# PBL기반의 장고(Django) 프로그래밍 기본

## Django Framework

강사 : 박종일

# Django

Django는...

> 프랑스의 유명한 작곡가 Django Reinhardt의 이름을 따서 명명된, 모델-뷰-컨트롤러(MVC)의 아키텍처 패턴을 따르는 Python 기반 웹 프레임워크.
>
> 쉽고 빠르게 웹사이트를 개발할 수 있도록 돕는 구성요소로 이루어져 있으며 강력한 라이브러리들을 그대로 사용할 수 있다는 큰 장점을 가지고 있다.

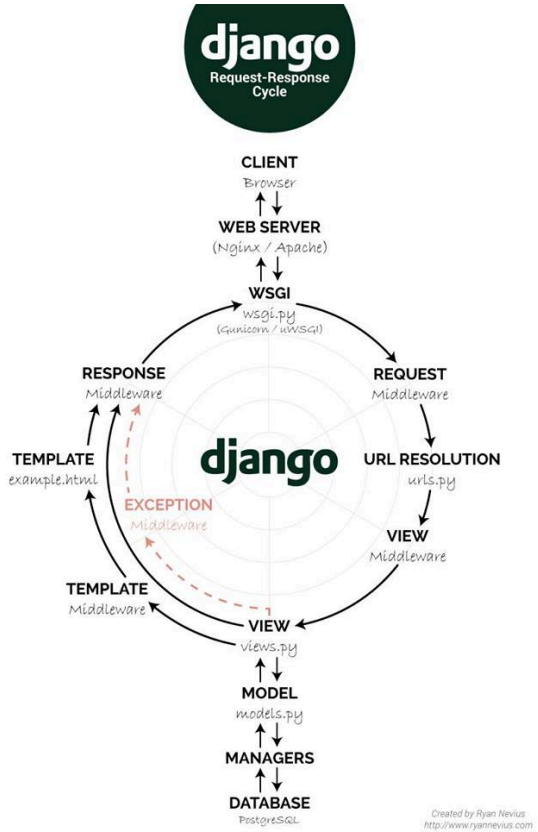
# Django 사용의 장점

- Django는 확장성이 뛰어난 것으로 알려져 있다. 특히 코드 재사용성 기능을 통해 개발자는 웹사이트에서 증가하는 트래픽에 쉽게 적응할 수 있다.
- Django를 기반으로 하는 웹사이트는 최적화가 쉽고 SEO 친화적. IP 주소가 아닌 URL을 통해 서버에서 Django 기반 애플리케이션의 유지가 가능.
- 코드 없음 지향: 코드 없는 프레임워크가 아니지만 개발자가 코드를 사용하지 않고 활용할 수 있는 패키지가 있다.
- 다용성: 데이터베이스 기반 웹사이트에 적합하며, 모든 유형의 웹 사이트를 만드는 데 사용할 수 있다.

# Django Cycle

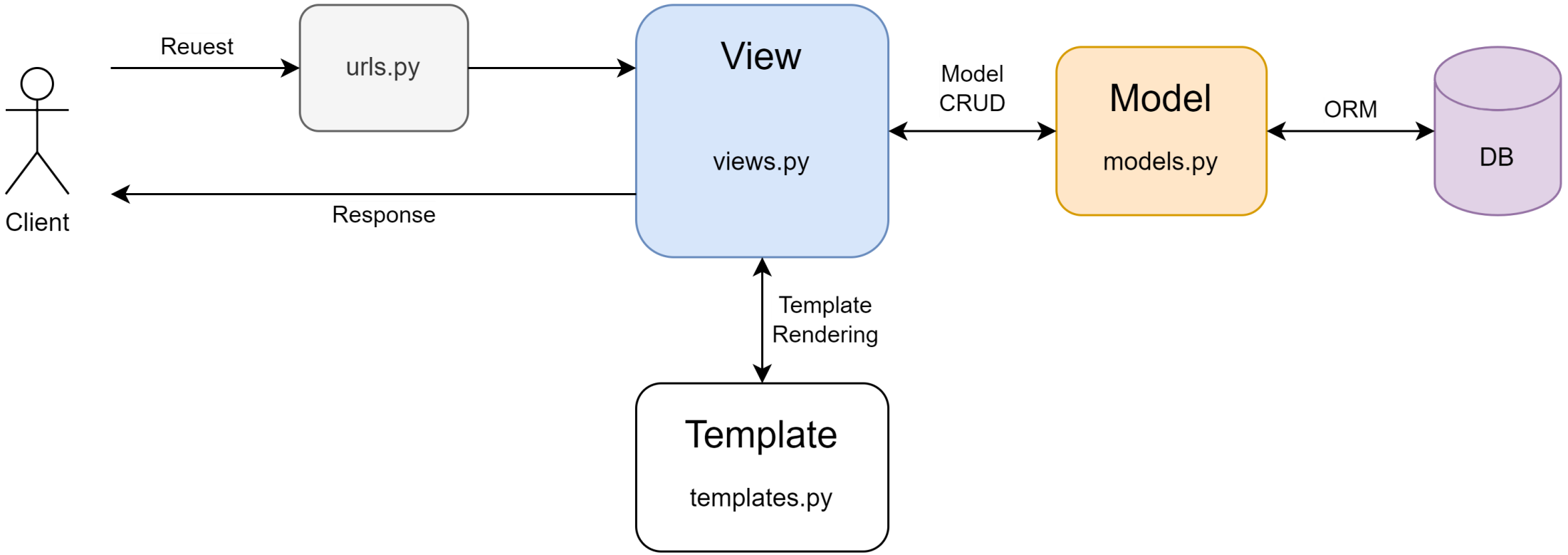
## WSGI(Web Server Gateway Interface)

- 웹 서버 소프트웨어와 파이썬으로 작성된 웹 응용 프로그램 간의 표준 인터페이스.
- 웹 서버를 통해 넘어오는 client의 request를 WSGI Server로 처리.
- ASGI(Asynchronous Server Gateway Interface)
  - 비동기 웹 서버
  - 프레임워크 및 애플리케이션 간의 표준 인터페이스를 제공하기 위한 WSGI의 정신적 후속 버전.



# MTV 패턴

MTV(Model, Template, View) 패턴은 웹 개발에서 활용하는 MVC 패턴에 대응되는 Django의 디자인 패턴이다.



## Model

> DB를 처리하는 부분.
> 클래스로 정의하며, 하나의 클래스가 하나의 DB table에 대응된다.
> ORM(Object Relational Mapping)을 지원하여 파이썬 코드로 DB의 CRUD를 처리.
> SQL 쿼리 및 파일 저장에 대한 처리도 가능.

## Template

> 브라우저에 출력될 화면을 처리하는 부분.
> View에서 처리된 로직의 결과인 context와 html 코드를 렌더링하여 Client에 전달.
> Django의 자체적인 Template 문법을 활용하여 html 내에서 context로 전달된 데이터를 출력.

# View

> URL Mapping 및 서비스 로직을 처리하는 부분.
> FBV(Function-Based View)와 CBV(Class-Based View)

## FBV

함수 기반 뷰 작성 방식으로 심플하고 가독성이 좋음

```
def some_url(request):
  if request.method == 'POST':
    # post 전송에 대한 처리 코드
  else:
    # get 전송에 대한 처리 코드
```

코드를 확장하거나 재사용하기 어려우며, 조건문으로 http method를 구분해야 한다.

## CBV

Class 기반 뷰 작성 방식으로 상속과 믹스인(Mixin) 기능을 이용할 수 있으며, 코드를 재사용하고 뷰를 체계적으로 구성할 수 있음.

```
class SomeUrlClass(View):
  def post(self, request):
    # post 전송에 대한 처리 코드
  def get(self, request):
    # get 전송에 대한 처리 코드
```

http method에 대한 처리를 조건문 대신 메소드 명으로 처리하여 코드 구조가 깔금하고 객체지향 기법을 활용할 수 있지만, FBV에 비하여 코드가 복잡하고 가독성이 떨어짐

# Django 개발환경 설정

설치할 것들

1. vscode
2. python

> vscode이 콘솔에서 python을 입력.
> Microsoft Store를 통해 설치하면 vscode에서 바로 사용 가능

3. vscode 확장(Extension)

- Python Extension
- Django Extension

# Django 프로젝트 생성 단계

## 1. Django 가상환경 생성

가상환경(Virtual Environments)이란 독립적인 파이썬의 실행 환경을 의미.

Django를 비롯하여 다양한 파이썬 프로젝트들에서 사용할 여러 패키지를 따로 관리하기 위해 분야별로 가상환경을 만들어서 따로 관리함.(프로젝트별로 다른 패키지를 사용할 수 있기 때문에..)

- 터미널 명령어

```
py -m venv '가상환경이름'
```

'가상환경이름'으로 폴더가 생성됨

## 2. 프로젝트 생성

가상환경을 실행한 상태에서 Django를 설치하고 그 후에 프로젝트를 생성.

### 1) 가상환경 실행

```
'가상환경이름'/Scripts/activate
```

### 2) Django 설치

```
pip install django
```

가상환경에 django가 설치됨

- 설치 확인

```
pip freeze
```

## 3) 프로젝트 생성

### (1) 프로젝트 생성

```
django-admin startproject project_name
```

> 프로젝트를 생성하면 'project_name' 폴더가 생성되고 그 하위에 'project_name' 폴더와 동일한 이름의 폴더가 하나 더 생성되어 설정관련 파이썬 파일들이 생성된다.

## 4) 앱 생성 및 등록

하나의 프로젝트에는 하나 이상의 앱이 작성될 수 있음.

앱(app, application)이란 웹을 구성하는 한 부분이라고 생각할 수 있음.

즉, 한 쇼핑몰 사이트(프로젝트)를 사용자 부분, 판매자 부분, 관리자 부분으로 구분하여 개발할 경우, 각 부분들을 개별 앱으로 작성한다.

