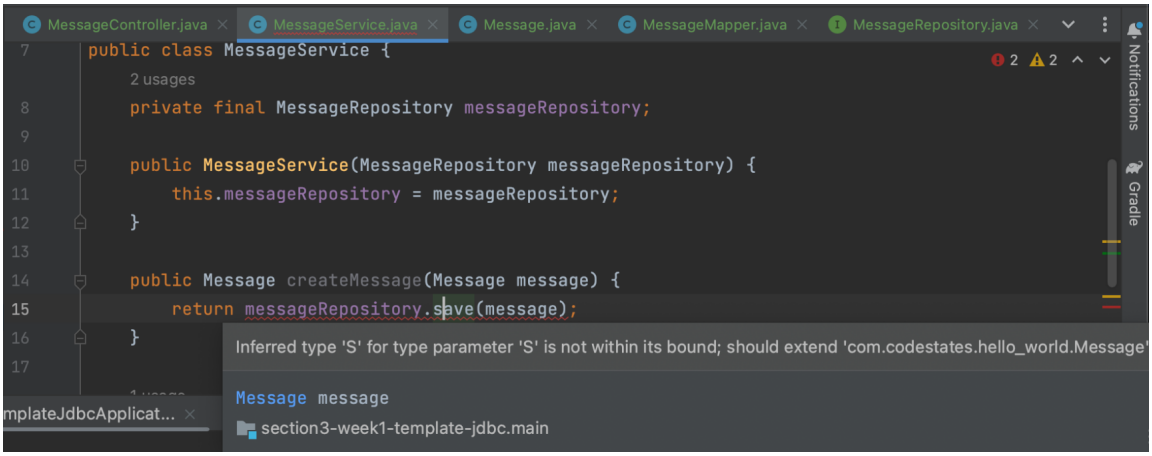
🎀 라이브 세션-2022.12.26(월)-Spring Data JDBC를 이용한 데이터 액세스 실습

