카카오페이 기술과제 (2022-09-30)

생성일	@2022년 9월 29일 오후 8:34
태그	

주요 버전 정보

- java 11
- spring boot 2.7.x
- gradle 6.9.2
- mysql 8.x

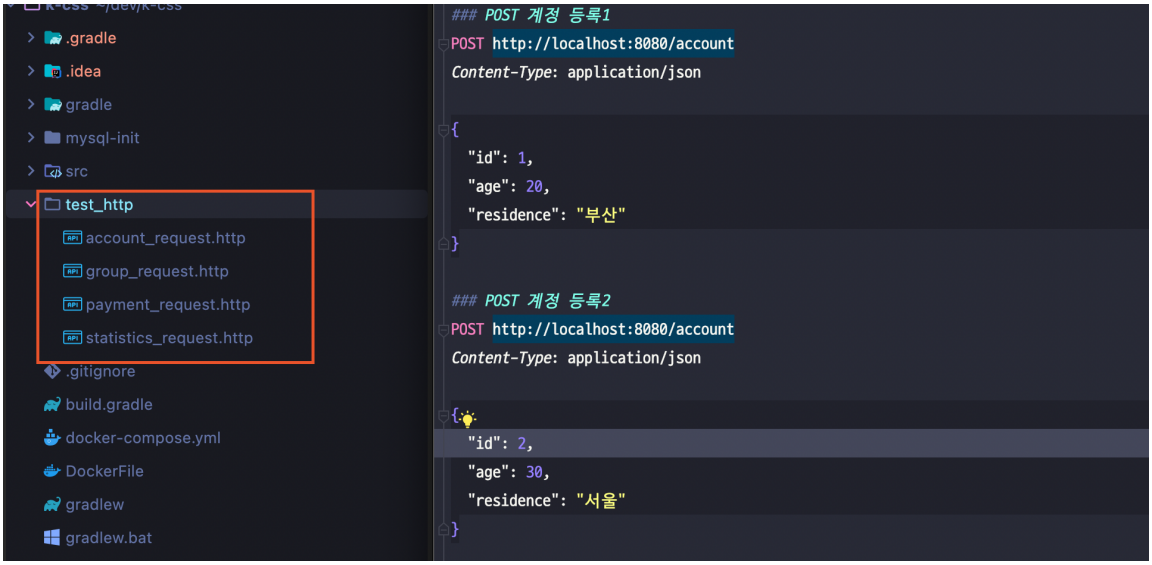
실행 방법

1. 빌드 및 docker-compose 명령어로 실행합니다.

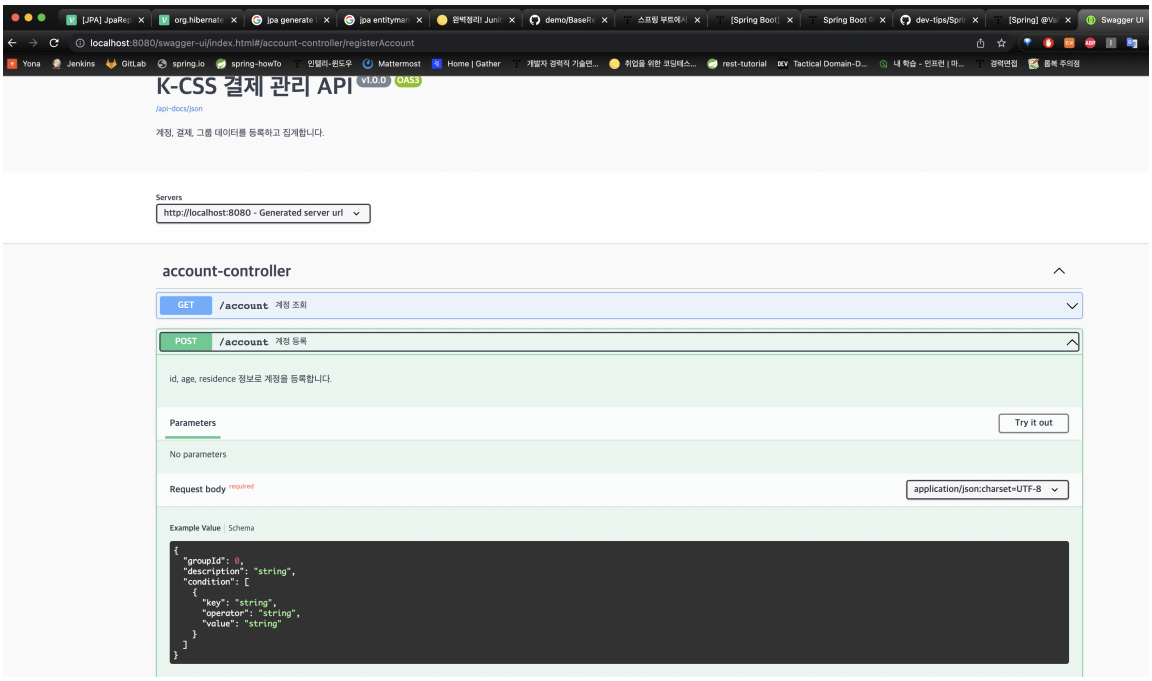
```
# boot jar 파일 생성을 위한 빌드 명령어
$ ./gradlew bootJar

# 도커 실행 명령어 - [spring boot app, MySQL 컨테이너 실행]
$ docker-compose up -d
```

