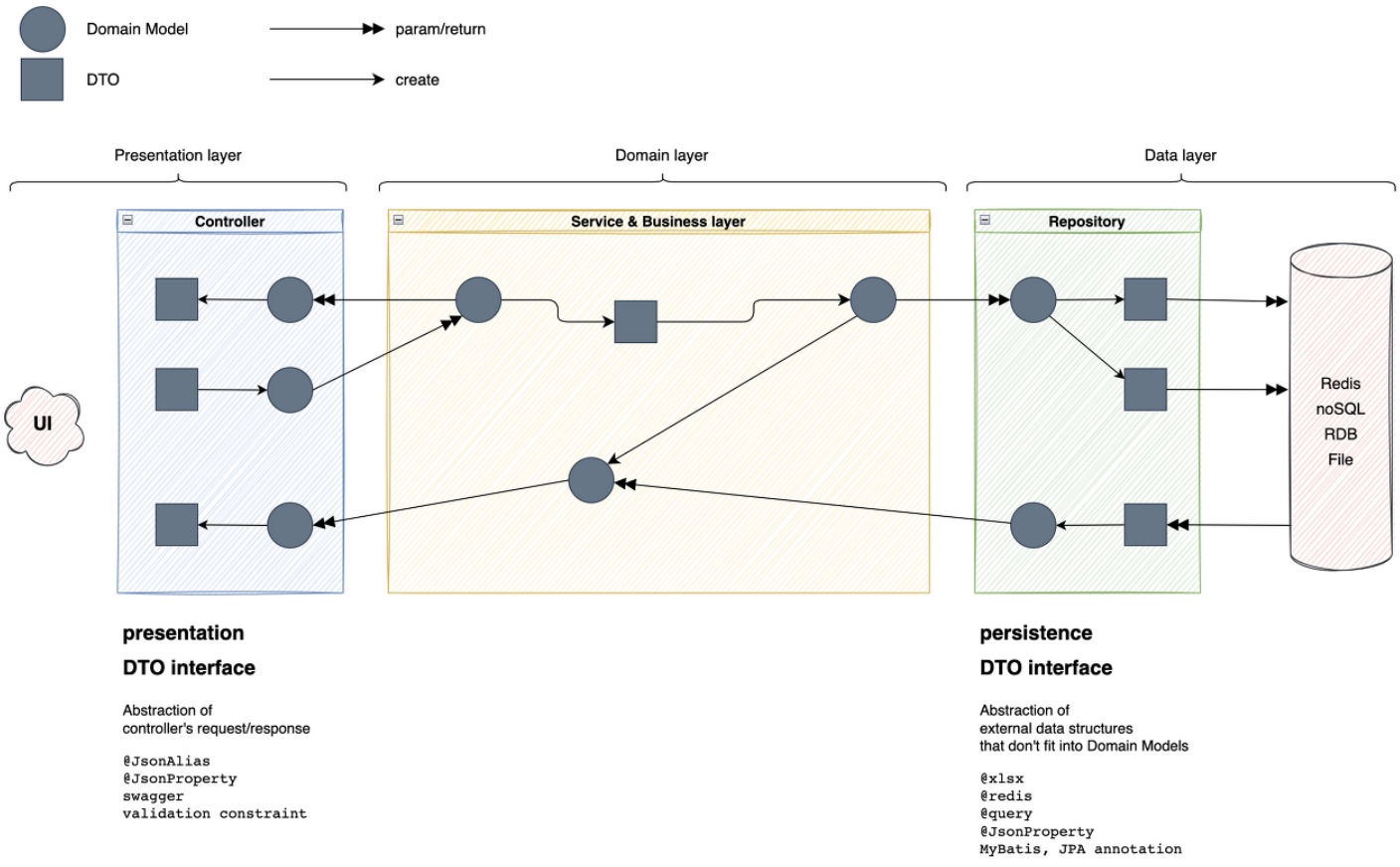
# [Spring] MVC Layering Architecture : Controller와 Service의 책임 나누기

