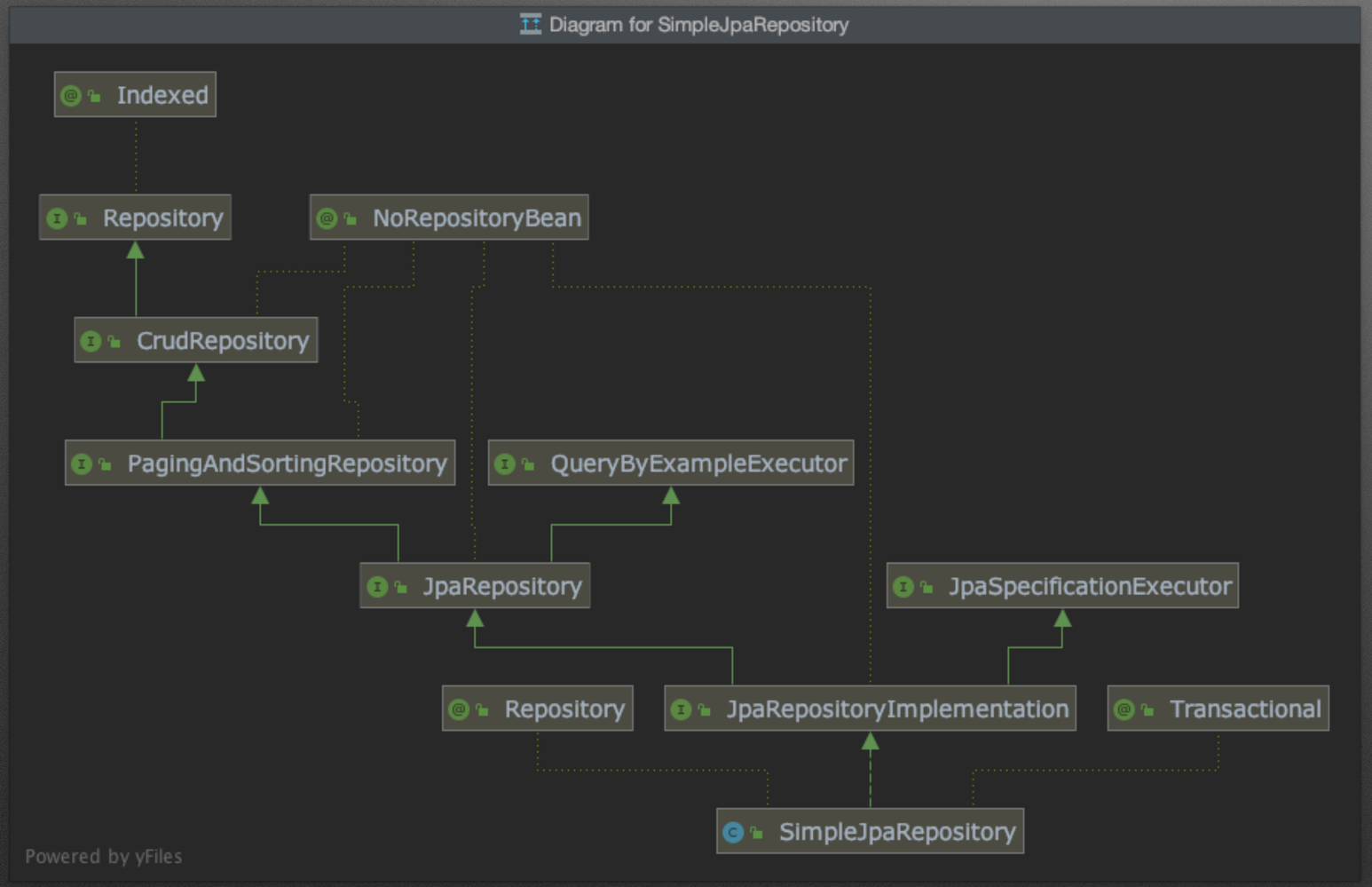
- bRepository.save(B("B")) 에서 예외가 발생했을 경우 트랜잭션 단위는 클라스 단위로 지정되기 때문에 aRepository.save(A("A")) Rollback 되지 않습니다.

트랜잭션 AOP 원리



- 스프링 부트에서는 기본적으로 CGLIB Proxy 방식으로 트랜잭션을 처리합니다. (이전에는 Dynamic Proxy 패턴)
- Proxy 패턴을 사용하기 때문에 클래스 단위로 트랜잭션이 시작되고 묶이게 됩니다.

JPA Repository는 interface인데 ?



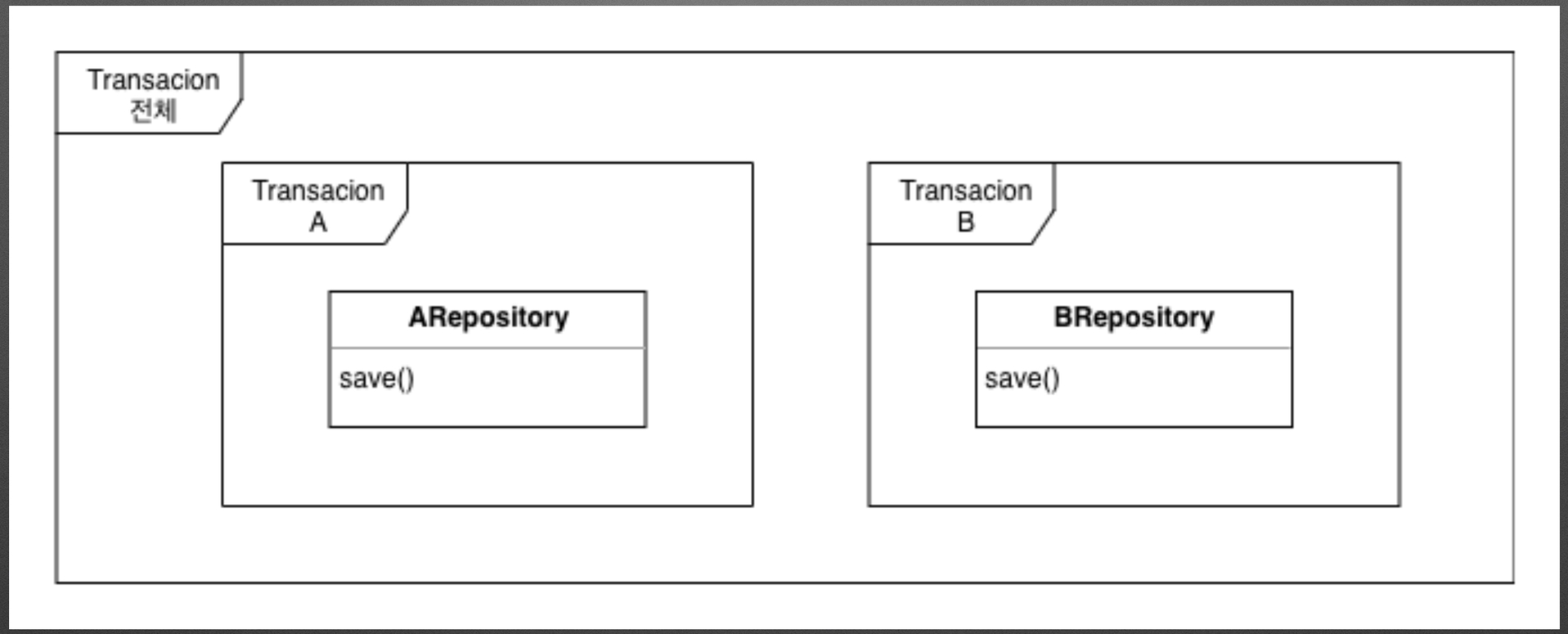

```
@Transactional
@Override
public <S extends T> S save(S entity) {

  if (entityInformation.isNew(entity)) {
    em.persist(entity);
    return entity;
  } else {
    return em.merge(entity);
  }
}
```

두 트랜잭션을 묶는 방법

```
@RestController
@RequestMapping("/sample")
class SampleController(
        private val aRepository: ARepository,
        private val bRepository: BRepository
) {
    @GetMapping
    @Transactional // aRepository, bRepository 트랜잭션을 묶는다.
    fun transaction() {
        aRepository.save(A("A"))
        bRepository.save(B("B")) // 여기에서 exception 발생시 위 save는 Rollback 된다.
    }
}
```

- @Transactional을 추가하게 되면 aRepository, bRepository 트랜잭션이 같이 묶이게 됩니다.
- 컨트롤러에 추가하는 것보다 별도의 트랜잭션를 담당하는 Service 영역을 추가하는 것이 바람직합니다.

두 트랜잭션을 묶는 방법



- 이렇게 되면 두 트랜잭션이 같이 묶이게 되고 bRepository.save(B("B")) 에서 exception이 발생하게 되면 bRepository.save(B("B")), aRepository.save(A("A"))이 모두 rollback이 진행됩니다.

트랜잭션 Isolation Level

- 스프링5 레시피 예제 참고

이름	설명
오염된 읽기 Duty Read	T2가 수정후 커밋하지 않은 필드를 T1이 읽는상황에서 **나중에 T2가 롤백되면 T1이 읽은 필드를 일시적인 값으로 더 이상 유효하지 않습니다.**
재현 불가한 읽기 Nonrepeatable Read	어떤 필드를 T1이 읽은 후 T2가 수정할 경우, **T1이 같은 필드를 다시 읽으면 다른 값을 얻습니다.**
허상 읽기 Phantom Read	T1이 테이블의 로우를 몇개를 읽은 후 T2가 같은 테이블에 새 로우를 삽입할 경우, 나중에 T1이 같은 테이블을 다시 읽으면 T2가 삽입한 로우가 보입니다.
소실된 수정 Lost Update	T1, T2가 모두 어떤 로우를 수정하려고 읽고 그 로우의 상태에 따라 수정하는 경우입니다. T1이 먼저 로우를 수정 후 커밋하기 전, T2가 T1이 수정한 로우를 똑같이 수정했다면 T1이 커밋한 후에 T2 역시 커밋하게 됩니다. 그러면 **T1이 수정한 로우를 T2가 넒어쓰게 되어 T1이 수정한 내용이 소실됩니다.**

격리 수준	설명
DEFAULT	DB 기본 격리 수준을 사용합니다.
READ_UNCOMMITTED	**다른 트랜잭션이 아직 커밋하지 않은 값을 한 트랜잭션이 읽을 수 있습니다.** 따라서 오염된 읽기, 재현 불가한 읽기, 허상 읽기 문제가 발생할 가능성이 있습니다.
READ_COMMITTED	**한 트랜잭션이 다른 트랜잭션이 커밋한 값만 읽을수 있습니다.** 이로써 오염된 값 읽기 문제는 해결되지만 재현 불가한 읽기, 허상 읽기 문제는 여전히 남습니다.
REPEATABLE_READ	트랜잭션이 어떤 필드를 여러번 읽어도 동일한 값을 읽도록 보장합니다. **트랜잭션이 지속되는 동안 다른 트랜잭션이 해당 필드를 변경할수 없습니다.** 오렴된 값 읽기, 재현 불가한 읽기 문제는 해결되지만 허상 읽기는 여전히 남습니다.
SERIALIZABLE	트랜잭션이 테이블을 여러 번 읽어도 정확히 동일한 로우를 읽도록 보장합니다. **트랜잭션이 지속되는 동안에는 다른 트랜잭션이 해당 테이블에 삽입, 수정, 삭제를 할 수 없습니다.** 동시성 문제는 모두 해소괴지만 성능이 현저히 떨어집니다.

READ_UNCOMMITTED, READ_COMMITTED 격리 수준

```java
public class JdbcBookShop extedns JdbcDaoSupport {

    @Transactional
    public void increaseStock(String isbn, int stock){
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + "- Prepareto increase book stock");

        getJdbcTemplate().update("update book_stock set stcok = stock +? where isbn =?", stock, isbn);

        System.out.println(threadName + "- Book stock increased by" + stock);
        sleep(threadName);

        System.out.println(threadName + "- Book stock rolled back");
        throw new RuntimeException("Increased by mistake");
    }

    @Transactional(isolation = Isolation.READ_UNCOMMITTED)
    public int checkStock(String isbn){
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + "- Prepare to check book stock");

        int stock = getJdbcTemplate().queryForObejct("select stcok from book_stock where isbn = ?", Inte

        System.out.println(threadName + "- Book stock is "+ stock);
        sleep(threadName);

        return stock;
    }

    private void sleep(String threadName) {
        System.out.println(threadName + " - Sleeping");

        try {
        Thread.sleep(10000);
        } catch (InterruptedException e) {
        }

        System.out.println(threadName + " - Wake up");
    }
}
```

- 동시성을 시뮬레이션 하기 위해 여러 스레드로 실행
- SQL문이 실행될 때 마다 강제로 **스레드를 10초간 sleep**
- increase() 메서드는 마지막 RuntimeException을 던지면서 트랜잭션이 롤백 진행

```java
public class Main {

    public staic void main(String[] args){
        final BookShop bookShop = context.getBean(BookShop.class);

        Thread thread1 = new Thread(()->{
            try {
                bookShop.increaseStock("0001", 5);
            } catch (RuntimeException e) {}
        }, "Thread1");


        Thread thread2 = new Thread(()->{
            bookShop.checkStock("0001");
        }, "Thread2");


        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (Exception e) {}

        thread2.start();

    }
}
```

```
Thread 1 Book stock 증가 준비중
Thread 1 Book stock 5 wmdrk
Thread 1 스레드 sleep
Thread 2 Book stock 제고 확인
Thread 2 Book stock 재고 15개
Thread 2 Sleeping
Thread 1 Wake Up
Thread 1 Book stock rolled back
Thread 2 Wake Up
```

```java
public class Main {

    public staic void main(String[] args){
        final BookShop bookShop = context.getBean(BookShop.class);

        Thread thread1 = new Thread(()->{
            try {
                bookShop.increaseStock("0001", 5);
            } catch (RuntimeException e) {}
        }, "Thread1");


        Thread thread2 = new Thread(()->{
            bookShop.checkStock("0001");
        }, "Thread2");


        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (Exception e) {}

        thread2.start();

    }
}
```

- 스레드 1이 도서 재고를 늘리고 잠듭니다. (아직 트랜잭션 롤백 이전)
- 스레드2 도서 재고 확인 (격리 수준이 READ_UNCOMMITTED 이므로 스레드2는 스레드1의 트랜잭션이 아직 변경 후 커밋하지 않은 재고의 값을 읽어옴)
- 스레드 1이 wake up 되면서 Exception 발생 하여 롤백
- 스레드2가 읽은 재곳값 15는 더이상 유효하지 않음 이런 증상을 오염된 읽기 Duty Read 라고 한다.

```java
public class Main {

    public staic void main(String[] args){
        final BookShop bookShop = context.getBean(BookShop.class);

        Thread thread1 = new Thread(()->{
            try {
                bookShop.increaseStock("0001", 5);
            } catch (RuntimeException e) {}
        }, "Thread1");


        Thread thread2 = new Thread(()->{
            bookShop.checkStock("0001");
        }, "Thread2");


        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (Exception e) {}

        thread2.start();

    }
}
```

- 이러한 문제는 격리 수준을 **READ_COMMITTED로 한 단계 올리면 해결**됩니다.
- 스레드2는 스레드 1이 트랜잭션을 롤백 하기 전까지 재곳값을 읽을 수 없습니다. **오염된 읽기는 한 트랜잭션이 아직 커밋하지 않은 갓을 다른 트랜잭션이 읽지 못하게 차단함으로써 방지** 할 수 있습니다.
- READ_COMMITED 격리 수준을 지원하는 DB는 **수정은 되었지만 아직 커밋하지 않은 로우에 수정 잠금을 걸어둔 상태** 입니다. 결국 **다른 트랜잭션이 이 트랜잭션이 커밋/롤백 되고 수정 잠금이 풀릴 때 까지 기다렸다가** 읽을 수 밖에 없습니다.

REPEATABLE_READ 격리수준

스레드 1이 도서 재고를 체크, 스레드 2가 도서 재고를 증가

1. 스레드 1이 재고를 체크하고 sleep (아직 트랜잭션이 커밋되지 않은 상태)
2. 스레드 2가 시작되어 도서 재고를 늘립니다. (격리수준 READ_COMMITTED 이므로 커밋되지 않은 트랜잭션이 읽은 재곳값을 수정할 수 있습니다.)
3. 스레드 1이 깨어 나면 도서 재고를 다시 읽으면 그 값은 이미 처음에 읽은 그 값이 아닙니다. 재현 불가한 읽기 Nonrepeatable Read 발생

REPEATABLE_READ 격리수준

해결 방법

- 이러한 문제를 재현 불가한 읽기 문제이며 한 트랜잭션이 동일한 필드를 다른 값으로 읽어들이는 모순이 발생합니다.
- 해결 방법은 REPEATABLE_READ으로 격리 수준을 변경하면 됩니다.

1. 스레드 1이 재고를 체크하고 sleep (아직 트랜잭션이 커밋되지 않은 상태)
2. 스레드 2는 스레드 1이 트랜잭션을 커밋하기 전까지 재곳값을 수정할 수 없습니다.

- REPEATABLE_READ 격리 수준을 지원하는 DB는 조회는 되었지만 아직 커밋하지 않은 로우에 읽기금을 걸어둔 상태입니다. 결국 다른 트랜잭션은 이 트랜잭션이 커밋/롤백하여 읽기 잠금이 풀릴 때까지 기다렸다가 수정 작업을 진행하게 됩니다.

SERIALIZABLE 격리 수준

1. 트랜잭션 1이 테이블에 여러 로우를 읽기
2. 트랜잭션 2가 같은 테이블에 여러 로우를 새로 insert
3. 트랜잭션 1이 같은 테이블을 다시 읽으면 자신이 처음 읽었을 때와 달리 새로 추가된 로우가 있음을 감지 (이것을 허상 읽기 라고함)

- 허상 읽기는 여러 로우가 연관된 점만 빼면 재현 불가한 읽기 문제와 비슷합니다.
- 허상 읽기 문제를 해결하려면 최고 격리 수준인 SERIALIZABLE 으로 올려야합니다.
- SERIALIZABLE 경우 전체 테이블에 읽기 잠금을 걸기 때문에 실행 속도는 가장 느립니다.

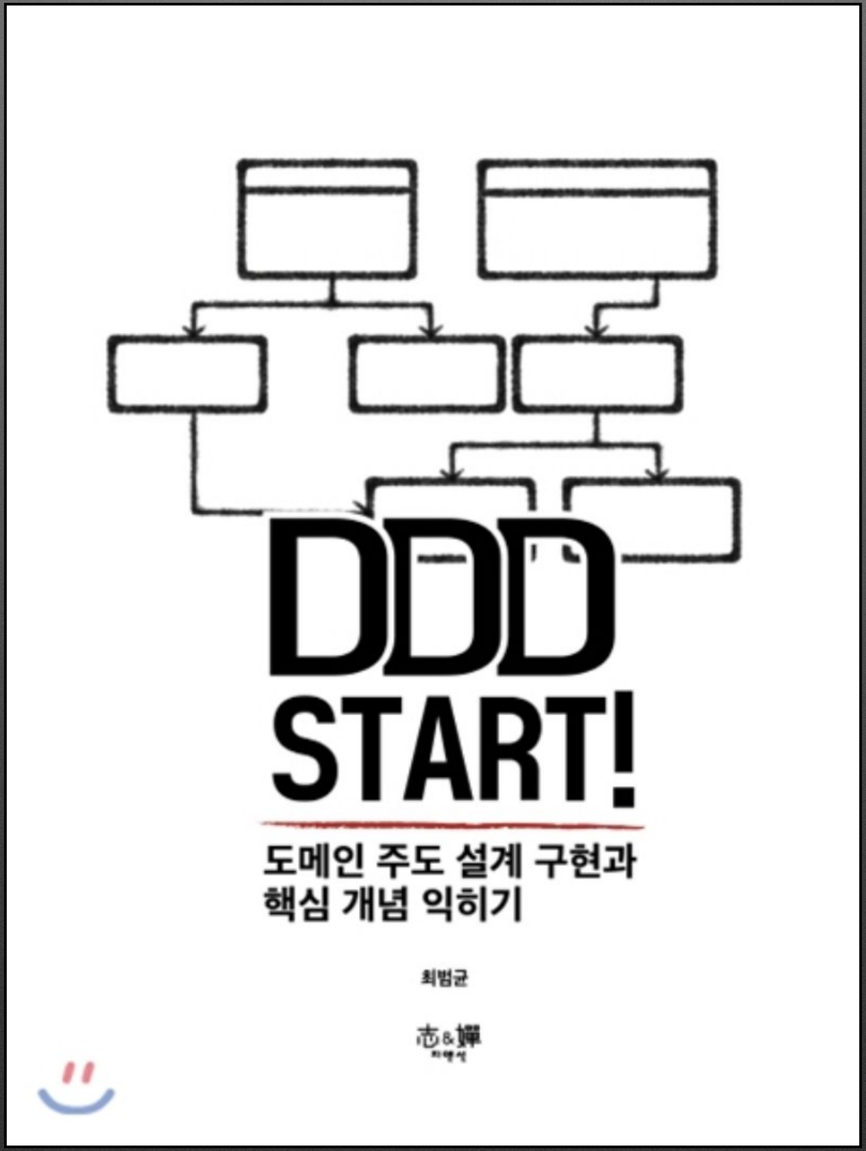
격리수준	더티리드	모순 판독	갱신 분실	팬텀리드
READE-UNCOMMITTED	O	O	O	O
READE-COMMITED	X	O	O	O
REPEATABLE-READ	X	X	O	O
SERIALIZABLE	X	X	X	X

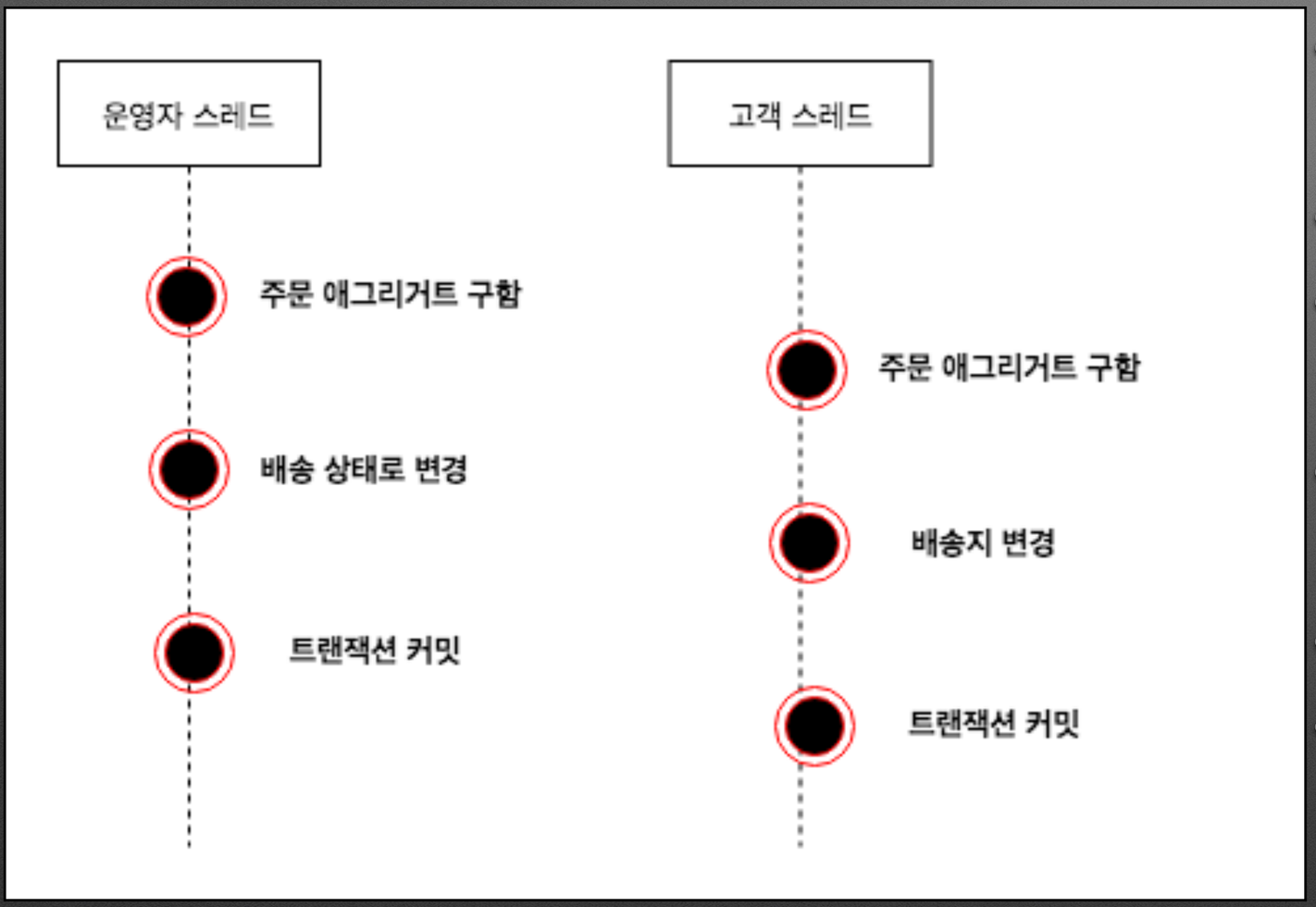
소실된 수정 : Lost Update

소실된 수정 : Lost Update

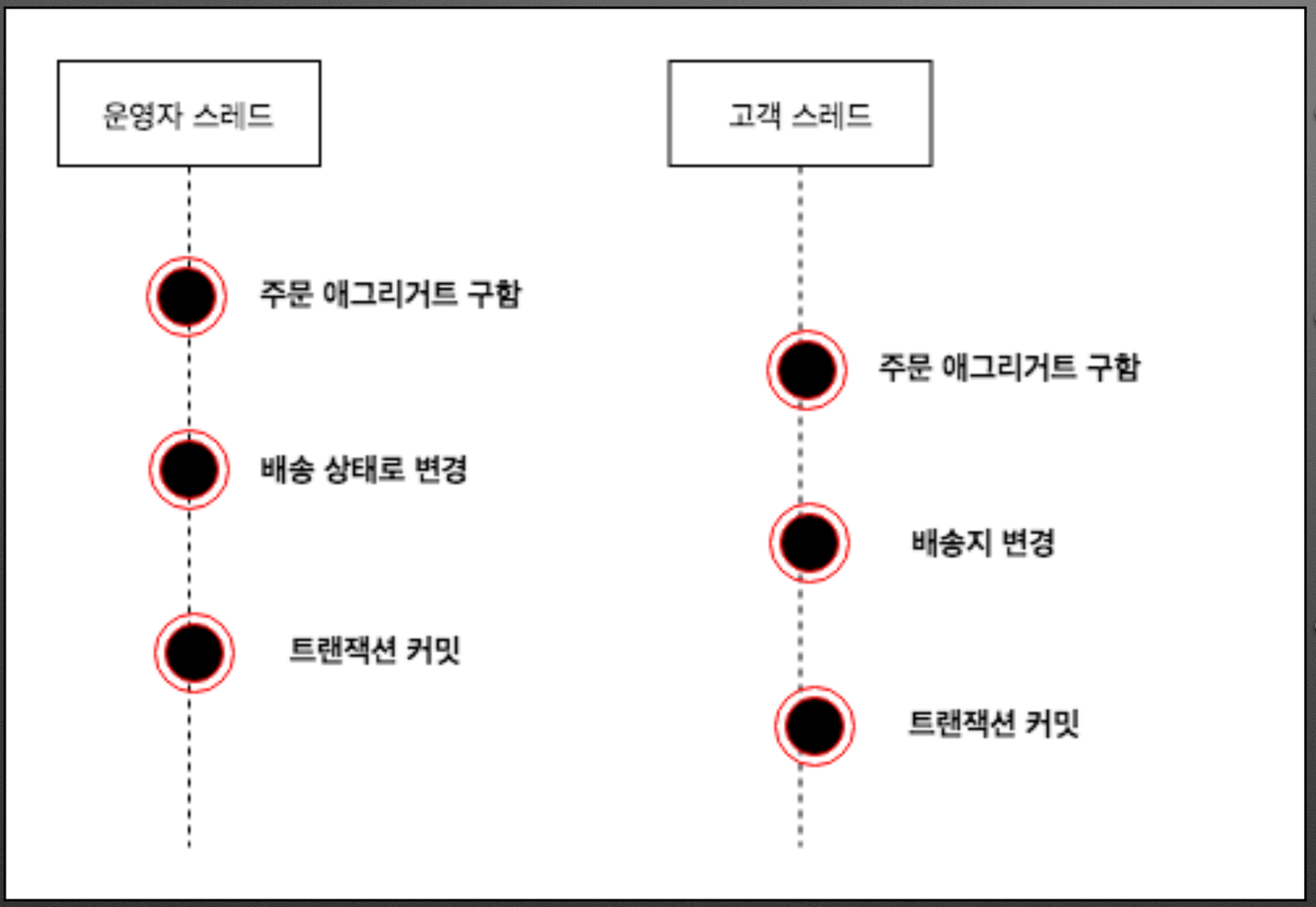
- T1, T2가 모두 어떤 로우를 수정하려고 읽고 그 로우의 상태에 따라 수정하는 경우입니다.
- T1이 먼저 로우를 수정 후 커밋하기 전, T2가 T1이 수정한 로우를 수정했다면 T1이 커밋한 후에 T2 역시 커밋하게 됩니다.
- 그러면 T1이 수정한 로우를 T2가 넒어쓰게 되어 T1이 수정한 내용이 소실됩니다.