테스트 방법

1. 첨부된 `.http` 파일 이용
    - root 디렉토리 하위 `test_http` 디렉토리 내에 API 호출을 위한 `.http` 파일을 첨부했습니다.



2. `api-docs(swagger)` 이용
   - 실행 방법을 참고하여 어플리케이션을 실행한 뒤 `http://localhost:8080/api-docs` 에 접속하여 테스트할 수 있습니다.



API 목록 및 실행 결과

API는 과제 가이드의 샘플과 최대한 동일하게 구성했습니다.

1. 계정 생성 및 조회

```
### 계정 생성 Request
POST http://localhost:8080/account
Content-Type: application/json

{
  "id": 1,
  "age": 20,
  "residence": "부산"
}

### 계정 생성 Response
...
{
  "success": true,
  "errorMessage": null
}

-----------------------

### 계정 조회 Request
GET http://localhost:8080/account?id=1
Accept: application/json

### 계정 조회 Response
...
{
  "id": 1,
  "residence": "부산",
  "age": 20,
  "payments": [
    {
      "paymentId": 1,
      "accountId": 1,
      "amount": 152000.0,
      "methodType": "송금",
      "itemCategory": "패션",
      "region": "충남"
    },
...
  ]
}
```

2. 결제 처리 및 조회

```
### 결제 처리(생성) Request
POST http://localhost:8080/payment
Content-Type: application/json
```

```
{
  "paymentId": 1,
  "accountId": 1,
  "amount": 152000.0,
  "methodType": "송금",
  "itemCategory": "패션",
  "region": "충남"
}

### 결제 처리(생성) Response
...
{
  "success": true,
  "errorMessage": null
}

-----------------------

### 결제 조회 Request
GET http://localhost:8080/payment?id=1
Accept: application/json

### 결제 조회 Response
{
  "paymentId": 1,
  "accountId": 1,
  "amount": 152000.0,
  "methodType": "송금",
  "itemCategory": "패션",
  "region": "충남"
}
```

3. 결제 그룹 생성, 삭제, 조회

```
### Group 생성 Request
POST http://localhost:8080/group
Content-Type: application/json

{
  "groupId": 1,
  "description": "카드 결제 그룹",
  "condition": [
    {
      "key": "methodType",
      "operator": "equals",
      "value": "CARD"
    }
  ]
}

### Group 생성 Response
...
```

```
{
  "success": true,
  "errorMessage": null
}

---------------------------

### 특정 Group 조회 Request
GET http://localhost:8080/group?id=1
Accept: application/json

### 특정 Group 조회 Response
...
[
  {
    "groupId": 1,
    "description": "카드 결제 그룹",
    "condition": [
      {
        "key": "methodType",
        "operator": "equals",
        "value": "CARD"
      }
    ]
  }
]


---------------------------

### 모든 Group 조회 Request
GET http://localhost:8080/group
Accept: application/json

### 모든 Group 조회 Response
...
[
  {
    "groupId": 1,
    "description": "카드 결제 그룹",
    "condition": [
      {
        "key": "methodType",
        "operator": "equals",
        "value": "CARD"
      }
    ]
  },
  {
    "groupId": 2,
    "description": "제주지역에서 1000 ~ 2000 사이 결제",
    "condition": [
      {
        "key": "region",
        "operator": "equals",
        "value": "제주"
      },
      {
```