2020.07.06 00:45 System Design & Arch

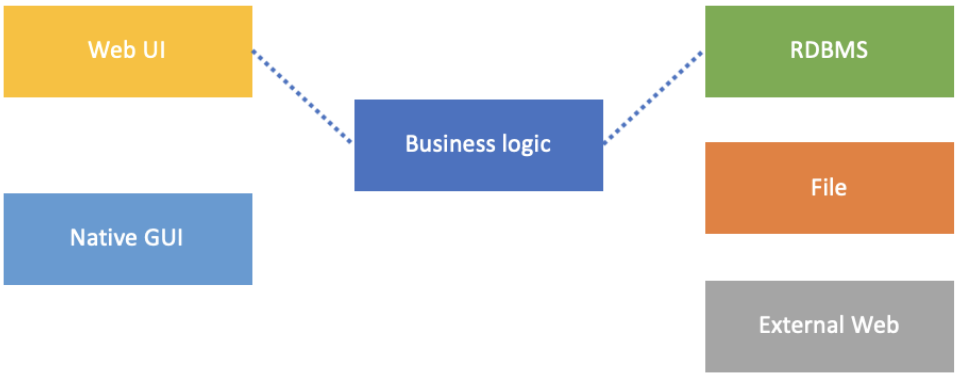
## 그림으로 정리한 Spring MVC Application Architecture




## 왜 layer가 필요한가?

### layer를 왜 분리할까? layer를 분리한다는 것에는 어떤 의미가 있는가?

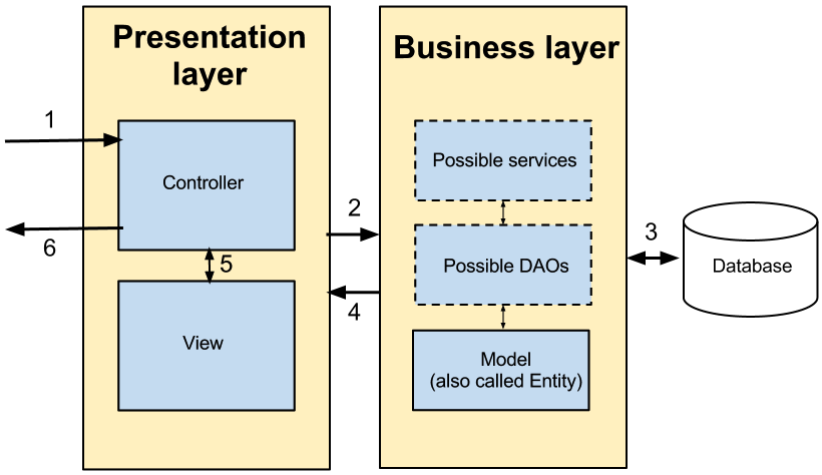
- layer를 나누게 되면, 다른 layer를 추상화 할 수 있다.
  - 추상화를 잘 했다면, **관심 분리** 를 통해 현재 작업하고 있는 layer에 집중할 수 있다.
  - 다른 layer의 모듈을 부품을 갈아끼우듯 변경할 수 있다. 각 layer가 자신의 세부사항을 몰라도 상관 없도록, 잘 추상화해서 제공하고 있었다면 가능하다.
- 컴포넌트 간의 의존 계층 관계를 깔끔하게 유지할 수 있다.
  - 각 layer를 넘나들면서 스파게티처럼 꼬여 있는 관계가 아니라, 위에서 아래로 떨어지는 간단한 구조 혹은 복잡한 참조는 같은 계층 내에서 끝내는 등 상대적으로 깔끔한 구조로 만들 수 있다.
  - 이는 곧 테스트 하기 쉬운 시스템으로 이어진다.
    - 예를 들어 data source에 mocking이 필요하다면 persistence layer를 mocking




## 이러한 layer들을 잘 분리하기 위한 개념으로, MVC 같은 디자인 패턴이 존재한다.

- MVC란?
  - https://martinfowler.com/eaaDev/uiArchs.html#ModelViewController
  - Model, View, Controller 세 가지 요소 간의 관계를 이용해 Presentation layer와 Business layer를 분리하는 패턴
- MVC 패턴은, *Presentation layer <> 나머지를 어떻게 잘 분리할 것이냐에 대한 패턴*이다. (MVVM, MVP도 마찬가지)
  - 즉, presentation 분리가 주 목적이므로, Business layer 뒤쪽이 어떻게 구성되는지와는 관련이 없다. (이름에서 유추할 수 있듯이)
  - 즉, Business layer가 Service sublayer를 별도로 두던, 어쩌든, MVC와는 별 상관이 없는 것이다.