## (1) 앱 생성

> myproject로 경로 이동 후

```
cd myproject
py manage.py startapp app_name
```

## (2) setting.py에 언어와 시간을 한국값으로 변경

(project_name/project_name/setting.py)

```
LANGUAGE_CODE = 'ko-kr'
TIME_ZONE = 'Asia/Seoul'
```

## (3) 앱 생성 후 프로젝트 구조

```
project_name/
├── manage.py
├── project_name/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── app_name/
    ├── migrations/
    │   └── __init__.py
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── models.py
    ├── tests.py
    └── views.py
```

## 5) 앱 실행

```
py manage.py runserver
```

> 실행 url : 127.0.0.1:8000
> 실행 중지 : ctrl + c

## Port 번호 변경

> Django는 기본 포트번호가 8000으로 지정되어 있음.
> 변경을 위해 따로 설정하는 파일은 없으며 실행 시 port 번호를 지정

```
py manage.py runserver 80
```

## 6) 가상환경 중지

```
deactivate
```

## 참고> 가상환경 중지 후 프로젝트를 재시작할 경우

> 경로를 잘 확인할 것.(프로젝트 폴더로 이동했을 경우 activate 안됨)
> 그 경우 ../'가상환경이름'/Scripts/activate 으로 가상환경을 실행할 것

# view와 url

> 프로젝트 이름 - myproject
>
> 앱 이름 - myapp

- views.py - 요청 url에 따라 해당 제공될 기능을 위한 서비스 로직을 작성
- urls.py - url을 등록하여 views에 작성된 서비스 로직용 함수(또는 클래스)와 해당 url을 매핑
  - URLconf(URL Configuration) - Django에서 URL과 일치하는 뷰를 찾기 위한 패턴들의 집합
  - 사용자의 request를 url로 분류하여 해당 view에 전달

# myapp의 views.py

> 사용자의 브라우저 화면에 출력할 내용을 처리하기 위한 뷰를 FBV 또는 CBV 방식으로 작성

함수 작성 형태 - index 페이지

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world.")
```

문법

```
def page_name(request):
    # 기능 처리 코드
    return HttpResponse(data, content_type)
```

## HttpResponse()

> 사용자의 request에 대한 처리 결과를 사용자 측에 전달(응답)하기 위한 API

- Django는 Request에 따라 메타데이터를 포함하는 HttpRequset객체를 생성한다.
- urls.py에서 정의한 View 함수에 첫 번째 인자로 해당 HttpRequest객체를 전달.
- 해당 View는 결과값을 HttpResponse나 JsonResponse 객체에 담아 전달.

## data

> 브라우저에 출력될 내용으로 직접 문자열로 html 태그를 작성하거나 처리된 내용을 담은 template을 지정

## content-type

> 전송되는 데이터의 MIME type을 지정
> 주로 파일 다운로드 시 활용하며 html 전송시는 거의 생략함

## url path 등록

> myproject/myproject/urls.py의 urlpatterns에 앱의 url path를 등록

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('myapp/', include('myapp.urls')),
    path('admin/', admin.site.urls), # 원래 있는 내용(admin 사이트 url)
]
```

기본적으로 view의 url들은 config용 (하위)myproject폴더의 urls.py에 등록하는데 include를 써서 고정적인 url을 쉽게 관리하거나, 앱별로 url을 관리 할 수 있다.

myapp에 'urls.py' 파일을 생성 후 다음을 작성

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name="index"),
]
```

> urlpatterns의 형식은 다음과 같다.

```
urlpatterns = [
    path("some_url1", views.page_name, name="page_name"),
    path("some_url2", views.page_name, name="page_name"),
    ...
]
```

## path()

```
path(route, view, kwargs=None, name=None)
```

### route

route는 URL 패턴을 가진 문자열.

> 요청이 처리될 때, 일치하는 패턴을 찾을 때 까지 요청된 URL을 각 패턴과 리스트의 순서대로 비교.

### view

views.py에 작성된 view 함수(또는 클래스)를 지정.

> Django에서 일치하는 패턴을 찾으면, HttpRequest 객체를 첫번째 인수로 하고, url에 포함된 전송 데이터(파라미터)를 추출하여 키워드 인수로 대상 view에 전달.

## kwargs

임의의 키워드 인수들은 대상 view에 딕셔너리로 전달.

urls.py

```
    ...
    path('', views.index, kwargs={'test':'test_data'}, name='index'),
```

views.py

```
def index(request, test):
    print(test)
    ...

# 출력 : test_data
```

### name

URL에 이름을 지으면, 템플릿을 포함한 Django 어디에서나 명확하게 참조할 수 있다.

```
<a href="some_url">이동</a>
```

이 경우 path 메소드의 'route'를 사용하여 페이지를 이동한다.

```
path('some_url', views.some, name='some'),
```

url 태그를 사용하게 되면 name의 값을 사용한다.

```
<a href="{% url 'some' %}">이동</a>
```

페이지의 url이 바뀌더라도 url 태그를 사용한 코드를 수정하지 않고 그대로 사용할 수 있다.(url 태그를 사용하는 이유이기도 함)

redirect() 메소드에서도 name 값을 활용한다.

## 앱 시작 url에 Root 경로 사용

http://127.0.0.1:8000 으로 접속하면 'Page not found' 페이지가 출력된다.

현재 앱에 접속하기 위한 주소는 http://127.0.0.1:8000/myapp 이 되어 있기 때문이다.

Root 경로(http://127.0.0.1:8000) 를 앱의 시작 경로로 사용하기 위해서는 myproject/myproject/urls.py의 urlpatterns에 앱의 url path를 다음과 같이 수정한다.

> path()의 route를 빈문자열로 작성

```
urlpatterns = [
    path('', include('myapp.urls')),
    path('admin/', admin.site.urls),
]
```

# Django Model

Database 테이블을 정의하기 위한 추상화된 클래스

Database를 처리하는 부분으로 myapp 폴더의 models.py 파일에 테이블을 위한 클래스를 작성하면 지정한 DB에 테이블을 작성.

> 프로젝트 이름 - myproject
> 앱 이름 - myapp

## myapp의 models.py

> 설계한 테이블을 클래스로 작성

```
from django.db import models
from django.urls import reverse
```

```
class Data_tbl(models.Model):
    # Fields
    col_name = models.Field(option)
    ...
    # Meta
    class Meta:
        verbose_name = '테이블 별칭'
        verbose_name_plural = '테이블 별칭 복수형'
        ordering = ['col_name',]

    # Methods
    def get_absolute_url(self):
        return reverse('myapp:myapp_datail', kwargs={'pk':self.id})

    def __str__(self):
        return self.col_name
    ...
```

## Model class

테이블의 컬럼을 구성하는 field, 모델에 대한 메타데이터 정의를 위한 meta, 다양한 메소드를 정의

### Fields

테이블 컬럼의 정보를 정의하기 위한 클래스 변수이며, Form 작성 시 input 태그와 연계되는 부분.

## Field Type

주요 필드 타입 클래스

| Field Type | 설명 |
| --- | --- |
| CharField | 제한된 문자열 필드 타입.<br>최대 길이를 max_length 옵션에 지정해야 한다. |
| TextField | 대용량 문자열을 갖는 필드 |
| IntegerField | 32 비트 정수형 필드 |
| BooleanField | true/false 필드 |
| DateTimeField | 날짜와 시간을 갖는 필드 |

## Field Type

| Field Type | 설명 |
| --- | --- |
| DecimalField | 소숫점을 갖는 decimal 필드 |
| BinaryField | 바이너리 데이타를 저장하는 필드 |
| FileField | 파일 업로드 필드 |
| ImageField | FileField의 파생클래스로서 이미지 파일인지 체크한다. |
| UUIDField | GUID (UUID)를 저장하는 필드 |

- CharField의 파생클래스로서, 이메일 주소를 체크를 하는 EmailField, IP 주소를 체크를 하는 GenericIPAddressField, 콤마로 정수를 분리한 CommaSeparatedIntegerField, 특정 폴더의 파일 패스를 표현하는 FilePathField, URL을 표현하는 URLField 등이 있다.
- 정수 사이즈에 따라 BigIntegerField, SmallIntegerField 을 사용할 수도 있다.
- Null 을 허용하기 위해서는 NullBooleanField를 사용한다.
- 날짜만 가질 경우는 DateField, 시간만 가질 경우는 TimeField를 사용한다.

## Field Option

필드 타입에 따른 여러 선택사항을 지정

| Field Option | 설명 |
| --- | --- |
| null (Field.null) | null=True 이면, Empty 값을 DB에 NULL로 저장한다. DB에서 Null이 허용된다.<br>예: models.IntegerField(null=True) |
| blank (Field.blank) | blank=False 이면, 필드가 Required 필드이다. blank=True 이면, Optional 필드이다.<br>예: models.DateTimeField(blank=True) |
| primary_key (Field.primary_key) | 해당 필드가 Primary Key임을 표시한다.<br>예: models.CharField(max_length=10, primary_key=True) |

## Field Option

| Field Option | 설명 |
| --- | --- |
| unique (Field.unique) | 해당 필드가 테이블에서 Unique함을 표시한다. 해당 컬럼에 대해 Unique Index를 생성한다.<br>예: models.IntegerField(unique=True) |
| default (Field.default) | 필드의 디폴트값을 지정한다.<br>예: models.CharField(max_length=2, default="WA") |
| db_column (Field.db_column) | 컬럼명은 디폴트로 필드명을 사용하는데, 만약 다르게 쓸 경우 지정한다. |

## Field Option

| Field Option | 설명 |
| --- | --- |
| auto_now (Field.auto_now) | True로 설정하면 model이 저장(수정)될 때마다 현재 날짜로 갱신한다.<br>예: models.DateTimeField(auto_now=True) |
| auto_now_add (Field.auto_now_add) | True로 설정하면 model이 처음 저장될 때 현재 날짜를 저장한다.<br>예: models.DateTimeField(auto_now_add=True) |

## verbose_name

> 외래키(관계) 관련 필드를 제외한, 각 필드 타입은 선택적으로 첫 번째 인자의 위치에 작성하거나 verbose_name='문자열'로 작성.
> 외래키 관련 필드는 첫 번째 인자로 관계된 테이블의 클래스가 위치함