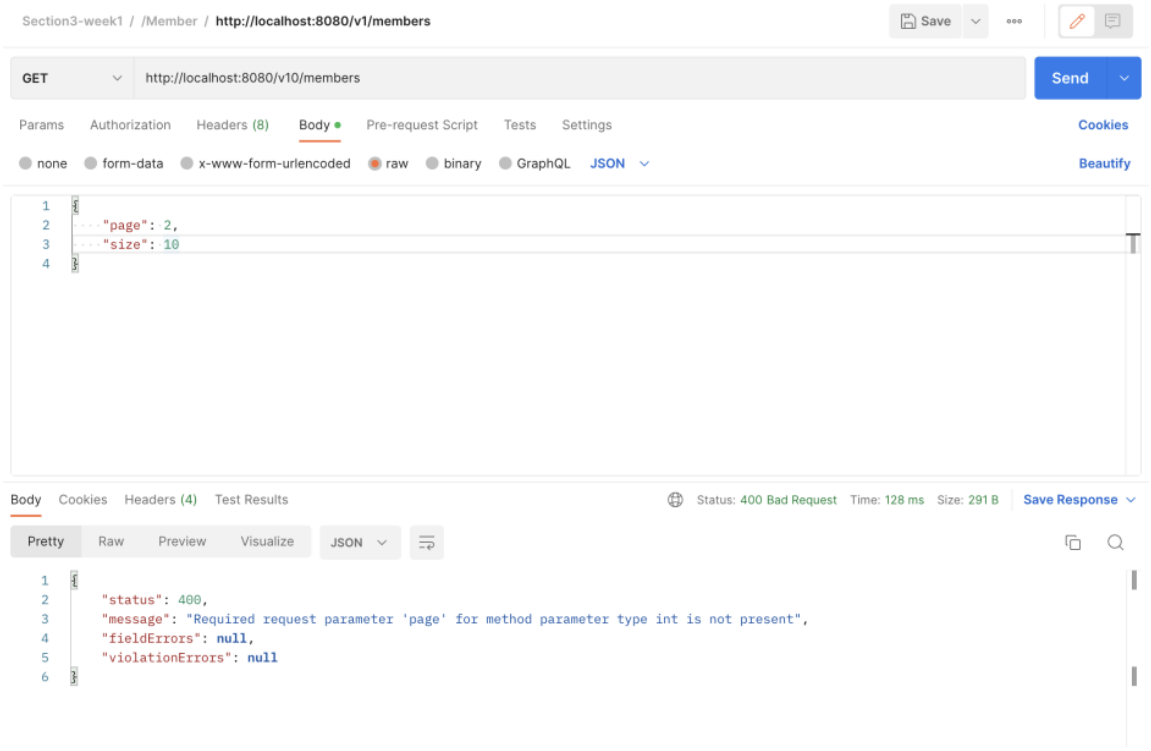
✅ 들어온 질문

- 아래와 같은 빨간줄이 떠요 ㅜㅜ



  - 클래스 import 오류입니다.

- 아래 그림처럼 request body를 전송했더니 그림과 같은 response body를 전달 받아요.



- `@RequestParam`으로 파라미터로 전달 받을 경우, **Query Parameter로 파라미터를 전송해야 합니다.**

- 아래 그림과 같은 오류가 떴습니다. 이유를 모르겠습니다.

```
Caused by: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'memberService' defined in file
[C:\Users\SSH\Documents\CST\be-homework-jdbc\build\classes\java\main\com\codestates\member\service\MemberService.class]: Unsatisfied dependency
expressed through constructor parameter 0; nested exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with
name 'memberRepository' defined in com.codestates.member.repository.MemberRepository defined in @EnableJdbcRepositories declared on
JdbcRepositoriesRegistrar.EnableJdbcRepositoriesConfiguration: Invocation of init method failed; nested exception is org.springframework.data
.repository.query.QueryCreationException: Could not create query for public abstract org.springframework.data.domain.Page com.codestates.member
.repository.MemberRepository.findAllByPage(org.springframework.data.domain.Pageable)! Reason: No property 'page' found for type 'Member'! Did you
mean ''name''?; nested exception is org.springframework.data.mapping.PropertyReferenceException: No property 'page' found for type 'Member'! Did
you mean ''name''?
```

- findByXXXXX() 메서드의 사용법에 오류가 있는 것 같습니다.

✅ 이 전 시간까지 배운내용 10분 리뷰

- **API 계층에 대한 리뷰는 오늘 이후로 졸업 할게요!**

✅ 지난 시간까지 배운 Spring Data JDBC 리뷰

1️⃣ DDD란?

- 도메인 주도 설계(Domain Driven Design)
- 한마디로 모든 기능이 도메인 모델 위주로 돌아가는 설계 기법
- 도메인(Domain)이란?
  - **비즈니스적인 어떤 업무 영역**
  - **우리가 실제로 현실 세계에서 접하는 업무의 한 영역**
- 코드로 이해

2️⃣ 빈약한 도메인 모델

MemberService(서비스 클래스)

```java
@Service
public class MemberService {
    private final MemberRepository memberRepository;
    private final JdbcTemplate jdbcTemplate;
    public MemberService(MemberRepository memberRepository, JdbcTemplate jdbcTemplate) {
        this.memberRepository = memberRepository;
        this.jdbcTemplate = jdbcTemplate;
    }

    public Member createMember(Member member) {...}

    public Member updateMember(Member member) {...}

    public Member findMember(long memberId) { return findVerifiedMember(memberId); }

    public List<Member> findMembers() {...}

    public void deleteMember(long memberId) {...}

    public Member findVerifiedMember(long memberId) {...}

    private void verifyExistsEmail(String email) {...}
}
```

서비스 클래스에 기능 집중

Member(도메인 엔티티 클래스)

```java
@Getter
@Setter
@NoArgsConstructor
public class Member {
    @Id
    private Long memberId;

    private String email;

    private String name;

    private String phone;
}
```