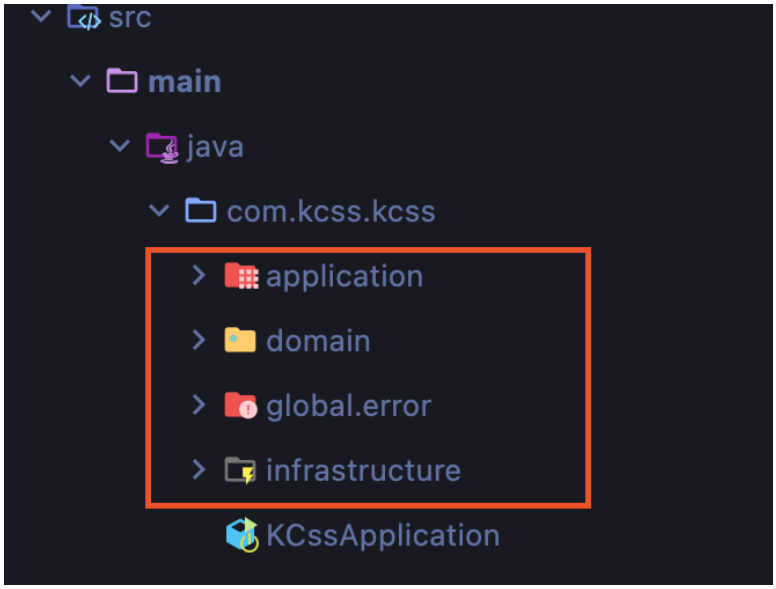
```
        "key": "amount",
        "operator": "between",
        "value": "[1000, 2000]"
      }
    ]
  },
...
]

---------------------------------

### 그룹 삭제 Requset
DELETE http://localhost:8080/group?id=3
Accept: application/json

### 그룹 삭제 Response
{
  "success": true,
  "errorMessage": null
}
```