```
col_name = model.CharField(verbose_name='컬럼명', max_length=200)
# 또는
col_name = model.CharField('컬럼명', max_length=200) # verbose_name은 생략 가능
# 또는
col_name = model.CharField(max_length=200, verbose_name='컬럼명')
```

> verbose_name은 admin 페이지나 Form(또는 ModelForm)을 활용한 입력 양식에서 출력할 컬럼의 별칭.
> 생략할 경우 필드의 이름(col_name)이 자동으로 사용된다.

## id field

기본적으로 Django는 각 모델에 자동으로 id 필드를 추가해준다.

- id 필드는 기본키(primary key) 컬럼으로 작성됨
- id 필드는 자동증가(Auto Increment) 컬럼으로 작성됨
- 다른 필드에 primary_key 옵션이 명시된 경우 자동으로 생성되지 않음

## Meta

Model 클래스의 내부 클래스로 모델의 취급 방법을 변경할 수 있음.

- db_table : 테이블의 이름을 지정
- verbose_name : admin 사이트에서 출력하는 테이블의 별칭을 지정
- verbose_name_plural : 테이블 별칭의 복수형(한글 별칭에 대해서는 적용안됨)
- ordering : 테이블의 정렬 기준 컬럼과 방향을 지정
    - 작성한 필드명으로 정렬을 수행하며, 필드명 앞에 '-'를 붙이면 내림차순으로 정렬한다.

## Methods

Django의 Model 클래스에 정의된 기본 메소드(save, delete 등)를 재정의하거나 admin 페이지에서의 인스턴스 출력을 위한 메소드를 작성

### __str__() 메소드

admin 페이지에서 저장된 데이터 인스턴스의 출력 내용을 지정한다.

> __str__() 메소드를 작성하지 않은 경우 'Model_class_name object (index)'로 출력
> __str__()' 메소드에서 특정 필드를 지정하면 해당 컬럼에 저장된 Data 값이 출력됨

admin 페이지에서의 처리에만 해당되므로 반드시 작성할 필요는 없음.

## get_absolute_url() 메소드

템플릿이나 redirect() 에서 해당 모델의 인스턴스를 사용할 때 코드를 간략화할 수 있으며 모델 접근에 대한 url 변경 시 일일이 변경해야하는 번거로움을 피할 수 있음

models.py

```
from django.urls import reverse

class DataModel(models.Model):
  ...
  def get_absolute_url(self):
    return reverse('myapp:data_detail', kwargs={'id': self.id})
```

```
<!-- 사용 전 -->
<a href="{% url 'myapp:data_detail' data.id %}">{{data.title}}</a>

<!-- 사용 후 -->
<a href="{{data.get_absolute_url}}">{{data.title}}</a>
```

model(views.py 포함)에서 변경하면 모든 html에서 변경 사항이 적용됨.

## 사용자 정의 메소드

row-level(행) 단위의 인스턴스 처리를 위한 메소드를 정의하여 활용할 수 있음.

예를 들어, 테이블의 두 컬럼을 조합하여 하나의 결과로 화면에 출력하는 경우

```
from django.db import models

class Data_tbl(models.Model):
    col_name = models.CharField(max_length=20)
    ...

    @property
    def id_col(self):
        return '%d : %s' % (self.id, self.col_name)

# 사용 시
data = Data_tbl.object.get(id=1)
data.id_col
# 출력 -> 1 : some_data
```

## Database 생성

models.py 작성 후 db 및 테이블을 생성하기 위한 작업

먼저, 서버가 가동 중이라면 서버는 중지시키고 작업한다.

작업 순서

makemigrations -> migrate

```
py manage.py makemigrations myapp
py manage.py migrate
```

테이블을 수정할 경우도 위와 같은 과정을 다시 수행한다.(서버중지 -> makemigrations -> migrate -> 서버실행)

## Migrations

새로 models.py에 작성하거나 변경한 내용을(예를 들면 field를 추가하거나, model을 삭제하는 것 등) 데이터베이스 스키마에 적용.

- Migrations는 내 모델에 적용된 변화를 모아서 각 migrations files에 적용시켜주는 것
- python 문법으로 쓰인 models.py를 SQL문으로 바꾸고 database에 적용시킬 준비를 하는 작업
- setting.py의 INSTALLED_APP에 생성한 App이 등록되어 있지 않으면 실행되지 않음
    - makemigrations 하기 전에 등록할 것!

## Migrate

데이터베이스에 변화된 내용을 실제 테이블에 적용해주는 것.

- 각 앱의 migrations file은 해당 앱 폴더의 migrations 폴더 안에 존재하고, 승인 처리되어 코드로서 분배된다.

> migrate 명령은 INSTALLED_APPS의 설정을 탐색하여, myproject/settings.py의 데이터베이스 설정과 app과 함께 제공되는 database migrations에 따라, 필요한 데이터베이스 테이블을 생성.
> 또한 template 파일을 사용할 경우, 파일 인식을 위해서도 앱을 등록해야 한다.

## setting.py에 앱이름 등록

> myproject/myproject 폴더에 setting.py를 연다.
> INSTALLED_APPS 항목의 마지막에 '앱이름'을 추가한다.

```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
    '앱이름', #<-- 추가`
]
```

# 간이 프로젝트용 DB 스키마

테이블 명 - data_tbl
컬럼

- id - 기본키, 자동증가 정수
- str_data - 문자열 저장 컬럼. 50자.
- int_data - 숫자 저장 컬럼
- reg_data - 생성 일시 저장 컬럼
- upd_data - 수정 일시 저장 컬럼

## Database 초기화

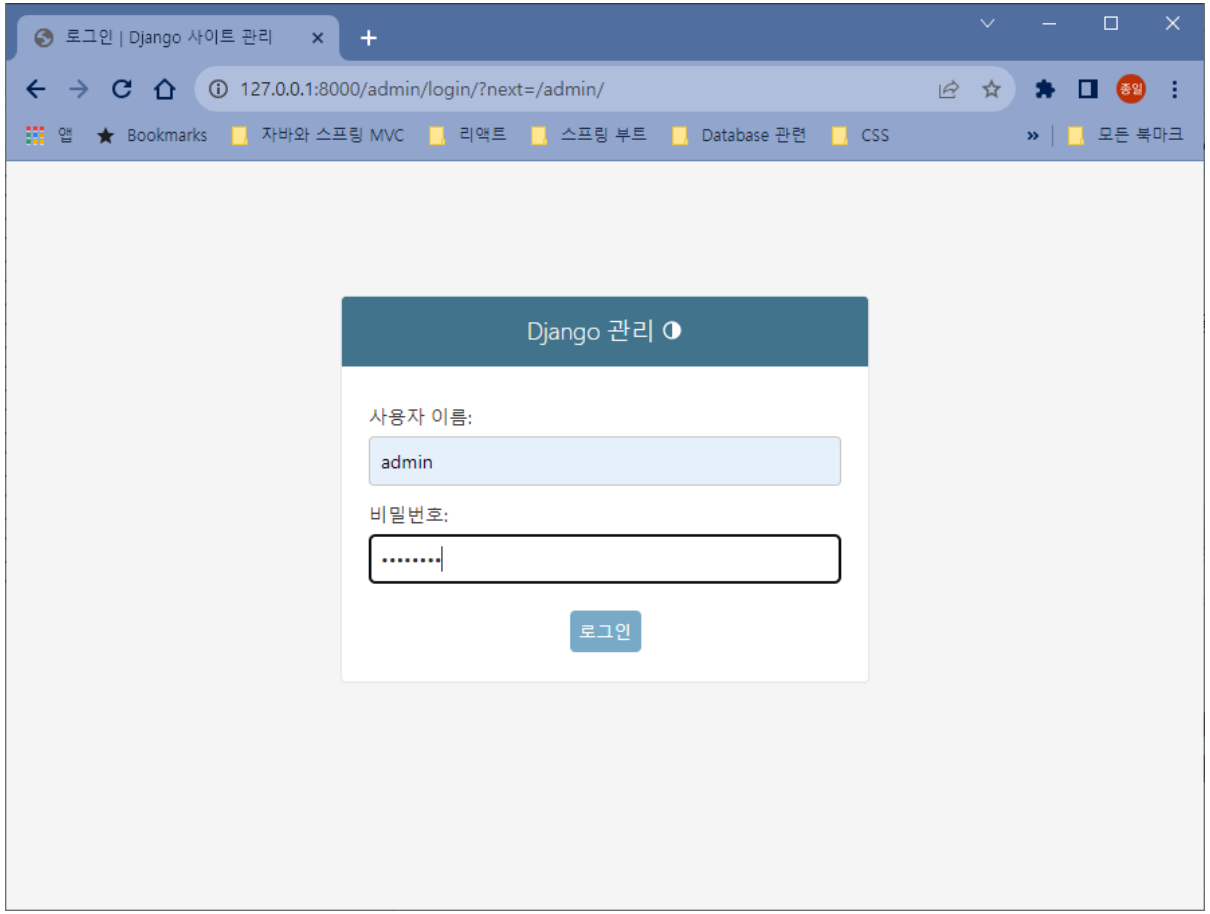
- 먼저 mirgrations 폴더의 파일 중 `__init__.py`를 제외한 나머지 파일을 모두 삭제한다.
- 다음 `db.sqlite3` 파일을 삭제한다.
- migration을 재수행한다.

# Admin 페이지

Django Admin은 모델의 CRUD 작업을 위한 사용자 인터페이스이다.

admin 페이지의 url

```
http://127.0.0.1:8000/admin
```



(하위)myproject 폴더의 urls.py에 admin 페이지에 대한 경로가 등록되어 있음.

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('', include('members.urls')),
    path('admin/', admin.site.urls), # admin 페이지의 url
]
```

admin 페이지로 들어가려면 사용자를 생성해야 하며, 생성한 사용자로 로그인이 필요.