기능이 없는 빈약한 도메인 모델

3️⃣ 풍부한 도메인 모델

MemberService(서비스 클래스)

```java
@Service
public class MemberService {
    private final MemberRepository memberRepository;
    private final JdbcTemplate jdbcTemplate;
    public MemberService(MemberRepository memberRepository, JdbcTemplate jdbcTemplate) {
        this.memberRepository = memberRepository;
        this.jdbcTemplate = jdbcTemplate;
    }
    ...
    ...
}
```

서비스 클래스의 기능 축소

기능 이전

Member(도메인 엔티티 클래스)

```java
@Getter
@Setter
@NoArgsConstructor
public class Member {
    @Id
    private Long memberId;

    private String email;

    private String name;

    private String phone;

    public Member createMember(Member member) {...}

    public Member updateMember(Member member) {...}

    public Member findMember(long memberId) { return findVerifiedMember(memberId);}

    public List<Member> findMembers() {...}

    public void deleteMember(long memberId) {...}

    public Member findVerifiedMember(long memberId) {...}

    private void verifyExistsEmail(String email) {...}
}
```

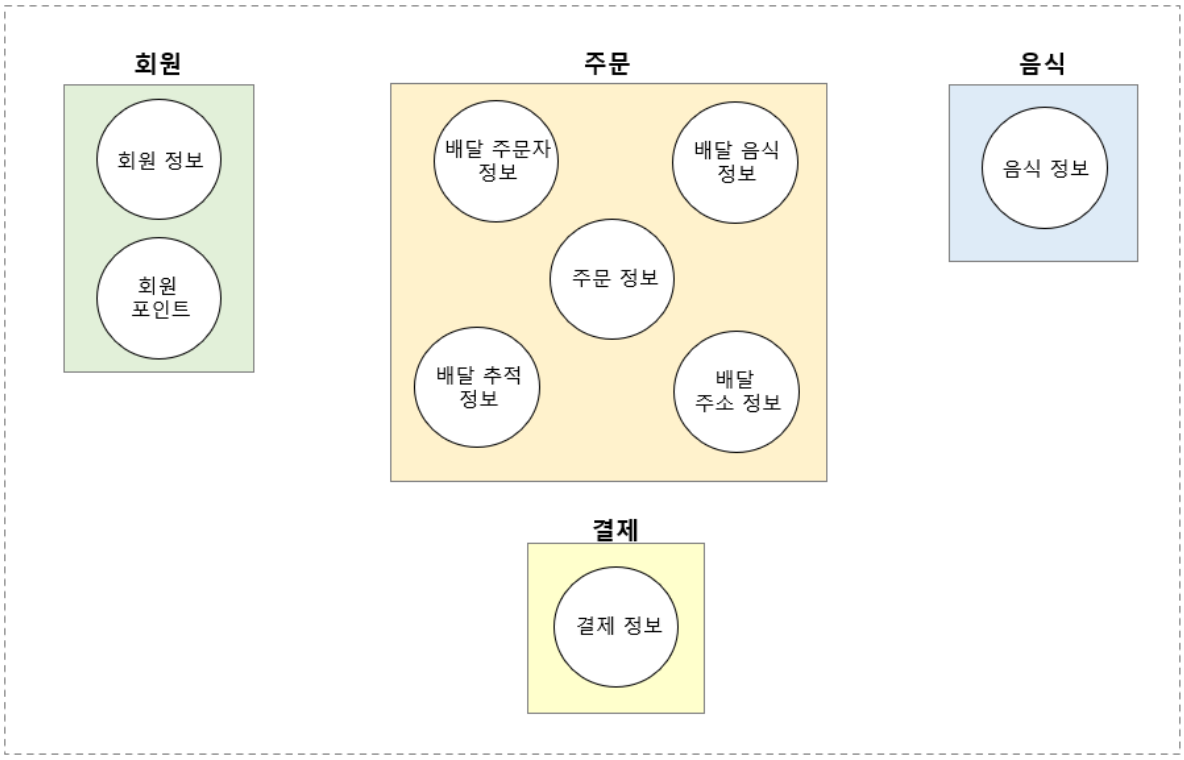
기능이 많은 풍부한 도메인 모델(Rich Domain)

4️⃣ 애그리거트(Aggregate)란?

