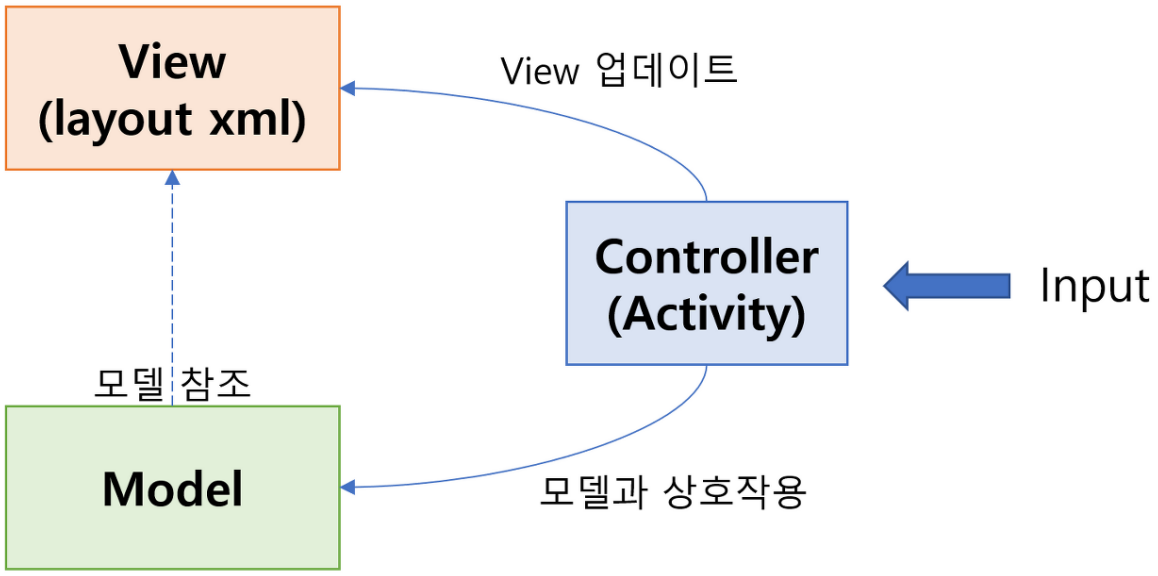
CHAPTER1. 디자인 패턴과 프로그래밍 패러다임

💡 각각의 구성 요소가 다른 요소들에게 영향을 미치지 않으려면?

1.1.8 MVC 패턴

MVC란 **M**odel **V**iew **C**ontroller의 약자로 에플리케이션을 세가지의 역할로 구분한 개발 방법론

사용자가 Controller를 조작하면 Controller는 Model을 통해서 데이터를 가져오고 그 정보를 바탕으로 시각적인 표현을 담당하는 View를 제어해서 사용자에게 전달



- 재사용성과 확장성 용이
- 애플리케이션이 복잡해질수록 모델, 뷰의 관계가 복잡해짐

모델(Model)

애플리케이션의 정보, 데이터를 나타낸다. 데이터베이스, 처음의 정의하는 상수, 초기화 값, 변수 등을 뜻한다. 이러한 데이터, 정보들의 가공을 책임지는 컴포넌트이다.

모델의 규칙

1. 사용자가 편집하길 원하는 모든 데이터를 가지고 있어야 한다.
    - 화면 안의 네모박스에 글자가 표현된다면, 네모박스의 화면 위치 정보, 네모박스의 크기 정보, 글자 내용, 글자의 위치 등
2. 뷰나 컨트롤러에 대해서 어떤 정보도 알지 말아야 한다.
    - 데이터 변경이 일어났을 때 모델에서 화면 UI를 직접 조정해서 수정할 수 있도록 뷰를 참조하는 내부 속성 값을 가지면 안된다는 말이다.
3. 변경이 일어나면, 변경 통지에 대한 처리 방법을 구현해야 한다.
    - 모델의 속성 중 텍스트 정보가 변경이 된다면, 이벤트를 발생시켜 누군가에게 전달해야 하며, 누군가 모델을 변경하도록 요청하는 이벤트를 보냈을 때 이를 수신할 수 있는 처리 방법을 구현해야 한다.

뷰(View)

Input 텍스트, 체크박스 항목 등과 같은 사용자 UI요소를 나타낸다. 데이터 및 객체의 입력, 그리고 보여주는 출력을 담당한다. 데이터를 기반으로 사용자들이 볼 수 있는 화면.