# 사용자 생성

사용자 생성에 필요한 정보

- Username - 사용자 계정 이름
- Email address - 이메일 주소(가짜 이메일 주소 사용 가능)
- Password

```
> py manage.py createsuperuser
사용자 이름 (leave blank to use 'user'): admin
이메일 주소: admin@example.com
Password:
Password (again):
비밀번호가 너무 일상적인 단어입니다.
비밀번호가 전부 숫자로 되어 있습니다.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

# Model 보이기

myapp/admin.py에 관리해야할 Model을 포함시켜야 한다.

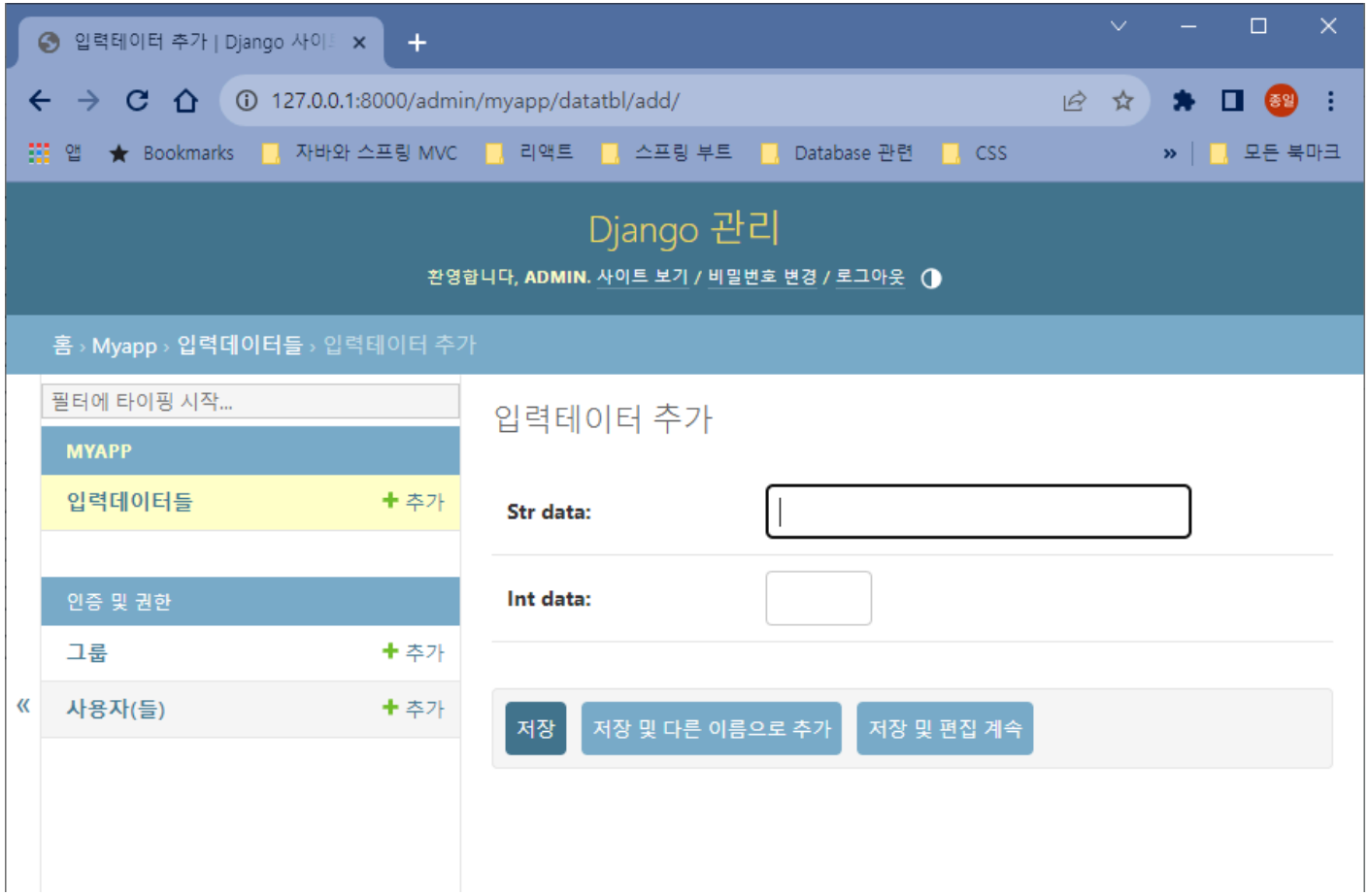
```
from django.contrib import admin
from .models import DataTbl

# Register your models here.
admin.site.register(DataTbl)
```

models.py에서 작성한 클래스를 등록하는 메소드 register()

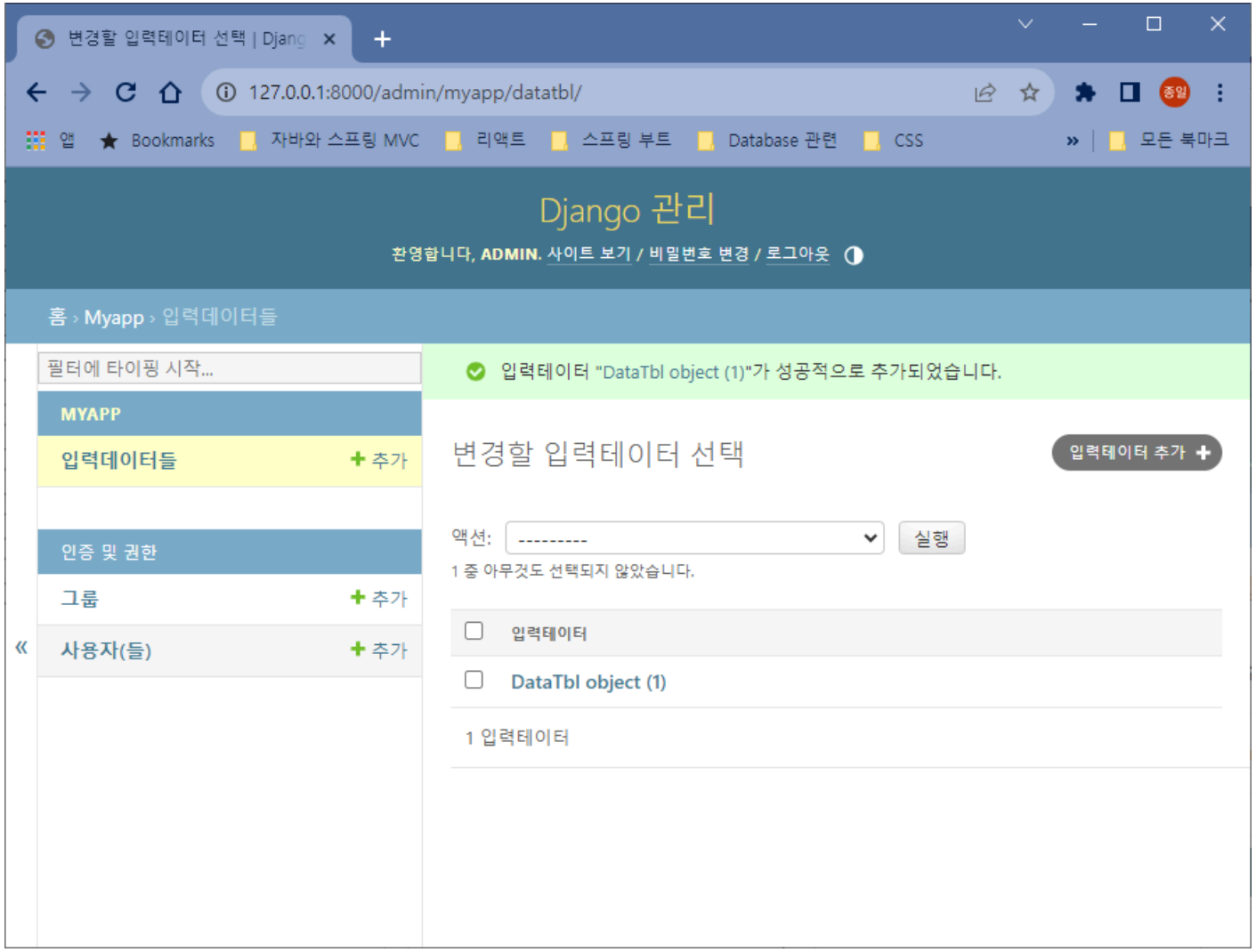
```
# 개별적인 클래스를 가져올 때
from .models import 클래스이름
# 모든 클래스를 가져올 때
from .models import *

admin.site.register(클래스이름)
```


이 화면에서 해당 테이블에 데이터를 추가하거나 수정/삭제할 수 있다.

# 정보 입력


입력테이터 추가 | Django 사이
127.0.0.1:8000/admin/myapp/datatbl/add/
앱
Bookmarks
자바와 스프링 MVC
리액트
스프링 부트
Database 관련
CSS
모든 북마크
Django 관리
환영합니다, ADMIN. 사이트 보기 / 비밀번호 변경 / 로그아웃
홈 › Myapp › 입력데이터들 › 입력테이터 추가
필터에 타이핑 시작...
MYAPP
입력데이터들
+ 추가
인증 및 권한
그룹
+ 추가
사용자(들)
+ 추가
입력테이터 추가
Str data:
Int data:
저장
저장 및 다른 이름으로 추가
저장 및 편집 계속

화면에서 정보를 입력하고 '저장'하면 다음과 같이 테이블이 저장되고 확인할 수 있다.

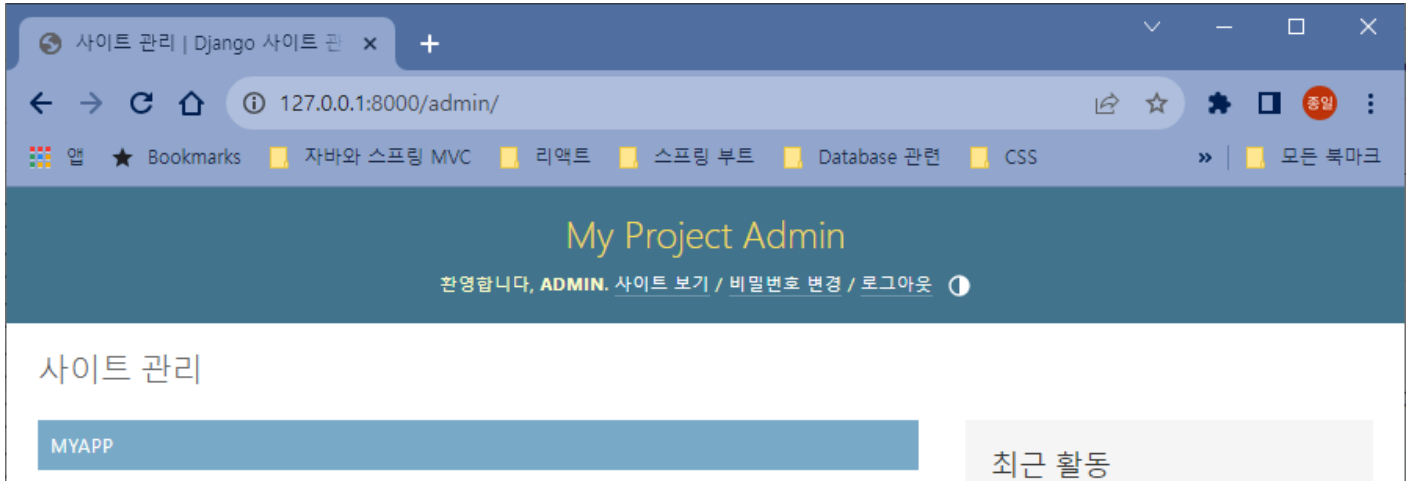


# Admin 페이지 수정

## Admin 페이지 제목 수정

하위 myproject 폴더의 urls.py에서 admin 페이지의 제목을 수정할 수 있다.