비슷한 업무 도메인들의 묶음

배달 주문 앱의 도메인 모델 예

배달 주문 앱 도메인에서의 애그리거트(Aggregate)

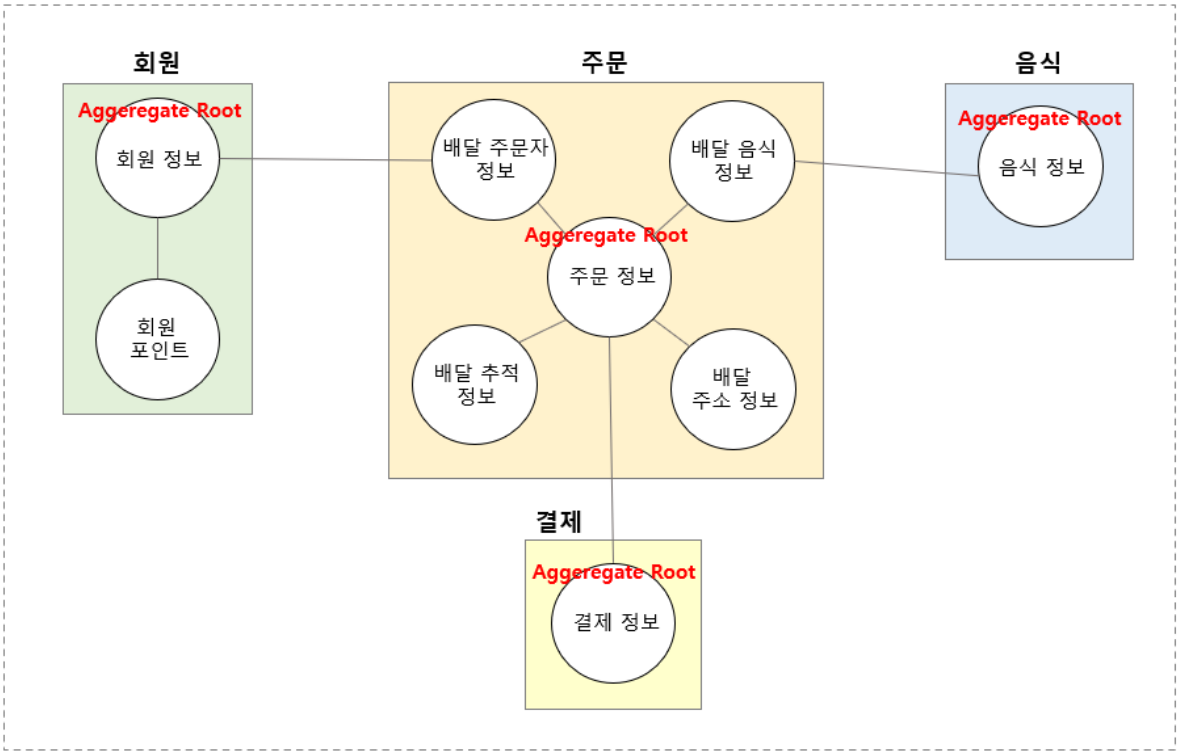


5️⃣ 애그리거트 루트(Aggregate Root)