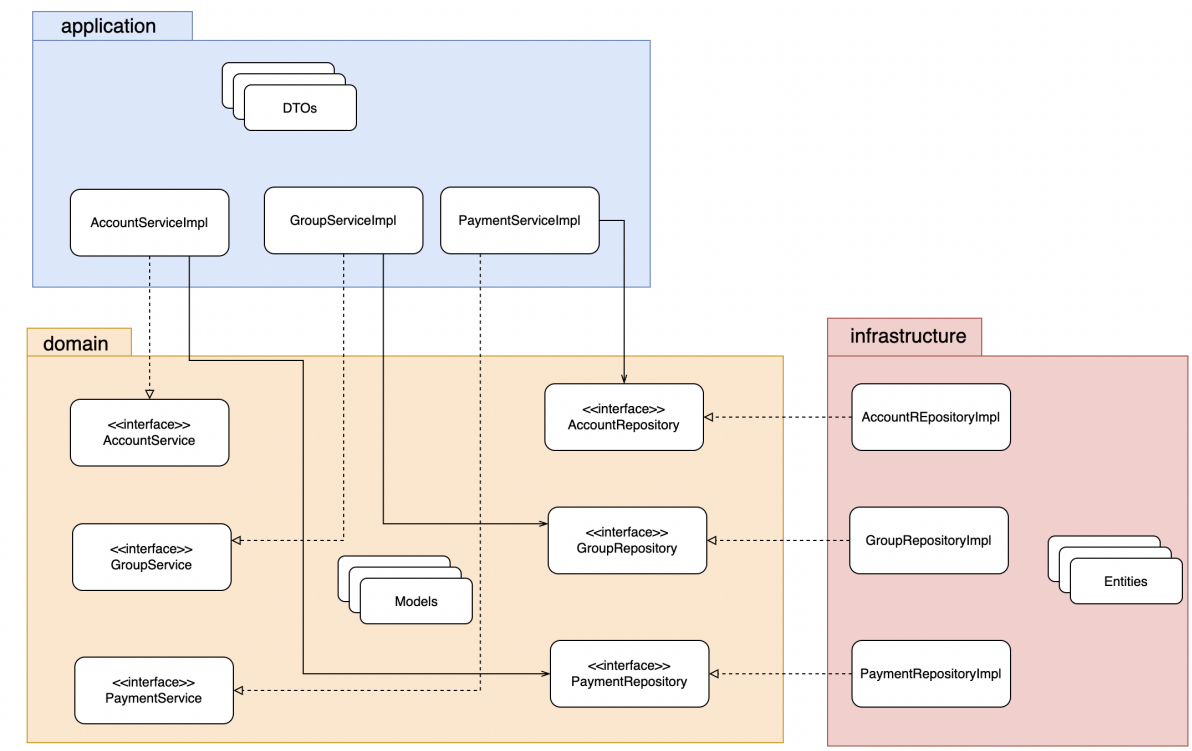
4. 그룹 집계 데이터 조회 API

```
### 그룹 집계 데이터 조회 Request
GET http://localhost:8080/statistics?groupId=3
Content-Type: application/json

### 그룹 집계 데이터 조회 Response
{
  "groupId": 1,
  "count": 2,
  "totalAmount": 605000.0,
  "avgAmount": 302500.0,
  "maxAmount": 305000.0,
  "minAmount": 300000.0
}
```

어플리케이션 구조

`application`, `domain`, `infrastructure`로 3계층 구성을 했습니다.





위 다이어그램에서 확인할 수 있듯, **저수준 모듈인 infrastructure, application 계층이 고수준 모듈인 domain 계층에 의존**하는 `의존성 역전 원칙(DIP)`를 적용했습니다.

application, infrastructure 계층은 domain 계층에 제공하는 **추상화(인터페이스)를 구현하거나 추상화에 의존**하게끔 하여 **고수준 모듈인 domain 계층은 application, infrastructure 변경으로부터 자유**롭게 구성했습니다.

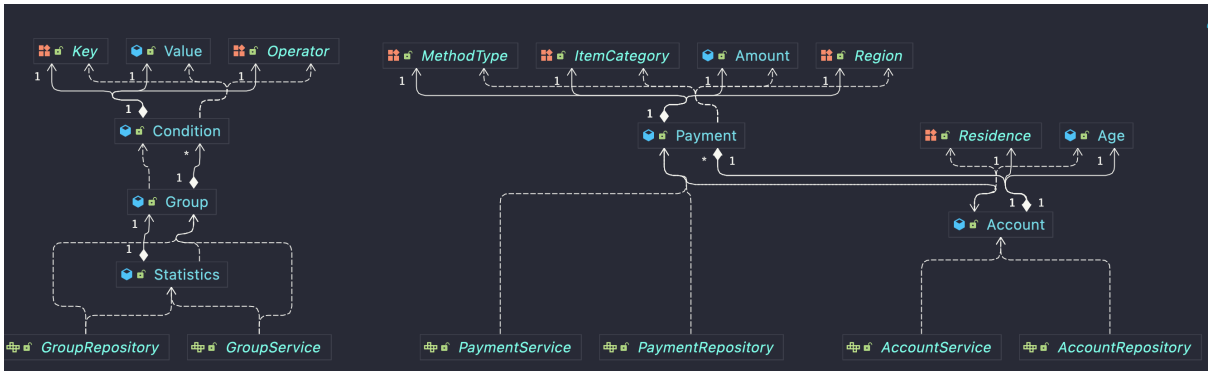
domain 영역

특정 프레임워크(Spring 등)나 데이터 접근 기술(JPA 등)에 의존적이지 않고 **핵심 도메인 규칙과 비즈니스 규칙을 담당**합니다.