```
admin.site.site_header = 'My Project Admin'
```




페이지 제목이 'Django 관리'에서 'My Project Admin'으로 변경된다.

## 브라우저 탭에 출력되는 title 수정

마찬가지 urls.py에서 수정한다.

```
admin.site.site_title = 'My Admin'
```


탭의 제목이 '사이트 관리 | My Admin'으로 변경된다.

# 목록 제목 수정

마찬가지 urls.py에서 수정한다.

```
admin.site.index_title = 'Project Apps'
```


목록 제목이 '사이트 관리'에서 'Project Apps'로 변경된다.(탭 제목도 함께 변경된다.)

## 테이블 목록 출력 항목 설정

admin.py에서 해당 모델에 대한 클래스를 작성

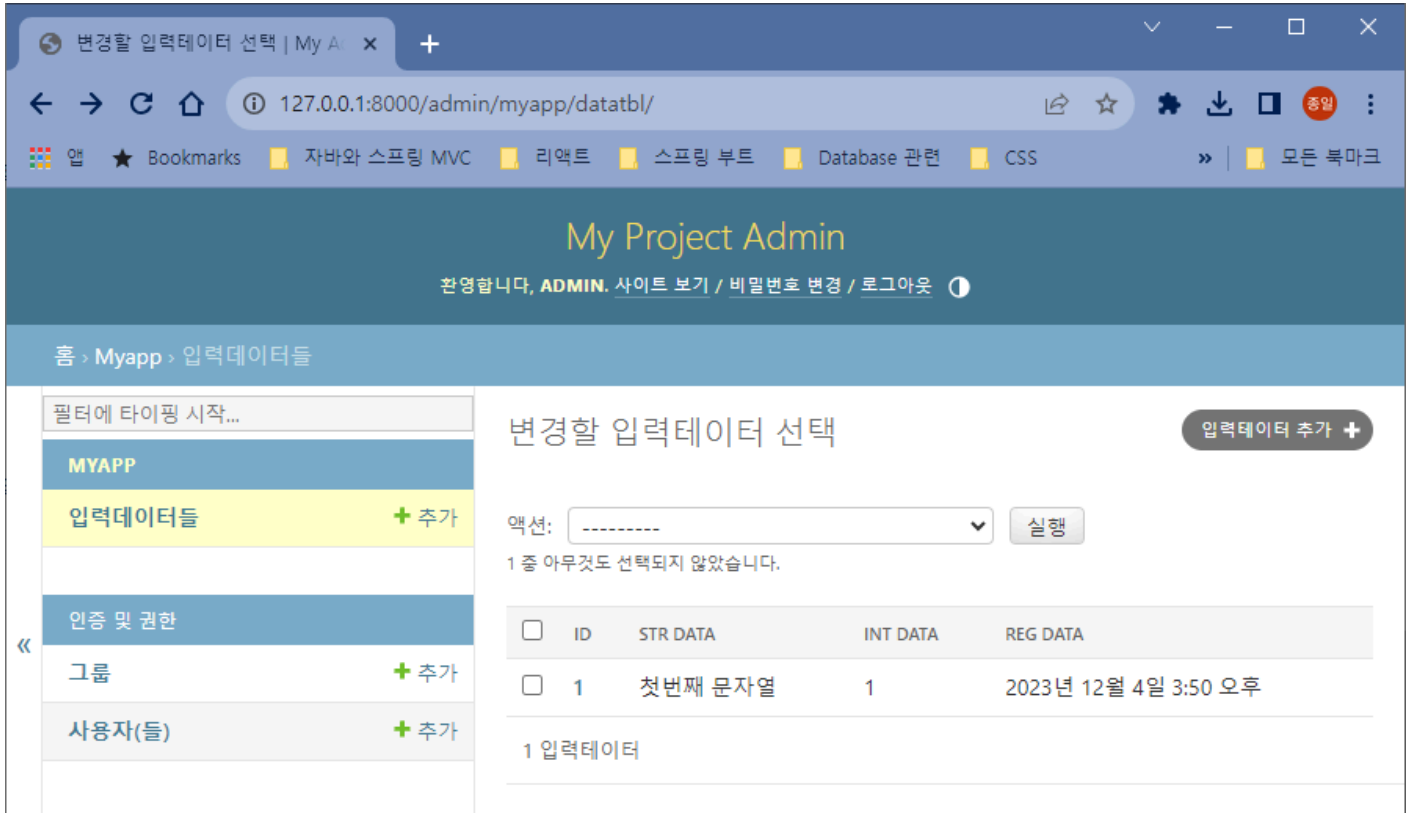
```
# Register your models here.
class DataTblAdmin(admin.ModelAdmin):
    list_display = ('id', 'str_data', 'int_data', 'reg_data')

admin.site.register(DataTbl, DataTblAdmin)
```

> 입력 양식 등 다양한 admin 사이트의 수정이 가능함

데코레이터를 사용한 등록 형식

```
@admin.register(DataTbl)
class DataTblAdmin(admin.ModelAdmin):
    list_display = ('id', 'str_data', 'int_data', 'reg_data')
```

# 처리 결과


변경할 입력테이터 선택 | My A
127.0.0.1:8000/admin/myapp/datatbl/
앱
Bookmarks
자바와 스프링 MVC
리액트
스프링 부트
Database 관련
CSS
모든 북마크
My Project Admin
환영합니다, ADMIN. 사이트 보기 / 비밀번호 변경 / 로그아웃
홈 › Myapp › 입력데이터들
필터에 타이핑 시작...
MYAPP
입력데이터들 + 추가
인증 및 권한
그룹 + 추가
사용자(들) + 추가
변경할 입력테이터 선택
입력테이터 추가 +
액션: ---------
실행
1 중 아무것도 선택되지 않았습니다.
ID
STR DATA
INT DATA
REG DATA
1
첫번째 문자열
1
2023년 12월 4일 3:50 오후
1 입력테이터

# Django Templates

내장된 template tag를 사용하여 파이썬코드를 html 코드로 변환하여 동적인 웹 페이지를 작성.

> 프로젝트 이름 - myproject
> 앱 이름 - myapp

# Template 문법

myapp 폴더 하위에 templates 폴더를 작성하고 html문서를 생성하여 django template 문법에 따라 작성.

## Template 주석

- 한줄 주석 - {# #}

```
{# 여기에 한줄로 주석을 작성 #}
```

- 여러줄 주석 - {% comment %} {% endcomment %}

```
{% comment %}
여러줄의 주석을 작성하는 경우
...
{% endcomment %}
```

## Template variable

{{ 식별자 }} : view에서 template으로 전달된 값을 출력하는 형식

'식별자'는 view에서 값을 저장한 객체의 key임.({key : value})

myapp/urls.py

```
urlpatterns = [
    path('', views.index, name='index'),
    path('testing', views.testing, name='testing'),
]
```

views.py

```
from django.http import HttpResponse
from django.template import loader
import datetime

def testing(request):
    template = loader.get_template('template.html')
    context = {
        'data': '출력할 내용',
        'date_data': datetime.datetime.now(),
    }
    return HttpResponse(template.render(context, request))
```

template.html

```
<p>{{data}}</p>
```

## Template Filter

변수의 표시에 변화를 줄 때 사용

변수 뒤에 '|'(pipe) 문자와 함께 사용

> 형식 : {{ value|filter }}

```
{{ date_data|date:'Y-m-d'}}
```

> 결과 : 2024-04-06

## 주요 필터

주요 필터 관련 문서 :
https://docs.djangoproject.com/ko/4.2/ref/templates/builtins/#std-templatefilter-add

## Template tag

변수 생성, 제어문 활용(if, for 등)을 위한 태그

실행할 태그를 {% %}로 묶어서 작성하며, 시작 태그와 종료 태그로 구성된다.

## 변수 생성 태그 : {% with %}

Template tag를 사용하여 직접 변수를 생성하여 활용할 수 있음

- 시작 태그 - {% with name='값' %}
- 종료 태그 - {% endwith %}

```
{% with name='user' %}
<h2>{{name}}님 안녕하세요.</h2>
{% endwith %}
```

### 조건 분기 태그 : {% if %}

변수의 값을 평가하여 true인 경우 코드 블록을 실행

- 시작 태그 : {% if condition %}
- 종료 태그 : {% endif %}

추가 조건을 처리하는 {% elif %}, false에 해당하는 {% else %}를 함께 사용
(조건 작성 시 연사자 앞뒤로 띄어쓰기 주의.)

```
{% if testing == 1 %}
  <h1>Hello</h1>
{% elif testing == 2 %}
  <h1>Welcome</h1>
{% else %}
  <h1>Goodbye</h1>
{% endif %}
```

## 활용 연산자

> 비교 연산자 : ==, !=, >, >=, <, <=
> 논리 연산자 : and, or
> 객체 내 항목의 존재 여부 : in/not in
> 두 객체의 동일 여부 : is/is not

### 반복 태그 : {% for %}

배열, 목록 또는 딕셔너리의 항목을 순서대로 반복하여 출력(또는 활용)할 때 사용

- 시작 태그 : {% for %}
- 종료 태그 : {% endfor %}

목록 데이터가 빈 상태의 처리를 위한 {% empty %}를 함께 사용

```
<ul>
  {% for item in object %}
    <li>{{ itme.name }}</li>
  {% empty %}
    <li>No item</li>
  {% endfor %}
</ul>
```

역순으로 반복할 경우 reversed를 붙인다.