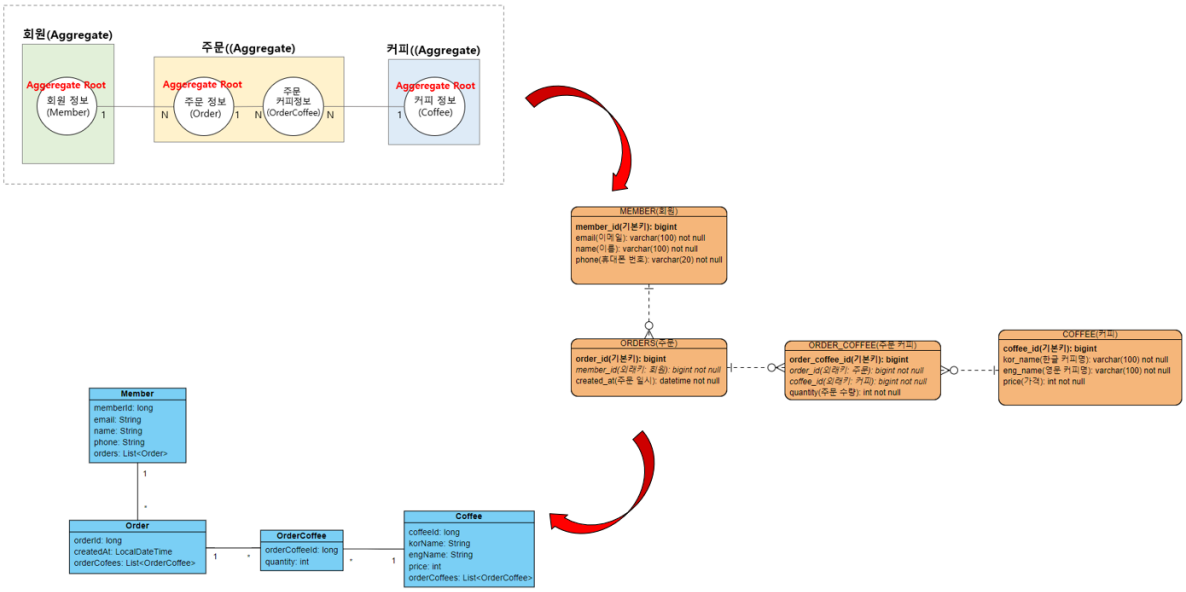
하나의 애그리거트를 대표하는 도메인

6️⃣ 배달 주문 앱 도메인에서의 애그리거트(Aggregate)와 애그리거트 루트(Aggregate Root)

- 애그리거트에서 대장 격인 도메인
- 다른 도메인과 직간접적으로 연결되는 도메인
- 데이터베이스의 테이블 간 관계에서는 부모 테이블이 애그리거트 루트, 자식 테이블은 애그리거트 루트가 아닌 다른 도메인이 된다.



✅ 커피 주문 샘플 애플리케이션 테이블 및 도메인 엔티티 설계



✅ 기능 구현

무엇부터 해야 될까요?

⭐ 두 가지 접근 방식

- 누가(클래스) 무엇(의존하는 클래스)을 우선적으로 필요로 하는지 생각해보기
  - Top Down
    - 상위 계층부터 구현
    - 일종의 TDD 방식으로 진행할 수 있다.
  - Bottom Up
    - 하위 계층부터 구현

1️⃣ 도메인 엔티티 클래스 정의 및 연관 관계 매핑

Member와 Order의 관계

Order와 Coffee의 관계

2️⃣ 리포지토리 구현

- 아래의 (1)과 같이 `@Param 애너테이션을 추가하지 않을 경우`, 오류가 나는 이유

CoffeeRepository

```
@Query("SELECT * FROM COFFEE WHERE COFFEE_ID = :coffeeId")
Optional<Coffee> findByCoffee(Long coffeeId);  // (1)
```

: Gradle로 빌드하지 않을 경우 발생할 수 있는 오류입니다.

Spring Data JDBC 공식 문서 내용

```
Spring fully supports Java 8’s parameter name discovery based on the -parameters
compiler flag.
By using this flag in your build as an alternative to debug information,
you can omit the @Param annotation for named parameters.
```

: `@Param` 애너테이션을 추가하지 않아도 이름 기반의 파라미터를 사용할 수 있다는 내용.