참고



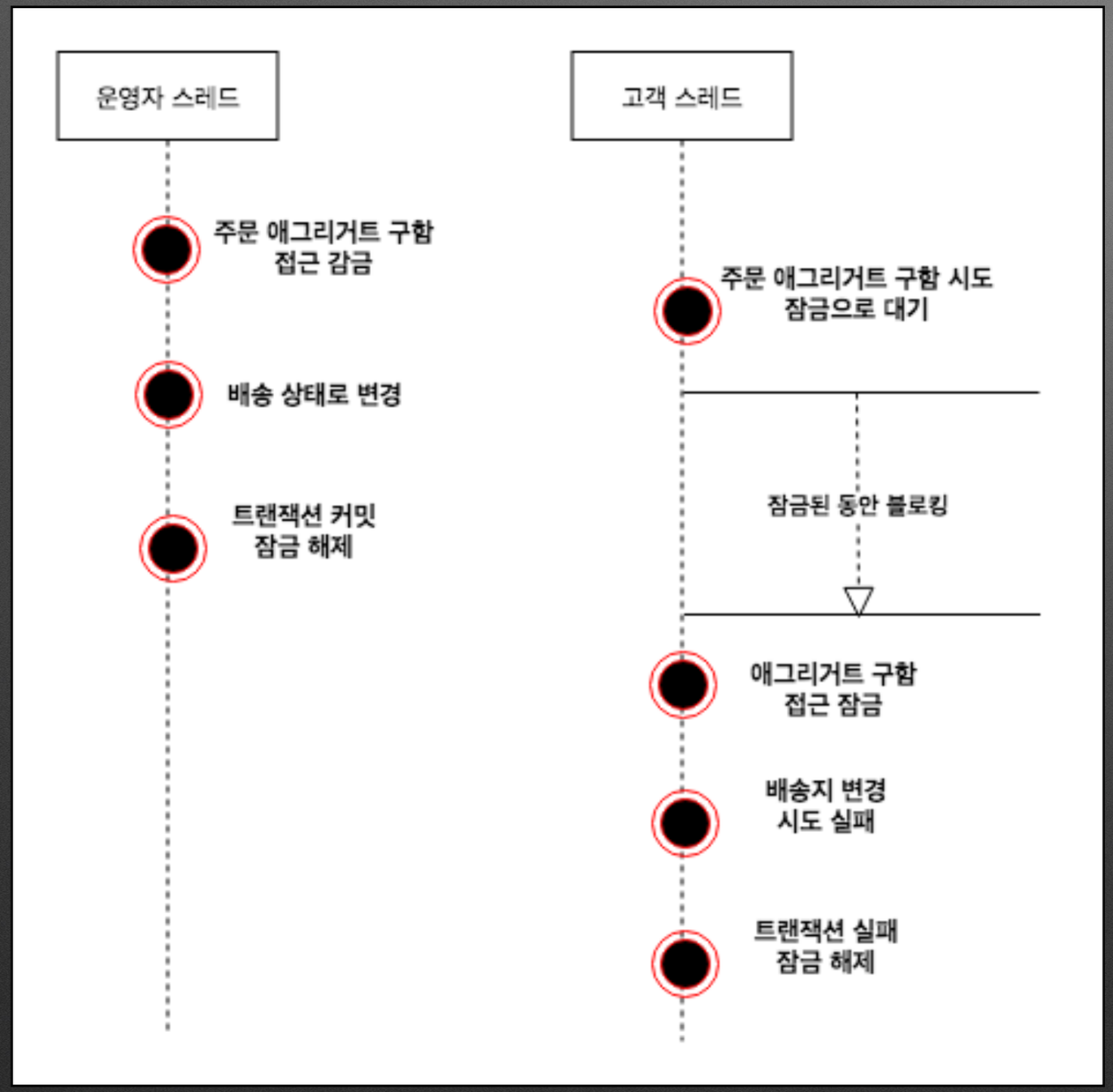


- 운영자와 고객이 동시에 한 주문 애그리거트를 수정하는 과정을 보여준다.
- 배송 상태로 변경되면 더 이상 배송지 변경은 할 수 없다.
- 운영자 스레드와 고객 스레드는 같은 주문 애그리거트를 나타내는 다른 객체를 구하게 된다.
- 이 상황에서 두 스레드는 각각 트랜잭션을 커밋할 때 수정한 내용을 DBMS에 반영한다.
- 배송 상태로 바뀌고 배송지 정보로 바뀌게 된다.
- 이 순서의 문제점은 운영자는 기존 배송지 정보를 이용해서 배송 상태로 변경했는데 그 사이 고객은 배송지 정보를 변경했다는 점이다.



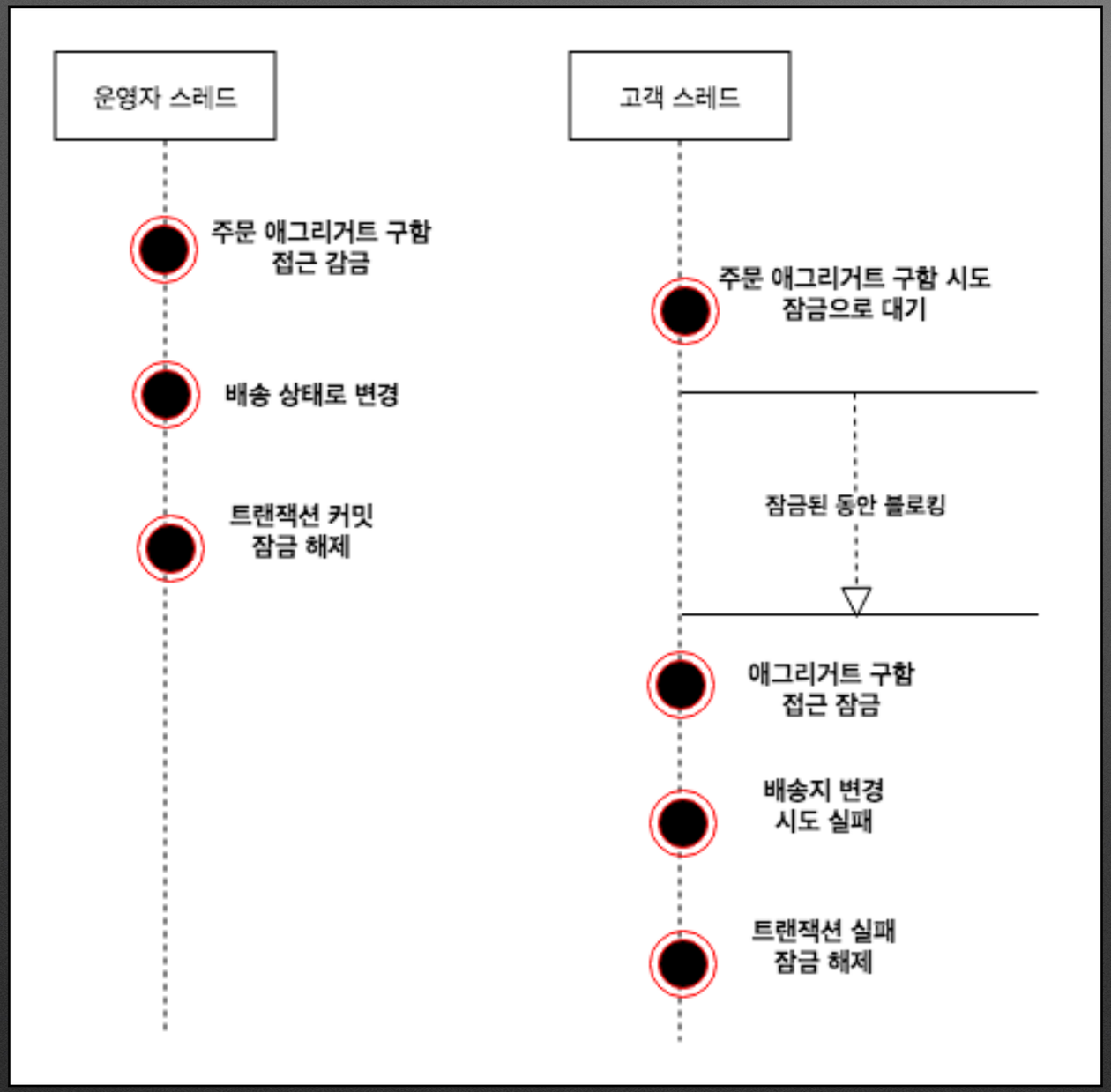
- 문제가 발생하지 않도록 하려면 다음 두 가지 중 하나를 해야한다.
- 운영자 배송지 정보를 조회하고 상태를 변경하는 동안 고객이 애그리거트를 수정하지 못하게 막는다. -> 선점 잠금
- 운영자가 배송지 정보를 조회한 이후 고객이 배송지 정보를 변경하면 운영자가 애그리거트를 다시 조회한 뒤 수정하도록 한다. -> 비선점 잠금

선점 잠금



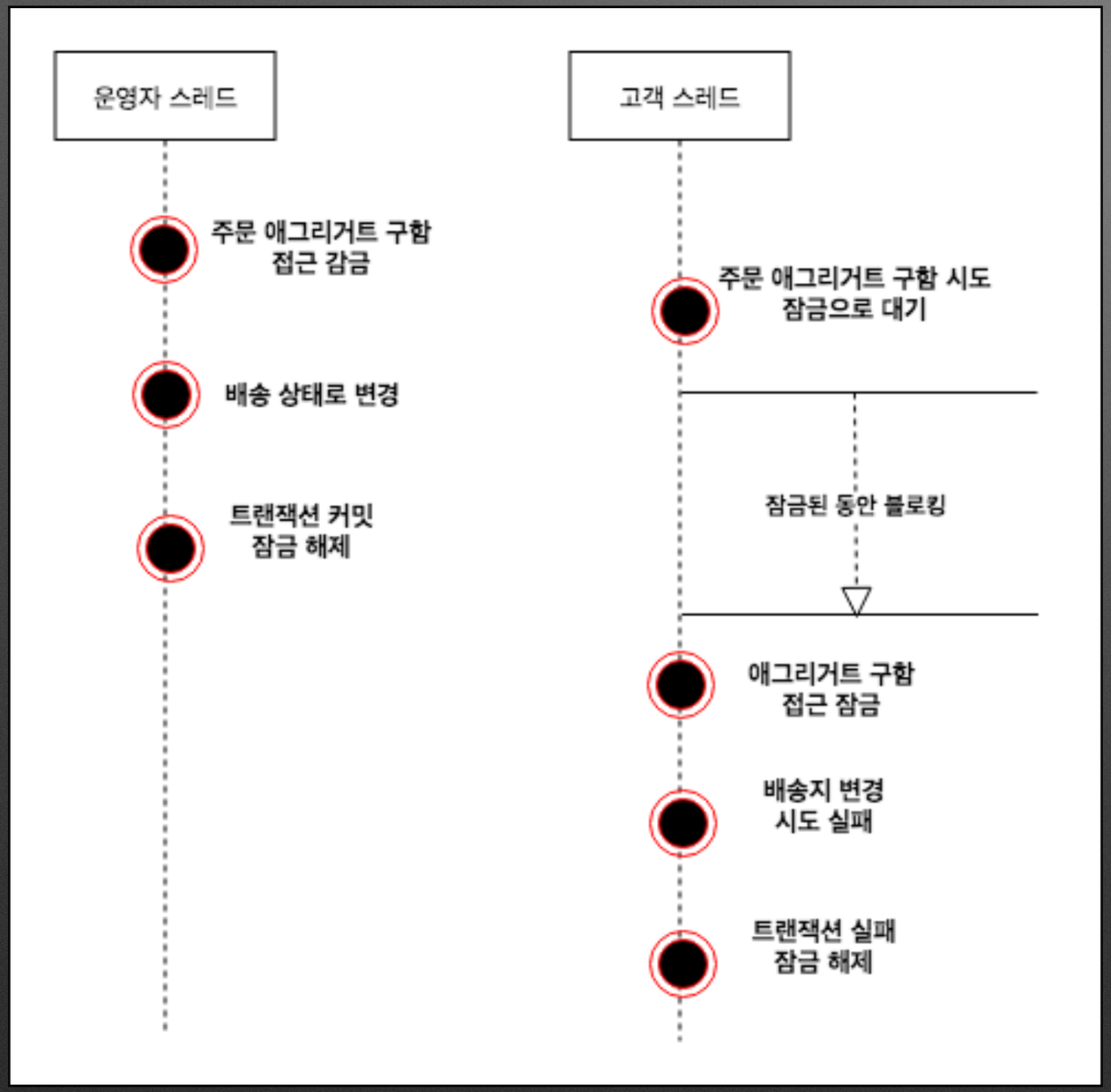
- 스레드1 선점 잠금방식으로 애그리거트를 구한 뒤 이에서 스레드2가 같은 애그리거트를 구하고 있는데, 이 경우 스레드2는 스레드1이 애그리거트에대한 잠금을 해제할 때 까지 블로킹된다.
- 스레드1이 애그리거트를 수정하고 트랜잭션을 커밋하면 잠금을 해제한다. 이 순간 대기하고 있던 스레드2가 애그리거트에 접근하게 된다. 스레드1이 트랜잭션을 커밋 뒤에 스레드2가 애그리거트를 구하게 되므로 스레드2는 스레드1이 수정한 애그리거트의 내용을 보게된다.
- 한 스레드가 애그리거트를 구하고 수정하는 동안 다른 스레드가 수정할 수 없음므로 동시에 애그리거트를 수정할 때 발생하는 데이터 충돌 문제를 해소할 수 있다.

선점 잠금 적용



1. 운영자 스레드가 먼저 선점 잠금 방식으로 주문 애그리거트를 구함
2. 고객 스레드는 운영자 스레드가 잠금을 해제할 때까지 고객 스레드는 대기 상태
3. 운영자 스레드가 배송 상태로 변경한 뒤에 트랜잭션을 커밋하면 잠금을 해제한다.
4. 잠금이 해제된 시점에 고객 스레드가 구하는 주문 애그리거트는 운영자 스레드가 수정한 배송 상태의 주문 애그리거트이다. 배송 상태이므로 주문 애그리거트는 배송지 변경 시 에러를 발생하거 트랜잭션이 실패한다.
5. 고객은 이미 배송이 시작되어 배송지를 변경할 수 없습니다.와 같은 안내 문구를 받게 됨

선점 잠금 적용



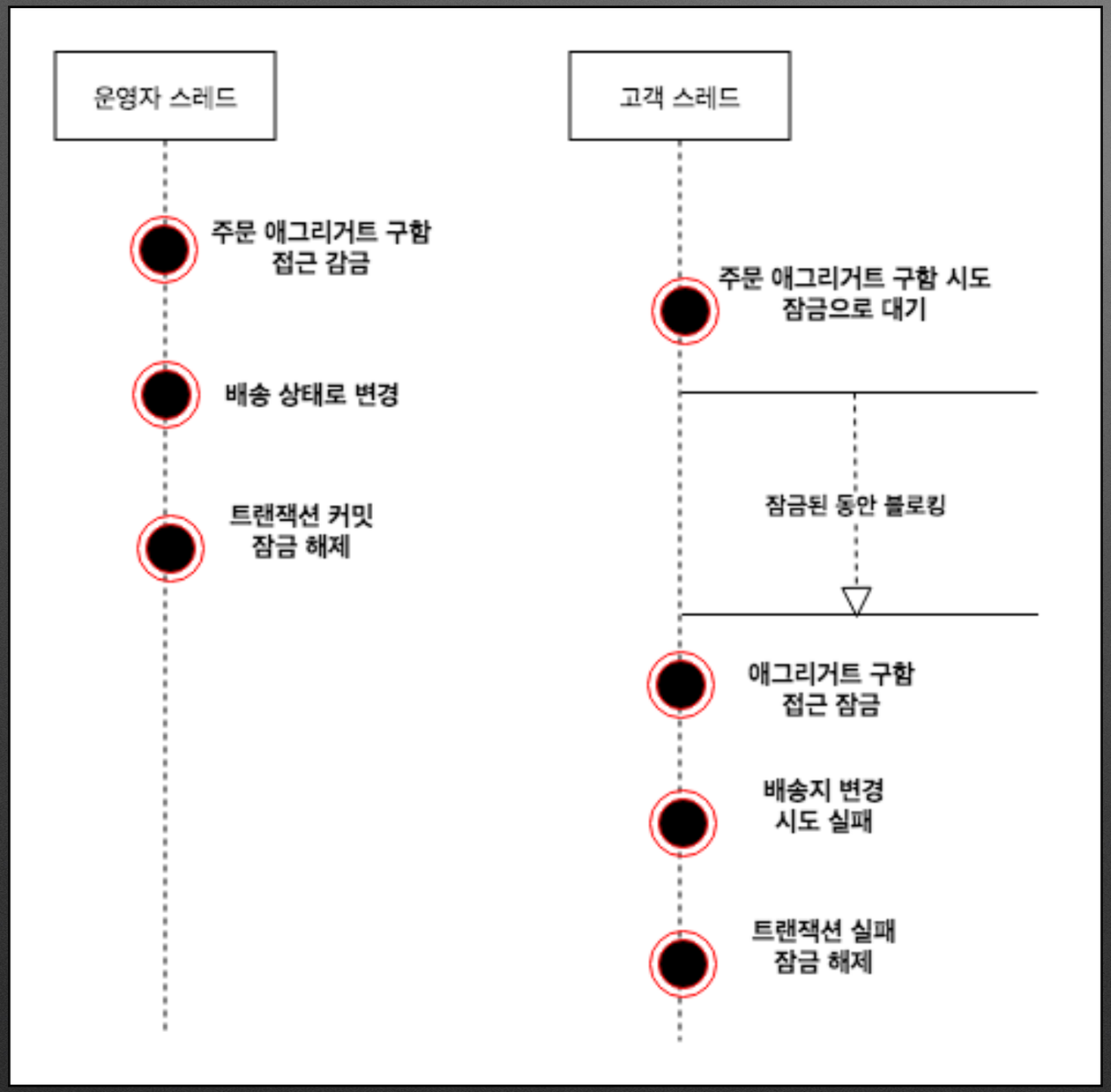
- 선점 잠금은 보통 DBMS가 제공하는 행 단위 잠금을 사용해서 구현한다. 오라클을 비롯한 다수 **DBMS가 for update와 같은 쿼리를 사용해서 특정 레코드에 한 사용자만 접근할 수 있는 잠금 장치**를 제공한다.
- JPA의 EntityManager는 LockModeType을 인자로 받는 find() 메서드를 제공하는데, LockModeType.PESSIMISTIC_WRITE를 값으로 전달하면 해당 **엔티티와 매핑된 테이블을 이용해서 선점 잠금 방식을 적용할 수 있다.**
- JPA 프로바이더와 DBMS에 따라 잠금 모드의 구현이 다른데, **하이버네티으의 경우 PESSIMISTIC_WRITE를 잠금 모드로 사용하면 for update 쿼리를 사용해서 선점 잠금을 구현한다.**

선점 잠금과 교착상태

1. 스레드 1: A 애그리거트에 대한 선점 잠금 구함
2. 스레드 2: B 애그리거트에 대한 선점 잠금 구함
3. 스레드 1: B 애그리거트에 대한 선점 잠금 시도
4. 스레드 2 : A 애그리거트에 대한 선점 잠금 시도

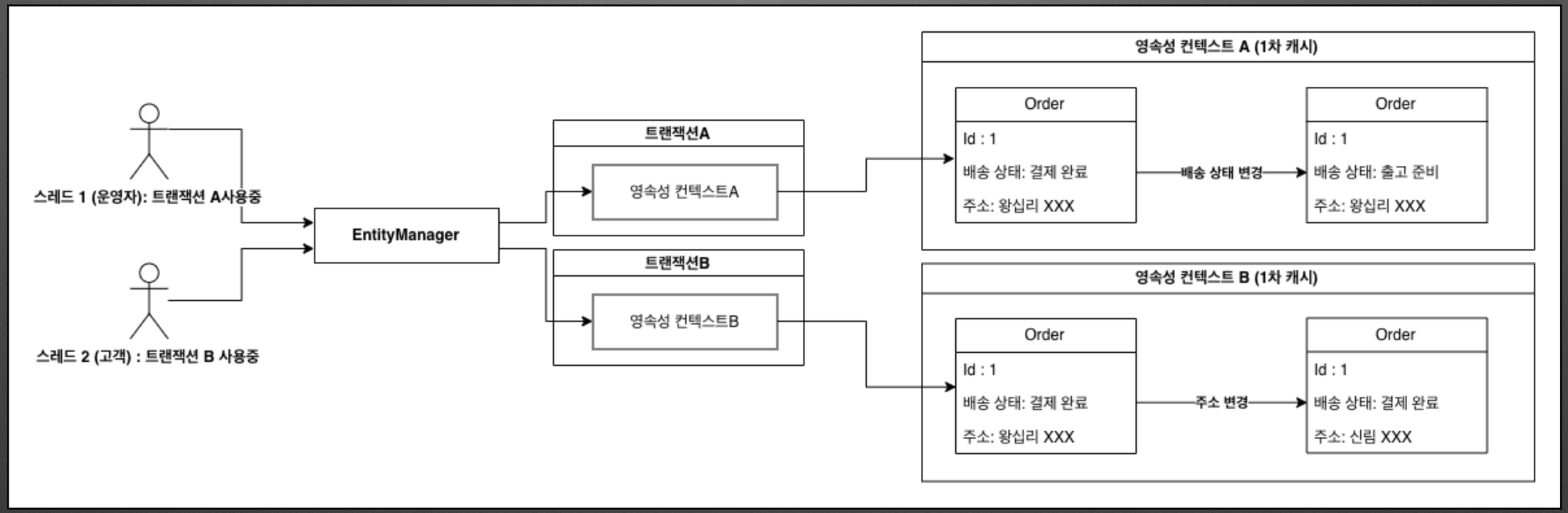
- 이 두 스레드는 상대방 스레드가 먼저 선점한 잠금을 구할수 없어 더 이상 다음 단계를 진행하지 못하게 된다. 즉 스레드 1과 스레드 2는 교착상태에 빠지게 된다.
- 선점 잠금에 따른 교착 상태는 상대적으로 사용자 수가 많을 때 발생할 가능성이 높고, 사용자 수가 많아지면 교착 상태에 빠지는 스레드가 더 빠르게 증가하게 된다. 더 많은 스레드가 교착 상태에 빠질수록 시스템은 점점 아무것도 할 수 없는 상황에 이르게 된다.
- 이런 문제가 발생하지 않도록 하려면 잠금을 구할 때 최대 대기 시간을 지정해야한다. JPA에서 선점 잠금을 시도할 때 최대 대기 시간을 지정하려면 다음과 같이 힌트를 사용하면 된다.
- JPA의 javax.persistence.lock.timeout 힌트는 잠금을 구하는 대기 시간을 밀리초 단위로 지정한다. 지정한 시간이내에 잠금을 구하지 못하면 익셉션을 발생 시킨다. (DBMS에 따라 힌트가 적용되지 않을 수도 있다. 적용 전 확인 필요)

선점 잠금의 한계 In JPA



- Isolcation Level이 REPEATABLE_READ의 경우 DB는 조회는 되었지만 아직 커밋하지 않은 로우에 읽기 잠금을 걸어둔 상태이다.
- 스레드 1이 애그리거트를 구하는 동안 접근 잠금을 걸고 애그리거트를 수정하는 동안 스레드2에서는 잠금 대기 상태이기 때문에 잠금된 동안은 블록킹이 된다.
- 하지만 JPA 환경에서는 REPEATABLE_READ의로 해당 문제는 해결할 수 없다.

선점 잠금의 한계 In JPA



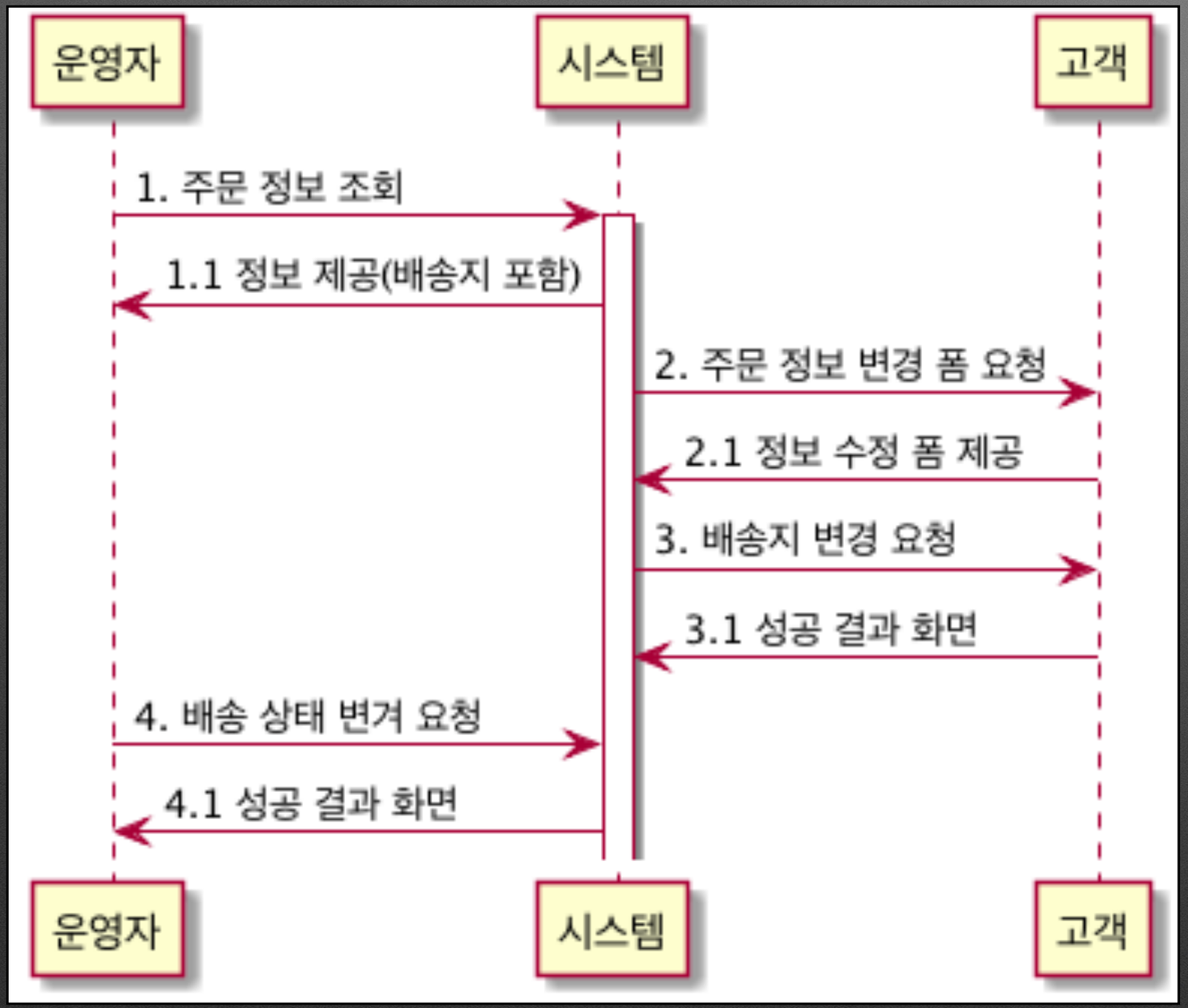
- JPA에서는 트랜잭션 마다 영속성 컨텍스트를 갖게 됩니다. 위 그림처럼 각각의 영속성 컨텍스트 1차 캐시에 저장된 주문 객체를 영속성 컨텍스트A는 배송 상태를 변경, 영속성 컨텍스트B는 주소를 변경하려고 한다. 결과적으로 모두 Commit 이 되어 데이터베이스에 반영되게 된다.
- REPEATABLE_READ의 경우 아직 커밋하지 않은 로우에 읽기 잠금을 걸어둔 상태이지만 동일한 시점(거의 동일한) 조회한 데이터는 1차 캐시에 저장되기 때문에 REPEATABLE_READ의로 해결할 수가 없다.

