웹 프레임워크

04장. Django의 핵심기능

Python-Django Web Framework

for Web Application

MVT방식의 템플릿 시스템

- 사용자에게 보여주는 화면, 즉 UIUser Interface를 담당하고 있는 기능이 템플릿 시스템.
- 템플릿 코드를 작성 시에 HTML 코드와 장고의 템플릿 태그가 섞여서 사용
- 중요한 점은 템플릿에서는 로직을 구현하는 것보다 사용자에게 어떻게 보여줄지에 대한 룩 앤 필을 표현

- 템플릿 태그의 특징
  - if태그, for태그 등이 있지만, 파이썬 언어와는 구별되는 문법의 템플릿 시스템 언어
  - 템플릿 태그를 템플릿 파일로 해석하는 과정을 장고에서는 렌더링이라 함

4.3.1 템플릿 변수

- 템플릿 시스템은 뷰 함수로부터 전달 받은 데이터(변수)를 출력.
- 변수명은 일반 프로그래밍의 변수명처럼 문자, 숫자, 밑줄(_)을 사용하여 이름을 정의
- 변수의 속성에 접근 할 수 있는 (.)도트 표현식도 가능
  - 템플릿 문법에서 도트를 사용하면 다음 순서로 찾기(lookup)를 시도함 예) foo.bar
    - foo가 딕셔너리 타입인지 확인, 그렇다면 foo['bar']로 해석
    - 그 다음은 foo의 속성을 찾음, bar라는 속성 이 있으면 foo.bar로 해석
    - 그것도 아니면 foo가 리스트인지를 확인, 그렇다면 foo[bar]로 해석

- 표현방법은 중괄호 2개로 감싸서 표현

```
{{ variable }}
```

4.3.2 템플릿 필터

- 필터링이란 어떤 객체나 처리결과에 필터명령을 적용해서 해당 명령에 맞게 최종 결과값을 변경하는 것
- 장고에서 필터는 파이프(|) 문자를 사용함

- name 변수값의 모든 문자를 소문자로 바꿔주는 필터

```
{{ name | lower }}
```

- 필터를 체인처럼 연결할 수도 있음

```
{{ text | escape | linebreaks }}
```

- 다음은 bio 변수값 중에서 앞에 30개의 단어만 보여 주고, 줄 바꿈 문자는 모두 없애 줌.

```
{{ bio | truncatewords:30 }}
```

4.3.2 템플릿 필터

- 필터의 인자에 빈칸이 있는 경우는 따옴표로 묶어서 출력

```
{{ list | join:" // " }}
```

- value 변수값이 False이거나 없는 경우, “nothing” (입력한 문자열) 으로 출력

```
{{ value | default:"nothing" }}
```

- 변수의 길이 값 반환

```
{{ value | length }}
```

- value 변수값에서 HTML 태그를 모두 제거

```
{{ value | striptags }}
```

4.3.2 템플릿 필터

- value 변수값이 1이 아니면 복수 접미사s를 붙여 출력함

```
{{ value | pluralize }}
```

- 사용자가 입력한 문자열로 출력하려면 필터에 파라미터를 추가함

```
{{ value }} or text {{ value | pluralize: “(사용자 입력)” }}
```

- add필터는 더하기 연산을 수행, 데이터 타입에 따라 다르게 출력

```
{{ value | add: “2” }} or {{ value | add: 2 }}
```

- 파이썬에서 사용하는 리스트 및 문자열 연산을 사용할 수 있음

```
{{ first | add: second }}
```

```
first = “python”, second = “Django” 의 결과 ‘pythondjango’
first = [1, 2, 3], second = [4, 5, 6] 의 결과 [1, 2, 3, 4, 5, 6]
first = “5”, second = “10” 의 결과 15
```

4.3.3 템플릿 태그

- 템플릿 태그는 아래와 같은 형식을 가지며, 템플릿 로직을 제어하는 역할을 수행
- 태그의 종류에 따라 시작 태그와 끝 태그 둘 다 있는 태그도 있다.

```
{% tag %} or {% startag %} … {% endtag %}
```

- {% for %} 태그
  - 항목들을 순회하면서 변수 또는 변수의 속성을 출력, for 태그의 경우 끝 태그가 있다.

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

변수명	설명
forloop.counter	현재까지 루프를 실행한 루프 카운트(1부터 카운트함)
forloop.counter0	현재까지 루프를 실행한 루프 카운트(0부터 카운트함)
forloop.revcounter	루프 끝에서 현재가 몇 번째인지 카운트한 숫자(1부터 카운트함)
forloop.revcounter0	루프 끝에서 현재가 몇 번째인지 카운트한 숫자(0부터 카운트함)
forloop.first	루프에서 첫 번째 실행이면 True 값을 가짐
forloop.last	루프에서 마지막 실행이면 True 값을 가짐
forloop.parentloop	중첩된 루프에서 현재의 루프 바로 상위의 루프를 의미함

표 **4-1** for 태그에 사용되는 변수들

4.3.3 템플릿 태그

{% if %} 태그

- if 제어문을 통해 로직 제어

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list | length }}
{% elif athlete_in locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

- 태그와 필터 연산자를 함께 사용 가능, 대부분의 필터가 string을 반환(length는 예외)

```
{% if athlete_list | length > 1 %}
```

- 아래와 같은 연산자를 사용할 수 있음

```
and, or, not, and not, ==, !=, <, >, <=, >=, in, not in
```

4.3.3 템플릿 태그

- {% csrf_token %} 태그
  - POST방식의 <form>을 사용하는 템플릿 코드에서는 CSRF공격을 방지하기 위해 {% csrf_token %} 태그를 사용
  - <form> 엘리먼트의 다음 줄에 넣어주면 됨

```
<form action=“ ” method = “POST”>
{% csrf_token %}

…(생략)…

</form>
```

Note_ CSRF공격이란?