뷰의 규칙

1. 모델이 가지고 있는 정보를 따로 저장해선 안된다.
    - 화면에 글자 등을 보여주기 위해 모델이 가지고 있는 정보를 전달 받는다. 그 정보를 유지하기 위해서 임의의 뷰 내부에 저장하면 안된다.
    - 단순히 네모 박스를 그리라는 명령을 받으면, 화면에 표시하기만 하고 그 화면을 그릴 때 필요한 정보들은 저장해선 안된다.
2. 모델이나 컨트롤러와 같이 다른 구성요소들을 몰라야 한다.
    - 모델과 같은 자기 자신을 빼고는 다른 요소는 참조하거나 어떻게 동작하는지 알아서는 안된다. 그냥 뷰는 데이터를 받으면 화면에 표현만 해준다.

3. 변경이 일어나면 변경통지에 대한 처리 방법을 구현해야만 한다.
    - 뷰에서는 화면에서 사용자가 화면에 표시된 내용을 변경하게 되면 이를 모델에게 전달해서 모델을 변경해야 할 것이다. 그 작업을 하기 위해 변경 통지를 구현한다.

컨트롤러(Controller)

controller는 model(데이터)과 view(UI)를 연결 시켜주는 다리 역할을 한다. 즉, 사용자가 데이터를 클릭하고 수정하는 것에 대한 이벤트들을 처리하는 부분을 뜻한다.

컨트롤러의 규칙

1. 모델이나 뷰에 대해 알고있어야 한다.
    - 모델이나 뷰는 서로의 존재를 모르고 변경을 외부에 알리고 수신하는 방법만 가지고 있다. 이를 컨트롤러가 중재하기 위해 모델과 그에 관련된 뷰에 대해 알고있어야 한다.
2. 모델이나 뷰의 변경을 모니터링 해야 한다.
    - 모델이나 뷰의 변경 통지를 받으면 이를 해석해서 각각의 구성 요소에게 통지를 해야한다.

MVC 패턴을 사용하는 이유

- 서로 분리되어 각자의 역할에 집중할 수 있게 된다.
- 유지보수성과 애플리케이션의 확장성, 유연성이 증가한다.
- 중복 코딩의 문제점이 사라진다.

- Spring, Angular.JS, Django 등

💡 MVC와 MTV(Django) 패턴의 차이는 뭘까?

MTV 패턴

모델(Model)

- MVC 패턴의 모델에 대응된며 DB에 저장되는 데이터를 의미한다. 모델은 클래스로 정의되며 하나의 **클래스가 하나의 DB Table이다.**
- 원래 DB를 조작하기 위해선 SQL을 다룰 줄 알아야 하지만 장고는 **ORM(Object Relational Mapping)기능을 지원하기 때문에 파이썬 코드로 DB를 조작할 수 있다**.

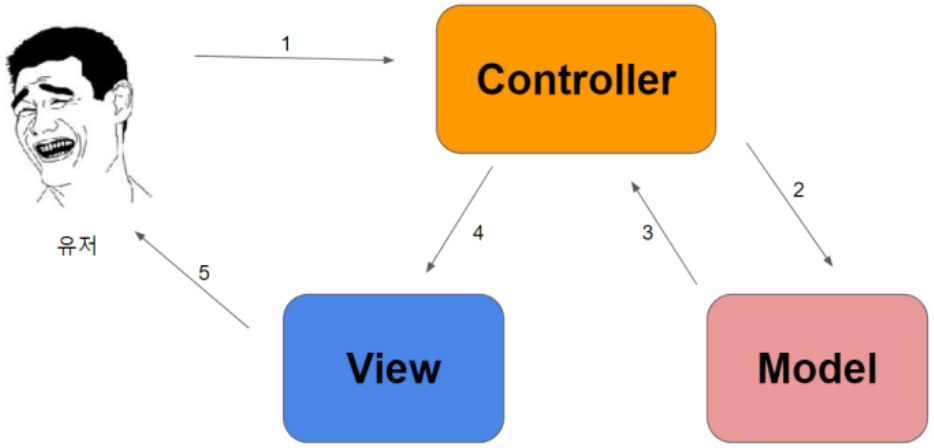
템플릿(Template)

- MVC 패턴의 뷰에 대응되며 유저에게 보여지는 화면을 의미한다.
- 장고는 뷰에서 로직을 처리한 후 html 파일을 context와 함께 렌더링하는데 이 때의 html 파일을 템플릿이라 칭한다.
- 장고는 자체적인 **Django Template 문법을 지원하며 이 문법 덕분에 html 파일 내에서 context로 받은 데이터를 활용할 수 있다.**