선점 잠금의 한계

- A와 B가 동시에 제목이 같은 공지사항을 수정한다고 가정하자 결과적으로 먼저 완료한 사용자의 수정사항은 사라지고 나중에 완료한 수정사항만 남게된다. 이것을 두 번의 갱신 분실 문제 second lost updates problem라 한다. 두 번의 갱신 분실 문제는 데이터베이스 트랜잭션의 범위를 넘어선다. 따라서 트랜잭션으로만 문제를 해결 할 수 없다. 이때는 3가지 선택 방법이 있다.

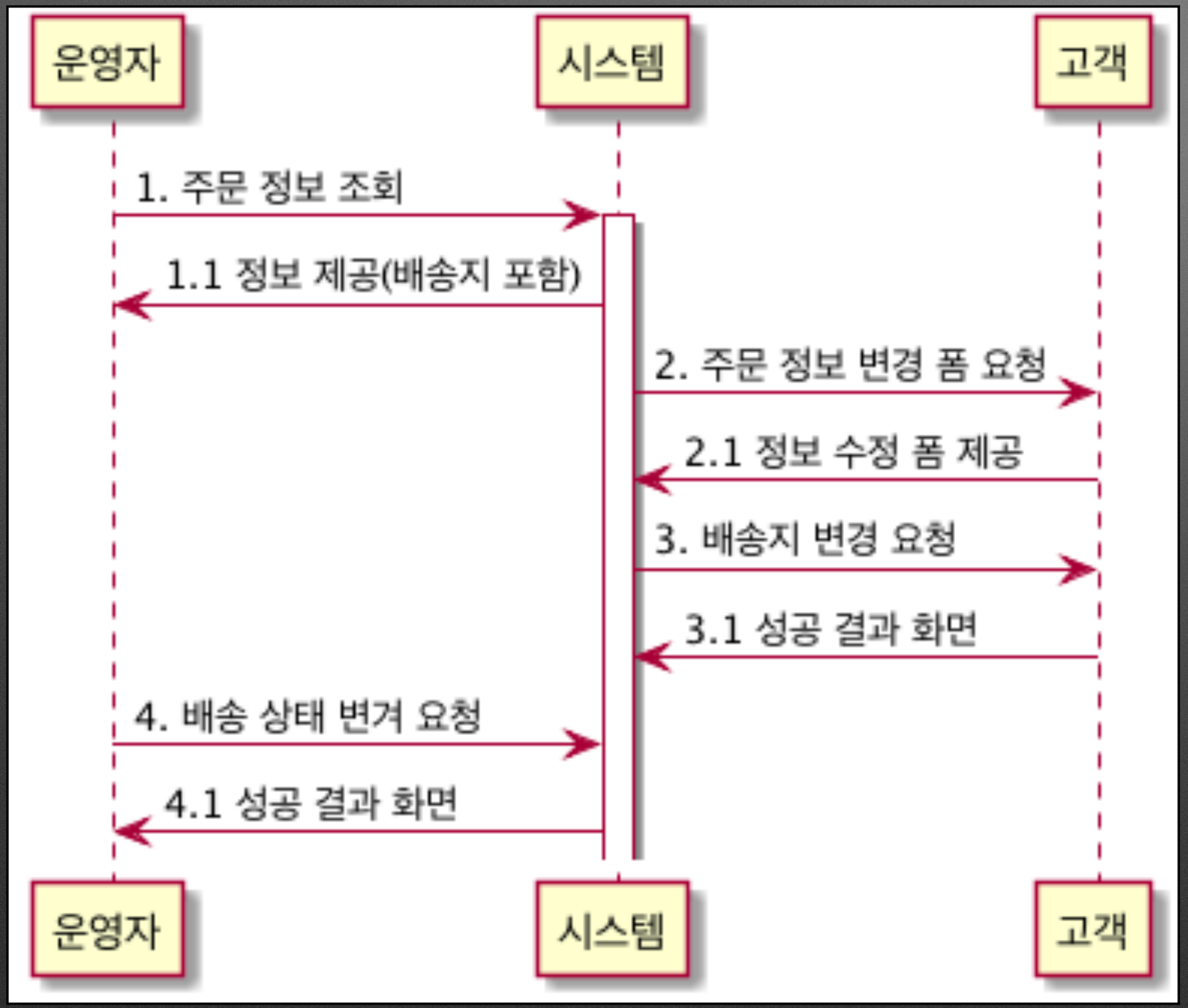
1. 마지막 커밋만 인정하기: 먼드들어온 A의 내용을 무시하고 마지막에 커밋한 B의 내용만 인정한다.
2. 최초 커밋만 인정하기: 사용자 A가 이미 수정을 완료했으므로 사용자 B가 수정을 완료할 때 오류가 발생한다.
3. 충돌하는 갱신 내용 병합하기: 사용자 A와 사용자 B의 수정사항을 병합한다.

비선점 잠금



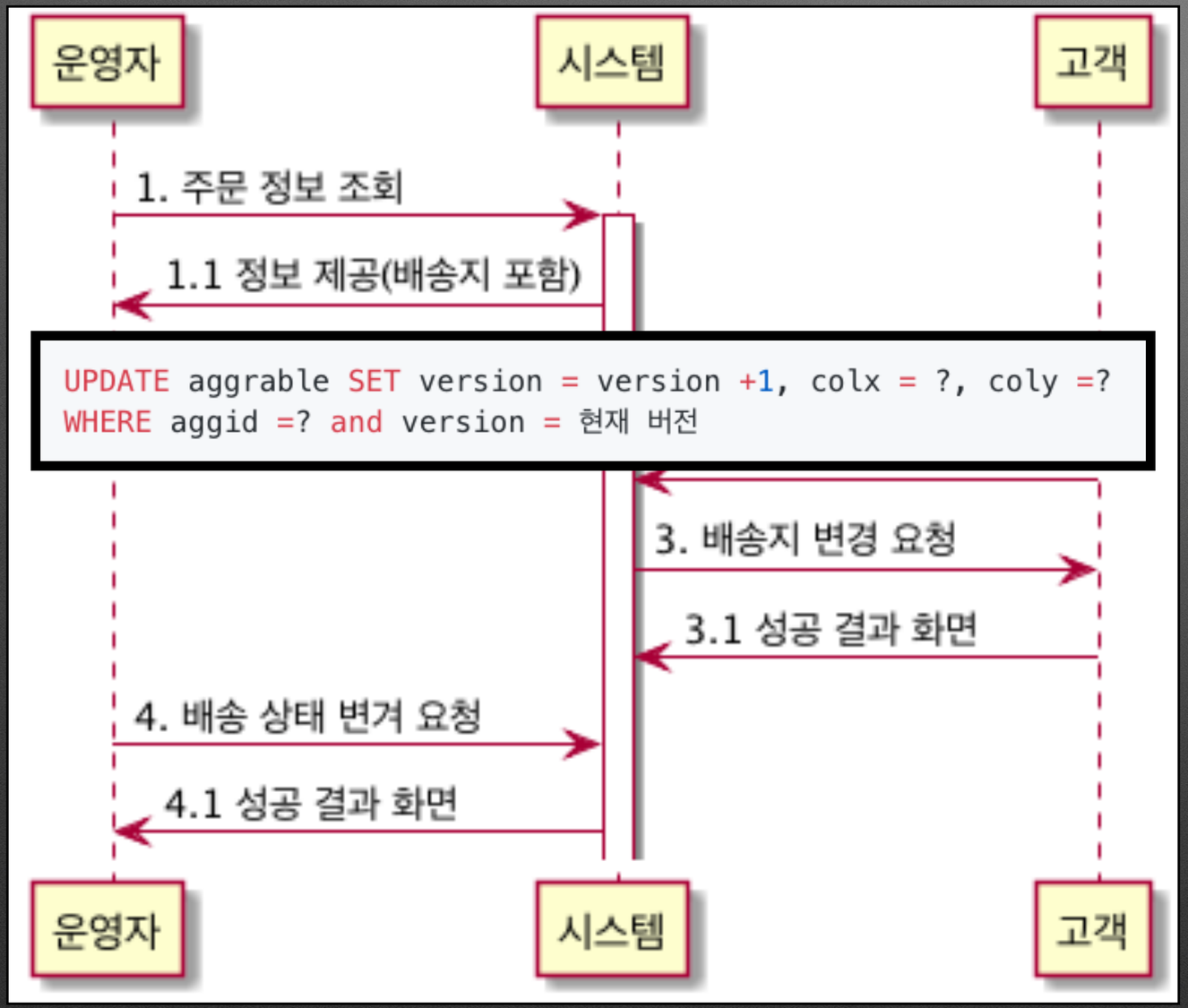
1. 운영자는 배송을 위해 주문 정보를 조회한다.
2. 고객이 배송지 변경을 위해 변경 폼을 요청한다. 시스템은 변경 폼을 제공한다.
3. 고객이 새로운 배송지를 입력하고 폼을 전송해서 배송지를 변경한다.
4. 운영자가 1번에서 조회한 주문 정보를 기준으로 배송지를 정하고 배송 상태 변경을 요청한다.

비선점 잠금



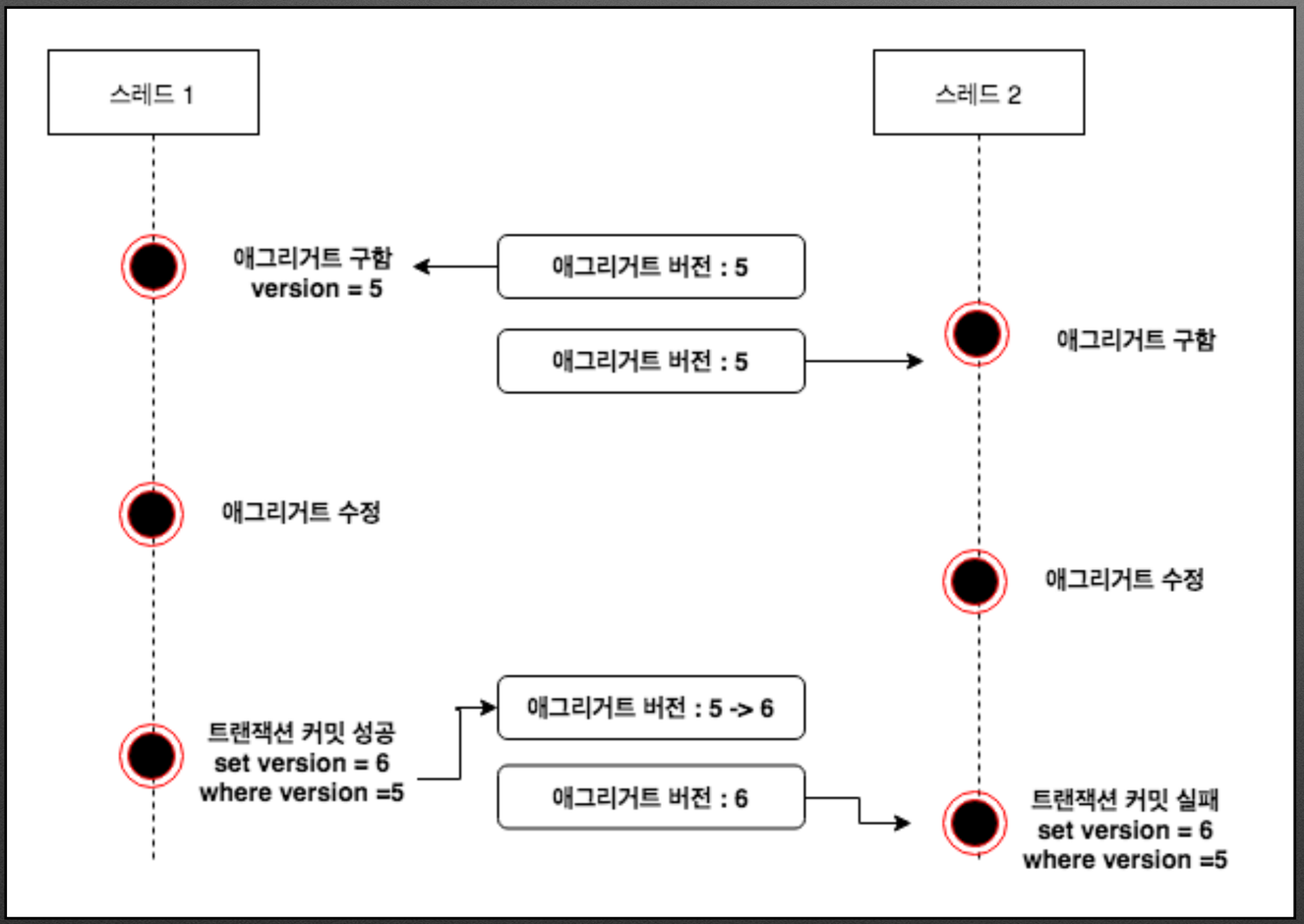
- 여기서 문제는 운영자가 배송지 정보를 조회하고 배송 상태로 변경하는 사이에 고객이 배송지를 변경한다는 것이다. 운영자는 고객이 변경하기 전에 배송지 정보를 이용해서 배송 준비를 한 뒤에 배송 상태로 변경하게 된다.
- 즉 배송 상태 변경 전에 배송지를 한 번 더 확인 하지 않으면 운영자는 다른 배송지로 물건을 발송하게 되고, 고객은 배송지를 변경했음에도 불구하고 엉뚱한 곳으로 주문한 물건을 받는 상황이 발생한다.
- 이 문제는 선점 잠금 방식으로 해결 할 수 없는데, 이 때 필요한 것이 비선점 잠금이다.
- 비선점 잠금 방식은 잠금을 해서 동시에 접근하는 것을 막는 대신 변경한 데이터를 실제 DBMS에서 반영하는 시점에 변경 가능 여부를 확인하는 방식이다.

비선점 잠금



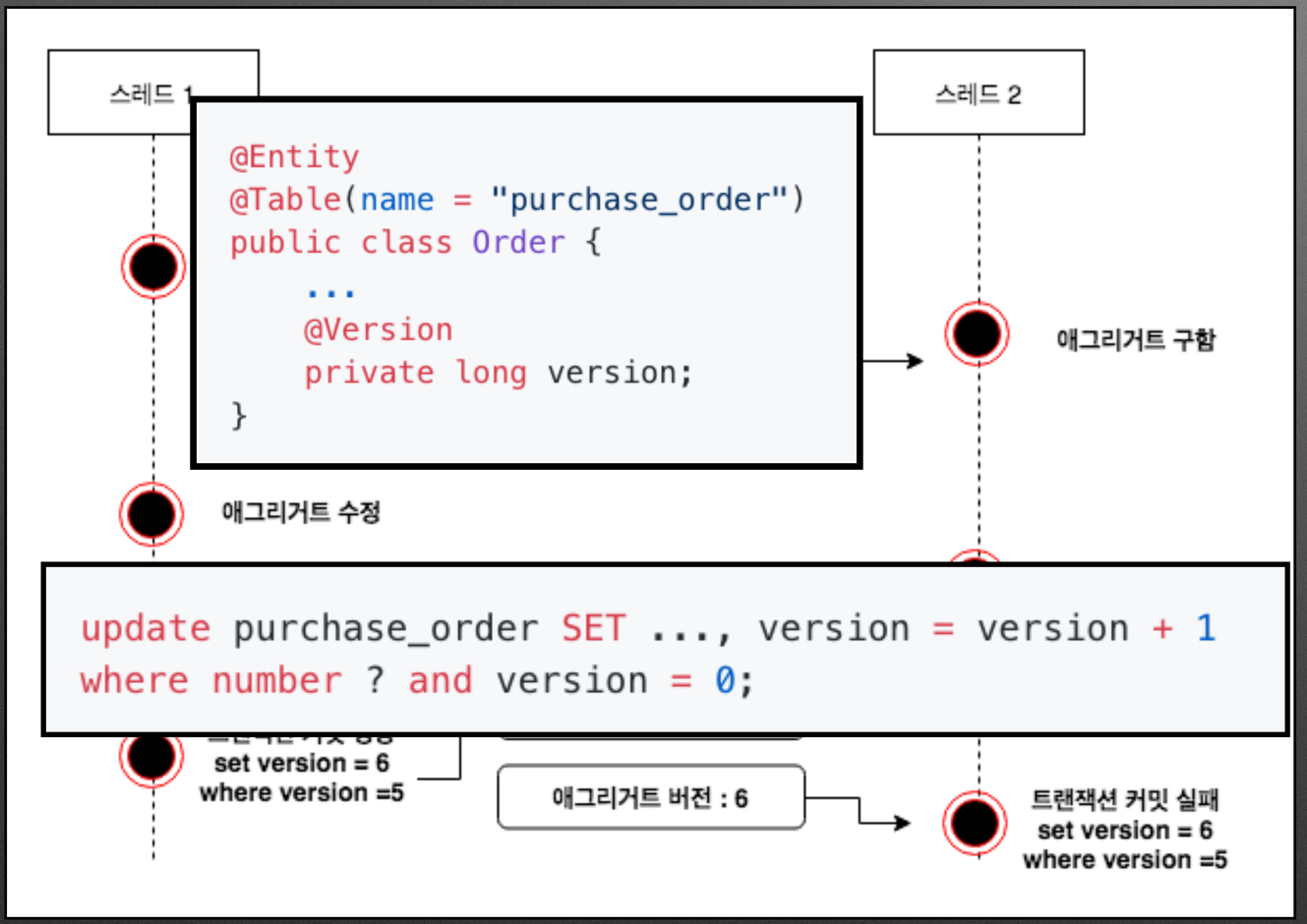
- 비선점 잠금을 구현하려면 애그리거트에 버전으로 사용할 숫자 타입의 프로퍼티를 추가해야한다. 애그리거트를 수정할 때마다 버전으로 사용할 프로피터 값이 1씩 증가하는데, 이때 다음과 같은 쿼리를 사용한다.
- 이 쿼리는 수정할 애그리거트와 매핑되는 테이블의 버전 값이 현재 애그리거트의 버전과 동일한 경우에만 데이터를 수정한다.
- 수정에 성공하면 버전 값을 1증가 시킨다. 따라서, 다른 트랜잭션이 먼저 데이터를 수정해서 버전 값이 바뀌면 데이터 수정에 실패하게 된다.

비선점 잠금을 이용한 트랜잭션 충돌 방지



1. 스레드 1 애그리거트 조회
2. 스레드 2 애그리거트 조회
3. 스레드 1 애그리거트 수정 시도 (스레드 2 보다 먼저 시도한다), 수정에 성공하고 버전은 6이 된다.
4. 스레드 2 애그리거트 수정 시도, 이미 애그리거트 버전이 6이므로 스레드2는 데이터 수정에 실패하게 된다

비선점 잠금을 이용한 트랜잭션 충돌 방지



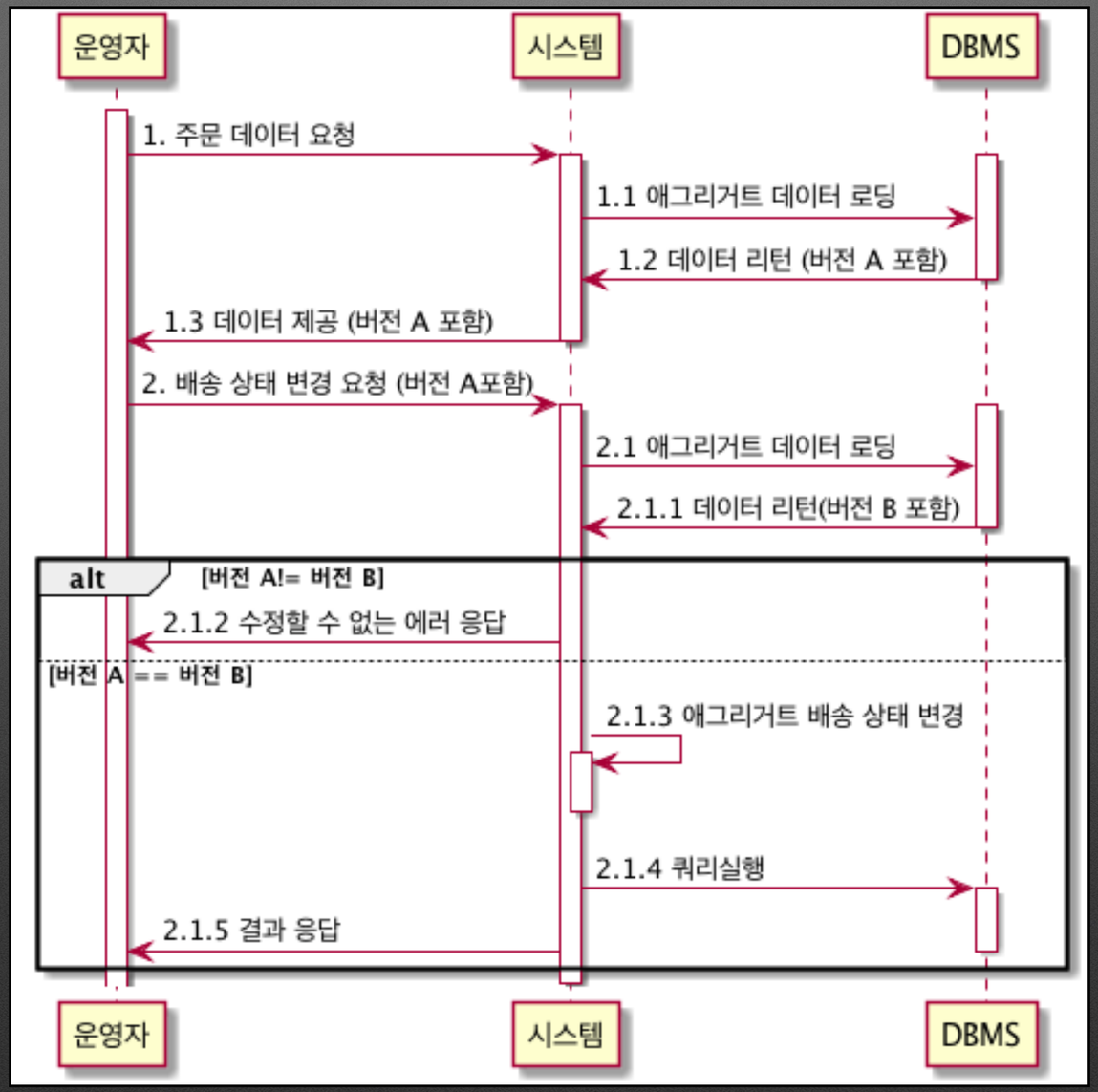
- JPA는 버전을 이용한 비선점 잠금을 기능을 지원한다. 다음과 같이 버전으로 사용할 필드에 **@Version 애노테이션을 붙이거 매핑되는 테이블 버전을 지정한 칼럼을 추가하기만 하면된다.**
- JPA는 엔티티가 변경되어 UPDATE 쿼리를 실행할 때 @Version에 명시한 필드를 이용해서 비선점 잠금 쿼리를 실행한다. 즉 **애그리거트 객체의 버전 10이면 UPDATE 쿼리를 실행할 때 다음과 같은 쿼리를 사용해서 버전 일치하는 경우에만 데이터를 수정한다.**

비선점 잠금을 이용한 트랜잭션 충돌 방지

```java
@Controller
public class OrderController {
    private ChangeShppingService changeShippingService;

    @PutMapping("/shpping")
    public String changeShipping(ChangeShippingsRequest changeReq){
        try {
            changeShppingService.changeShpping(changeReq);
            return "changeShppingSuccess";
        } catch (OptimistickLockingFailureException ex){
            // 누군가 먼저 같은 주문 애그리거트를 수정 했음으로
            // 트랜잭션 충돌이 일어났다는 메시지를 보여준다.
            return "changeShppingTxConflict";

        }
    }
}
```

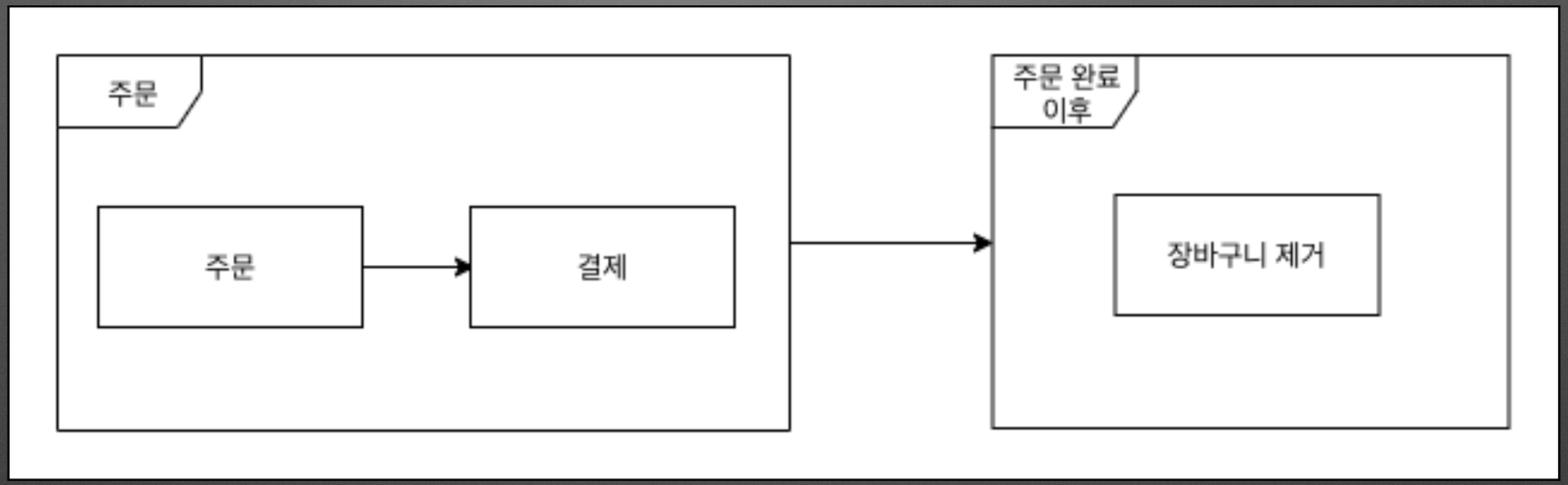
- 응용 서비스 버전에 대해 알 필요가 없다. Repository에 필요한 애그리거트를 구현하고 알맞은 기능을 실행하면 된다.
- 기능을 실행하는 과정에서 애그리거트의 데이터가 변경되면 JPA트랜잭션 종료 시점에 비선점 잠금을 위한 쿼리를 실행 한다.