```
{% for item in object reversed %}
```

### forloop

forloop 객체를 활용하여 반복 횟수 등의 목록 index나 count와 같은 상태(status) 값을 활용할 수 있다.

| variable | 설명 |
| --- | --- |
| forloop.counter0 | 0부터 시작하는 반복 횟수 출력 |
| forloop.counter | 1부터 시작하는 반복 횟수 출력 |
| forloop.revcounter0 | 0부터 시작하는 반복 횟수를 역순으로 출력 |
| forloop.revcounter | 1부터 시작하는 반복 횟수를 역순으로 출력 |

### forloop

| variable | 설명 |
| --- | --- |
| forloop.first | 첫번째 반복이면 True |
| forloop.last | 마지막 반복이면 True |
| forloop.parentloop | 중첩된 반복일 경우 외부 반복 객체를 참조<br>예: forloop.parentloop.counter |

예)

```
{% for item in lists %}
<li>{{forloop.count}} : {{item}}</li>
{% endfor}
```

## CSRF(Cross Site Request Forgery) : {% csrf_token %}

> CSRF : 웹사이트 취약점 공격의 하나로, 사용자가 자신의 의지와는 무관하게 공격자가 의도한 행위(수정, 삭제, 등록 등)로 특정 웹사이트에 요청하게 하는 공격

CSRF 공격을 방어하기 위한 태그로 `<form>` 에서 ModelForm을 사용하기 전에 작성.

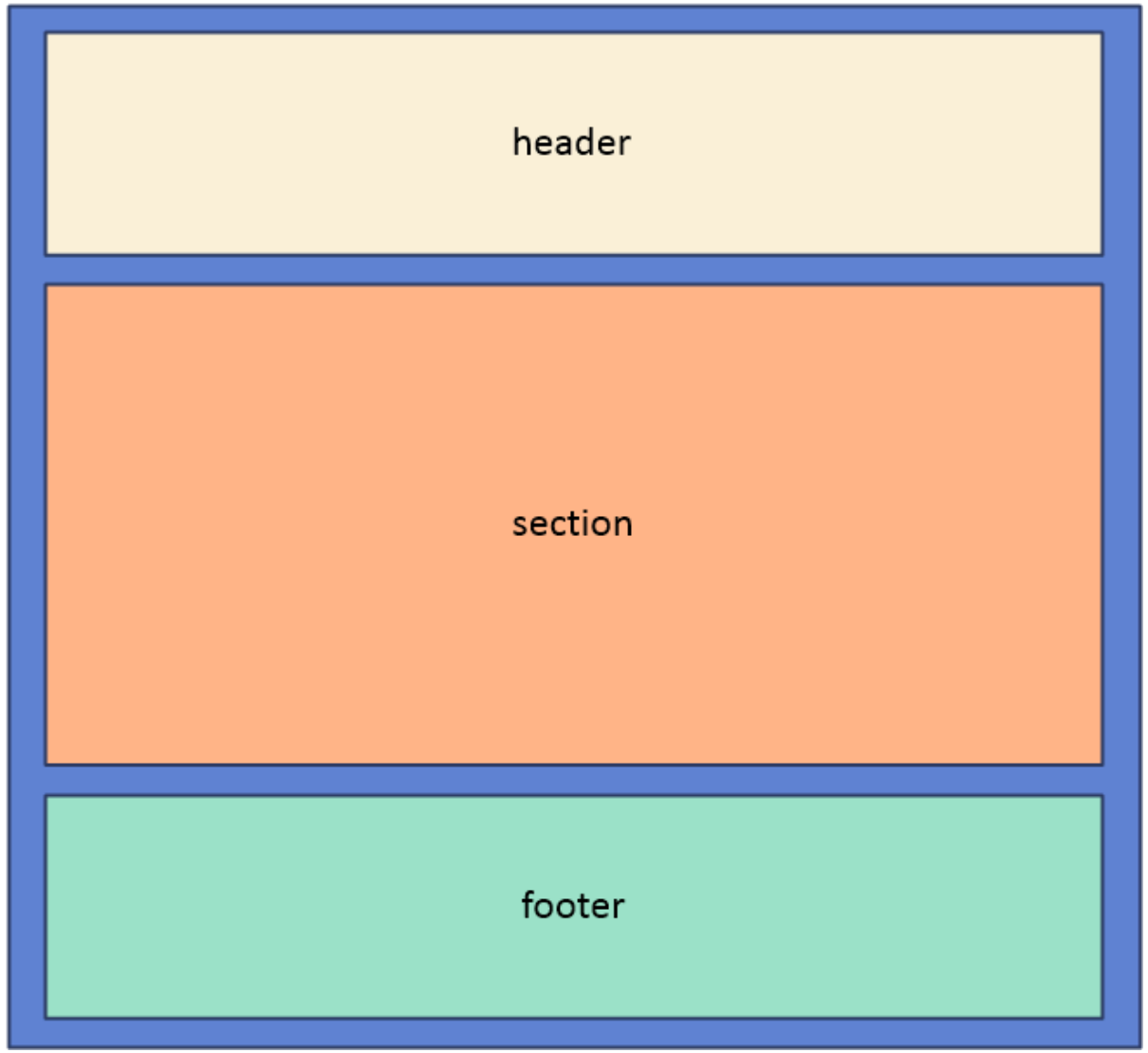
```
<form method="post">
    {% csrf_token %}
    <div>
        <label>문자열</label>
        {{ form.some_input1 }}
    </div>
    <div>
        <label>숫자</label>
        {{ form.some_input2 }}
    </div>
    <input type="submit" value="Send">
</form>
```

### 동작 과정

- 사용자가 해당 페이지에 접속하면 Django에서 자동으로 csrf_token을 클라이언트로 보내어 cookie에 저장
- 사용자가 form을 모두 입력한 후 전송
- form과 cookie의 csrf_token을 함께 POST로 전송
- 전송된 token의 유효성을 검증
- 유효한 요청이면 요청을 처리
- token이 유효하지 않거나(없거나 값이 잘못된 경우) 검증 오류 시에는 403 Forbidden Response 반환

## Template 확장

header, navigation, footer 등 모든 template에서 공통적인 부분을 하나의 template에 모아 놓는 방식



그림과 같은 구조의 페이지를 작성할 경우, master.html은 다음과 같이 작성한다.

```
<!DOCTYPE html>
<html lang="ko">
<head>
    <meta charset="UTF-8">
    <title>페이지 제목</title>
</head>
<body>
    {% include "header.html" %}
    {% block section %}{% endblock %}
    {% include "footer.html" %}
</body>
</html>
```

## 공통된 영역의 파일 포함 - {% include "some.html" %}

별도로 작성한 html 파일을 특정 페이지에 포함시킬 때 사용하는 template 태그

위의 예에서 header와 footer를 따로 파일로 작성하여 모든 페이지에 포함시키는 형식으로 활용할 수 있다.

master.html은 모든 페이지의 공통된 부분을 처리하는 페이지이므로 결과적으로 모든 페이지에 동일한 header와 footer가 포함된다.

> 별도의 html 파일에는 <html>, <head> 태그 등은 작성하지 않고 <body>에 들어갈 요소관련 태그만 작성

## 개별적인 페이지 영역 처리 - {% block xxx %} ... {% endblock %}

개별적인 화면을 위한 영역은 block 태그를 사용한다. 이 영역은 개별적인 내용이 나오는 html 페이지를 각각 작성하여 포함시키도록 한다.

예를 들어, index.html 페이지는 master 페이지를 확장(extends)시켜서 head 부분을 처리하고 section 부분만 작성하는 방식으로 완성한다.

```
{% extends 'master.html' %}
{% block section %}
    index.html에서 보여질 요소...
{% endblock %}
```

# Static resource 처리

Static resource(정적 자원)는 배경 이미지, 스타일 시트, Javascript 등을 나타낸다.

## static 폴더 설정

project_name/project_name/setting.py에 이미 설정되어 있다.

```
...
STATIC_URL = 'static/'
...
```

프로젝트 폴더 하위에 static 폴더를 작성하여 해당 파일을 저장

```
project_name/
├── manage.py
├── project_name/
└── app_name/
    └── static/
        ├── css/
        ├── images/
        └── js/
```

- 각 자원들은 폴더로 구분하여 처리하는 것이 좋다.
  - 이미지 파일 - images
  - 스타일 시트 - css
  - Javascript - js

## Static url

myproject/myproject 폴더의 urls.py에 static url을 작성한다.

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    ...
] + static(settings.STATIC_URL, document_root=settings.STATIC_URL)
```

## Template 안에서 static 불러오기 - {% load static %}

html 파일에서 정적 자원을 활용하기 위해서 사용하는 태그.

반드시 자원 활용 전에 먼저 작성되어야 한다.

```
{% load static %}
<!DOCTYPE html>
```

## static 활용 태그 - {% static '경로/파일' %}

static 파일을 활용하는 요소에서 사용하는 태그.(위의 예와 같이 사용)

```
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
```

# QuerySet

Django ORM(Object Relational Mapping)에서 제공하는 데이터 타입으로 DB에서 가져온 객체 목록.

DB 테이블의 데이터를 불러오기 위한 작업은 view와 template에서 수행한다.

> 프로젝트 이름 - myproject
> 앱 이름 - myapp
> 테이블 모델 - DataTbl
> 컬럼 : str_data, int_data, reg_data, upd_data

## objects

QuerySet을 만들어 주는 manager.

manager는 models.py에서 정의한 클래스를 django에서 활용하기 위한 queryset 형태로 만들어 주는 역할을 담당.