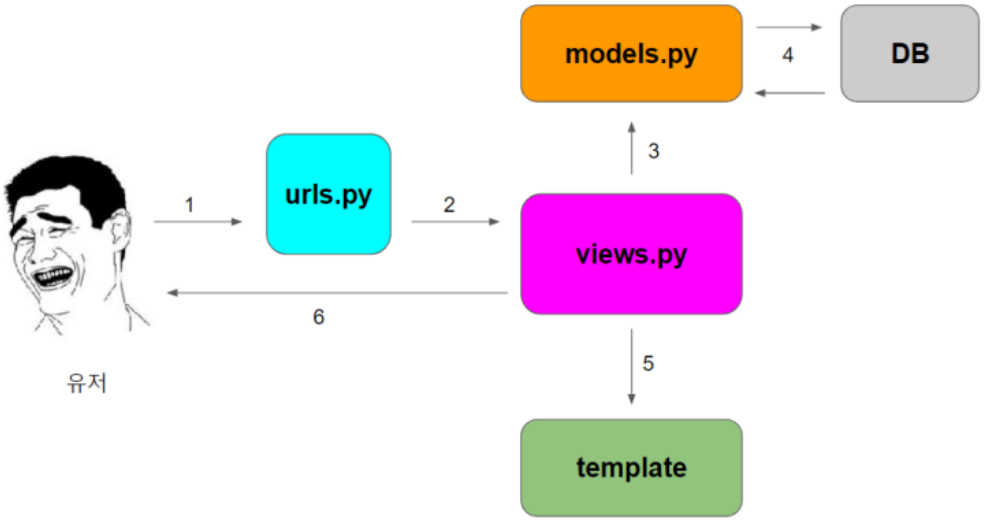
뷰(View)

- MVC 패턴의 컨트롤러에 대응되며 요청에 따라 **적절한 로직을 수행하여 결과를 템플릿으로 렌더링하며 응답한다.**
- 다만 항상 템플릿을 렌더링 하는 것은 아니고 백엔드에서 데이터만 주고 받는 경우도 있다.

MVC, MTV 비교



1. 유저가 컨트롤러에 요청을 보낸다.(예시 : 뷰에 있는 Submit 버튼을 누른다)
2. 컨트롤러가 모델에 요청사항대로 데이터를 수정할 것을 지시한다.
3. 모델은 지시받은대로 데이터를 수정하고 컨트롤러에게 완료되었음을 알려준다.
4. 컨트롤러는 수정된 데이터를 토대로 표시할 뷰를 결정하고 화면을 출력할 것을 지시한다.
5. 뷰는 지시받은대로 화면을 출력한다..



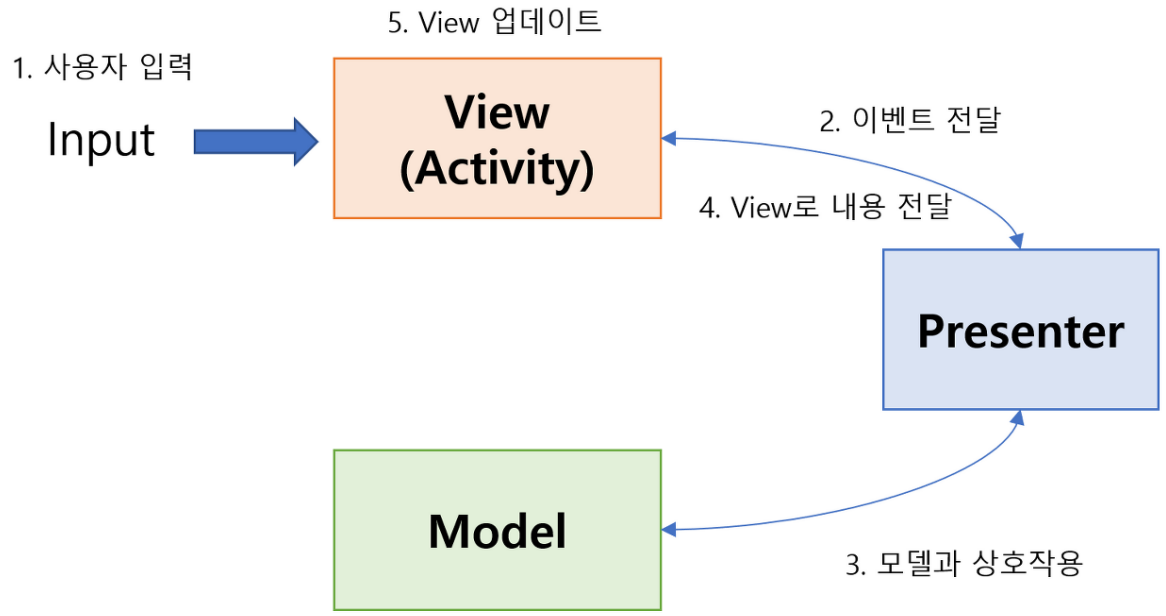
1. 유저가 특정 url로 요청을 보낸다.