비선점 트랜잭션 충돌 문제 해결 Flow



Application Event

Application Event



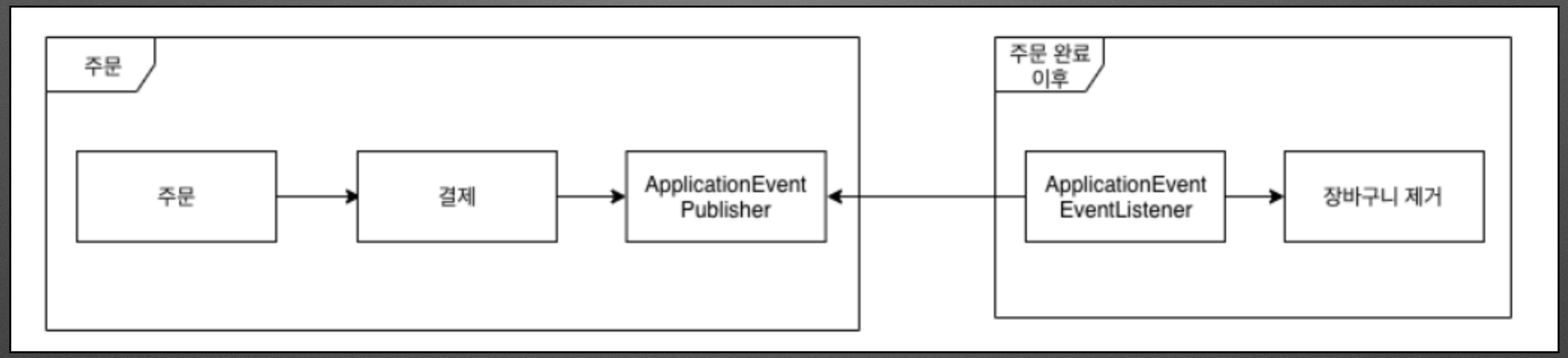
- 장바구니 제거 (장바구니에 등록된 상품 주문시)
- 제품 재고 변경
- 주문 완료 알람 (시스템, 유저 등등)

Application Event

```
class OrderService(
        ... // 필요한 의존성 주입, 강한 관계를 갖는 경우 필요한 의존성 들이 많아진다.
) {

    fun order(): Order {
        orderRepository.save(order) // 주문 저장
        stockService.adjust(order) // 재고 조정
        cartService.remove(order) // 장바구니 제거 (장바구니 상품중 주문 완료된 상품 제거)
        return order
    }
}
```

- OrderService 객체에 의존성 주입받아야 하는 항목들이 많아지는 것이 대표적인 외부와 결합관계를 갖게 되는 대표적인 신호입니다. 주문, 주문 완료 이후 로직이 추가될 때마다 OrderService의 변경이 야기됩니다. 이런 문제를 Application Event으로 쉽게 해결 할 수 있습니다.

Application Event



- 주문-결제 이후 결제 완료 이벤트를 발생시키고 주문 완료 이후 로직들은 Application Event Subscriber에서 구현하게 작성하는 구조를 갖을 수 있습니다.
- OrderService에 필요 의존성이 줄게 되고 자연스럽게 해당 서비스의 책임이 작아지게 되며 이벤트에 의해 느슨한 결합관계를 갖게 됩니다.

Application Event

```
@Service
class OrderService(
        private val orderRepository: OrderRepository,
        private val itemRepository: ItemRepository,
        private val eventPublisher: ApplicationEventPublisher
) {

    fun order(itemCode: List<String>): Order {
        val items = itemRepository.findByCodeIn(itemCode) // 상품(Item) 조회
        val orderItems: MutableList<OrderItem> = mutableListOf()
        items.mapTo(orderItems) { OrderItem(it.code, it.price, it.name) }
        val order = orderRepository.save(Order(orderItems))
        eventPublisher.publishEvent(OrderCompletedEvent(itemCode)) // 주문 완료 이벤트 발생,
        return order
    }
}
```

```
@Component
class OrderEventHandler(
        private val cartRepository: CartRepository
) {

    @EventListener
    fun orderEventHandler(event: OrderCompletedEvent) {
        cartRepository.deleteAllByCodes(codes = event.itemCodes)
    }

}
```

- 요청 받은 상품(Item)를 주문 하고, 주문 완료 event를 발생시키게 됩니다. 장바구니 제거 관련 코드는 존재하지 않고 주문 완료 이벤트만 발생시킵니다.
- 이벤트 리스너를 담당합니다. 주문 완료 이벤트가 발생하면 해당 리스너가 동작하게 되고 장바구니에서 주문 상품들을 제거하는 코드입니다.

JPA

JPA Setter 사용하지 않기

Setter 메소드는 의도를 갖기 힘들다.

```
public Account updateMyAccount(long id, AccountDto.MyAccountReq dto) {
    final Account account = findById(id);
    account.setAddress("value");
    account.setFistName("value");
    account.setLastName("value");
    return account;
}
```

- 위의 코드는 회원 정보의 성, 이름, 주소를 변경하는 코드로 여러 setter 메소드들이 나열돼있습니다.
- setter들은 회원 정보를 변경하기 위한 나열이라서 메소드들의 의도가 명확히 드러나지 않습니다.

Setter 메소드는 의도를 갖기 힘들다.

```
public static class MyAccountReq {
                private Address address;
                private String firstName;
                private String lastName;
}
```

```
public Account updateMyAccount(long id, AccountDto.MyAccountReq dto) {
    final Account account = findById(id);
    account.updateMyAccount(dto);
    return account;
}
// Account 도메인 클래스
public void updateMyAccount(AccountDto.MyAccountReq dto) {
    this.address = dto.getAddress();
    this.fistName = dto.getFistName();
    this.lastName = dto.getLastName();
}
```

- Account 도메인 클래스에 updateMyAccount 메소드를 통해서 회원정보업데이트를 진행했습니다. 위의 코드보다 의도가 명확히 드러납니다.
- 위는 MyAccountReq 클래스입니다. 회원 정보 수정에 필요한 값 즉 변경될 값에 대한 명확한 명세가 있어 DTO를 두는 것이 바람직합니다.

객체의 일관성을 유지하기 어렵다

```
public Account updateMyAccount(long id, AccountDto.MyAccountReq dto) {
    final Account account = findById(id);
    account.setEmail("value");
    return account;
}
```

- setter 메소드가 있을 때 객체에 언제든지 변경할 수 있게 됩니다.
- 회원 변경 메소드뿐만이 아니라 모든 곳에서 이메일 변경이 가능하게 됩니다.
- 변경이 불가능 한 항목에 setter 메서드를 두지 않는다는 방법도 있지만 관례로 setter는 모든 멤버필드에 대해서 만들기도 하거니와 실수 조금이라도 덜 할 수 있게 하는 것이 바람직한 구조라고 생각합니다.

JPA 연관관계 설정

OneToOne 관계 설정 팁

```java
public class Coupon {
    @Id
    @GeneratedValue
    private long id;

    @Column(name = "discount_amount")
    private double discountAmount;

    @Column(name = "use")
    private boolean use;

    @OneToOne()
    private Order order;
}

public class Order {
    @Id
    @GeneratedValue
    private long id;

    @Column(name = "price")
    private double price;

    @OneToOne
    @JoinColumn()
    private Coupon coupon;
}
```

- 주문과 쿠폰 엔티티가 있다.
- 주문 시 쿠폰을 적용해서 할인받을 수 있다.
- 주문과 쿠폰 관계는 1:1 관계 즉 OneToOne 관계이다.
- 주의 깊게 살펴볼 내용은 다음과 같습니다.

- 외래 키는 어디 테이블에 두는 것이 좋은가?
- 양방향 연관 관계 편의 메소드
- 제약 조건으로 인한 안정성 및 성능 향상

외래 키는 어디 테이블에 두는 것이 좋은가?

```
// Order가 연관관계의 주인일 경우
@OneToOne
@JoinColumn(name = "coupon_id", referencedColumnName = "id")
private Coupon coupon;

@OneToOne(mappedBy = "coupon")
private Order order;

// coupon이 연관관계의 주인일 경우
@OneToOne(mappedBy = "order")
private Coupon coupon;

@OneToOne
@JoinColumn(name = "order_id", referencedColumnName = "id")
private Order order;
```

- 일대다 관계에서는 다 쪽에서 외래 키를 관리 하게 되지만 상대적으로 일대일 관계 설정에는 외래 키를 어느 곳에 두어야 하는지를 생각을 해야 합니다.
- JPA 상에서는 외래 키가 갖는 쪽이 연관 관계의 주인이 되고 연관 관계의 주인만이 데이터베이스 연관 관계와 매핑되고 외래 키를 관리(등록, 수정, 삭제)할 수 있기 때문입니다.

Order가 주인일 경우 장점 : INSERT SQL이 한번 실행

```
Tests passed: 1 of 1 test – 148 ms
OrderServiceTes 148 ms
  order_쿠폰할인 148 ms
Hibernate: drop table delivery if exists
Hibernate: drop table delivery_log if exists
Hibernate: drop table orders if exists
Hibernate: create table account (id bigint generated by default as identi
Hibernate: create table coupon (id bigint generated by default as identit
Hibernate: create table delivery (id bigint generated by default as ident
Hibernate: create table delivery_log (id bigint generated by default as i
Hibernate: create table orders (id bigint generated by default as identit
Hibernate: alter table account add constraint UK_q0uja26qgu1atulenwup9rxy
Hibernate: alter table delivery_log add constraint FKfs49km0ea809mth3oq4s
Hibernate: alter table orders add constraint FKa5ei0aklq6wrjl8vrr7ied3bx
Hibernate: select coupon0_.id as id1_1_0_, coupon0_.discount_amount as di
Hibernate: insert into orders (id, coupon_id, price) values (null, ?, ?)
couponId : 1
Hibernate: drop table account if exists
Hibernate: drop table coupon if exists
Hibernate: drop table delivery if exists
Hibernate: drop table delivery_log if exists
Hibernate: drop table orders if exists

Process finished with exit code 0
```

- order 테이블에 coupon_id 칼럼을 저장하기 때문에 주문 SQL은 한 번만 실행됩니다.
- 반면에 coupon이 연관 관계의 주인일 경우에는 coupon에 order의 외래 키가 있으니 order INSERT SQL 한 번, coupon 테이블에 order_id 칼럼 업데이트 쿼리 한번 총 2번의 쿼리가 실행됩니다.

```
// order가 연관 관계의 주인일 경우 SQL
insert into orders (id, coupon_id, price) values (null, ?, ?)

//coupon이 연관 관계의 주인일 경우 SQL
insert into orders (id, price) values (null, ?)
update coupon set discount_amount=?, order_id=?, use=? where id=?
```

Order가 주인일 경우 단점 : 연관 관계 변경 시 취약

```java
public class Coupon {
    @Id
    @GeneratedValue
    private long id;

    @Column(name = "discount_amount")
    private double discountAmount;

    @Column(name = "use")
    private boolean use;

    @OneToOne()
    private Order order;
}

public class Order {
    @Id
    @GeneratedValue
    private long id;

    @Column(name = "price")
    private double price;

    @OneToOne
    @JoinColumn()
    private Coupon coupon;
}
```

- 기존 요구사항은 주문 한 개에 쿠폰은 한 개만 적용이 가능 했기 때문에 OneToOne 연관 관계를 맺었지만 하나의 주문에 여러 개의 쿠폰이 적용되는 기능이 추가되었을 때 변경하기 어렵다는 단점이 있습니다.
- order 테이블에 coupon_id 칼럼을 갖고 있어서 여러 개의 쿠폰을 적용하기 위해서는 coupon 테이블에서 order_id 칼럼을 가진 구조로 변경 해야 합니다.

  OneToMany 관계에서는 연관 관계의 주인은 외래 키를 갖는 엔티티가 갖는 것이 바람직합니다. 비즈니스 로직 변경은 어려운 게 없으나 데이터베이스 칼럼들을 이전 해야 하기 때문에 실제 서비스 중인 프로젝트에는 상당히 골치 아프게 됩니다.

연관 관계의 주인 설정

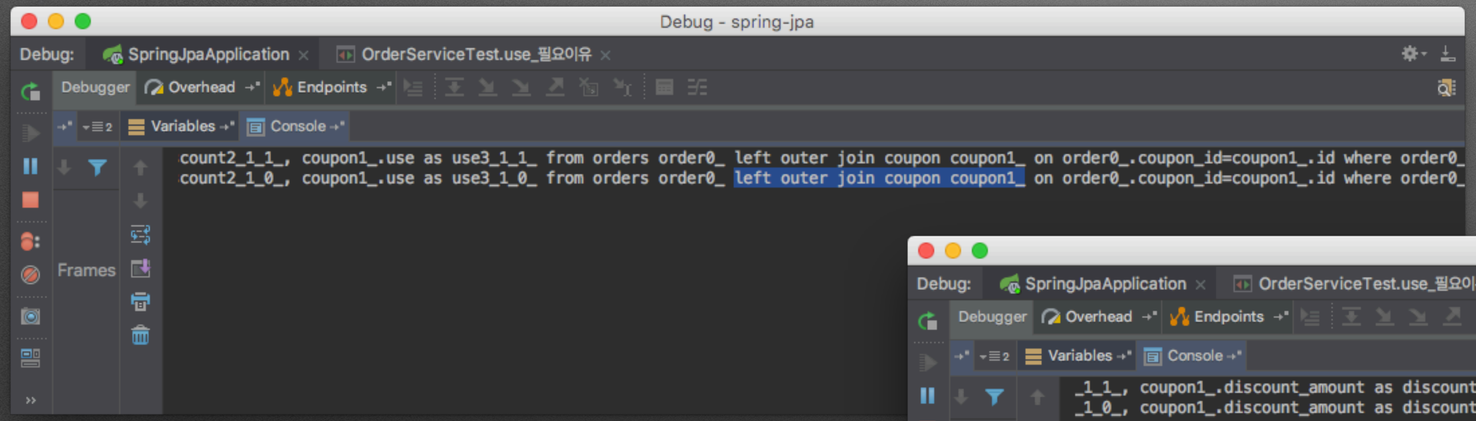
- OneToOne 관계를 맺으면 외래 키를 어디에 둘 것인지, 즉 연관 관계의 주인을 어디에 둘 것인지는 많은 고민이 필요 합니다. 제 개인적인 생각으로는 OneToMany로 변경될 가능성이 있는지를 판단하고 변경이 될 가능성이 있다고 판단되면 Many가 될 엔티티가 관계의 주인이 되는 것이 좋다고 봅니다. 또 애초에 OneToMany를 고려해서 초기 관계 설정을 OneToMany로 가져가는 것도 좋다고 생각합니다.
- 연관 관계가 정말 OneToOne 관계인지 깊은 고민이 필요하고 해당 도메인에 대한 지식도 필요 하다고 생각합니다. 예를 들어 개인 송금 관계에서 입금 <-> 출금 관계를 가질 경우 반드시 하나의 입금 당 하나의 출금을 갖게 되니 이것은 OneToOne 관계로 맺어가도 무리가 없다고 판단됩니다. (물론 아닌 때도 있습니다. 그래서 해당 도메인에 대한 지식이 필요 한다고 생각합니다)
- 주인 설정이라고 하면 뭔가 더 중요한 것이 주인이 되어야 할 거 같다는 생각이 들지만 연관 관계의 주인이라는 것은 외래 키의 위치와 관련해서 정해야 하지 해당 도메인의 중요성과는 상관관계가 없습니다.

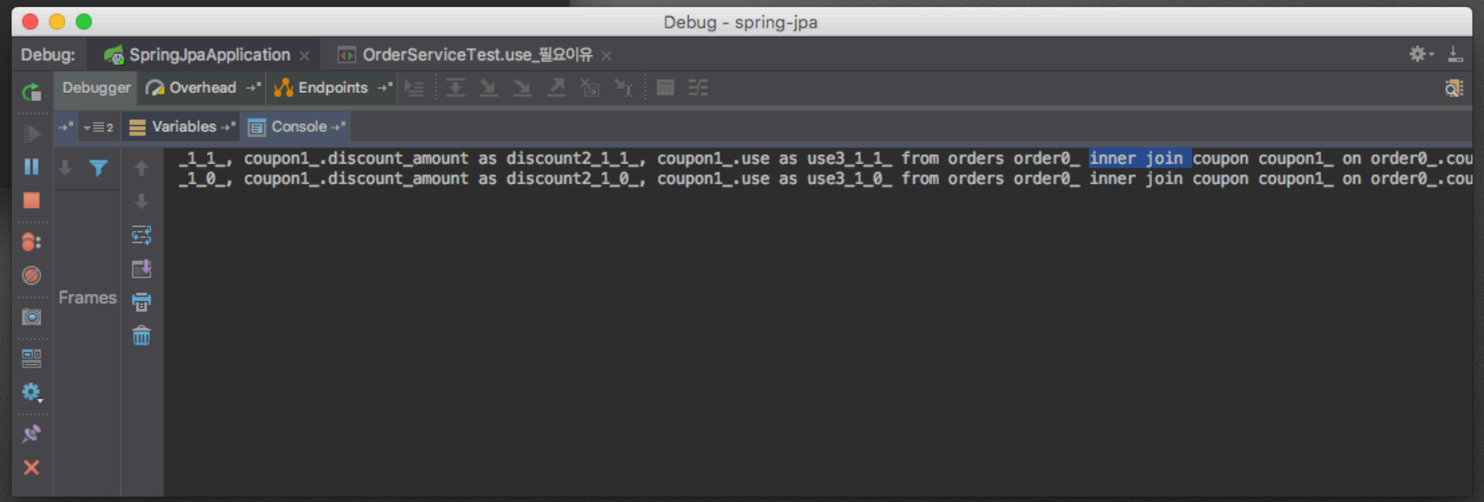
제약 조건으로 인한 안정성 및 성능 향상

```
public class Order {
    ...

    @OneToOne
    @JoinColumn(name = "coupon_id", referencedColumnName = "id", nullable = false)
    private Coupon coupon;
}
```

- 모든 주문에 할인 쿠폰이 적용된다면 @JoinColumn의 nullable 옵션을 false로 주는 것이 좋습니다. NOT NULL 제약 조건을 준수해서 안전성이 보장됩니다.
- JPA 기반으로 DDL을 작성하지 않더라도 해당 도메인의 Spec으로 표시하는 편이 좋다고 생각합니다.

제약 조건으로 인한 안정성 및 성능 향상





- 외래 키에 NOT NULL 제약 조건을 설정하면 값이 있는 것을 보장합니다. 따라서 JPA는 이때 내부조인을 통해서 SQL을 만들어 주고 이것은 외부 조인보다 성능과 최적화에 더 좋습니다.
- 물론 모든 경우에 적용할 수는 없고 반드시 외래 키가 NOT NULL인 조건에만 사용할 수 있습니다. 예를 들어 쿠폰과 회원 연관관계가 있을 때 쿠폰은 반드시 회원의 외래 키를 참조하고 있어야 합니다. 이런 경우 유용하게 사용할 수 있습니다.

양방향 연관관계 편의 메소드

```java
// Order가 연관관계의 주인일 경우 예제
class Coupon {
    ...
    // 연관관계 편의 메소드
    public void use(final Order order) {
        this.order = order;
        this.use = true;
    }
}

class Order {
    private Coupon coupon; // (1)
    ...
    // 연관관계 편의 메소드
    public void applyCoupon(final Coupon coupon) {
        this.coupon = coupon;
        coupon.use(this);
        price -= coupon.getDiscountAmount();
    }
}

// 주문 생성시 1,000 할인 쿠폰 적용
public Order order() {
    final Order order = Order.builder().price(10_000).build(); // 10,000 상품주문
    Coupon coupon = couponService.findById(1); // 1,000 할인 쿠폰
    order.applyCoupon(coupon);
    return orderRepository.save(order);
}
```

- 양방향 연관 관계일 경우 위처럼 연관 관계 편의 메소드를 작성하는 것이 좋습니다. 위에서 말했듯이 연관 관계의 주인만이 외래 키를 관리 할 수 있으니 applyCoupon 메소드는 이해하는데 어렵지 않습니다.

양방향 연관관계 편의 메소드가 필요한 이유

```
public void use(final Order order) {
//  this.order = order; 해당코드를 주석했을 때 테스트 코드
    this.use = true;
}
@Test
public void use_메서드에_order_set_필요이유() {
    final Order order = orderService.order();
    assertThat(order.getPrice(), is(9_000D)); // 1,000 할인 적용 확인
    final Coupon coupon = order.getCoupon();
    assertThat(coupon.getOrder(), is(notNullValue())); // 해당 검사는 실패한다.
}
```

- order를 바인딩하는 코드를 주석하고 해당 코드를 돌려보면 실패하게 됩니다.
- 일반적으로 생각했을 때 order 생성 시 1,000할인 쿠폰을 적용했기 때문에 해당 쿠폰에도 주문 객체가 들어갔을 거로 생각할 수 있습니다. 하지만 위의 주석시킨 코드가 그 기능을 담당했기 때문에 쿠폰 객체의 주문 값은 null인 상태입니다.
- 즉 순수한 객체까지 고려한 양방향 관계를 고려하는 것이 바람직하고 그것이 안전합니다.

OneToMany 관계 설정

```java
@Entity
public class Delivery {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Embedded
    private Address address;

    @OneToMany(mappedBy = "delivery", cascade = CascadeType.PERSIST, orphanRemoval = true, fetch = Fet
    private List<DeliveryLog> logs = new ArrayList<>();
}

@Entity
public class DeliveryLog {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false, updatable = false)
    private DeliveryStatus status;

    @ManyToOne
    @JoinColumn(name = "delivery_id", nullable = false, updatable = false)
    private Delivery delivery;
}
```

- 배송이 있고 배송의 상태를 갖는 배송 로그가 있습니다.
- 배송과 배송 상태는 1:N 관계를 갖는다.
- 배송은 배송 상태를 1개 이상 반드시 갖는다

Delivery 저장

```java
class Delivery {
    public void addLog(DeliveryStatus status) {
        this.logs.add(DeliveryLog.builder()
                .status(status)
                .delivery(this) // this를 통해서 Delivery를 넘겨준다.
                .build());
    }
}

class DeliveryLog {
    public DeliveryLog(final DeliveryStatus status, final Delivery delivery) {
        this.delivery = delivery;
    }
}

class DeliveryService {
    public Delivery create(DeliveryDto.CreationReq dto) {
        final Delivery delivery = dto.toEntity();
        delivery.addLog(DeliveryStatus.PENDING);
        return deliveryRepository.save(delivery);
    }
}
```

- Delivery가 시작되면 DeliveryLog는 반드시 PENDDING이어야 한다고 가정했을 경우 편의 메소드를 이용해서 두 객체에 모두 필요한 값을 바인딩시켜줍니다.

Delivery 저장

```java
class Delivery {
    public void addLog(DeliveryStatus status) {
        this.logs.add(DeliveryLog.builder()
                .status(status)
                .delivery(this) // this를 통해서 Delivery를 넘겨준다.
                .build());
```

CaseCade PERSIST 설정

```
// cascade 없는 경우
Hibernate: insert into delivery (id, address1, address2, zip, created_at, update_at) values (null, ?, ?,

// cascade PERSIST 설정 했을 경우
Hibernate: insert into delivery (id, address1, address2, zip, created_at, update_at) values (null, ?, ?,
Hibernate: insert into delivery_log (id, created_at, update_at, delivery_id, status) values (null, ?, ?,
```

```java
class DeliveryService {
    public Delivery create(DeliveryDto.CreationReq dto) {
        final Delivery delivery = dto.toEntity();
        delivery.addLog(DeliveryStatus.PENDING);
        return deliveryRepository.save(delivery);
    }
}
```

- CaseCade PERSIST을 통해서 Delivery 엔티티에서 DeliveryLog를 생성할수 있게 설정합니다.
- CaseCade PERSIST가 없을 때 실제 객체에는 저장되지만, 영속성 있는 데이터베이스에 저장에 필요한 insert query가 동작하지 않습니다.

고아 객체 (orphanRemoval) : DeliveryLog 삭제

```
public Delivery removeLogs(long id) {
    final Delivery delivery = findById(id);
    delivery.getLogs().clear(); // DeloveryLog 전체 삭제
    return delivery; // 실제 DeloveryLog 삭제 여부를 확인하기 위해 리턴
}
```

```
// delete SQL
Hibernate: delete from delivery_log where id=?
```

- JPA는 부모 엔티티와 연관 관계가 끊어진 자식 엔티티를 자동으로 삭제하는 기능을 제공하는데 이것을 고아 객체 제거라 합니다.
- 이 기능을 사용해서 부모 엔티티의 컬렉션에서 자식 엔티티의 참조만 제거하면 자식 엔티티가 자동으로 삭제돼서 개발의 편리함이 있습니다.
- Delivery 객체를 통해서 DeliveryLog를 위처럼 직관적으로 삭제 할 수 있습니다.

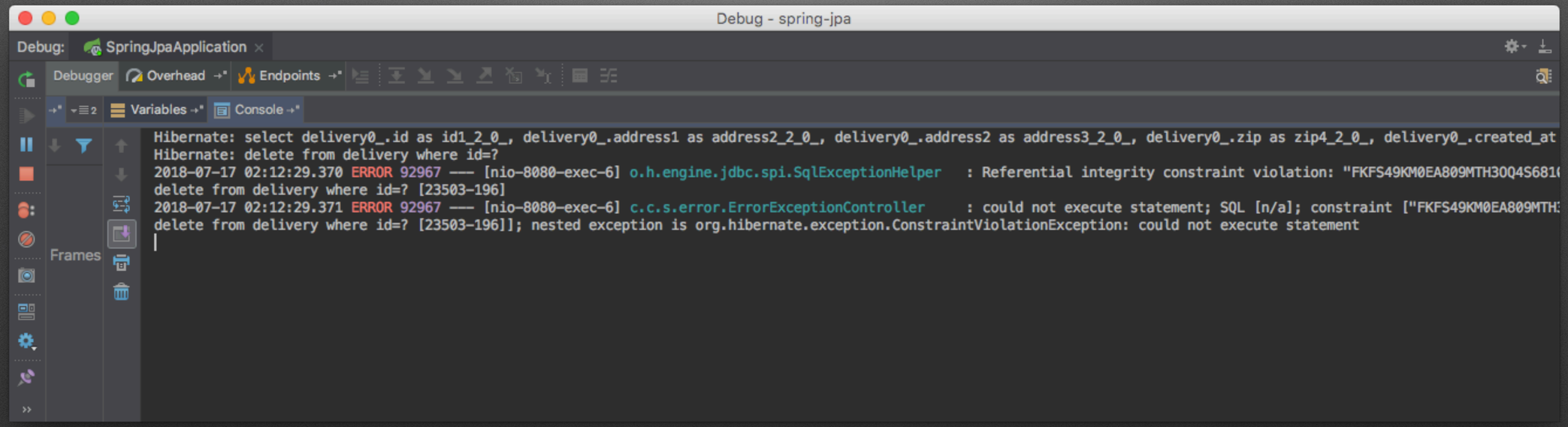
고아 객체 (orphanRemoval) : Delivery 삭제

```
public void remove(long id){
    deliveryRepository.delete(id);
}
```

```
// delete SQL
Hibernate: delete from delivery_log where id=?
Hibernate: delete from delivery where id=?
```

- delivery, deliverylog 참조 관계를 맺고 있어 Delivery만 삭제할 수 없습니다.
- delete SQL을 보시다시피 delivery_log 부터 제거 이후 delivery를 제거하는 것을 알 수 있습니다.
- 이처럼 orphanRemoval 속성으로 더욱 쉽게 삭제 할 수 있습니다.

고아 객체 (orphanRemoval) : 설정이 없는 경우



- DeliveryLog 삭제 같은 경우에는 실제 객체에서는 clear() 메서드로 DeliveryLog가 삭제된 것처럼 보이지만 영속성 있는 데이터를 삭제하는 것은 아니기에 해당 Delivery를 조회하면 DeliveryLog가 그대로 조회됩니다.
- 실수하기 좋은 부분이기에 반드시 삭제하고 조회까지 해서 데이터베이스까지 확인하시는 것을 권장해 드립니다

Embedded를 적극 활용

자료형의 통일

```
class Account {
    // 단순 String
    @email
    @Column(name = "email", nullable
    private String email;

    // Email 자료형
    @Embedded
    private Email email;
}

class Email {
    @Email
    @Column(name = "email", nullable = false, unique = true)
    private String value;
}
```

```
public Account findByEmail(final Email email) { //단순 문자열일 경우 (final String email)
    final Account account = accountRepository.findByEmail(email);
    if (account == null) throw new AccountNotFoundException(email);
    return account;
}
```

- String 자료형에서 Email 자료형으로 통일이 됩니다. 자료형이 통일되면 많은 더욱 안전성이 높아지는 효과가 있습니다.
- 이메일로 회원을 조회 할 때 단순 문자열일 경우에는 굳이 이메일 형식을 맞추지 않고도 단순 문자열을 통해서 조회할 수 있습니다. 이것은 편하게 느껴질지는 모르나 안전성에는 좋다고 생각하지 않습니다.
- 위처럼 단순 조회용뿐만이 아니라 Email에 관련된 모든 자료형을 단순 String에서 Email로 변경함으로써 얻을 수 있는 이점은 많습니다.