# 여러개의 결과 검색

### all()

```
DataTbl.objects.all()
```

QuerySet의 전체(테이블의 전체) 데이터를 불러온다. 다음의 SQL query문과 같다.

```
SELECT * FROM DataTbl
```

## filter()

```
DataTbl.objects.filter(str_data='문자열')
```

id가 1인 데이터를 불러온다. 다음의 SQL query문과 같다.

```
SELECT * FROM DataTbl
WHERE str_data = '문자열'
```

## exclude()

```
DataTbl.objects.exclude(id=2)
```

id가 2인 데이터를 제외한 나머지 데이터를 불러온다. 다음의 SQL query문과 같다.

```
SELECT * FROM DataTbl
WHERE id != 2
```

## Chaining

```
DataTbl.objects.filter(str_data='문자열').exclude(id=2)
```

filter와 exclude를 연속적으로 사용할 수 있다. id가 2인 데이터를 제외한 str_data가 '문자열'인 데이터를 불러온다. 다음의 SQL query문과 같다.

```
SELECT * FROM DataTbl
WHERE str_data = '문자열' AND id != 2
```

## Lookup Filter

filter(), exclude() 메소드에서 사용할 수 있는, 필드(컬럼)별 구체적인 값에 대한 비교를 가능하게 하는 Django의 내장 모듈. 연속된 두개의 '_'(underscore)로 시작.

### __contains, __icontains

특정 문자가 포함된 데이터를 검색할 때 사용. __contains는 대소문자를 구분하며, __icontains는 대소문자를 구분하지 않는다.

```
DataTbl.objects.filter(str_data__contains='abc')
```

### __startwith, __endwith

특정 문자로 시작(startwith)하거나 끝(endwith)나는 데이터를 검색할 때 사용.

```
DataTbl.objects.filter(str_data__startwith='ab')
```

## __gt, __lt

지정한 값보다 크(gt)거나 작은(lt) 데이터를 검색할 때 사용.

```
DataTbl.objects.filter(id__gt=2)
```

## __isnull

True로 지정 시 해당 컬럼이 null인 데이터만, False로 지정 시 해당 컬럼이 null이 아닌 데이터만 검색.

```
DataTbl.objects.filter(str_data__isnull=True)
```

### __year, __month, __day, __date

date 타입의 필드에서 특정 년(__year), 월(__month), 일(__day) 혹은 특정 날짜(__date : YY-MM-DD 형식)의 데이터를 검색.

### AND/OR

두개 이상의 조건을 AND 또는 OR을 이용하여 결합할 수 있다.

- AND 조건 : 두 개 이상의 쿼리 셋을 '&' 로 연결
- OR 조건 : 두 개 이상의 쿼리 셋을 '|'로 연결

```
DataTbl.objects.filter(str_data__isnull=True) & DataTbl.objects.filter(str_data__icontains='abc')
```

# 하나의 결과 검색

## get()

```
DataTbl.objects.get(id=1)
```

id가 1인 데이터 하나를 불러온다. 다음의 SQL query문과 같다.

```
SELECT * FROM DataTbl WHERE id = 1
```

## filter와의 차이

filter는 데이터가 없을 경우 빈(empty) queryset을 불러오지만, get의 경우 DoesNotExist 메시지를 출력.
get은 데이터가 여러개일 경우 MultipleObjectsReturned라는 메시지를 출력.
또 get은 chaining을 할 수 없다.

# 기타 메소드

## count()

QuerySet에 포함된 데이터의 개수를 구하는 데 사용.

```
DataTbl.objects.count()
```

## exists()

filter, exclude와 chining하여 사용. 해당 데이터가 있으면 True, 없으면 False를 리턴.

```
DataTbl.objects.filter(str_data__icontains='abc').exists()
```

## values(), values_list()

QuerySet의 내용을 dictionary 형태로 변환하여 반환. (template에 전달하기 위해 변환이 필요)

인자가 없을 경우 모든 컬럼의 데이터를 반환하지만, 특정 컬럼을 넣으면 해당 컬럼의 데이터만 반환.

values_list는 내용을 list 형태로 변환하여 반환.

## order_by()

인자에 작성한 컬럼명을 기준으로 정렬한 데이터를 반환.
기본은 오름차순 정렬이며, 컬럼명 앞에 '-'(minus)를 붙이면 내림차순으로 정렬.

# ModelForm을 활용한 CRUD

ModelForm을 활용하면 model에서 정의한 내용을 기반으로 template의 form을 작성할 수 있다.

## Form vs ModelForm

- Form은 직접 입력을 위한 필드를 정의해야 하는데, 이 코드는 model.py에서 정의한 모델 필드와 거의 동일하기 때문에 중복된 작업이 된다.
- ModelForm은 모델과 입력을 위한 필드를 지정하면 자동으로 template의 form에 필요한 필드를 생성한다.

## ModelForm의 장점

- 폼을 위한 HTML을 작성할 필요가 없다.
- 데이터의 유효성 검사를 자동으로 수행한다.
- 잘못된 데이터 입력 시 에러 메시지를 출력해준다.
- 재사용이 가능하다.

> 프로젝트 이름 - myproject
> 앱 이름 - myapp
> 테이블 모델 - DataTbl
> 컬럼 - str_data, int_data, reg_data, upd_data

# Forms.py

app 폴더(myapp)에 form.py를 생성하여 다음과 같이 라이브러리를 import한다.

```
from django.forms import *
from .models import DataTbl

class DataForm(ModelForm):
    class Meta:
        model = DataTbl
        fields = '__all__'
        exclude = ('reg_data', 'upd_data',)
        widgets = {
            'str_data': TextInput(attrs = {
                'placeholder': '문자열',
            }),
            'int_data': NumberInput(attrs = {
                'placeholder': '숫자',
            }),
        }
```

## Meta class

Model의 정보를 작성하는 내부 클래스.

ModelForm에서 사용하는 model을 지정하고 필드의 포함 및 제외 등을 정의.

## model

models.py에 작성한 model을 지정

## fields

form으로 생성할 필드를 지정.

- __all__ : model의 모든 필드를 지정하는 값.
- 모든 필드를 지정하지 않는 경우 튜플로 생성할 필드를 지정
    - fields = ('field1', 'field2')

### exclude

제외할 필드를 지정. 튜플로 작성.

### widgets

HTMl input 태그에 적용될 attributes를 지정하기 위해 사용. dictionary로 작성.

```
widgets = {
    'field1': TextInput(attrs = {
        'class': 'some_class',
        'id': 'some_id',
        ...
    }),
    'field2': ...
}
```

# 데이터 입력 처리

## 입력 페이지 작성

writeForm.html에 <input> 대신 form.py에서 작성한 field를 사용한다.

```
<div>
    <form method="post" class="i-form">
        {% csrf_token %}
        <div>
            <label>문자열</label>
            {{ form.str_data }}
        </div>
        <div>
            <label>숫자</label>
            {{ form.int_data }}
        </div>
        <input type="submit" value="Send">
    </form>
</div>
{% endblock %}
```

## view 처리

form.py에 작성한 ModelForm class를 import

```
from .form import DataForm
```

HTML 전송 method에 따라 '입력 페이지로의 이동'인지 '데이터의 저장'인지를 구별하여 하나의 메소드로 처리

- request.method == 'GET' : 입력 페이지로의 이동
- request.method == 'POST' : 입력 데이터 DB insert

```
def write(request):
    if request.method == 'POST':
        data_form = DataForm(request.POST)
        if data_form.is_valid():
            data = data_form.save(commit=False)
            data.save()
            return redirect('index')
    else:
        title = "처음으로"
        data_form = DataForm()
        template = loader.get_template('writeForm.html')
        context = {
            'title': title,
            'form': data_form,
        }
        return HttpResponse(template.render(context, request))
```

**is_valid()**

ModelForm에 정의되어 있는 유효성 검사 메소드.

각 필드에 맞지 않는 데이터 입력 시 저장하지 못하게 막는다.

**save()**

Form에 바인딩된 데이터로 DB 객체를 만들어 저장하는 메소드.

기존 model의 instance가 있다면(수정하는 경우라면) DB Update를 수행하고, 없다면 DB Insert를 수행.

Transaction을 위해 인자로 commit을 False로 설정하고 이후 save()를 재실행하는 방식으로 활용할 것.

(파일 업로드 처리와 같은 경우 파일명 등의 추가 정보를 ModelForm에 넣은 다음 insert 하기 위한 방식이기도 함.)

**ModelForm(arg1, arg2)**

ModelForm class를 상속받아 작성한 클래스의 instance 생성 시 arg1은 HTML에서 넘어온 데이터.

- DB insert 시에는 arg1만 작성한다.
- DB update 시에는 arg1은 HTML에서 넘어온 데이터
- arg2는 QuerySet의 instance.

save()를 실행하면 arg1의 데이터로 DB를 update 한다.

arg1에 instance를 작성하는 경우 QuerySet의 instance(데이터)로 template에 출력할 필드에 데이터를 지정하는 것이다.(수정 처리 부분에서 확인할 것)

## url 처리

myapp/urls.py

```
urlpatterns = [
    ...
    path('write', views.write, name='write'),
    ...
]
```

# 데이터 수정 처리

입력 처리와 마찬가지로 HTML 전송 method에 따라 '입력 페이지로의 이동'인지 '데이터의 저장'인지를 구별하여 하나의 메소드로 처리

- request.method == 'GET' : 수정 페이지로의 이동
- request.method == 'POST' : 수정 데이터 DB update

template(index.html)

```
<a href="{% url 'update' id=item.id %}">{{item.str_data}}</a>
```

urlpatterns(urls.py)

```
urlpatterns = [
    ...
    path('update/<int:id>', views.update, name='update'),
]
```

## parameter의 처리

Client에서 Server로 데이터를 전송할 때 사용하는 방식(상세 페이지 이동 등에서 활용)

- Query Parameter

```
/data?id=1
```

- Path Variable

```
/data/1
```

## Query Parameter 활용

Query Parameter를 사용하는 경우 template에는 다음과 같이 작성한다.

```
<a href="data?id={{item.id}}">데이터 전송</a>
```

url의 urlpatterns에는 다음과 같이 작성한다.

```
urlpatterns = [
    ...
    path('data', views.data, name='data'),
    ...
]
```

view의 메소드에서는 다음과 같이 작성한다.

```
def data(request):
    id = request.GET['id']
    ...
```

## Path Variable 활용

Path Variable을 사용하는 경우 template에는 다음과 같이 작성한다.

```
<a href="{% url 'data' id=item.id %}">데이터 전송</a>
```

url의 urlpatterns에는 다음과 같이 작성한다.

```
urlpatterns = [
    ...
    path('data/<int:id>', views.data, name='data'),
    ...
]
```

view의 메소드에서는 다음과 같이 작성한다.

```
def data(request, id):
    ...
```

## get_object_or_404()

QuerySet의 get()는 검색한 데이터가 없을 경우 DoesNotExist 메시지를 반환한다. 이 결과를 처리할 때 Http404 예외를 발생시켜 사용자에게 해당 결과가 없음을 보여주어야 한다.