3️⃣ 서비스 클래스

4️⃣ 기타 주문 기능 수정으로 변경된 클래스

- Controller
- Mapper
- DTO

✅ N + 1 문제(Advanced)

1️⃣ 주문한 커피(OrderCoffee)를 조회하는 로직의 문제점

- 1건의 주문 정보당 **여러 건의 주문한 커피 정보를 DB에서 추가로 조회하는 문제**
- 특정 주문 정보 1건에 주문한 커피 정보가 3건일 경우, 매번 커피 정보를 조회할 때의 쿼리 발생 수
  - 주문 정보(1) x 주문한 커피 정보(3건) = **3건의 쿼리 발생**
- 특정 주문 정보 10건에 주문한 커피 정보가 3건일 경우, 매번 커피 정보를 조회할 때의 쿼리 발생 수
  - 주문 정보(10) x 주문한 커피 정보(3건) = **30건의 쿼리 발생**

2️⃣ 개선 방법

- 쿼리 빌드 메서드를 이용한 방법
  - **IN 조건절 이용**

- 네이티브 쿼리에서의 **JOIN을** 이용한 방법
  - **JOIN** + **읽기 전용 엔티티**를 이용해 한 번에 데이터를 조회할 수 있다.
    - 특정 주문 정보 10건에 주문한 커피 정보가 3건일 경우 조회할 때의 쿼리 발생 수

```
SELECT O.*, OC.QUANTITY, C.* FROM ORDERS O
INNER JOIN ORDER_COFFEE OC ON O.ORDER_ID = OC.ORDER_ID
INNER JOIN COFFEE C ON OC.COFFEE_ID = C.COFFEE_ID
ORDER BY O.ORDER_ID DESC
```

    - 주문 정보 + 주문한 커피 정보를 JOIN으로 한번에 가져오므로 주문에 해당하는 **10건의 쿼리 발생**

- 요구 사항에 따라 DTO와 Mapper가 필요없을 수도 있다.

✅ 데이터 타입이 서로 다른 DTO와 엔티티 클래스를 매핑하는 또 다른 방법

- **어차피 양쪽 클래스의 데이터 타입만 맞춰주면 된다.**
- 문제는 **데이터 타입을 어디서 맞춰주느냐** 하는 부분
  - **Mapper에서 맞춰주기**
    - Mapper가 할일이 많아진다.
    - 대신에 DTO 클래스와 Entity 클래스의 코드가 단순해진다.
    - Mapping 관련된 로직은 Mapper만 들여다보면 되므로 유지보수가 상대적으로 더 용이하다.
    - Mapping이라는 일을 Mapper가 하는게 자연스럽다.
  - DTO 클래스 또는 Entity 클래스에서 맞춰주기
    - Mapper의 코드가 상대적으로 간결해진다.
    - DTO 또는 Entity 클래스 코드의 양과 복잡도가 늘어난다.
    - 자칫 DTO와 Mapper 둘 다 관리해야하므로 유지보수가 상대적으로 더 어렵다.
    - Mapper가 있다면 Mapper가 일을 하도록 시키는게 바람직하지 않을까?

✅ 실습 과제 리뷰

- 페이지네이션 실습 과제 solution 코드 설명

✅ 페이지네이션 처리 방식

1️⃣ 오프셋 방식

- 가져와야 되는 데이터까지 오프셋 개수 만큼 카운팅해서 찾아가는 방식
- **직접 찾아가야 되므로 시간이 좀 걸릴 수 있다.**
- 100만 건 ~ 200만 건 정도에서 클라이언트가 심하게 느리다고 할 정도는 아님
- 쿼리 예

```
SELECT *
FROM MEMBER
ORDER BY MEMBER_ID DESC LIMIT 10 OFFSET 500_000;
```

2️⃣ 커서 방식

- WHERE 절을 조건으로 마지막으로 조회한 부분부터 탐색하므로 속도가 빠르다.
- **마지막 조회한 위치를 가지고 있어야 한다.**
- 쿼리 예

```
SELECT *
FROM MEMBER
WHERE ID < {이 전에 조회한 마지막 MEMBEr_ID}  //<- 커서
ORDER BY MEMBER_ID DESC LIMIT 10;
```