풍부한 객체 (Rich Object)

```
public class Email {
    ...
    public String getHost() {
        int index = value.indexOf("@");
        return value.substring(index);
    }

    public String getId() {
        int index = value.indexOf("@");
        return value.substring(0, index);
    }
}
```

- 이메일 아이디와 호스트값을 추출해야 하는 기능이 필요해질 경우 기존 String 자로 형일 경우에는 해당 로직을 Account 도메인 객체에 추가하든, 유틸성 클래스에 추가하든 해야 합니다.
- 도메인 객체에 추가할 때는 Account 도메인 클래스가 갖는 책임들이 많아집니다. 또 이메일은 어디서든지 사용할 수 있는데 Account 객체에서 이 기능을 정의하는 것은 올바르지 않습니다.
- 유틸성 클래스에 추가하는 것 또한 좋지 않아 보입니다. 일단 유틸성 클래스에 해당 기능이 있는지 알아봐야 하고 기능이 있음에도 불구하고 그것을 모르고 추가하여 중복 코드가 발생하는 일이 너무나도 흔하게 발생합니다.
- 이것을 Email 형으로 빼놓았다면 아래처럼 Email 객체를 사용하는 곳 어디든지 사용할 수 있습니다. 해당 기능은 Email 객체가 해야 하는 일이고 또 그 일을 가장 잘할 수 있는 객체입니다. 또한 코드가 아주 이해하기 쉽게 됩니다. 객체의 기능이 풍부해집니다.

재사용성

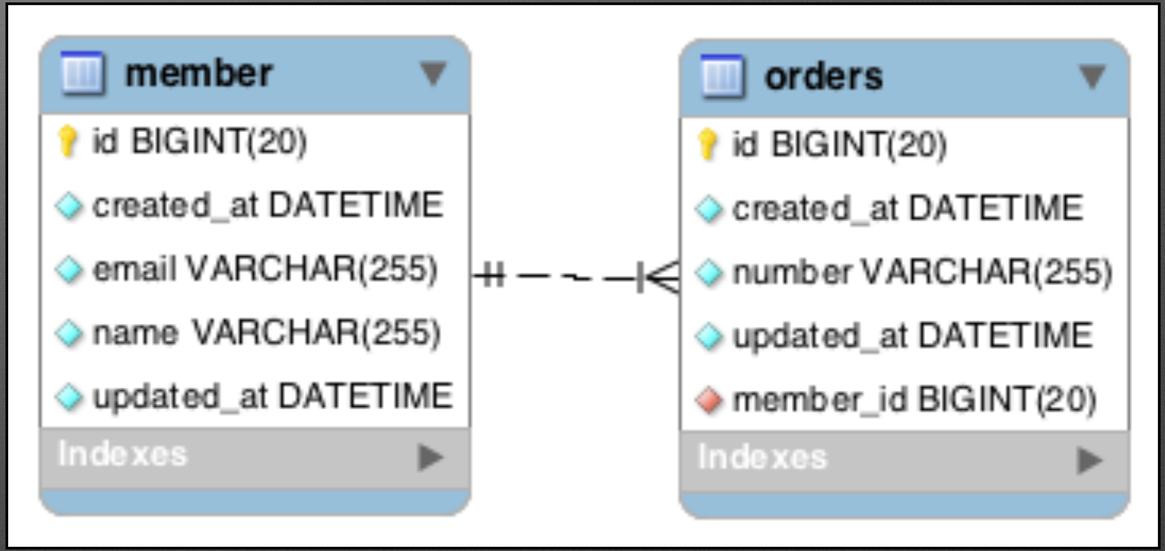
```
class Remittance {
    //자료형이 없는 경우
    @Column(name = "send_amount") private double sendAamount;
    @Column(name = "send_country") private String sendCountry;
    @Column(name = "send_currency") private String sendCurrency;

    @Column(name = "receive_amount") private double receiveAamount;
    @Column(name = "receive_country") private String receiveCountry;
    @Column(name = "receive_currency") private String receiveCurrency;

    //Money 자료형
    private Money snedMoney;
    private Money receiveMoney;
}
class Money {
    @Column(name = "amount", nullable = false, updatable = false) private double amount;
    @Column(name = "country", nullable = false, updatable = false) private Country country;
    @Column(name = "currency", nullable = false, updatable = false) private Currency currency;
}
```

- 위처럼 Money라는 자료형을 두고 금액, 나라, 통화를 두면 도메인을 이해하는데 한결 수월할 뿐만 아니라 수많은 곳에서 재사용 할 수 있습니다. 사용자에게 해당 통화로 금액을 보여줄 때 소숫자리 몇 자리로 보여줄 것인지 등등 핵심 도메인일수록 재사용성을 높여 중복 코드를 제거하고 응집력을 높일 수 있습니다.

JPA N+1

ERD

```
@Entity
@Table(name = "member")
class Member private constructor() {

    ...
    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY)
    var orders: Set<Order> = emptySet()
        private set
}

@Entity
@Table(name = "orders")
class Order private constructor() {
    ...

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "member_id", nullable = false, updatable = false)
    lateinit var member: Member
        private set
```



즉시 로딩 N+1

```
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size: 10
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size: 10
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size: 10
Hibernate:
    select
```

```
@Test
internal fun `즉시 로딩 n+1`() {
    // fetch = FetchType.EAGER 의 경우
    val members = memberRepository.findAll()
}
```

지연로딩과 N+1

```
  activeProfiles = '{}', propertySourceLocations = '{}', propertySc
  .PropertyMappingContextCustomizer@0, org.springframework.boot.tes
  .DuplicateJsonObjectContextCustomizerFactory$DuplicateJsonObjectC
  'src/main/webapp', contextLoader = 'org.springframework.boot.test
  .ServletTestExecutionListener.populatedRequestContextHolder' -> t
Hibernate:
    /* select
        generatedAlias0
    from
        Member as generatedAlias0 */ select
            member0_.id as id1_0_,
            member0_.created_at as created_2_0_,
            member0_.email as email3_0_,
            member0_.name as name4_0_,
            member0_.updated_at as updated_5_0_
        from
            member member0_
2019-10-26 23:48:34.175  INFO 1245 --- [           main] o.s.t.c.t
   경우$jpa_n_plus_1@MemberJpaTest, testException = [null], mergedCo
  activeProfiles = '{}', propertySourceLocations = '{}', propertySc
  .PropertyMappingContextCustomizer@0, org.springframework.boot.tes
  .DuplicateJsonObjectContextCustomizerFactory$DuplicateJsonObjectC
  'src/main/webapp', contextLoader = 'org.springframework.boot.test
  .ServletTestExecutionListener.populatedRequestContextHolder' -> t
```

```
@Test
internal fun `지연 로딩 n+1`() {
    // fetch = FetchType.LAZY 의 경우
    val members = memberRepository.findAll()
}
```

지연로딩과 N+1

```
'org.springframework.test.context.web.ServletTestExecutionListener.resetReq
Hibernate:
    /* select
        generatedAlias0
    from
        Member as generatedAlias0 */ select
            member0_.id as id1_0_,
            member0_.created_at as created_2_0_,
            member0_.email as email3_0_,
            member0_.name as name4_0_,
            member0_.updated_at as updated_5_0_
        from
            member member0_
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size : 10
2019-10-27 00:24:26.266  INFO 2388 --- [          main] o.s.t.c.transaction.
```

```
@Test
internal fun `지연로딩인 n+1`() {
    val members = memberRepository.findAll()

    // 회원 한명에 대한 조회는 문제가 없다
    val firstMember = members[0]
    println(firstMember.orders.size)
}
```

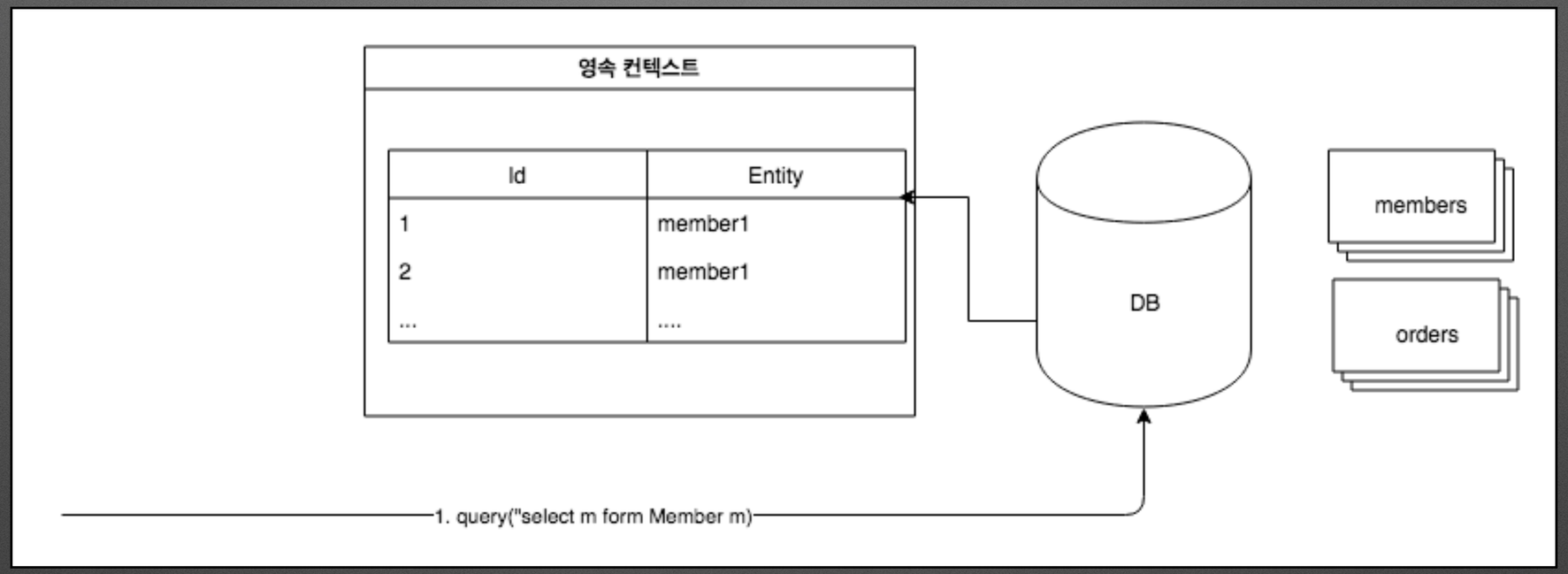
지연로딩과 N+1

```
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size: 10
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size: 10
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size: 10
Hibernate:
    select
```

```
@Test
internal fun `지연로딩인 n+1`() {
    val members = memberRepository.findAll()

    // 회원 한명에 대한 조회는 문제가 없다
    val firstMember = members[0]
    println("order size : ${firstMember.orders.size}")

    // 조회한 모든 회원에 대해서 조회하는 경우 문제 발생
    for(member in members){
        println("order size: ${member.orders.size}")
    }
}
```

- 모든 Member에 대해서 주문을 조회하는 경우 N+1문제가 발생합니다.

N+1원인



- JPQL 특징이 있습니다. findById() 같은 경우에는 엔티티를 영속성 컨텍스트에서 먼저 찾고 영속성 컨텍스트에 없는 경우에 데이터베이스에 찾는 반면 JPQL은 항상 데이터베이스에 SQL을 실행해서 결과를 조회합니다. 그리고 아래와 같은 작업을 진행하게 됩니다.

1. JPQL을 호출하면 데이터베이스에 우선 적으로 조회한다.
2. 조회한 값을 영속성 컨텍스트에 저장한다.
3. 영속성 컨텍스트에 조회할 때 이미 존재하는 데이터가 있다면(같은 영속성 컨텍스트에서 이미 조회한 유저가 있는 경우) 데이터를 버린다.

N+1원인

즉시 로딩인 경우

```
val members = memberRepository.findAll()
```

JPQL에서 동작한 쿼리를 통해서 members에 데이터가 바인딩 됩니다. **그 이후 JPA에서는 글로벌 패치 전략(즉시 로딩)을 받아들여 해당 member의 연관관계인 order에대해서 추가적인 레이지 로딩이 진행되어 N+1을 발생시킵니다.**

지연 로딩인 경우

```
val members = memberRepository.findAll()
```

JPQL에서 동작한 쿼리를 통해서 members에 데이터가 바인딩 됩니다. **JPA가 글로벌 패치 전략을 받아들이지만 지연 로딩이기 때문에 추가적인 SQL을 발생시키지 않습니다.** 하지만 위에서 본 예제처럼 레이지로 딩으로 추가적인 작업을 진행하게되면 결국 N+1 문제가 발생하게 됩니다.

- **JPQL을 실행하면 JPA는 이것을 분석해서 SQL을 생성합니다. JPQL 입장에서는 즉시 로딩, 지연 로딩과 같은 글로벌 패치 전략을 무시하고 JPQL만 사용해서 SQL을 생성합니다.**

해결 방법 : Batch Size

```
@Entity
@Table(name = "member")
class Member private constructor() {
    ...

    @BatchSize(size = 5) // Batch size를 지정한다
    @OneToMany(mappedBy = "member", fetch = FetchType.EAGER) // 즉시 로딩으로 설정
    var orders: List<Order> = emptyList()
        private set
}
```

```
Hibernate:
    /* load one-to-many com.example.jpanplus1.member.Member.orders */ select
        orders0_.member_id as member_i5_1_1_,
        orders0_.id as id1_1_1_,
        orders0_.id as id1_1_0_,
        orders0_.created_at as created_2_1_0_,
        orders0_.member_id as member_i5_1_0_,
        orders0_.number as number3_1_0_,
        orders0_.updated_at as updated_4_1_0_
    from
        orders orders0_
    where
        orders0_.member_id in (
            ?, ?, ?, ?, ?
        )
Hibernate:
    /* load one-to-many com.example.jpanplus1.member.Member.orders */ select
        orders0_.member_id as member_i5_1_1_,
        orders0_.id as id1_1_1_,
        orders0_.id as id1_1_0_,
        orders0_.created_at as created_2_1_0_,
        orders0_.member_id as member_i5_1_0_,
        orders0_.number as number3_1_0_,
        orders0_.updated_at as updated_4_1_0_
    from
        orders orders0_
    where
        orders0_.member_id in (
            ?, ?, ?, ?, ?
        )
```

- @BatchSize(size = 5) 에노테이션을 통해서 설정한 size 만큼 데이터를 미리 로딩 한다. 즉 연관된 엔티티를 조회할때 size 만큼 where in 쿼리를 통해서 조회하게되고 size를 넘어가게 되면 추가로 where in 쿼리를 진행합니다.
- spring.jpa.properties.hibernate.default_batch_fetch_size=1000 설정으로 properties 설정으로 글로벌하게 설정 가능

해결 방법 : Fetch 조인 사용

```
interface MemberRepository : JpaRepository<Member, Long> {

    @Query(
            "select m from Member m left join fetch m.orders"
    )
    fun findAllWithFetch(): List<Member>
}


@Test
internal fun `페치 조인 사용`() {
    val members = memberRepository.findAllWithFetch()

    // 조회한 모든 회원에 대해서 조회하는 경우에도 N+1 문제가 발생하지 않음
    for (member in members) {
        println("order size: ${member.orders.size}")
    }
}
```

```
Hibernate:
    /* select
        m
    from
        Member m
    left join
        fetch m.orders */ select
            member0_.id as id1_0_0_,
            orders1_.id as id1_1_1_,
            member0_.created_at as created_2_0_0_,
            member0_.email as email3_0_0_,
            member0_.name as name4_0_0_,
            member0_.updated_at as updated_5_0_0_,
            orders1_.created_at as created_2_1_1_,
            orders1_.member_id as member_i5_1_1_,
            orders1_.number as number3_1_1_,
            orders1_.updated_at as updated_4_1_1_,
            orders1_.member_id as member_i5_1_0__,
            orders1_.id as id1_1_0__
        from
            member member0_
        left outer join
            orders orders1_
                on member0_.id=orders1_.member_id
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
```

- 가장 많이 사용하는 방법인 fetch을 통해서 조인 쿼리를 진행하는 것입니다. Fetch 키워드를 사용하게 되면 연관된 엔티티나 컬렉션을 한 번에 같이 조회할 수 있습니다.
- 즉 Fetch 조인을 사용하게 되면 연관된 엔티티는 프록시가 아닌 실제 엔티티를 조회하게 되므로 연관관계 객체까지 한 번의 쿼리로 가져올 수 있습니다.

Fetch 조인과 일반 조인 차이

```
interface MemberRepository : JpaRepository<Member, Long> {

    @Query(
            "select m from Member m join m.orders"
    )
    fun findAllWithFetch(): List<Member>
}
```

```
@Test
internal fun `페치 조인 키워드 제거`() {
    val members = memberRepository.findAllWithFetch() // 패치 타입 Lazy 경우

    // 패치 조인하지 않은 상태에서는 N+1 문제 발생
    for (member in members) {
        println("order size: ${member.orders.size}")
    }
}
```

```
Hibernate:
    /* select
        m
    from
        Member m
    join
        m.orders */ select
            member0_.id as id1_0_,
            member0_.created_at as created_2_0_,
            member0_.email as email3_0_,
            member0_.name as name4_0_,
            member0_.updated_at as updated_5_0_
        from
            member member0_
        inner join
            orders orders1_
                on member0_.id=orders1_.member_id
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
order size: 10
Hibernate:
    select
        orders0_.member_id as member_i5_1_0_,
        orders0_.id as id1_1_0_,
        orders0_.id as id1_1_1_,
        orders0_.created_at as created_2_1_1_,
        orders0_.member_id as member_i5_1_1_,
        orders0_.number as number3_1_1_,
        orders0_.updated_at as updated_4_1_1_
    from
        orders orders0_
    where
        orders0_.member_id=?
```

- 출력되는 SQL을 보면 조인을 통해서 연관관계 컬렉션까지 함께 조회되는 것으로 생각할 수 있습니다. 하지만 JPQL은 결과를 반환할 때 연관관계까지 고려하지 않고 select 절에 지정한 엔티티만 조회하게 됩니다.
- 따라서 컬렉션은 초기화하지 않은 컬렉션 레퍼를 반환하게 되고 컬렉션이 없기 때문에 Lazy 로딩이 발생하게 되고 결과적으로 N+1 문제가 발생하게 됩니다.

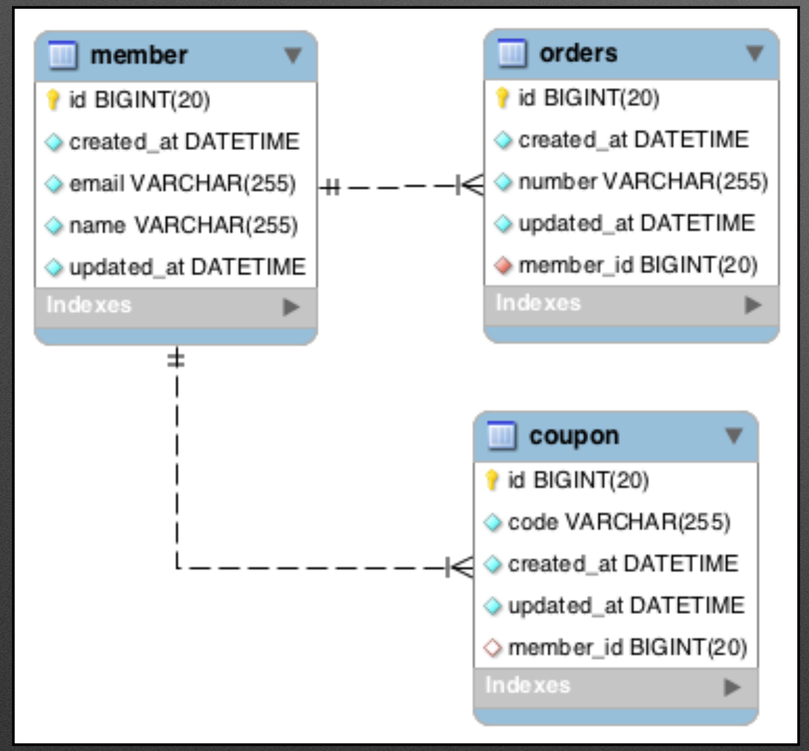
Fetch 조인과 한계
컬렉션을 Fetch 조인하면 페이징 API를 사용할 수 없다.

```
interface MemberRepository : JpaRepository<Member, Long> {
    @Query(
            value = "select m from Member m left join fetch m.orders",
            countQuery = "select count(m) from Member m"
    )
    fun findAllWithFetchPaging(pageable: Pageable): Page<Member>
}
```

```
@Test
internal fun `컬렉션을 페치 조인하면 페이징 API를 사용할 수 없다`() {
    val page = PageRequest.of(0, 10)
    val members = memberRepository.findAllWithFetchPaging(page)


    // 조회한 모든 회원에 대해서 조회하는 경우에도 N+1 문제가 발생하지 않음
    for (member in members) {
        println("order size: ${member.orders.size}")
    }
}
```

- 해당 쿼리에서는 limit offset 관련된 쿼리문이 없습니다. 하이버네이트에서 컬렉션을 페치 조인하고 페지이 API를 사용하면 메모리에서 페이징 처리를 진행합니다.
- 즉 데이터베이스에서는 FULL Scan 한 이후 모든 데이터를 메모리에 올린 이후 limit에 맞게 데이터를 만들게 됩니다.
- 데이터베이스에 Full Sacn 하는 것도 문제지만 그것을 메모리에 올리기 때문에 메모리를 심하게 잡아먹게 됩니다. 컬렉션이 아닌 단일 값 연관 필드의 경우에는 패치 조인을 사용해도 페이징 API를 사용할 수 있습니다.

Fetch 조인과 한계
둘 이상 컬렉션을 페치할 수 없다.




```fun
@Entity
@Table(name = "member")
class Member private constructor() {
    ...

    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY)
    var orders: List<Order> = emptyList()
        private set

    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY) // 새로운 컬렉션 쿠폰 추가
    var coupons: List<Coupon> = emptyList()
        private set
}
```

```fun
@Query(
        value = "select m from Member m left join fetch m.orders left join fetch m.coupons",
        countQuery = "select count(m) from Member m"
    )
fun findAllWithFetchPaging2(pageable: Pageable): Page<Member>
```

```fun
@Test
internal fun `둘 이상 컬렉션을 페치할 수 없다`() {
    val page = PageRequest.of(0, 10)
    val members = memberRepository.findAllWithFetchPaging2(page)

    // 조회한 모든 회원에 대해서 조회하는 경우에도 N+1 문제가 발생하지 않음
    for (member in members) {
        println("order size: ${member.orders.size}")
    }
}
```

```
Tests failed: 1 of 1 test – 2 ms
    at org.springframework.data.repository.core.support.RepositoryFactorySupport$QueryExecutorMethodInterceptor.mapMethodsToQuery(RepositoryFactorySupport.j
    at org.springframework.data.repository.core.support.RepositoryFactorySupport$QueryExecutorMethodInterceptor.lambda$new$0(RepositoryFactorySupport.java:5
    at java.util.Optional.map(Optional.java:215)
    at org.springframework.data.repository.core.support.RepositoryFactorySupport$QueryExecutorMethodInterceptor.<init>(RepositoryFactorySupport.java:559)
    at org.springframework.data.repository.core.support.RepositoryFactorySupport.getRepository(RepositoryFactorySupport.java:332)
    at org.springframework.data.repository.core.support.RepositoryFactoryBeanSupport.lambda$afterPropertiesSet$5(RepositoryFactoryBeanSupport.java:297)
    at org.springframework.data.util.Lazy.getNullable(Lazy.java:212)
    at org.springframework.data.util.Lazy.get(Lazy.java:94)
    at org.springframework.data.repository.core.support.RepositoryFactoryBeanSupport.afterPropertiesSet(RepositoryFactoryBeanSupport.java:300)
    at org.springframework.data.jpa.repository.support.JpaRepositoryFactoryBean.afterPropertiesSet(JpaRepositoryFactoryBean.java:121)
    at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.invokeInitMethods(AbstractAutowireCapableBeanFactory.java:1862)
    at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:1799)
    ... 107 more
Caused by: java.lang.IllegalArgumentException: org.hibernate.loader.MultipleBagFetchException: cannot simultaneously fetch multiple bags: [com.example.jpa
    at org.springframework.orm.jpa.ExtendedEntityManagerCreator$ExtendedEntityManagerInvocationHandler.invoke(ExtendedEntityManagerCreator.java:360) <3 inter
    at org.springframework.data.jpa.repository.query.SimpleJpaQuery.validateQuery(SimpleJpaQuery.java:87)
    ... 136 more
Caused by: org.hibernate.loader.MultipleBagFetchException: cannot simultaneously fetch multiple bags: [com.example.jpanplus1.member.Member.orders, com.exam
    ... 144 more
```

- 컬렉션의 카테시안 곱이 만들어지므로 하이버네이트는 주의해야 합니다. 하이버네이트는 `annot simultaneously fetch multiple bag` 예외가 발생하게 됩니다. 가장 쉬운 해결 방법으로는 자료형을 List -> Set으로 변경하는 것입니다.

Fetch 조인과 한계
둘 이상 컬렉션을 페치할 수 없다.

```
@Entity
@Table(name = "member")
class Member private constructor() {
    ...

    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY)
    var orders: Set<Order> = emptySet()
        private set

    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY)
    var coupons: Set<Coupon> = emptySet()
        private set
}
```

```
Hibernate:
    /* select
        m
    from
        Member m
    left join
        fetch m.orders
    left join
        fetch m.coupons */ select
            member0_.id as id1_1_0_,
            orders1_.id as id1_2_1_,
            coupons2_.id as id1_0_2_,
            member0_.created_at as created_2_1_0_,
            member0_.email as email3_1_0_,
            member0_.name as name4_1_0_,
            member0_.updated_at as updated_5_1_0_,
            orders1_.created_at as created_2_2_1_,
            orders1_.member_id as member_i5_2_1_,
            orders1_.number as number3_2_1_,
            orders1_.updated_at as updated_4_2_1_,
            orders1_.member_id as member_i5_2_0__,
            orders1_.id as id1_2_0__,
            coupons2_.code as code2_0_2_,
            coupons2_.created_at as created_3_0_2_,
            coupons2_.member_id as member_i5_0_2_,
            coupons2_.updated_at as updated_4_0_2_,
            coupons2_.member_id as member_i5_0_1__,
            coupons2_.id as id1_0_1__
        from
            member member0_
        left outer join
            orders orders1_
                on member0_.id=orders1_.member_id
        left outer join
            coupon coupons2_
                on member0_.id=coupons2_.member_id
Hibernate:
    /* select
        count(m)
    from
        Member m */ select
            count(member0_.id) as col_0_0_
        from
            member member0_
order size: 10
order size: 10
order size: 10
```

- 컬렉션의 카테시안 곱이 만들어지므로 하이버네이트는 주의해야 합니다. 하이버네이트는 `annot simultaneously fetch multiple bag` 예외가 발생하게 됩니다. 가장 쉬운 해결 방법으로는 자료형을 List -> Set으로 변경하는 것입니다.

JPA 플러시

플러시 란 ?

- JPA는 엔티티를 영속성 컨텍스트에서 관리합니다. 영속성 컨텍스트에 있는 내용을 데이터베이스에 반영하는 것을 플러시라고 합니다.
- 보통 트랜잭션을 커밋하면 영속성 컨텍스트의 변경 내용을 데이터베이스에 동기화(등록, 수정, 삭제) 작업을 진행하게 됩니다.

영속성 컨텍스트를 플러시 하는 3 가지 방법

플러시는 영속성 컨텍스의 변경 내용을 데이터베이스에 반영한다. 플러시를 실행하면 구체적으로 다음과 같은 일이 일어난다

1. 변경 감자기 동작해서 영속성 컨텍스트에 있는 모든 엔티티를 스냅샷과 비교 해서 수정된 엔티티를 찾는다. 수정 엔티티는 수정 쿼리를 만들어 쓰기 지연 SQL 등록한다.
2. 쓰기 지연 SQL 의 저장소의 쿼리를 데이터베이스에 전송한다. (등록, 수정, 삭제 쿼리)

영속성 컨텍스트를 플러시 하는 방법은 3가지다.

1. em.flush()를 직접 호출한다.
2. JPQL 쿼리 실행 시 플러시가 자동 호출된다.
3. 트랜잭션 커밋 시 플러시가 자동 호출된다.