> 그래서 MVC라는 이름만 가지고는 전체 아키텍쳐가 어떤 형태로 구성되어 있나를 전혀 유추 할 수가 없다.




MVC 예제

### 보편적으로 많이 사용하는 layer 구분

- Presentation(view) layer
- Service layer
- Business(Domain) layer
- Persistence(data) layer

> 필요에 따라 기타 sublayer를 포함 할 수도 있다. 일반적으로 이러한 구조를 많이 사용하기는 하지만, 절대로 따라야 하는 원칙이나 진리가 아니다. 상황에 맞게 적용하는 센스가 필요하다.

# 각 layer의 역할은 무엇인가? 추상화하고자 하는 것은?

## Presentation layer

- UI와 표현과 관련된 코드가 위치한 layer
- view가 jsp 인지, react 인지, app 인지, native GUI인지, CLI 인지는 여기서 결정한다. (다음 layer로 view에 따라 달라지는 specific한 내용을 전달하지 않는다.)
- UI 변경 될 때 같이 변경되어야만 하는 대상은 대체로 presentation layer에 속할 확률이 크다. (Martin Fowler 의견)
- Spring MVC Architecture에서는 View + Controller 로 정의할 수 있다.

**Controller는 Presentation layer인가? Service layer인가?**

https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)#Controller
여기서는 application/service layer에 속한다고 말하고 있으나, Presentation layer에 속한다고 보는게 더 적절할 것 같다.

1. https://martinfowler.com/eaaDev/SupervisingPresenter.html
   => Presentation은 Controller와 View 두 가지로 분리할 수 있다고 말하고 있다.
2. 때때로 View에서 수행하지 못하는 Presentation 로직을 처리하기 위해서라도 Controller는 Presentation layer에 있는 것이 맞다고 봄.
3. 애초에 MVC 자체가 Presentation code와 나머지 code를 어떻게 분리할것이냐에 대한 패턴이기도 함.
4. UI를 Native GUI로 변경하거나, 앱으로 변경한다고 가정했을 때 Controller가 아예 제거되거나, 변경이 발생해야 함.
5. 그리고 UI가 jsp라고 생각해보면, 어떤 jsp page에 결과를 전달할 것인지 선택하는 역할도 맡고 있다.

## Service layer

- https://martinfowler.com/eaaCatalog/serviceLayer.html
- **Domain Model을 묶어서 이 소프트웨어에서 사용 가능한 핵심 작업 집합을 설정하는 계층**
  - 이 소프트웨어가 수행해야 하는 작업은 무엇인가?
  - 이 소프트웨어에 내릴 수 있는 명령은 무엇인가?
- 보통 도메인 모델의 비즈니스 로직 하나를 호출하는 것 만으로는 복잡한 작업을 처리할 수 없음.
  - [도메인 모델 여러개를 불러와 요청을 가공하고, 비즈니스 로직을 호출하고, 응답을 조정해서 또 다른 비즈니스 로직을 호출] 이런 작업을 해주는 상위 layer가 있어야 함.
  - 이렇게 여러 비즈니스 로직들을 의미있는 수준으로 묶어 제공하는게 Service layer의 역할
- 게다가 FE, Gateway 등 다양한 엔드포인트로부터 작업 요청을 받는 상황을 가정해 보면.
  - 각 엔드포인트의 종류와 목적이 다르더라도, 공통적으로 사용하는 작업이 있다면 service layer에서 처리하는 것이 적절할 수 있음. (본질적으로 그 작업이 service에 들어가는게 맞느냐를 먼저 따져야 하지만, 이게 모호한 경우가 있다.)
- 결국 가장 핵심에 가까운 API를 제공하는 계층이 Service layer라고 볼 수 있음.
  - Controller가 그래보이겠지만, 이건 UI layer에 가깝다.
  - 요청이 UI를 통해 들어온거라면 Controller를 거치겠지만, 다른 경로로 들어왔다면? 내부 API 호출이라면? Controller를 거치지 않거나 다른 layer를 통해서 Service layer에 접근할 수도 있기 때문에...