`Account`, `Payment`, `Group` 도메인 모델로 구성되고 각 모델들은 정의된 도메인 규칙(나이는 0세 이상, 결제 수단은 카드와 송금, 지역은 서울, 대전 등)을 구현합니다.

또한 Service, Repository에 대한 추상화를 제공하여 외부 계층(application, infrastructure)에서 구현할 수 있게끔 합니다.

domain 계층의 컴포넌트 의존 그래프



주요 도메인은 `Group`, `Payment`, `Account`로 각 모델 내의 요소(Condition, Acount, Residence 등)들은 Value Object로 구성하여 각 객체들에게 값 검증과 같은 책임을 부여했습니다.

`MethodType`, `ItemCategory`들의 경우 **한글과 영문 모두 수용**해야 한다는 요구사항으로 받아들여 호환이 되도록 구성했습니다.(CARD ← → 카드), (패션 ← → FASHION)

또한 `Group`의 `Condition`을 구성하는 `Key`, `Value`, `Operator`는 **현재 지원하는 연산자, 피연산자**에 대한 정의를 해두었습니다.

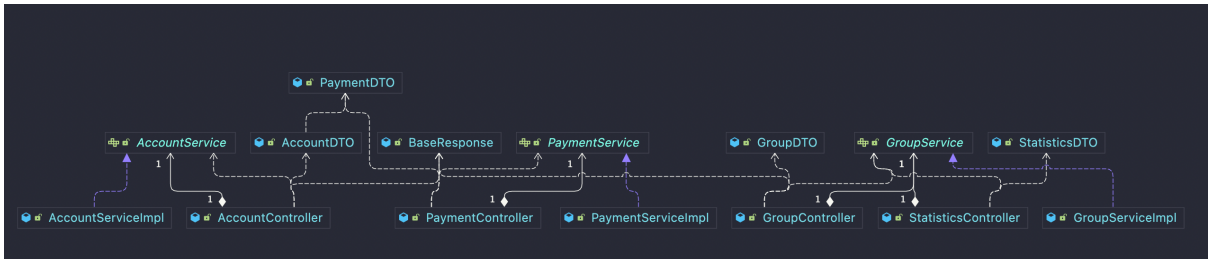
application 영역

application 영역에서는 표현 영역(사용자 요청을 받고 응답)하는 역할까지 함께 담당하도록 구성했습니다.

domain 영역에서 제공하는 Service 인터페이스를 구현하고 Repository 인터페이스에 의존성을 갖습니다.

외부로부터 요청(Http)을 받아 어플리케이션 로직을 실행하고 domain model과 dto 사이의 변환을 합니다.

application 계층의 주요 컴포넌트 의존 그래프



Service 구현체들은 트랜잭션의 시작과 끝을 담당하기 때문에 `@Transactional` 어노테이션이 선언되어 있습니다.

또한 각 계층에서 발생한 BusinessException을 비롯하여 다양한 Exception을 처리하기 위한 GlobalExceptionHandler를 두어서 예기치 못한 예외가 발생하더라도 일관된 응답 처리를 수행합니다.

Infrastructure 영역

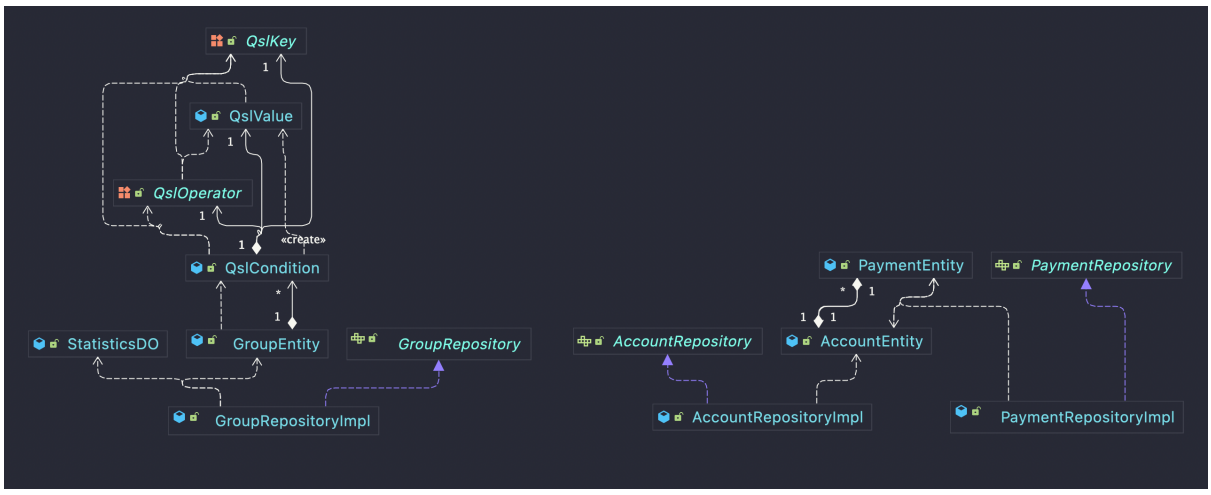
데이터 접근 기술에 대한 선택과 구현을 담당합니다.