플러시를 직접 호출하는 경우

- 엔티티 매니저의 flush() 메서드를 직접 호출해서 영속성 컨텍스트를 강제로 플러시 한다.
- 테스트나 다른 프레임워크와 JPA 함께 사용할 때는 제외하고 거의 사용하지 않는다.

트랜잭션 커밋 시 플러시가 자동 호출

- 데이터베이스에 변경 내용을 SQL로 전달하지 않고 트랜잭션만 커밋하면 어떤 데이터도 데이터베이스에 반영되지 않는다.
- 따라서 트랜잭션을 커밋하기 전에 꼭 플러시를 호출해서 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영해야 한다.
- JPA는 이런 문제를 예방하기 위해서 트랜잭션 커밋할 때 플러시를 자동으로 호출한다.

JPQL 쿼리 실행시 플러시 자동 호출

```
@Test
internal fun `JPQL 쿼리 실행시 플러시 자동 호출`() {
    val teamA = Team("teamA")
    val teamB = Team("teamB")

    em.persist(teamA)
    em.persist(teamB)


    val teams = query.select(qTeam)
            .from(qTeam)
            .fetch()

    for (team in teams) {
        println("team : $team")
    }
}
```

- JPQL이나 QueryDSL 같은 객체지향 쿼리를 호출할 플러시가 실행된다.

- 1. teamA, teamB를 영속성 컨텍스트에 저장한다.
- 2. QueryDSL으로 Team 전체를 조회 한다.
- 3. QueryDSL 쿼리 시점에 teamA, teamB 플러시를 이르켜 데이터베이스에 commit하지 않았다면 QueryDsl으로 조회한 값은 없을 것이다.

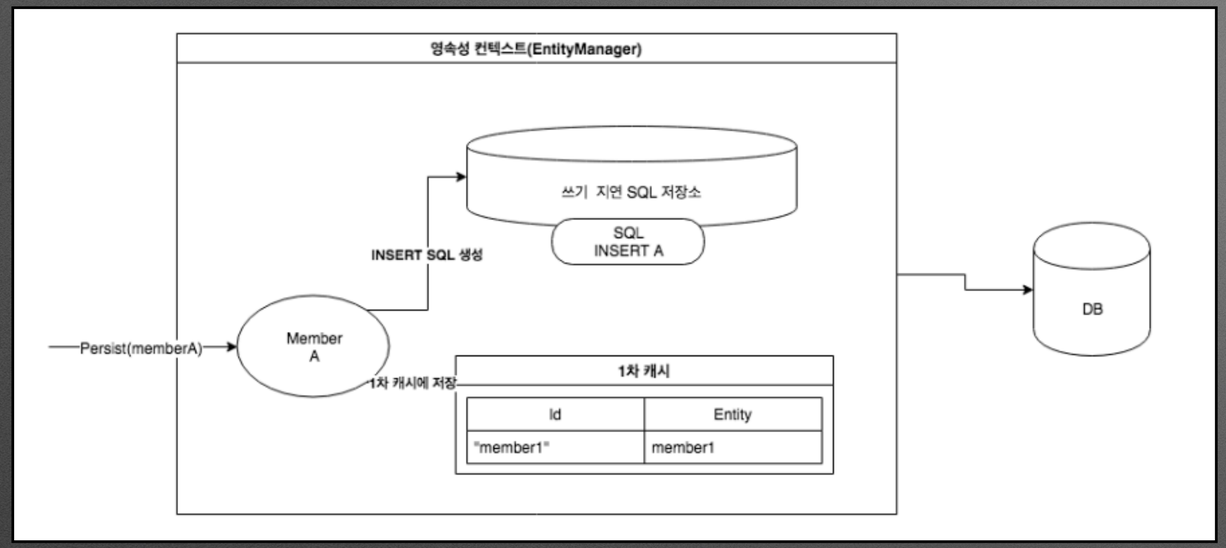
- 이런 결과가 나오기 때문에 쿼리를 실행하기 직전에 영속성 컨텍스트를 플러시해서 변경 내용을 데이터베이스에 반영해야 한다.
- JPA는 이런 문제를 예방하기 위해서 JPQL을 실행할 때도 플러시르 자동 호출한다. 참고로 식별자를 기준으로 조회하는 find() 메서드는 호출되지 않는다.

엔티티 등록

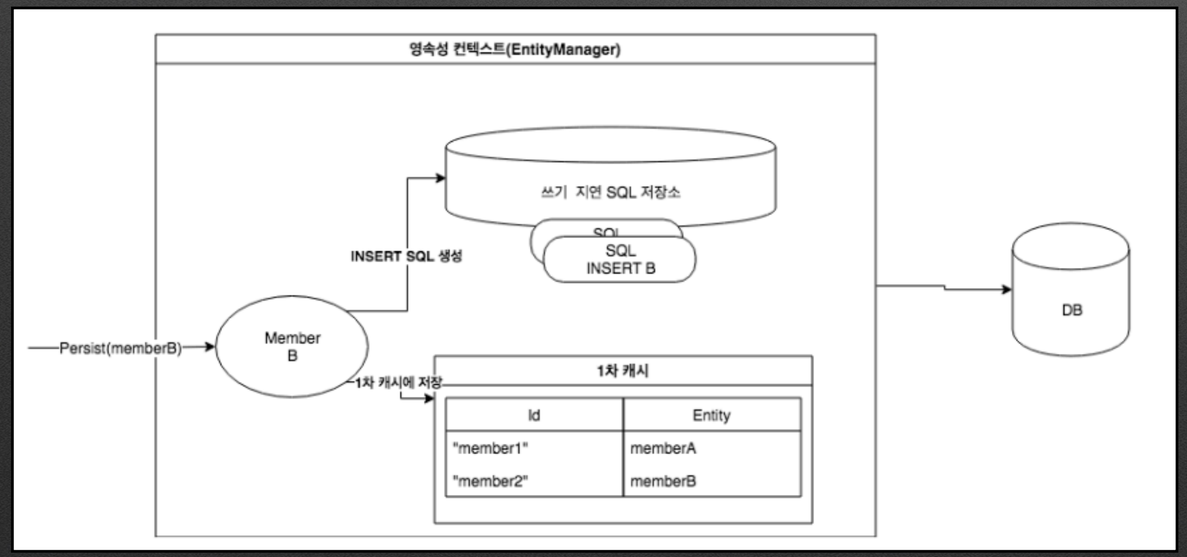
```
EntityMaanger em  = emf.createEnttiyManager();
ENtityTranscation transaction = em.getTransaction();
// 엔티티 매니저는 데이터 변경 시 트랜잭션을 시작해야한다.

transaction.begin();

em.persist(memberA);
em.persist(memberB);

// 여기까지 Insert SQL을 데이터베이스에 보내지 않는다.

// Commit을 하는 순간 데이터베이스에 Insert SQL을 보낸다
transaction.commit();
```

- 엔티티 매니저는 트랜잭션을 커밋하기 직전까지 데이터베이스에 엔티티를 저장하지 않고 **내부 쿼리 저장소에 INSERT SQL을 모아둔다**.
- **트랜잭션을 커밋할 때 모아둔 쿼리를 데이터베이스에 보내느데 이것을 트랜잭션을 지원하는 쓰기 지연 이라 한다.**

엔티티 등록

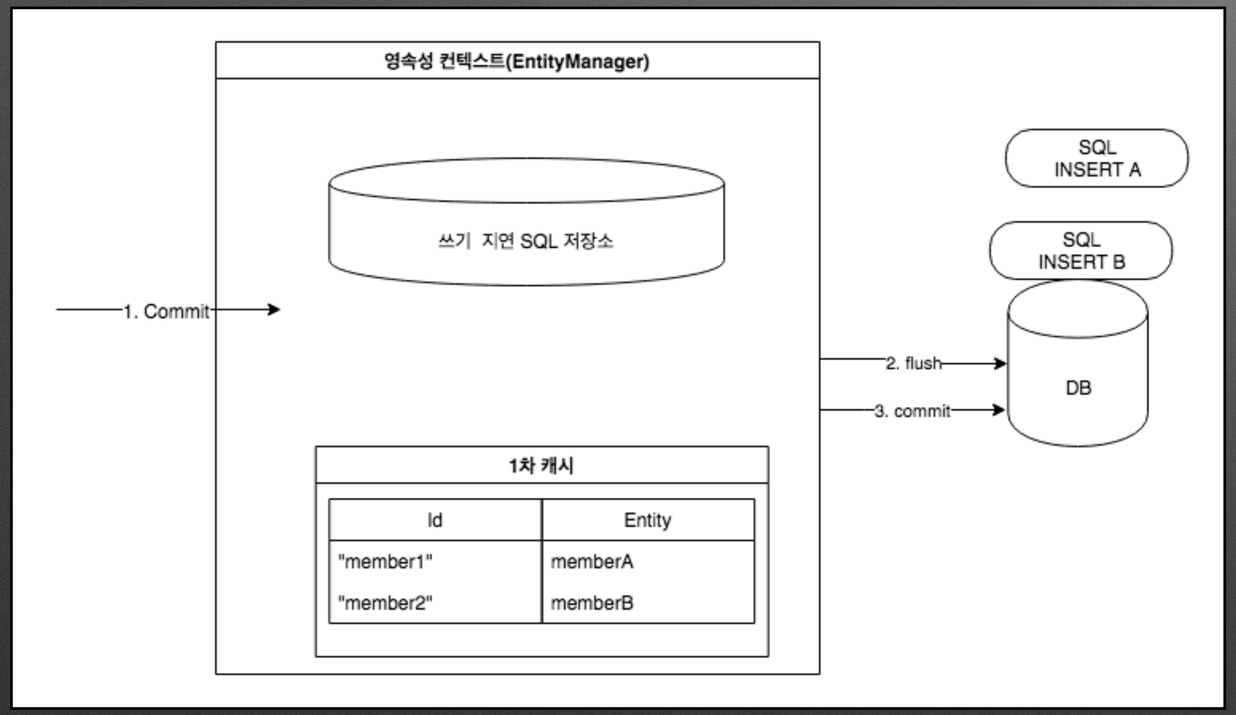


- 회원 A를 영속화 했다. 영속성 컨텍스트는 1차 캐시에 회원 엔티티를 저장하면서 동시에 회원 엔티티 정보로 등록 쿼리를 만든다. 그리고 만들어진 **등록 쿼리를 쓰기 지연 SQL 저장소에 보관**한다.



- 다음으로 회원 B를 영속화했다. 마찬가지로 회원 엔티티 정보로 등록 쿼리를 생성해서 쓰지 지연 SQL 저장소에 보관한다. 현재 쓰기 지연 SQL저장소 에는 등록 쿼리가 2건이 저장되어 있다.

엔티티 등록



- 마지막으로 트랜잭션을 커밋했다. 트랜잭션을 커밋하면 엔티티 매니저는 우선 영속성 컨텍스트를 플러시한다.
- 플러시는 영속성 컨텍스트의 변경 내용을 데이터베이스에 동기화하는 작업인데 이때 등록, 수정, 삭제한 엔티티를 데이터베이스에 반영한다.
- 즉, 쓰기 지연 SQL 저장소에 모인 쿼리를 데이터베이스에 보낸다. 이렇게 영속성 컨텍스트의 변경 내용을 데이터베이스에 동기화한 후에 실제 데이터베이스 트랜잭션을 커밋한다.
- flush가 먼저 동작하고 데이터베이스에 동기화한 후에 실제 데이터베이스 트랜잭션을 커밋한다.

트랜잭션을 지원하는 쓰기 지연이 가능한 이유

```
begin(); // 트랜잭션 시작

save(A);
save(B);
save(C);

commit(); // 트랜잭션 커밋
```

1.데이터를 저장하는 즉시 등록 쿼리를 데이터베이스에 보낸다. 예제에서 save() 메서드를 호출할 때 마다 즉시 데이터베이스에 등록 쿼리를 보낸다. 그리고 마지막에 트랜잭션을 커밋한다.

2.데이터를 저장하면 등록 쿼리를 데이터베이스에 보내지 않고 메모리에 모아 둔다. 그리고 트랜잭션을 커밋할 때 모아 둔 등록 쿼리를 데이터베이스에 보낸다.

트랜잭션을 지원하는 쓰지 지연과 성능 최적화 : 트랜잭션을 지원하는 쓰지 이연과 JDBC 배치

```
insert(member1); // INSERT INTO ...
insert(member2); // INSERT INTO ...
insert(member3); // INSERT INTO ...
insert(member4); // INSERT INTO ...
insert(member5); // INSERT INTO ...

commit();
```

- 네트워크 호출 한번은 단순한 메소드를 수만 번 호출하는 것보다 더 큰 비용이 든다.
- 이 코드는 5번의 INSERT SQL과 1번의 커밋으로 총 6번 데이터 베이스와 통신한다.
- 이것을 최적화하라면 5번의 INSERT SQL을 모아서 한 번에 데이터베이스로 보내면 된다.
- JDBC가 제공하는 SQL 배치 기능을 사용하면 SQL을 모아서 데이터베이스에 한 번에 보낼 수 있다. 하지만 이 기능을 사용하라면 많은 코드를 수정해야한다.
- JPA는 플러시 기능이 있이므로 SQL 배치 기능을 효과적으로 사용할 수 있다.

트랜잭션을 지원하는 쓰지 지연과 성능 최적화 : 트랜잭션을 지원하는 쓰지 이연과 JDBC 배치

```
em.persist(new Member()); // 1
em.persist(new Member()); // 2
em.persist(new Member()); // 3
em.persist(new Member()); // 4
em.persist(new Orders()); // 1-1,
em.persist(new Member()); // 1
em.persist(new Member()); // 2
```

- hibernate.jdbc.batch_size 속성의 값을 50으로 주면 최대 50건씩 모아서 SQL 배치를 실행한다. 하지만 SQL 배치는 같은 SQL일 때만 유효하다. 중간에 다른 처리가 들어가면 SQL 배치를 다시 시작한다.
- 1,2,3,4를 모아서 하나의 SQL 배치를 실행하고 1-1를 한 번 실행하고 1,2을 모아서 실행한다. 따라서 총 3번의 SQL 배치를 실행한다.
- 모든 경우에 사용할 수 있는 것은 아니다. 엔티티가 영속 상태가 되려면 식별자가 꼭 필요하다. 그런데 IDENTITY 식별자 생성 전략은 엔티티를 데이터베이스에 저장해야 식별자를 구할 수 있으므로 em.persist()를 호출하는 즉시 INSERT SQL이 데이터베이스에 전달된다. 따라서 쓰지 지연을 활용한 성능 최적화를 할 수가 없다.

트랜잭션을 지원하는 쓰기 지연과 애플리케이션 확장성

```
em.persist(new Member()); // 1
em.persist(new Member()); // 2
em.persist(new Member()); // 3
em.persist(new Member()); // 4
em.persist(new Orders()); // 1-1,
em.persist(new Member()); // 1
em.persist(new Member()); // 2
```

- 트랜잭션을 지원하는 쓰기 지연의 가장큰 장점은 데이터베이스 테이블 로우에 락이 걸리는 시간을 최소한다는 것이다. 이 기능은 트랜잭션을 커밋해서 영속성 컨텍스트를 플러시하기 전까지는 데이터베이스에 데이터를 등록, 수정, 삭제 하지 않는다. 따라서 커밋 전까지 데이터베이스 로우에 락을 걸지 않는다.

트랜잭션을 지원하는 쓰기 지연과 애플리케이션 확장성

```
update(memberA); // UPDATE SQL Member A
비지니스로직A(); // UPDATE SQL ...
비지니스로직B(); // UPDATE SQL ...
commit();
```

- JPQL를 사용하지 않고 SQL 직접다루면 update(memberA)를 호출할 때 **UPDATE SQL을 실행하면 데이터베이스 테이블 로우에 락을 건다**.
- 이 락은 비지니스 로직A(), 비지니스 로직B()를 모두 수행하고 commit()을 호출할 때까지 유지된다.
- 트랜잭션 격리 수준에 따라 다르지만 보통 많이 사용하는 커밋된 읽기 Read Committed 격리 수준이나 그 이상에는 데이터베이스에 **현재 수정 중인 데이터(로우)를 수정하려는 다른 트랜잭션은 락이 풀릴 때까지 대기한다**.

트랜잭션을 지원하는 쓰기 지연과 애플리케이션 확장성

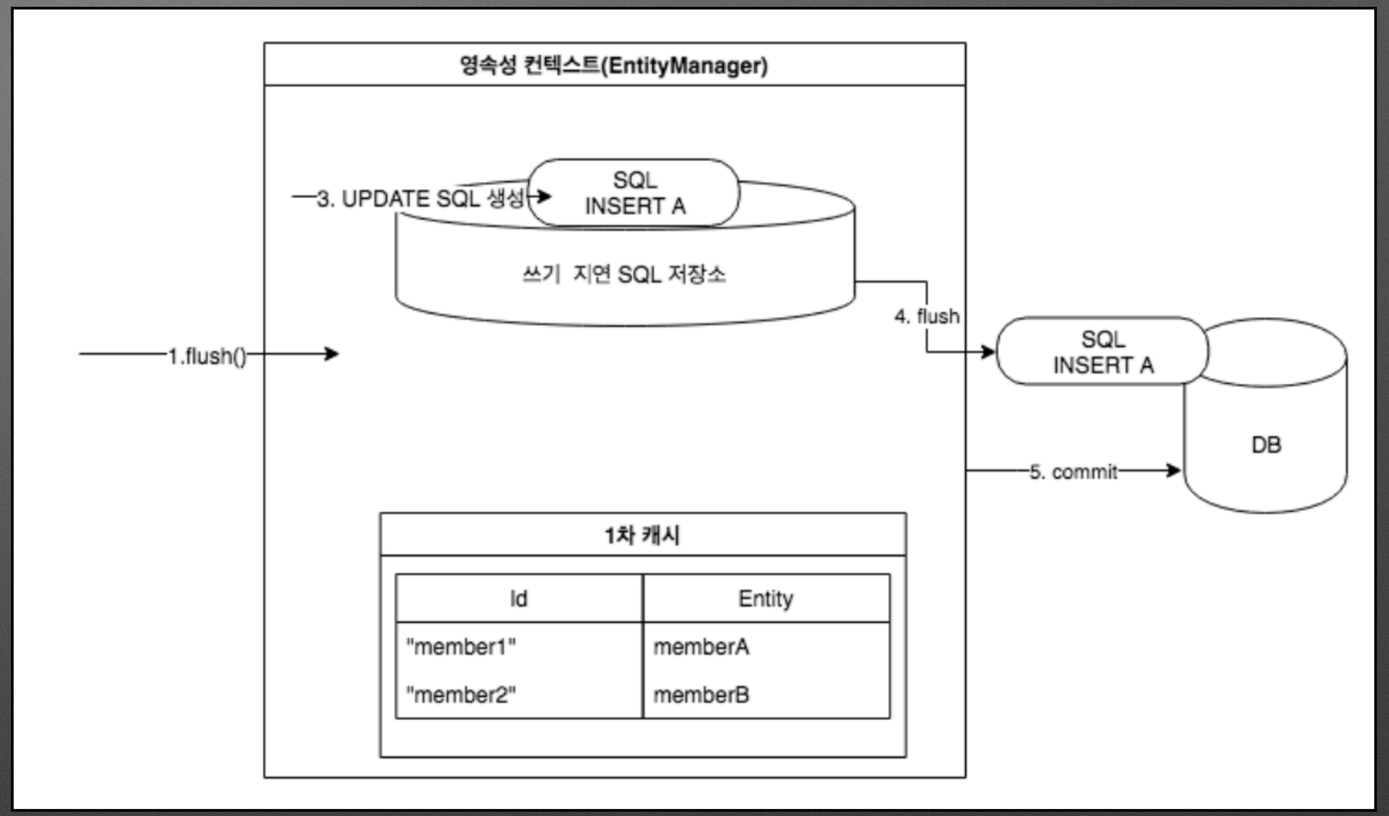
```
update(memberA); // UPDATE SQL Member A
비지니스로직A(); // UPDATE SQL ...
비지니스로직B(); // UPDATE SQL ...
commit();
```

- JPA는 커밋을 해야 플러시를 호출하고 데이터베이스에 수정 쿼리를 보낸다.
- 예제에서 commit()을 호출할 때 UPDATE SQL을 실행하고 바로 데이터베이스 트랜잭션을 커밋한다. 쿼리를 보내고 보내고 바로 트랜잭션을 커밋하므로 결과적으로 데이터베이스에 락이 걸리는 시간을 최소화 한다. 이는 동시에 더 많은 트래잭션을 처리할 수 있다는 장점이 된다.

엔티티 수정

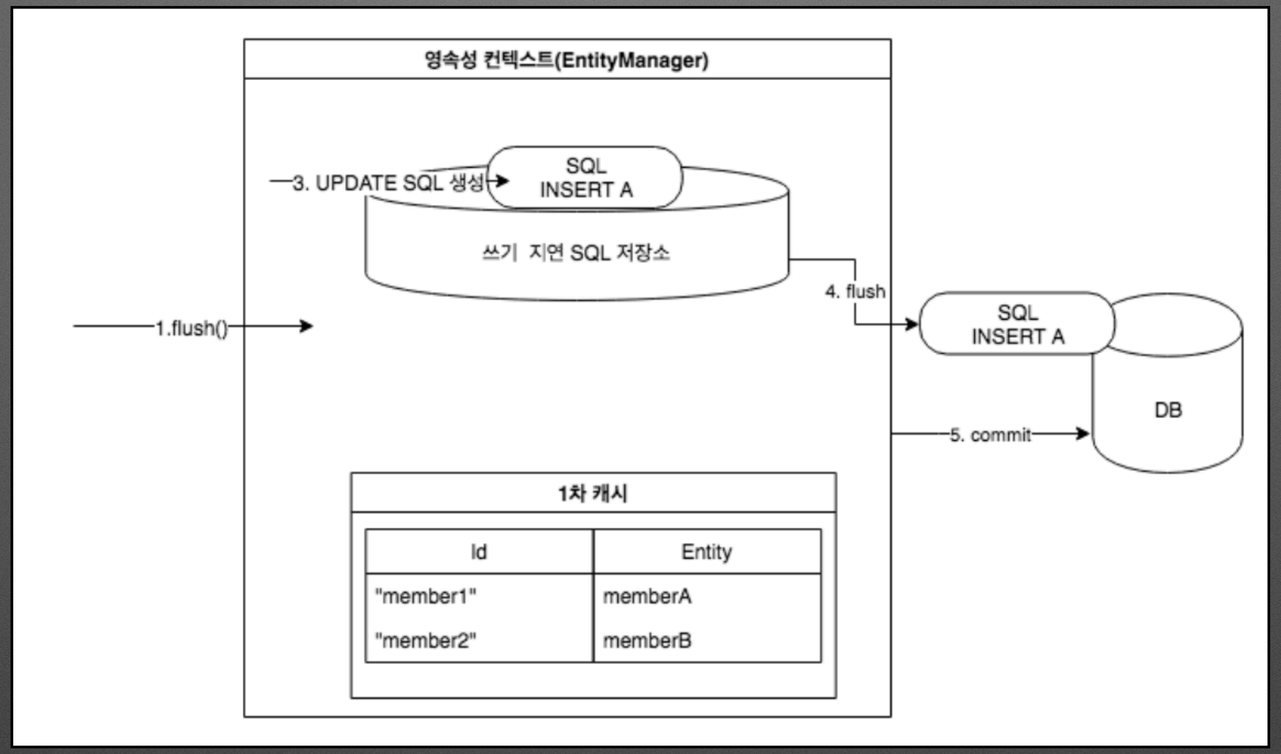
변경 감지

```
EntityMaanger em  = emf.createEnttiyManager();
ENtityTranscation transaction = em.getTransaction();
transaction.begin(); // 트랜잭션 시작

// 영성속 텐티티 조회
Member memberA = em.find(Member.class, "memberA");

// 영속성 엔티티 데이터 수정

memberA.setUsername("hi");
memberA.setAge(10);

//em.update(member) 이런 코드가 있어야하지 않을까?

transaction.commit(); // 트랜잭션 커밋
```



- JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해두는데 이것을 스냅샷이라고 한다. 그리고 플러시 시점에서 스냅샵과 엔티티를 비교해서 변경된 엔티티를 찾는다.

엔티티 수정



1.트랜잭션을 커밋하면 엔티티 매니저 내부에서 먼저 플러시가 호출된다.

2.엔티티와 스냅샵을 비교해서 변경된 엔티티를 찾는다.

3.변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보낸다.

4.쓰기 지연 저장소의 SQL을 데이터베이스에 보낸다.

5.데이터베이스 트랜잭션을 커밋한다.

읽기 전용 트랜잭션

- 스프링 프레임워크를 사용하면 트랜잭션을 읽기 전용 모드로 설정할 수 있다. @Transactional(readOnly = true) 옵션을 주면 스프링 프레임워크가 하이버네이트 세션의 플러시 모드를 MANUAL로 설정한다
- 그렇게되면 강제로 플러시를 호출하지 않은 한 플러시가 일어나지 않는다. 따라서 트랜잭션을 커밋해도 영속성 컨텍스트 플러시하지 않는다.
- 영속성 컨텍스트를 플러시하지 않으니 엔티티의 등록, 수정, 삭제는 당연히 동작하지 않는다. 플러시 할 때 일어나는 스냅샷비교와 같은 무거운 로직을 수행하지 않으므로 성능에 향상된다.
- 변경 감지는 영속성 컨텍스타 관리하는 영속 상태의 엔티티에만 적용된다. 비영속, 준영속처럼 영속성 컨텍스트의 관리를받지 못하는 엔티티는 값을 변경해도 데이터베이스에 반영되지 않는다.

엔티티 삭제

```
Member meberA = em.find(Member.class, "memberA"); // 삭제할 대상 엔티티 조회
em.remove(memberA); // 엔티티 삭제
```

- 엔티티를 삭제하려면 먼저 삭제 대상 엔티티를 조회해야한다. em.remove()에 삭제 대상 엔티티를 넘겨주면 엔티티를 삭제한다.
- 물론 엔티티를 즉시 삭제하는 것이 아니라 엔티티 등록과 비슷하게 삭제 쿼리를 쓰기 지연 데이터베이스에 삭제 쿼리를 전달한다.

JPA 영속성 컨텍스트 주의 점

엔티티

```
@Entity
@Table(name = "member")
data class Member(
        @Column(name = "username", nullable = false)
        var username: String,

        @Column(name = "age", nullable = false)
        var age: Int = 0,

        @ManyToOne(fetch = FetchType.LAZY, optional = false)
        @JoinColumn(name = "team_id", nullable = false)
        var team: Team
) : EntityAuditing()

@Entity
@Table(name = "team")
data class Team(
        @Column(name = "name", nullable = false)
        var name: String
) : EntityAuditing() {

    @OneToMany(mappedBy = "team")
    var members: MutableList<Member> = mutableListOf()
}
```

- Member, Team 객체가 있고 Team 1: Member N 관계

엔티티

```
import com.example.querydsl.domain.QMember.member as qMember
import com.example.querydsl.domain.QTeam.team as qTeam

@SpringBootTest
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
@Transactional
class PersistenceContextTest(
        private val em: EntityManager
) {

    val query = JPAQueryFactory(em)

    @Test
    internal fun `persistence context test`() {
        //given
        val teamA = Team("teamA")
        em.persist(teamA)

        val member1 = Member(username = "member1", age = 10, team = teamA)
        val member2 = Member(username = "member2", age = 20, team = teamA)
        em.persist(member1)
        em.persist(member2)

        //when
        val team = query
                .selectFrom(qTeam)
                .join(qTeam.members, qMember).fetchJoin()
                .where(qTeam.name.eq("teamA"))
                .fetchOne()!!

        //then
        val members = team.members
        then(members).anySatisfy {
            then(it.username).isIn("member1", "member2")
        }
    }
}
```

- teamA를 영속화 이후, member1, member2를 영속화를 진행합니다. member1, member2는 teamA 소속이 됩니다.
- 그리고 fetch join통해서 teamA와 teamA에 속한 member를 조회하고 있습니다. 그리고 then절에서 member1, member2가 teamA에 있는지 검증을 진행합니다.
- 이 테스트 코드는 실패합니다.