- 단, 핵심 로직은 Business layer에 두고, Service layer는 얇게 유지하는 것이 맞다.

**presentation layer에 들어갈 대상과 service layer에 들어갈 대상을 구분하는 기준?**

- FE, Gateway 등 다양한 엔드포인트로부터 작업 요청을 받는 상황을 가정해 보자.
  - 현재 UI가 웹이라면, Native GUI / CLI / 앱 도 동시에 제공하게끔 변경했을 때 웹 Presentation 코드와 다른 UI Presentation 코드에 중복이 있는지 생각해본다.
  - 각 엔드포인트의 종류와 목적이 다르더라도, 공통적으로 사용하는 작업이 있다면 service layer에서 처리하는 것이 적절할 수 있음. (본질적으로 그 작업이 service에 들어가는게 맞느냐를 먼저 따져야 하지만, 이게 모호한 경우가 있다.)

- UI에 종속적인 로직인지, 아니면 그 소프트웨어의 핵심 api 로직인지 생각해본다.
  - 예를 들어 계좌 정보와 카드 정보를 동시에 표시하는 web UI를 제공하고 있었는데, 이를 app UI로 변경하는 상황을 생각해보니 바뀐 app UI에서도 여전히 계좌 정보와 카드 정보를 동시에 제공하겠다 싶은 생각이 든다면.
  - 이쯤 되면 계좌 정보와 카드 정보를 service layer에서 묶어서 반환해도 되기는 하지만... 이건 그냥 UI가 바뀌었는데 우연히 필요로 하는 정보가 그대로일 가능성이 큼.
  - 정말로 카드 정보와 계좌 정보를 묶어서 반환하는걸 모든 인터페이스에 핵심 api로 제공해야겠다면 service layer에서, UI specific하다면 Presentation layer에서 묶어서 반환하면 된다.

## Business layer (Domain Model)

- Domain Model에 대해서
- https://martinfowler.com/bliki/AnemicDomainModel.html
  - Domain Model 클래스에 field, getter, setter만 두고 단순히 DTO 처럼 사용하며 로직은 다 Service에 넣어버리는 것은 **안티패턴**이라는 의견.
  - 나도 그렇게 생각하는 것이, 객체라는건 data(state) 및 그와 연관된 logic을 가지고 있는 기본 단위다. 실세계를 사상하려면 state + logic이어야 한다.
  - anemic domain model 구조로 가면... 결국 Service Class에 로직이 다 들어가고, Model Class에는 필드밖에 없는데, Class의 "책임"이나 응집도를 생각해 보았을 때 Model Class에서 로직을 가지고 있는게 더 자연스럽지 않나? 싶은 경우를 꽤 자주 마주하게 된다.
  - **즉, 단순 요청/응답/전달을 위한 DTO와 비즈니스로직에서 사용하는 actor 그 자체인 Model 클래스는 다르게 바라보아야 한다.**
  - e.g., Model : User / DTO : UserRequest & UserResponse

## Persistence layer

- data source 추상화 : 어디에 저장하고 어디서 읽어오는지?
- data structure 추상화 : domain model 구조와 잘 들어맞지 않는 DB schema 추상화
- business layer에서 Domain Model을 받아 data source에 저장하고,
- data source에서 읽어와 Domain Model로 변환 후 business layer로 반환
- Repository와 DataMapper의 책임

## 참고 문서


- https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
- https://docs.oracle.com/cd/E19644-01/817-5448/dgdesign.html
- [마틴파울러] Presentation - Domain - Data Layering





**'System Design & Arch' 카테고리의 다른 글**

[리팩터링 2판] 3장 Bad Smells in Code →
CQRS : Command and Query Responsibility Segregation →
**[Spring] MVC Layering Architecture : Controller와 Service의 책임 나누기** ✓
[Spring] MVC : Model 클래스와 요청/응답 Data 클래스, Map을 사용하는 것? →
공통 비즈니스 로직 분리(제휴사 인터페이스 통합 및 클래스 설계) →
Exception 처리, 어떻게 하는게 좋을까? →


MVC

이름

암호