domain 영역에서 제공하는 Repository 인터페이스를 구현합니다.

데이터 접근 기술은 **JPA + QueryDSL**을 사용했습니다.

domain 계층의 domain model들을 ORM(JPA) entity로 변환하고 영속화 하는 책임을 수행합니다.

infrastructure 계층의 주요 컴포넌트 의존 그래프

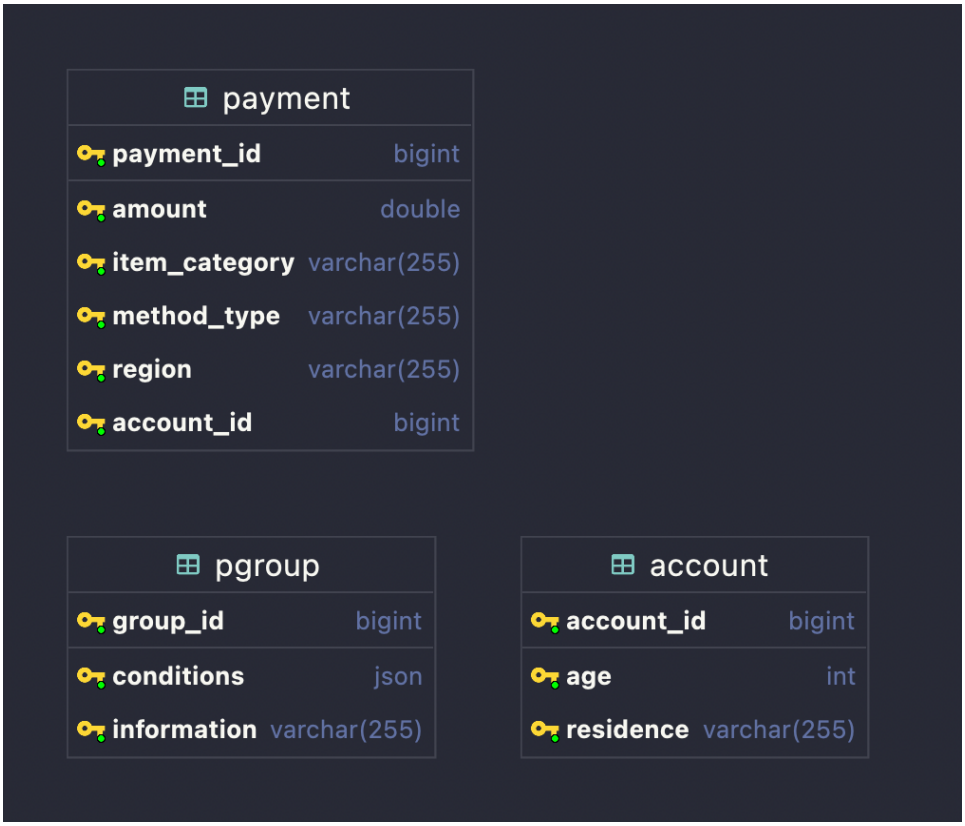


결제 그룹 데이터의 집계 책임을 수행하는 `GroupEntity` 구성 컴포넌트들은 `Qsl`이라는 prefix를 갖는데 집계를 위한 연산 결과 객체가 `QueryDSL` 표현식이기 때문에 domain 계층의 Group 모델 구성 컴포넌트들과의 구분을 위해 지정했습니다.