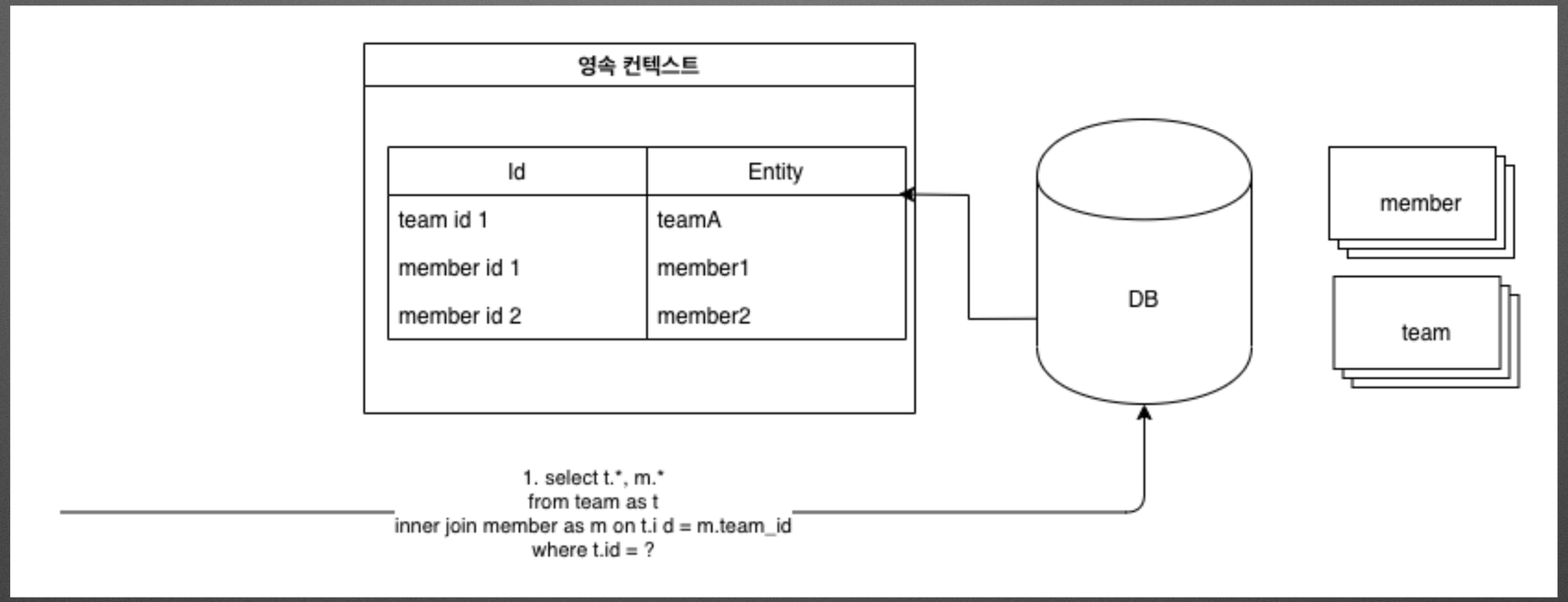
엔티티

```
Variables
this = {PersistenceContextTest@9978}
teamA = {Team@9981} "Team(name=teamA)"
member1 = {Member@9982} "Member(username=member1, age=10, team=Team(name=teamA))"
member2 = {Member@9983} "Member(username=member2, age=20, team=Team(name=teamA))"
team = {Team@9981} "Team(name=teamA)"
    members = {PersistentBag@9984}  size = 0
    name = "teamA"
    id = {Long@9990} 1
    createdAt = {LocalDateTime@9991} "2020-01-26T23:12:58.240"
    updatedAt = {LocalDateTime@9992} "2020-01-26T23:12:58.240"
    members = {PersistentBag@9984}  size = 0
```

```
Hibernate:
    insert
    into
        team
        (id, created_at, updated_at, name)
    values
        (null, ?, ?, ?)
2020-01-26 23:12:58.249 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [1] as [TIMESTAMP] - [2020-01-26T23:12:58.240]
2020-01-26 23:12:58.254 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [2] as [TIMESTAMP] - [2020-01-26T23:12:58.240]
2020-01-26 23:12:58.255 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [3] as [VARCHAR] - [teamA]
Hibernate:
    insert
    into
        member
        (id, created_at, updated_at, age, team_id, username)
    values
        (null, ?, ?, ?, ?, ?)
2020-01-26 23:12:58.264 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [1] as [TIMESTAMP] - [2020-01-26T23:12:58.263]
2020-01-26 23:12:58.264 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [2] as [TIMESTAMP] - [2020-01-26T23:12:58.263]
2020-01-26 23:12:58.265 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [3] as [INTEGER] - [10]
2020-01-26 23:12:58.265 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [4] as [BIGINT] - [1]
2020-01-26 23:12:58.266 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [5] as [VARCHAR] - [member1]
Hibernate:
    insert
    into
        member
        (id, created_at, updated_at, age, team_id, username)
    values
        (null, ?, ?, ?, ?, ?)
2020-01-26 23:12:58.267 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [1] as [TIMESTAMP] - [2020-01-26T23:12:58.267]
2020-01-26 23:12:58.267 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [2] as [TIMESTAMP] - [2020-01-26T23:12:58.267]
2020-01-26 23:12:58.268 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [3] as [INTEGER] - [20]
2020-01-26 23:12:58.268 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [4] as [BIGINT] - [1]
2020-01-26 23:12:58.268 TRACE 66365 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [5] as [VARCHAR] - [member2]
```

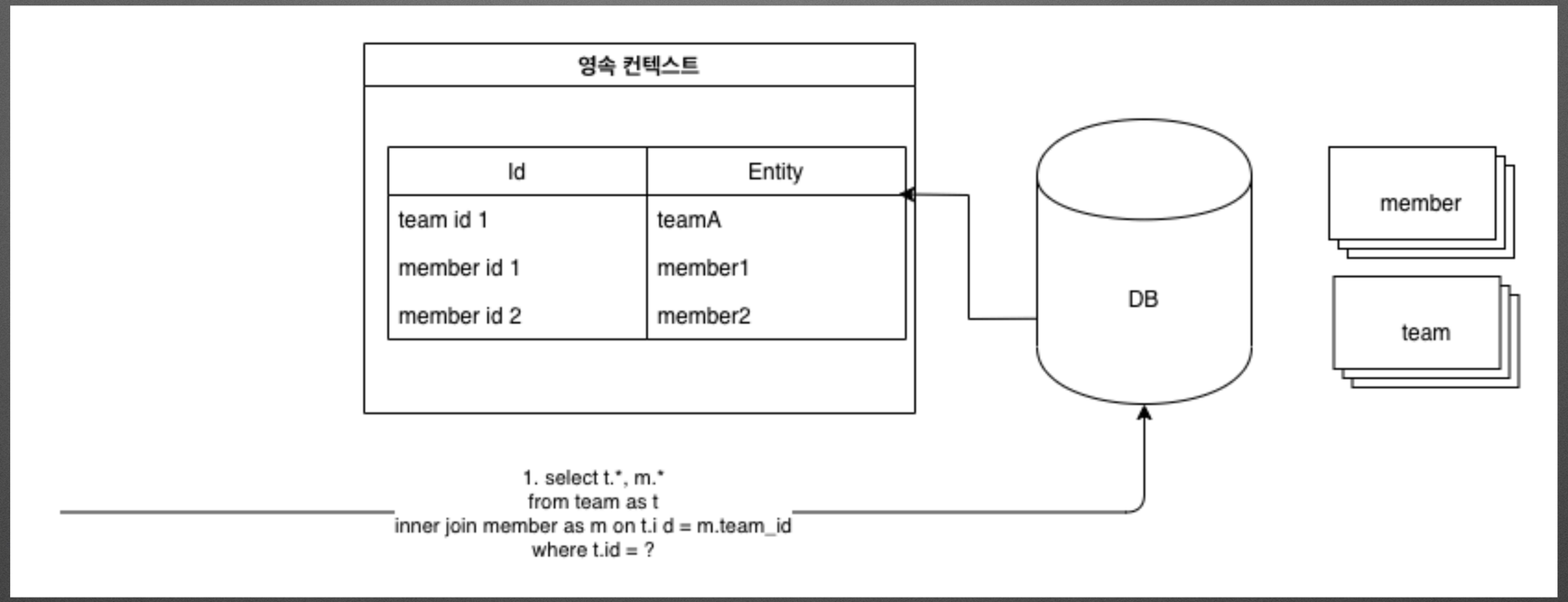
- 디버깅 모드로 보면 **memebers szie가 0**인 것을 확인할 수 있습니다.
- teamA insert, member1, memeber2 insert 쿼리를 확인할 수 있습니다.
- 그렇다면 왜 해당 테스트 코드가 실패할까요? 그 이유를 알기 위해서는 **영속성 컨텍스트의 1차 케시의 저장 메커니즘을 이해해야** 합니다.

영속성 컨텍스트 1차 캐시 저장 메커니즘



- findById() 같은 경우는 영속성 컨텍스트를 먼저 찾고 영속성 컨텍스트에 해당 엔티티가 있으면 그 값을 바로 리턴합니다. 이를 1차 캐시라고 말합니다.
- 반면 JPQL은 영속성성 컨텍스트를 먼저 조회하지 않고 데이터베이스에 query하여 결과를 가져옵니다. 그리고 아래와 같은 흐름으로 영속성 컨텍스트를 저장을 시도합니다.

영속성 컨텍스트 1차 캐시 저장 메커니즘



1. JPQL을 호출하면 데이터베이스에 우선적으로 조회한다.
2. 조회한 값을 영속성 컨텍스트에 저장을 시도한다.
3. 저장을 시도할 때 해당 데이터가 이미 영속성 컨텍스트에 존재하는 경우(영속성 컨텍스트에서는 식별자 값으로 식별) 데이터베이스에서 조회한 신규 데이터를 버린다.

테스트가 실패하는 이유

```
    ...

    @Test
    internal fun `persistence context test`() {
    //given
    val teamA = Team("teamA")
    em.persist(teamA)

    val member1 = Member(username = "member1", age = 10, team = teamA)
    val member2 = Member(username = "member2", age = 20, team = teamA)
    em.persist(member1)
    em.persist(member2)

    ...
}
```

```
//when
val team = query
        .selectFrom(qTeam)
        .join(qTeam.members, qMember).fetchJoin()
        .where(qTeam.name.eq("teamA"))
        .fetchOne()!!
```

- fetch join를 사용했기 때문에 team, member 객체를 한번에 가져오게 됩니다.
- 하지만 teamA가 이미 영속성 컨텍스트에 존재 하므로 해당 쿼리를 통해서 가져온 데이터를 버리게 됩니다.

해결 하는 방법 : Team 객체에 Member 추가하기

```
@Test
internal fun `persistence context test`() {
    //given
    val teamA = Team("teamA")
    em.persist(teamA)

    val member1 = Member(username = "member1", age = 10, team = teamA)
    val member2 = Member(username = "member2", age = 20, team = teamA)
    em.persist(member1)
    em.persist(member2)

    teamA.members.add(member1) // teamA에 member1 추가
    teamA.members.add(member2) // teamA에 member2 추가

    ...
}
```

- member1, member2영속화 이후 teamA 객체에 영속화된 member 객체를 추가하는 코드를 작성하면 됩니다.
- 그렇게 되면 fetch join을 통해서 데이터베이스에 query를 진행하고 해당 데이터가 영속성 컨텍스트에 이미 있으므로 데이터를 버리게 되더라도 이미 member1, memeber2가 존재하는 상태이기 때문에 문제가 되지 않습니다.

해결 하는 방법 : Team 객체에 Member 추가하기

```
@Test
internal fun `persistence context test`() {
    //given
    val teamA = Team("teamA")
    em.persist(teamA)

    val member1 = Member(username = "member1", age = 10, team = teamA)
    val member2 = Member(username = "member2", age = 20, team = teamA)
    em.persist(member1)
    em.persist(member2)

    teamA.members.add(member1) // teamA에 member1 추가
    teamA.members.add(member2) // teamA에 member2 추가

    ...
}
```

- 양방향 연관 관계의 경우 편의 메서드를 작성해서 이런 문제를 미연에 방지하는 것이 좋습니다. 물론 필요에 따라서 양방향 양방향 연관 관계를 작성하는 것은 상관없지만 가능하면 단방향 연관 관계를 유지하는 것이 좋다고 생각합니다.
- 양방향 관계가 되면 생각보다 신경 쓸 것이 많아지게 되어 복잡도가 증가하게 됩니다. 또 OneToMany, ManyToOne 같은 양방향 연관 관계의 경우 N+1 문제도 신경 쓸 것들이 많아지게 됩니다.

해결 하는 방법 : 영속성 컨텍스트 초기화

```
@Test
internal fun `persistence context test`() {
    //given
    val teamA = Team("teamA")
    em.persist(teamA)

    val member1 = Member(username = "member1", age = 10, team = teamA)
    val member2 = Member(username = "member2", age = 20, team = teamA)
    em.persist(member1)
    em.persist(member2)

//        teamA.members.add(member1)
//        teamA.members.add(member2)

    em.flush()
    em.clear()

    ...
}
```

- em.flush()을 통해서 영속성 컨텍스트의 내용을 데이터베이스에 반영하고 em.clear()을 통해서 영속성 컨텍스트를 모두 초기화 합니다.
- 영속성 컨텍스트의 데이터를 모두 제거, 영속성 컨텍스츠가 초기화 되었기 때문에 fetch join의 결과가 모두 영속성 컨텍스트의 반영됩니다.

JPA Bulk 작업시 주의점

JPA Bulk 작업시 주의점

```
internal class BulkTest(
        private val em: EntityManager
) : SpringBootTestSupport() {

    val query = JPAQueryFactory(em)

    @BeforeEach
    internal fun setUp() {
        val teams = listOf(
                Team("team_1"),
                Team("team_2"),
                Team("team_3"),
                Team("team_4"),
                Team("team_5"),
                Team("team_6"),
                Team("team_7"),
                Team("team_8"),
                Team("team_9"),
                Team("team_10")
        )

        for (team in teams) {
            em.persist(team)
        }

    }
```

```
    @Test
    internal fun `bulk test`() {
        // team 전체를 조회한다. team name은 team_x 이다.
        val teams = query.selectFrom(qTeam).fetch()

        for (team in teams) {
            println("before update team : $team")
        }

        val ids = teams.map { it.id!! }

        // team  name 전체를 none name으로 변경한다.
        val updateCount = query.update(qTeam)
                .set(qTeam.name, "none name")
                .where(qTeam.id.`in`(ids))
                .execute()

        println("update count : $updateCount")

        for (team in teams) {
            println("after update team : $team")
        }

        val newSelectTeams = query.selectFrom(qTeam).fetch()

        for (team in newSelectTeams) {
            println("new select team : $team")
        }
    }
}
```

JPA Bulk 작업시 주의점

```
before update team : Team(name=team_1)
before update team : Team(name=team_2)
before update team : Team(name=team_3)
before update team : Team(name=team_4)
before update team : Team(name=team_5)
before update team : Team(name=team_6)
before update team : Team(name=team_7)
before update team : Team(name=team_8)
before update team : Team(name=team_9)
before update team : Team(name=team_10)
```

- 코드는 간단합니다. team_1, team_2...team_10을 저장하고 query dsl update를 이용해서 team name을 none name으로 변경하는 것입니다.
- 변경 제대로 변경이 되었는지 확인하는 확인하는 반복문이 있습니다.

JPA Bulk 작업시 주의점

```
Hibernate:
    update
        team
    set
        name=?
    where
        id in (
            ? , ? , ? , ? , ? , ? , ? , ? , ? , ?
        )
2020-01-31 02:08:59.130 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [1] as [VARCHAR] - [none name]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [2] as [BIGINT] - [1]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [3] as [BIGINT] - [2]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [4] as [BIGINT] - [3]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [5] as [BIGINT] - [4]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [6] as [BIGINT] - [5]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [7] as [BIGINT] - [6]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [8] as [BIGINT] - [7]
2020-01-31 02:08:59.131 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [9] as [BIGINT] - [8]
2020-01-31 02:08:59.132 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [10] as [BIGINT] - [9]
2020-01-31 02:08:59.132 TRACE 51643 --- [    Test worker] o.h.type.descriptor.sql.BasicBinder      : binding parameter [11] as [BIGINT] - [10]
update count : 10
```

- 로그를 보면 team name 1 ~ 10까지 제대로 출력되는 것을 확인할 수 있습니다.
- update query, update count가 정상적으로 출력됩니다. 이제 업데이트된 team을 확인하는 아래 코드의 로그를 확인해보겠습니다

```
for (team in teams) {
    println("after update team : $team")
}
```

```
after update team : Team(name=team_1)
after update team : Team(name=team_2)
after update team : Team(name=team_3)
after update team : Team(name=team_4)
after update team : Team(name=team_5)
after update team : Team(name=team_6)
after update team : Team(name=team_7)
after update team : Team(name=team_8)
after update team : Team(name=team_9)
after update team : Team(name=team_10)
```

예상했던 no

- 예상했던 none name 이 아닌 이전 데이터가 출력 되는 것을 획인 할 수 있습니다. 그렇다면 새로 query dsl으로 조회 쿼리를 해보겠습니다.

JPA Bulk 작업시 주의점

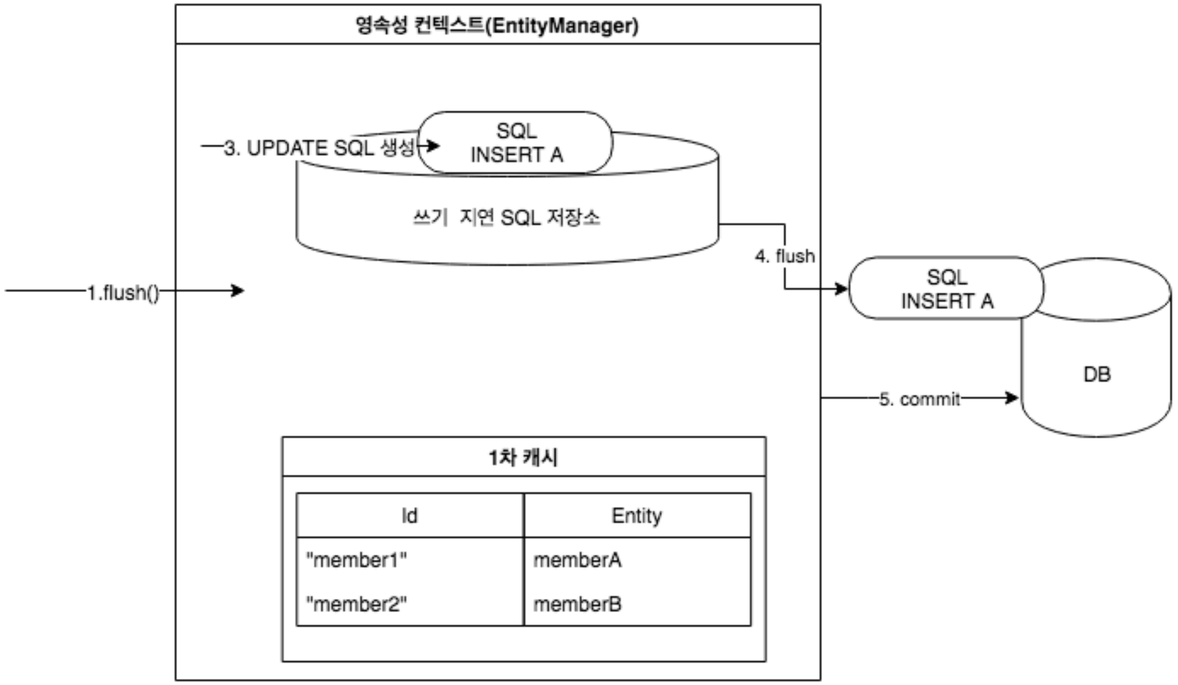
```
val newSelectTeams = query.selectFrom(qTeam).fetch()

for (team in newSelectTeams) {
    println("new select team : $team")
}
```

```
new select team : Team(name=team_1)
new select team : Team(name=team_2)
new select team : Team(name=team_3)
new select team : Team(name=team_4)
new select team : Team(name=team_5)
new select team : Team(name=team_6)
new select team : Team(name=team_7)
new select team : Team(name=team_8)
new select team : Team(name=team_9)
new select team : Team(name=team_10)
```

- 신규로 조회한 데이터에서도 team name이 변경되지 않은 것을 확인할 수 있습니다. 왜 변경되지 않은 것일 까요?

Team 엔티티가 변경되지 않은 이유

엔티티 수정 메커니즘



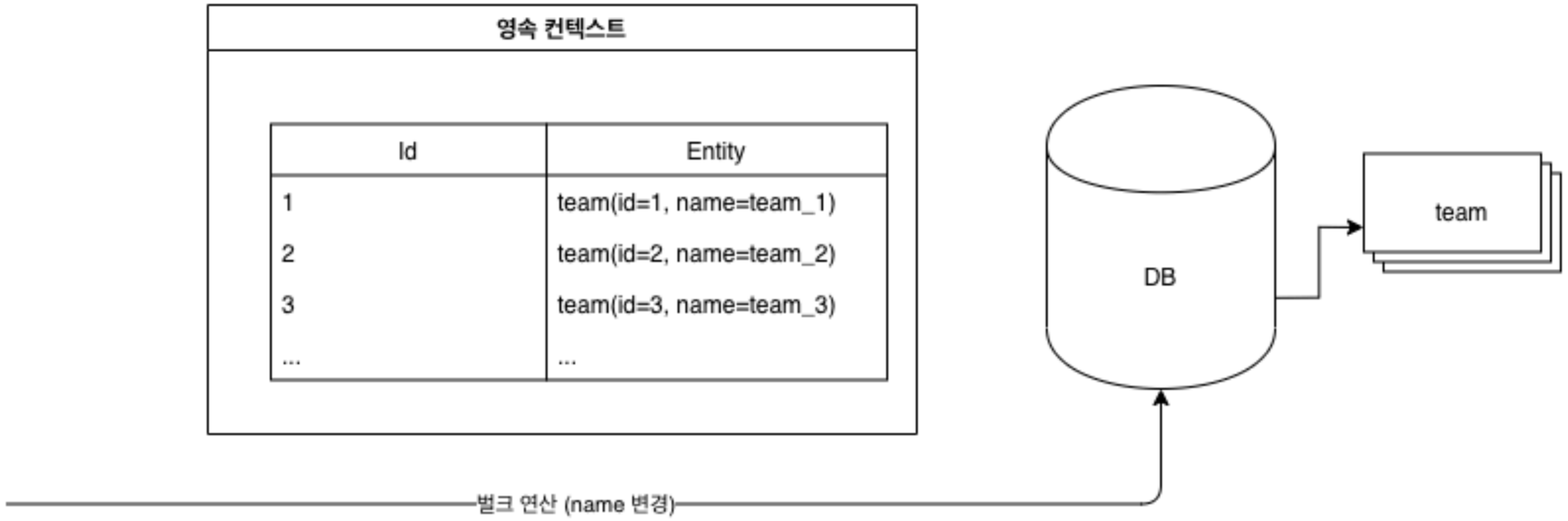
엔티티의 변경사항을 을 데이터베이스에 반영하는 것을 변경 감지(dirty checking)이라고 합니다. JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해두고. 플러시 시점에서 스냅샷과 엔티티를 비교해서 변경된 엔티티를 데이터베이스에 반영하게 됩니다. 즉 아래와 같은 절차를 진행하게 됩니다.

1. **트랜잭션을 커밋 하면 엔티티 매니저 내부에서 먼저 플러시가 호출된다.**
2. **엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾는다.**
3. 변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보낸다.
4. 쓰기 지연 저장소의 SQL을 데이터베이스에 보낸다.
5. **데이터베이스 트랜잭션을 커밋 한다.**

Team 엔티티가 변경되지 않은 이유

벌크 수정 메커니즘

하지만 벌크성 작업은 영속성 컨텍스트와 무관합니다. 벌크 연산은 영속성 컨텍스트를 통하지 않고 아래 그림처럼 **데이터베이스에 직접 쿼리를 진행합니다.**



즉 영속성 컨텍스트와 무관하니 기존 영속성 컨텍스트에 존재하는 Team의 name들은 변경이 없는 것입니다.

Team 엔티티가 변경되지 않은 이유

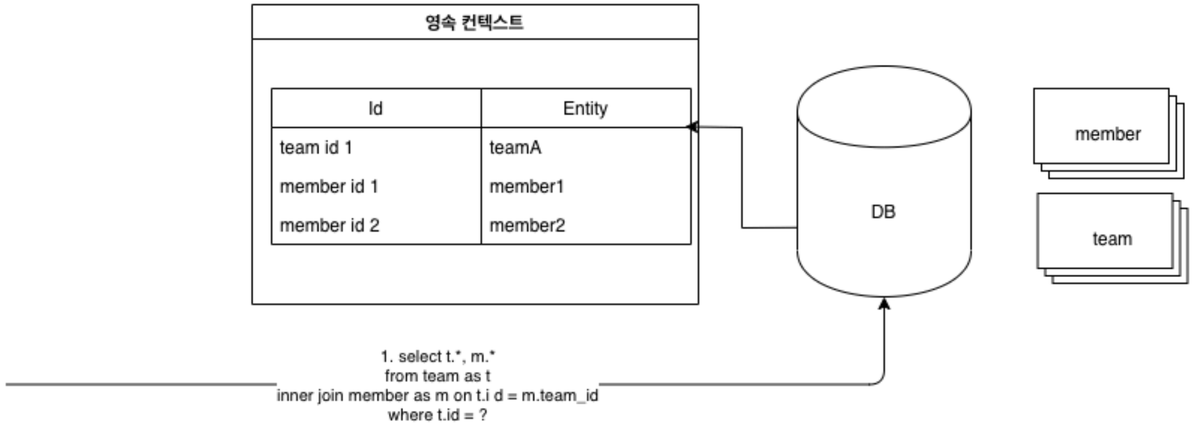
영속성 컨텍스트는 1차 캐시된 것이 우선이다.

`after update team` 를 조회했을 때는 영속성 컨텍스트와 무관했기 때문에 기존 데이터가 나왔지만 **신규 조회를 한 아래 코드 왜 변경 전 name이 출력 되었을까요? 그 이유는 영속성 컨텍스트 저장 메커니즘에 있습니다.**

```
val newSelectTeams = query.selectFrom(qTeam).fetch()

for (team in newSelectTeams) {
    println("new select team : $team")
}
```

영속성 컨텍스트 저장 메커니즘

영속성 컨텍스트 1차 캐시 저장 메커니즘 관련 포스팅에서 이미 포스팅한 적이 있습니다. 해당 글을



findById() 같은 경우는 영속성 컨텍스트를 먼저 찾고 영속성 컨텍스트에 해당 엔티티가 있으면 그 값을 바로 리턴합니다. 이를 1차 캐시라고 말합니다. **반면 JPQL은 영속성성 컨텍스트를 먼저 조회하지 않고 데이터베이스에 query 하여 결과를 가져옵니다. 그리고 아래와 같은 흐름으로 영속성 컨텍스트를 저장을 시도합니다.**

1. JPQL을 호출하면 데이터베이스에 우선적으로 조회한다.
2. 조회한 값을 영속성 컨텍스트에 저장을 시도한다.
3. 저장을 시도할 때 해당 데이터가 이미 영속성 컨텍스트에 존재하는 경우(영속성 컨텍스트에서는 식별자 값으로 식별) 데이터베이스에서 조회한 신규 데이터를 버린다.

즉 이미 영속성컨텍스트에 team들이 존재하기 때문에 신규 조회한 영속성 컨텍스트를 버리게 되는 것입니다. 영속성 컨텍스트에서는 ID를 기반으로 식별작을 합니다.

해결 방법

해결 방법은 간단합니다. **영속성 컨텍스트를 초기화(clear)을 하면 됩니다.** 영속성 컨텍스트가 초기화됐기 때문에 신규 조회한 `team` 객체들이 모두 영속성 컨텍스트에 그대로 반영되게 됩니다.

```
...
em.clear() // 영속성 컨텍스트를 초기화 한다.

val newSelectTeams = query.selectFrom(qTeam).fetch()

for (team in newSelectTeams) {
    println("new select team : $team")
}
```

```
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
new select team : Team(name=none name)
```

신규 조회한 `team name` 제대로 변경된 것을 확인할 수 있습니다.