즉, get_object_or_404()는 get()과 404 처리를 합친 것이다.

# 데이터의 삭제 처리

template(index.html)

```
<a href="{% url 'delete' id=item.id %}">[삭제]</a>
```

urlpatterns(urls.py)

```
urlpatterns = [
    ...
    path('delete/<int:id>', views.delete, name='delete'),
]
```

view method(views.py)

```
def delete(request, id):
    data = get_object_or_404(DataTbl, id=id)
    data.delete()
    return redirect('index')
```

## delete()

QuerySet에서 해당 데이터를 삭제하고 이를 DB에 반영.

# Custom Error Page

404, 500 등의 에러 페이지를 원하는 디자인으로 구성할 수 있다.

> 프로젝트 이름 - myproject
> 앱 이름 - myapp

## Setting.py

프로젝트를 생성하면 Debug 모드는 기본적을 True로 설정된다. 이 경우 Django의 기본 에러페이지를 사용하기 때문에 False로 변경한다.

```
...
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False
...
```

그 다음 문장인 ALLOWED_HOSTS는 서버 접근이 허용된 도메인 네임 또는 IP를 작성한다. 개발 중이기 때문에 'localhost'와 '127.0.0.1'을 함께 추가한다.

```
...
ALLOWED_HOSTS = ['localhost','127.0.0.1']
...
```

DEBUG를 False로 설정하고 ALLOWED_HOSTS 작성하지 않을 경우 서버는 실행되지 않는다.

## Error Page

'templates' 폴더 밑에 error 폴더를 생성하여 404.html(또는 다른 이름의) 페이지를 작성한다.

> 기본 설정으로 error 폴더를 생성하지 않고 templates 폴더에 작성하는 것도 가능하며, 각 에러 페이지의 이름은 stauts 코드와 같은 이름을 사용할 수도 있다.

- 404 오류 -> 404.html
- 500 오류 -> 500.html 등

# Messages Framework

Django의 messages framework은 사용자에게 메시지를 표시하는 유용한 도구이다. 성공 메시지, 경고 메시지, 오류 메시지 등을 쉽게 생성하고 표시할 수 있다.

Message Framework 다음과 같은 특징을 갖는다.

- 1회성 메시지를 담는 용도로 사용한다.
- HttpRequest 인스턴스를 통해 메시지를 담을 수 있다.
- 메시지는 1회 노출되고 사라진다.

## Message level

다양한 메시지 레벨: Django의 messages framework은 다양한 메시지 레벨을 제공합니다. 주요 메시지 레벨에는 다음이 포함됩니다.

- DEBUG: 개발 목적으로 사용되며 주로 디버그 정보를 제공합니다.
- INFO: 일반적인 정보를 제공합니다.
- SUCCESS: 작업이 성공적으로 완료되었음을 나타냅니다.
- WARNING: 사용자에게 경고를 제공합니다.
- ERROR: 오류 메시지를 제공합니다.

# File 처리

영화 정보 관리 사이트 프로젝트로 파일 관련 처리를 진행한다.

> 프로젝트 이름 - movieinfo
> 앱 이름 - movies
> DB - Movie

포스터 이미지의 저장 및 수정, 삭제, 다운로드를 처리.

## File Upload

ImageField를 사용하기 위해서는 파이썬 이미지 처리 라이브러리인 pillow가 필요하다.

```
pip install pillow
```

FileField를 사용할 경우 pillow를 설치할 필요 없음

## ImageField와 FileField의 차이

ImageField

> ImageField는 이미지 파일을 저장하기 위한 특수한 필드. 파일을 업로드할 때, 이미지인지를 확인하고, 필요에 따라 썸네일 생성과 같은 추가적인 작업을 수행할 수 있다.(이미지 크기 조정이나 필터링 등)

FileField

> FileField는 모든 종류의 파일을 저장하기 위한 범용 필드. 이미지 뿐만 아니라 문서, 비디오, 오디오 등 모든 종류의 파일을 다룰 수 있다. 이미지인지 여부를 검증하지 않기 때문에 이미지 파일 이외의 파일을 업로드할 수 있다.

파일 정보 저장을 위해 models.py에 FileField 또는 ImageField를 설정

```
class Movie(models.Model):
    ...
    mposter = models.FileField('',upload_to='images/',blank=True)
```

또는(pillow를 설치했다면)

```
class Movie(models.Model):
    ...
    mposter = models.ImageField('',upload_to='images/',blank=True)
```

FileField 또는 ImageField에는 upload_to로 파일을 저장할 경로를 지정한다.

저장을 위한 기본 경로의 설정은 settings.py에서 처리한다.

```
...
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

movieinfo/movieinfo 폴더의 urls.py에 settings.py에서 설정한 경로 정보를 가져오기 위해 다음을 추가한다.

```
...
from django.urls import re_path as url
from django.conf import settings
from django.views.static import serve

urlpatterns = [
    ...
    url(r'^media/(?P<path>.*)$', serve, {'document_root':settings.MEDIA_ROOT}),
    ...
]
```

> 파일의 저장 경로는 'movieinfo/media/images/'가 된다.

```
movieinfo/
├── manage.py
├── media
│   └── images
├── movieinfo/
└── movies/
    └── static/
```

form.py의 ModelForm에 각 입력 필드의 속성을 설정

request에서 multipart로 전송되는 파일 정보를 꺼내 movie 객체의 해당 필드에 넣은 다음 DB에 저장한다.

HTML의 form 태그에는 반드시 enctype을 'multipart/form-data'로 설정해야 한다.

```
<form method="post" enctype="multipart/form-data">
...
```

## File Download

파일의 위치 정보와 물리적인 파일 로딩을 위해 다음과 같이 views.py에 import를 추가한다.

```
from django.conf import settings # 파일 저장위치를 setting.py에서 가져오기 위해 추가
import os                        # 물리적인 파일접근을 위해 추가
```

settings.py에 설정된 media 폴더의 위치와 DB에 저장된 file의 경로 및 파일명을 path로 설정한다.

os.path.basename() 메소드로 파일명과 확장자를 구할 수 있다.

with ... as는 파일 처리 후 자동으로 파일 객체를 해제해 주기 때문에 유용하다.

HttpResponse를 사용하여 파일을 전송할 때 Content-Disposition 헤더를 사용하여 파일명을 지정한다.

저장 위치로부터 해당 파일을 읽어와서 HttpResponse 객체에 담아서 사용자 컴퓨터로 전송한다.

Content-Disposition 헤더에 파일명을 설정하는데, 이때 한글 파일명은 UTF-8로 인코딩하고, 그 결과를 Latin-1로 디코딩한다.

- Latin-1으로 디코딩하는 이유는 HTTP 헤더의 Content-Disposition 필드에는 ASCII 문자 집합만 사용 가능하기 때문에 UTF-8로 인코딩된 문자열을 직접 사용할 수 없기 때문이다.

urls.py는 다음과 같다.

```
urlpatterns = [
    ...
    path('download', views.download, name='download'),
    ...
]
```

detail.html의 이미지에 다운로드의 링크를 건다.

```
<a href="/download?poster={{movie.mposter}}">
    <img class="poster" src="/media/{{movie.mposter}}">
</a>
```

## File Update

본 예제에서와 같이 하나의 파일만 저장하는 프로그램일 경우 기존 파일을 삭제하고 새 파일을 저장하는 과정이 필요하다.

DB에 데이터를 저장하기 전 또는 후에 이런 과정을 진행하게 되는데 이때 Django의 Signal을 활용한다.

> Django의 Signal은 애플리케이션에서 발생하는 특정 이벤트를 감지하고 처리하기 위한 메커니즘이다. 이벤트가 발생하면 Signal을 트리거하고, 이에 연결된 리스너 함수들이 실행된다. Signal은 애플리케이션 내에서 여러 가지 작업을 자동화하거나 이벤트에 대한 처리를 추가하기 위해 사용된다.

이 예제어서 파일 수정을 위해 model 객체의 signal을 활용한다.

model 객체의 Signal은 다음과 같다.

| Signal | 설명 |
| --- | --- |
| pre_save | model 객체가 저장되기 전에 발생하는 signal. 객체가 저장되기 전에 어떤 처리를 수행하려는 경우에 사용. |
| post_save | model 객체가 저장된 후에 발생하는 signal. 객체가 저장된 후에 추가적인 작업을 수행하려는 경우에 사용. |
| pre_delete | model 객체가 삭제되기 전에 발생하는 signal. 객체가 삭제되기 전에 어떤 처리를 수행하려는 경우에 사용. |
| post_delete | model 객체가 삭제된 후에 발생하는 signal. 객체가 삭제된 후에 추가적인 작업을 수행하려는 경우에 사용. |

Signal과 연결된 리스너 함수를 만들 때, receiver 데코레이터를 사용한다.

파일의 수정은 models.py에서 처리한다. 먼저 receiver와 물리적인 처리를 위한 os를 import한다.

```
from django.dispatch import receiver
import os
```

파일 수정을 객체 저장 전에 먼저 수행하기 위해 pre_save를 활용한다.

## File Delete

파일의 삭제는 데이터의 삭제 후에 진행하도록 post_delete를 활용한다. 해당 코드는 다음과 같다.

models.py

```
@receiver(models.signals.post_delete, sender=Movie)
def file_delete(sender, instance, **kwargs):
    for field in instance._meta.fields:
        field_type = field.get_internal_type()
        if field_type == 'FileField' or field_type == 'ImageField':
            ori_file = getattr(instance, field.name)
            if ori_file and os.path.isfile(ori_file.path):
                os.remove(ori_file.path)
```

> 완성된 코드는 project 폴더에 있습니다.