2. urlConf를 통해 해당 url과 매핑된 뷰를 호출한다..
3. 호출된 뷰는 요청에 따라 적절한 로직을 수행하며 그 과정에서 모델에게 CRUD를 지시한다.
4. 모델은 ORM을 통해 DB와 소통하며 CRUD를 수행합니다.
5. 그 후 뷰는 지정된 템플릿을 렌더링하고
6. 최종 결과를 응답으로 반환한다.

1.1.9 MVP 패턴

Model, View, Presenter로 구성된 디자인 패턴.

MVP의 핵심 설계는 MVC와 다르게 **UI(View)와 로직(Model)을 분리**하고, 서로 간에 상호작용을 다른 객체(Presenter)에 그 역할을 줌으로써, 서로의 영향(의존성)을 최소화.



모델(Model)

- 프로그램 내부적으로 쓰이는 데이터를 저장하고, 처리하는 역할(비즈니스 로직)
- View 또는 Presenter 등 다른 어떤 요소에도 의존적이지 않은 독립적인 영역

뷰(View)

- UI를 담당하며 안드로이드에서는 Activity, Fragment가 대표적인 예
- Model에서 처리된 데이터를 Presenter를 통해 전달받아서 유저에게 보여줌
- 유저의 행동(Action) 및 Activity 생명 주기 상태 변경을 주시하며 Presenter에 전달하는 역할
- Presenter를 이용하여 데이터를 주고받기 때문에 Presenter에 매우 의존적임

프레젠터(Presenter)

- Model과 View사이의 매개체.
- Model과 View를 매개체라는 점에서 Controller와 유사하지만, View에 직접 연결되는 대신 인터페이스를 통해 상호작용한다는 차이가 있음
- 인터페이스를 통해 상호작용하므로 MVC가 가진 테스트 문제와 함께 모듈화/유연성 문제 역시 해결할 수 있음
- View에게 표시할 내용(Data)만 전달하며 어떻게 보여줄 지는 View가 담당

MVP의 장점

- MVC와는 다르게 코드가 깔끔해진다.
- Model과 View의 결합도를 낮추면, 새로운 기능 추가 및 변경을 할때 마다 관련된 부분만 코드를 수정하면 되기 때문에 확장성이 개선된다.

MVP의 단점

- 어플리케이션이 복잡해질수록 **View와 Presenter 사이의 의존성**이 강해지는 문제
- MVC의 Controller처럼 추가 비즈니스 로직에 집중되는 경향

MVC, MVP 디자인 패턴의 차이

1. 역할 분리
    - MVC
      모델은 애플리케이션의 데이터와 비즈니스 로직을 처리
      뷰는 사용자 인터페이스를 표시
      컨트롤러는 사용자 입력을 받고 모델 및 뷰 간의 상호 작용을 조정
    - MVP
      모델은 여전히 데이터와 비즈니스 로직을 처리
      뷰는 사용자 인터페이스를 표시
      프리젠터는 사용자 입력을 받고 모델 및 뷰를 직접 조정
2. 의존성 관리
    - MVC
      뷰와 모델 간의 양방향 의존성이 존재할 수 있다.
      즉, 뷰는 모델의 변경 사항을 감지하여 업데이트해야 하고, 모델은 뷰에 의존하여 변경 사항을 통지해야 한다.
    - MVP
      모델은 뷰나 프리젠터와 독립적으로 존재하며, 뷰와 프리젠터는 서로 알지만 모델과 직접적으로 소통하지 않는다.
3. 테스트 용이성
    - MVC
      뷰와 모델 간의 결합도가 높기 때문에 유닛 테스트 작성이 어려울 수 있다.
    - MVP
      뷰와 프리젠터 간의 인터페이스를 통해 뷰를 모의(Mock)하거나 가짜(Fake) 뷰를 사용하여 테스트 작성이 쉽다.
4. 정리
    - MVC
      뷰와 모델 간의 직접적인 의존성을 허용하고 컨트롤러가 중간자 역할
    - MVP
      뷰와 모델 간의 강력한 분리를 통해 테스트 용이성을 높이고 프리젠터가 중간자 역할