JPA 선호하는 패턴

- 최근 JPA를 3년 가까이 사용하면서 개인적인으로 선호하는 패턴들을 한 번 정리하려고 한다.
- 어디까지 개인적으로 선호하는 패턴으로 굳이 이런 가이드를 따르지 않아도 된다.

컬럼 에노테이션 사용

```
@Entity
@Table(name = "member")
data class Member(
    @Column(name = "username", nullable = false)
    var username: String,

    @Column(name = "age", nullable = false)
    var age: Int = 0,

    @Column(name = "email", nullable = false, unique = true, updatable = false)
    var email: String
)
```

- 칼럼 에노테이션과 멤버 필명이 동일한 경우 칼럼에노테이션을 생략하는 경우도 있다. 그래서 일치하지 않은 것들만 작성하는 방법도 있지만 나는 모든 멤버 필드에 칼럼 에노테이션을 작성하는 패턴을 선호한다.
- 칼럼 에노테이션을 통해 nullable, unique, updatable 등등 여러 메타 정보를 전달해 줄 수 있다.
- 물론 필요한 경우에만 추가하고 기본적으로 작성하지 않아도 되는 패턴을 선택해도 되지만 개인적으로 이런 예매한 예외를 두는 가이드보다 조금 불편하더라도 다 작성하는 패턴이 좋다고 본다.
- 멤버 필드 리네임 관련 리팩토링에 과감해질 수 있다. 멤버 필드는 카멜케이스, 데이터베이스는 스네이크 케이스를 사용하는 경우가 흔한데 칼럼명이 긴 경우 가끔 오타 때문에 실수할 수 도 있다. 그러기 때문에 애초에 칼럼 에노테이션으로 다 작성하는 게 리팩토링 할 때도 안전성이 있다. 칼럼 에노테이션을 작성하면 멤버 필드 rename 관련 리팩토링을 과감하게 할 수 있다.

컬럼 에노테이션 사용

```
@Entity
@Table(name = "member")
data class Member(
    @Column(name = "username", nullable = false)
    var username: String,

    @Column(name = "age", nullable = false)
    var age: Int = 0,

    @Column(name = "email", nullable = false, unique = true, updatable = false)
    var email: String
)
```

- 칼럼 에노테이션과 멤버 필명이 동일한 경우 칼럼에노테이션을 생략하는 경우도 있다. 그래서 일치하지 않은 것들만 작성하는 방법도 있지만 나는 모든 멤버 필드에 칼럼 에노테이션을 작성하는 패턴을 선호한다.
- 칼럼 에노테이션을 통해 nullable, unique, updatable 등등 여러 메타 정보를 전달해 줄 수 있다.
- 물론 필요한 경우에만 추가하고 기본적으로 작성하지 않아도 되는 패턴을 선택해도 되지만 개인적으로 이런 예매한 예외를 두는 가이드보다 조금 불편하더라도 다 작성하는 패턴이 좋다고 본다.
- 멤버 필드 리네임 관련 리팩토링에 과감해질 수 있다. 멤버 필드는 카멜케이스, 데이터베이스는 스네이크 케이스를 사용하는 경우가 흔한데 칼럼명이 긴 경우 가끔 오타 때문에 실수할 수 도 있다. 그러기 때문에 애초에 칼럼 에노테이션으로 다 작성하는 게 리팩토링 할 때도 안전성이 있다. 칼럼 에노테이션을 작성하면 멤버 필드 rename 관련 리팩토링을 과감하게 할 수 있다.

엔티티에 과도한 에노테이션은 작성하지 않는다

```
@Entity
@Table(
    name = "member",
    indexes = [
        Index(columnList = "username"),
        Index(columnList = "age")
    ],
    uniqueConstraints = [
        UniqueConstraint(columnNames = ["username", "age"])
    ]
)
data class Member(
    ...
)
```

- 사실 위 칼럼 에노테이션 사용과는 좀 반대되는 개념이긴 한데 너무 과도한 에노테이션으로 엔티티 클래스의 비즈니스 코드의 집중도를 떨어트리기 때문에 선호하지 않는다.
- 물론 위 작업을 아주 간결하게 표현할 수 있다면 가능하면 작성하는 것도 좋지만, 해당 속성들은 비교적 변경사항이 잦기 때문에 작성하지 않는 것을 선호하는 이유이기도 하다.

조인 칼럼을 사용하자

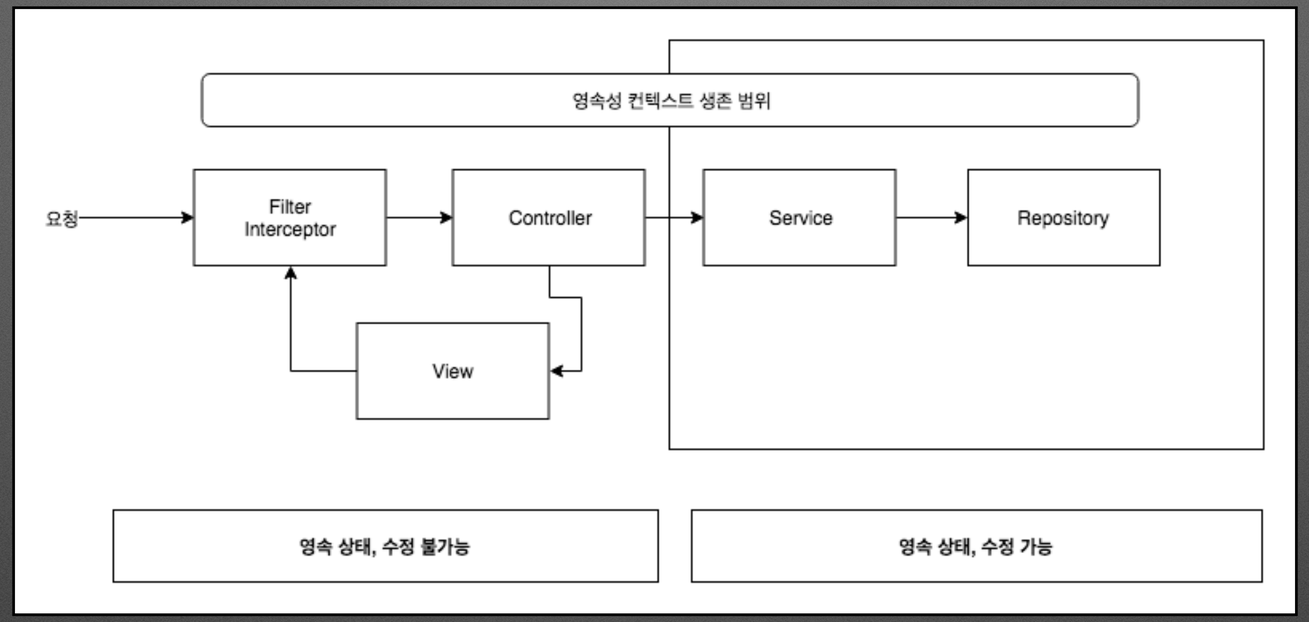
```
data class Member(
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "team_id", nullable = false) // 생략 가능
    var team: Team
) : EntityAuditing()
```

- OneToOne, ManyToOne 정보 같은 경우 연관관계의 주인에서 FK를 갖게 되기 때문에 칼럼 에노테이션을 작성할 수 있다. 연관관계 에노테이션으로 작성하면 기본적으로 PK 기반으로 되기 때문에 생략 가능하다.
- 하지만 칼럼 에노테이션을 사용하는 이유와 마찬가지로 조인 칼럼 에노테이션을 작성하는 것을 선호한다. nullable, unique, updatable 등 정보를 많이 표현해 준다.
- 무엇보다 nullabel 설정에 따른 조인 전략이 달라질 수 있어 성능상 이점을 얻을 수 있는 부분이 있다.
- 익숙하지 않은 프로젝트에서 칼럼명으로 엔티티 클래스를 찾는 경우가 있는데 이런 경우 해당 칼럼명으로 쉽게 검색할 수 있어 약간의 장점이 있다.

양방향 보다는 단방향으로 설정하자

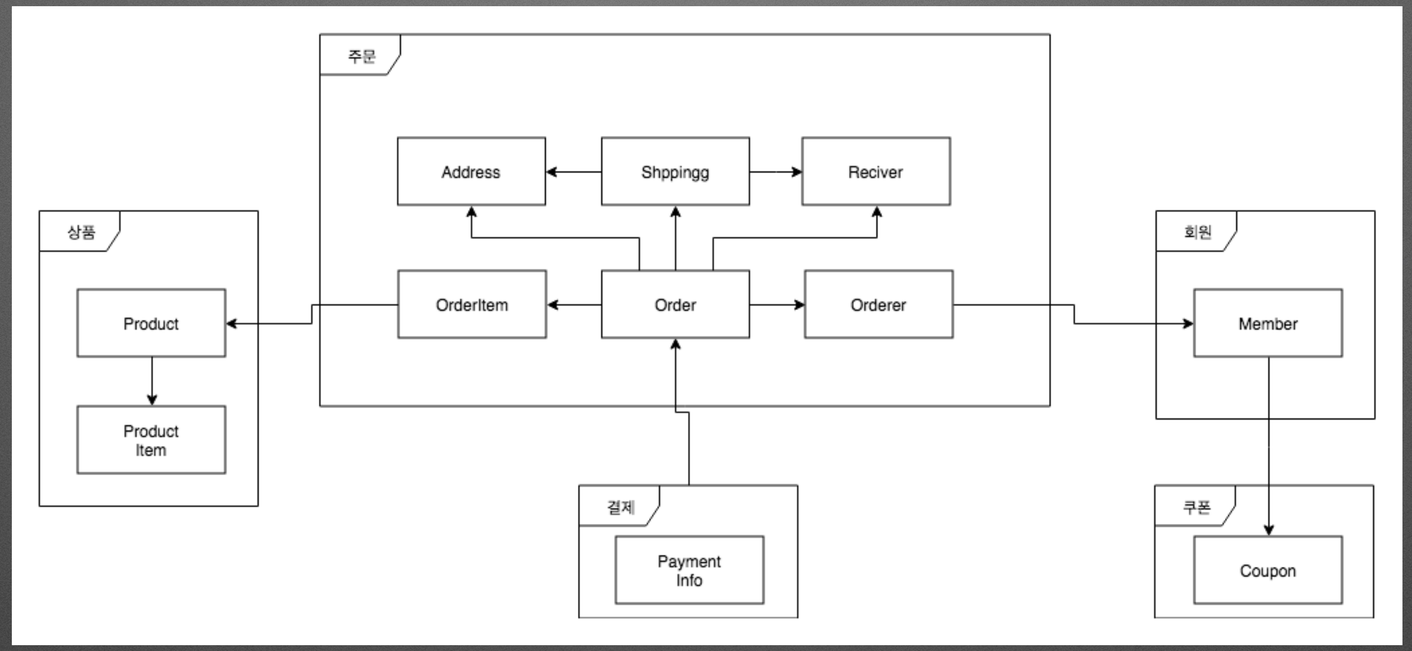
- 기본적으로 JPA 연관관계 설정은 단방향으로 설정하는 것을 선호한다. 사실 이것은 선호하기보다는 가이드 쪽에 가깝다.
- 양방향 연관관계를 작성하면 생각 보다 고려할 부분이 많다. 양방향 편의 메서드, 디펜던시 사이클 문제가 있기 때문이다.
- 무엇보다 OnyToMany, ManyToOne 관계는 설정은 선호하지 않는다. OneToMany를 관계를 갖게 되면 N+1 문제(물론 OneToMany를 관계에서만 N+1 문제가 발생하지는 않는다.)
- OneToMany 2개 이상 갖는 경우 카테시안 곱 문제로 MultipleBagFetchException 발생한다. 양방향 메서드를 반드시 지정해야 할 이유가 없다면 단방향 관계를 지향하는 것이 좋다고 생각한다.

Open Setting View false로 두자



- OSVI는 영속성 컨텍스트를 View 레이어까지 열어 둘 수 있다. 영속성 컨텍스트가 살아있으면 엔티티는 영속 상태로 유지된다. 따라서 뷰에서도 지연 로딩을 사용할 수 있다. 하지만 View에서 트랜잭션을 계속 가지고 있는 것이 좋다고 생각하지 않는다.
- 기존 OSVI 문제를 해결하기 위해서 비즈니스 계층에서만 트랜잭션을 유지하는 방식의 OSVI를 스프링에서 지원해주고 있긴 하지만 ModelAndView를 사용하지 않고 단순하게 API로 JSON 같은 응답을 내려주는 서버로 사용한다면 View false를 false 두는 것이 좋다.(기본이 true이다)

객체 그래프 탐색은 어디까지 할것인가? 좀 끊자!



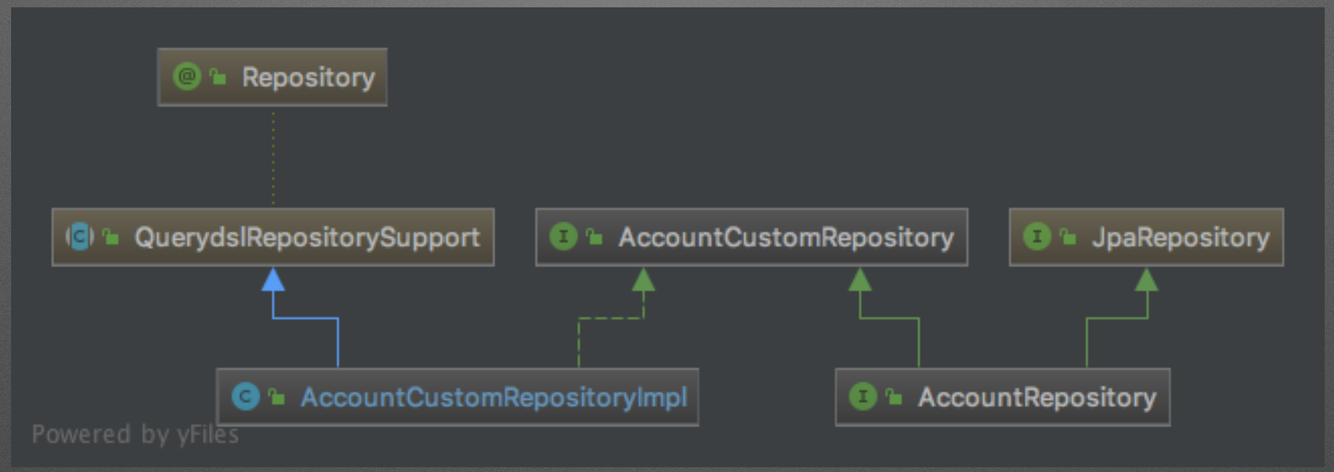
- 관계형 데이터베이스는 FK를 통해서 연관 탐색을 계속 진행할 수 있다. 마찬가지로 JPA도 객체 그래프 탐색을 통해서 연관 탐색을 계속 진행할 수 있다. 이것은 그래프 탐색의 오용이라고 생각한다.
- DDD를 프로덕션 레벨에서 진행해본 적은 없지만 DDD에서 주장하는 에그리거트 단위, 그 단위가 다르면 객체 탐색을 끊는 것은 매우 동의한다.

객체 그래프 탐색은 어디까지 할것인가? 좀 끊자!

```
@Entity
@Table(name = "orders")
data class Order(
    @Column(name = "amount", nullable = false)
    var amount: BigDecimal,

    @Embedded
    var orderer: Orderer

) : EntityAuditing()

@Embeddable
data class Orderer(
    @Column(name = "member_id", nullable = false, updatable = false)
    var memberId: Long,

    @Column(name = "email", nullable = false, updatable = false)
    var email: String
)
```

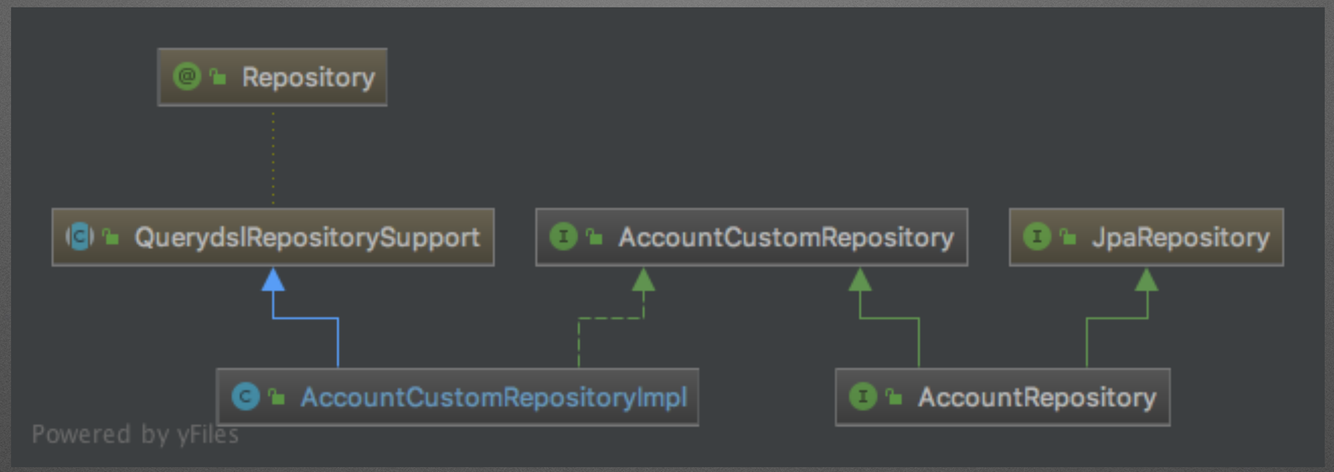
- 연관관계를 명확하게 끊는 것이 옳다고 본다. 연관관계를 계속 탐색할 수 있는 것도 문제지만 연관관계를 갖게 되면 주문을 조회한 이후에 회원 정보를 변경할 수도 있다. **Order 객체의 책임과 역할이 어디까지 인가를 봤을 때 이렇게 명확하게 끊는 것**이 옳다고 본다. 만약 **조회가 필요한 경우 Orderer객체에 memberId가 있으니 회원정보가 필요시에 조인**해서 가져오면된다.
- 이것도 가이드 수준으로 생각하고 있지만 아직 DDD를 실무에서 진행해본 적이 없어 막연하게 드는 생각이라 가능하면 에그리거트 기준으로 연관관계를 끊는 것이 좋다고 생각만 하고 있다.

쿼리 메서드는 선호하지 않는다.



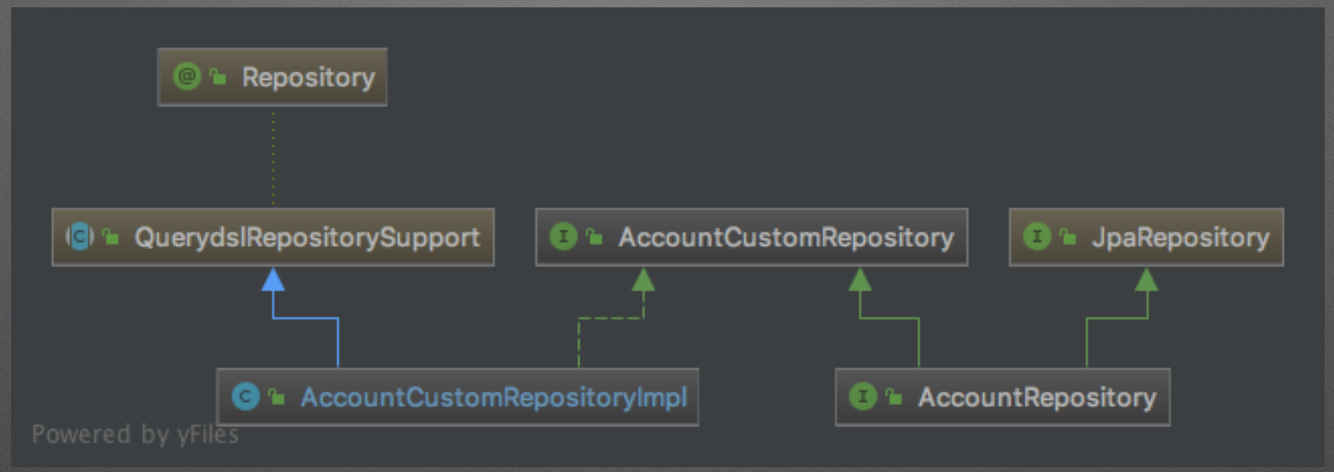
- 이건 정말 개인적인 선호도이다. 일단 조건이 까다로운 조회용 코드인 경우 쿼리메서드로 표현하면 너무 장황해서 코드 가독성이 좋지 않다고 본다.
- 쿼리 메서드를 사용하지 않고 QuerydslRepositorySupport를 이용한 Query DSL 기반으로 모두 작성하는 것을 선호한다. 물론 findByEmail 같은 것들은 쿼리 메서드가 더 편리하고 직관적이다고 생각하지만 위에서도 한 번 언급했지만 예외를 하나를 허용하면 추가적인 예외가 생기게 되기 때문에 아주 명확한 가이드가 없다면 모두 Query DSL 기반으로 작성하는 것을 선호한다.

쿼리 메서드는 선호하지 않는다.



- 또 엔티티의 멤버 필드 변경 시에도 쿼리 메서드의 변경돼야 하는 것도 문제라고 생각한다. 물론 인텔리 제이에서 멤버 필드 rename 시에 적절하게 변경해주기도 하고, 스프링에서 쿼리 메서드가 문제가 있는 경우 스프링 구동시 예외가 발생하기 때문에 최소한의 안전장치가 있긴 하지만(이것도 없었다면 쿼리 메서드는 사용하지 않는 것을 가이드 했을 거 같다) 이러한 문제 때문에도 쿼리 메서드를 선호하지 않는다.
- 물론 Query DSL도 멤버 필드 변경 시에는 문제가 발생한다. 그 문제는 Qxx.class관련 Syntax이기 때문에 더 명확하다는 장점이 있다고 본다.

쿼리 메서드는 선호하지 않는다.



- 지극히 주관적인 생각이지만 쿼리 메서드는 인터페이스 기반으로 작성하기 때문에 테스트 코드를 작성하지 않는 경우도 많다. 반면 **Query DSL 세부 클래스를 작성하기 때문에 뭔가 테스트를 더 작성하게 하는 심리적인 이유도 있는 거 같다.**
- 위 코드처럼 **QuerydslRepositorySupport을 이용하면 Repository를 통해서 세부 구현체의 코드를 제공하기 때문에 이런 식의 패턴을 선호한다.** 해당 패턴은 step-15: Querydsl를 이용해서 Repository 확장하기 (1)에서 자세하게 다룬 적 이 있다.

Auditing 상속 구조로 사용하자

```
@EntityListeners(value = [AuditingEntityListener::class])
@MappedSuperclass
abstract class EntityAuditing {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long? = null
        private set

    @CreationTimestamp
    @Column(name = "created_at", nullable = false, updatable = false)
    lateinit var createdAt: LocalDateTime
        private set

    @UpdateTimestamp
    @Column(name = "updated_at", nullable = false)
    lateinit var updatedAt: LocalDateTime
        private set

}

@Entity
@Table(name = "member")
data class Member(
    @Column(name = "username", nullable = false)
    var username: String,

    @Column(name = "age", nullable = false)
    var age: Int = 0,

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "team_id", nullable = false)
    var team: Team
) : EntityAuditing()
```

- @MappedSuperclass, @EntityListeners를 사용하면 **반복적인 id, createdAt, updatedAt을 코드를 상속받아 해결할 수 있다.** 예전에는 이런 식의 상속 구조는 올바르지 않다고 생각해서 이런 패턴에 거부감이 있었다.

- 하지만 요즘에는 잘 사용하고 있다. **순수한 객체지향 코드를 유지하는 것도 중요하지만 결국 실리성 있는 부분을 택하는 것도 중요하다고 본다.** 특히 코틀린으로 넘어오면서 id, createdAt, updatedAt 등 우리가 직접적으로 핸들링하지 않는 값에 대한 바인딩 방법이 많기 때문에 각각 엔티티에서 처리하면 코드 통일성이 떨어진다. 그래서 이렇게 처리하는 것이 좋다고 생각한다.

부록

Properties 관리

안티패턴 : Environment

```java
public class AntiSamplePropertiesRunner implements ApplicationRunner {
    private final Environment env;

    @Override
    public void run(ApplicationArguments args)  {
        final String email = env.getProperty("user.email");
        final String nickname = env.getProperty("user.nickname");
        final int age = Integer.valueOf(env.getProperty("user.age"));
        final boolean auth = Boolean.valueOf(env.getProperty("user.auth"));
        final int amount = Integer.valueOf(env.getProperty("user.amount"));

        log.info("=========ANTI=========");
        log.info(email); // "yun@test.com"
        log.info(nickname); // yun
        log.info(String.valueOf(age)); // 27
        log.info(String.valueOf(auth)); // true
        log.info(String.valueOf(amount)); // 100
        log.info("=========ANTI=========");
    }
}
```

properties

```yaml
user:
  email: "yun@test.com"
  nickname: "yun"
  age: 28
  auth: false
  amount: 101
```

정확한 자료형 확인의 어려움

- key 값으로 어느 정도 유추할 수 있지만 어디까지나 유추이지 정확한 자료형을 확인하기 위해서는 properties에서 value 값을 기반으로 자료형을 확인해야 합니다.
- amount 값이 100 이기 때문에 int 타입으로 바인딩시켰지만 amount 값은 소수로 값이 변경될 수도 있습니다. 이 또한 값을 통해서 자료형을 유추했기 때문에 발생한다고 생각합니다.

변경시 관리의 어려움

- email의 키값이 email-address로 변경됐을시 getProperty() 메서드를 통해서 바인딩 시킨 부분들은 모두 email-address로 변경해야 합니다.
- 변경하는 것도 문제지만 만약 1개의 메소드라도 실수로 놓쳤을 경우 에러 발생 시점에 runtime으로 넘어가게 되고 해당 에러가 NullPointException이 발생하기 전까지는 확인하기 어렵습니다.

추천 패턴 : ConfigurationProperties

```java
@Configuration
@ConfigurationProperties(prefix = "user")
@Validated
public class SampleProperties {
    @Email
    private String email;
    @NotEmpty
    private String nickname;
    private int age;
    private boolean auth;
    private double amount;

    // getter, setter
}

public class SamplePropertiesRunner implements ApplicationR
    private final SampleProperties properties;
    @Override
    public void run(ApplicationArguments args)  {
        final String email = properties.getEmail();
        final String nickname = properties.getNickname();
        final int age = properties.getAge();
        final boolean auth = properties.isAuth();
        final double amount = properties.getAmount();

        log.info("==================");
        log.info(email); // yun@test.com
        log.info(nickname); // yun
        log.info(String.valueOf(age)); // 27
        log.info(String.valueOf(auth)); // true
        log.info(String.valueOf(amount)); // 100.0
        log.info("==================");
    }
}
```

```yaml
user:
  email: "yun@" // 이메일 형식 올바르지 않음 -> @Email
  nickname: "" // 필수 값 -> @NotEmpty
```

```
Binding to target com.cheese.springjpa.properties.SampleP

    Property: user.email
    Value: yun@
    Reason: 이메일 주소가 유효하지 않습니다.

Binding to target com.cheese.springjpa.properties.SampleP

    Property: user.nickname
    Value:
    Reason: 반드시 값이 존재하고 길이 혹은 크기가 0보다 커야 합니다.
```

추천 패턴 : ConfigurationProperties Immutable Properties

```kotlin
@ConstructorBinding
@ConfigurationProperties(prefix = "user")
@Validated
data class UserProperties(
        @field:Email
        val email: String,

        @field:NotEmpty
        val nickname: String,

        @field:Min(10)
        val age: Int,

        @field:NotNull
        val auth: Boolean,

        @field:Min(10)
        val amount: Double,

        @DateTimeFormat(pattern = "yyyy-MM-dd")
        @field:NotNull
        val date: LocalDate
)
```

```kotlin
@SpringBootApplication
@EnableConfigurationProperties(UserProperties::class)
class KotlinJunit5Application

fun main(args: Array<String>) {
    runApplication<KotlinJunit5Application>(*args)
}
```

- @ConstructorBinding을 추가하고 data class으로 넘겨받을 값을 생성자로 지정합니다.
- 스프링 부트 2.2.1 버전부터는 @ConfigurationProperties scanning 방식이 disabled 되면서 @EnableConfigurationProperties을 따로 지정해야 합니다.

END