`GroupEntity`의 중요한 책임 중 하나가 **그룹 집계 조건에 대한 표현식을 만드는 것**으로 각 조건 `QslCondition`들이 QueryDSL의 `BooleanExpression`을 반환하고 `GroupEntity`는 이 `BooleanExpression`을 모아서(Array) 반환해줍니다.

그리고 QslCondition은 BooleanExpression을 반환하기 위해 `QslOperator`, `QslKey`, `QslValue` 객체들에 의존하고, 각 객체들은 자신에게 할당된 책임 (연산 처리, 피연산 처리, 피연산 값 적절한 타입으로 변환 등)을 수행함으로써 하나의 BooleanExpression을 반환합니다.

DB 테이블



회고

요구사항 분석

K-CSS의 요구사항을 구현하기 위해 도메인 규칙이 무엇인지 데이터 샘플과 request/response 샘플을 보고 분석을 시작했습니다.

Account, Payment와 같은 정적 모델들 보다 Group에 대한 요구사항을 이해하기 위해 노력했습니다.

몇 가지 예시를 보고 어플리케이션이 원하는 기능이 무엇인지 파악이 되었고, 유연한 결제 그룹 집계를 위한 구조에 대해서 고민하기 시작했습니다.

어플리케이션 구조 고민

처음에는 두 가지 생각이 들었습니다.

1. Infrastructure 영역에서는 Group 데이터를 영속화 하는 책임만 두고 **실제 집계는 Domain** 영역의 로직으로 두는 방법
2. Infrastructure 영역에서 실제 집계까지 하여 결과를 반환하는 방법.

첫 번째 방법의 장점은 **집계 알고리즘을 도메인 모델의 책임으로 할당**함으로써 특정 기술에 의존적이지 않고 구현이 용이하며 필요 시 좀 더 복잡한 통계 연산까지 유연하게 구현 가능하다는 장점이 있습니다.

단점으로는 집계 연산 수행을 위해 모든 결제 데이터를 메모리에 올려서 직접 구현하기에는 **리소스의 낭비가 상당**히 심하기 때문에 조금만 데이터 사이즈가 커져도 어플리케이션이 동작을 멈출 위험이 있습니다.
또한 데이터 접근 기술이 제공하는 성능상의 이점을 이용하지 못하기 때문에 **반응성 측면에도 문제**가 된다고 판단했습니다.

두 번째 방법은 집계 연산을 데이터 접근 기술에 맡김으로써 특정 라이브러리나 프레임워크에 의존적이고 저장소가 변경 되었을 때 영향을 크게 받는다는 단점이 있지만 성능상의 이점을 얻고, 리소스 낭비를 최소화 할 수 있습니다.

두 가지 방법의 트레이드 오프를 고려했을 때 **대용량 데이터 처리** 요구사항을 만족시키기 위해서 **Infrastructure 영역에서 해결**하는 방향으로 결정했습니다.

세부 구현 고민

`GroupEntity`가 갖는 책임을 명확하게 정의했습니다.

> Dynamic Query가 가능하게끔 조건(where)절을 생성한다.

그렇다면 동적으로 `where` 생성을 위해서 어떤 기술을 사용할지를 고민했고 `MyBatis`, `JPQL`, `QueryDSL` 로 선택지를 좁혔습니다.

각 기술들의 동적 쿼리 생성 방법에 대해 비교한 결과 `QueryDSL`로 선택을 했기 때문에 이제 `GroupEntity`가 갖는 책임은 `QueryDSL`의 동적 where 표현식인 `BooleanExpression[]` 반환이 되었습니다.

이러한 흐름으로 `GroupEntity`의 책임을 수행하기 위해 `QslKey`, `QslOperator`, `QslValue`에 각각 피연산, 연산, 피연산 값 검증 및 변환에 대한 책임을 할당하여 구현했습니다.