Cross-Site Request Forgery는 사이트간 요청 위조 공격이라고도 표현한다. 웹 사이트의 취약점을 공격하는 방식 중 하나로 특정 웹 사이트에서 이미 인증을 받은 사용자를 이용하여 공격을 시도한다. **강의자료 ch03_part02 P.67 참조**

4.3.3 템플릿 태그

{% url %} 태그

- URLconf를 참조하는 URL을 만드는 기능을 수행

```
<form action = "{% url ‘polls:vote’ question.id %}" method = "POST">
```

- 태그의 주 목적은 소스코드에 URL을 하드 코딩하는 것을 방지

```
<form action = "/polls/3/vote/" method = "POST">
```

- /polls/라는 URL을 /blog/로 변경 해야 하는 경우에 URLconf뿐만 아니라 해당 URL을 사용하는 모든 HTML코드 수정
- /3/이라는 숫자는 런타임에 따라 결정되기 때문에 변수처리가 필요, 직접 변경할 경우 마찬가지로 모든 HTML코드에서 수정

```
{% url 'namespace:view-name' arg1 arg2 %}
```

- namespace : app_name 변수에 정의한 이름 공간
- view-name : urls.py파일에서 정의한 URL패턴
- argN : URL 스트링의 Path Converter부분에 들어갈 인자로 없을 수도 있고 여러 개 인경우는 빈칸으로 구분

4.3.3 템플릿 태그

- {% with %} 태그
  - 템플릿 안에서 특정 변수에 값을 저장해 두고 재사용 할 수 있도록 하는 기능

```
{% with total=business.employees.count %}
    {{ total }} employees{{ total | pluralize }}
{% endwith %}
```

  - 주의사항
    - total=business… 변수 사용시 파이썬 문법과 같이 변수 사이에 =의 앞과 뒤에 공백사용 금지
    - .count 함수는 ORM객체의 경우 내부에서 .count함수 지원, 일반 리스트 또는 딕셔너리 자료형은 지원 X

- {% load %}태그
- 사용자 정의 태그 및 필터를 로딩함, 태그 및 필터를 개발자가 필요에 따라 만들어서 사용 가능

```
{% load somelibrary package.otherlibrary %}
{% load foo bar from somelibrary %}
```

4.3.4 템플릿 주석

- 첫번째는 한 줄 주석문으로, 아래처럼 {# #} 형식을 따름. 한 문장의 전부 또는 일부를 주석 처 리하는 방법

```
{# greeting #}hello
```

- {# #} 주석문 내에 템플릿 코드가 들어있어도 정상적으로 주석 처리됨

```
{# {% if foo %}bar{% else %}foobar #}
```

- 여러 줄의 주석문은 아래처럼 {% comment %} 태그를 사용

```
{% comment “Optional note” %}
<p>Commented out text here</p>
{% endcomment %}
```

4.3.5 HTML 이스케이프

- Django 템플릿 시스템은 기본적으로 출력되는 모든 변수에 대해 자동 이스케이프를 수행
  - 예를 들어 변수 {{ user_input }}에 다음 데이터가 들어 있을 때

```
<script>alert("XSS Attack!")</script>
```

  - 장고는 HTML에 사용되는 의미있는 특수 문자를 엔티티로 변환 하여 출력한다.

```
&lt;script&gt;alert(&quot;XSS Attack!&quot;)&lt;/script&gt;
```

사용자 화면에는 정상적으로 출력되어 보여진다.
HTML의 기능만 제거됨

- Django의 이스케이프 기능은 웹 보안의 중요한 요소로, XSS(Cross Site Scripting) 공격을 방지하기 위해 템플릿 렌더링 시 자동으로 데이터를 HTML-safe(안전하게 출력가능한 문자열) 변환 처리함

4.3.5 HTML 이스케이프

- 만약 아래 데이터를 엔티티로 변환하지 않고 그대로 사용한다면?

```
<script>alert("XSS Attack!")</script>
```

  - <script>에서 <, >는 태그로 인식
  - 브라우저는 태그의 시작과 끝으로 인식하기 때문에 브라우저가 자바스크립트를 실행 (XSS공격)
  - 사용자의 의도와는 다르게 경고 스크립트가 작동된다.

- 사용자가 입력한 데이터를 그대로 렌더링하는 것은 위험, 장고는 위와 같은 결과를 방지하기 위해서 자동 이스케이프 기능을 제공.

문자	의미	HTML 엔티티	출력 결과
<	태그 시작	<	<
>	태그 종료	>	>
&	엠퍼샌드	&amp;	&
"	큰 따옴표	&quot;	"
'	작은 따옴표	' 또는 '	'

4.3.5 HTML 이스케이프

- HTML 이스케이프 기능을 비활성화 하는 방법
  - 첫번째, safe 필터 사용

```
This will not be escaped : {{ data | safe }}
```

  - 두번째, {% autoescape %} 태그를 사용

```
{% autoescape off %}
Hello {{ name }}
{% endautoescape %}
```

- Text로 처리된 것을 확인 하는 방법 :
  - 브라우저에서 F12 -> Source -> page -> 페이지 선택
  - 또는 결과 화면에서 Ctrl + U

4.3.6 템플릿 상속

- 템플릿 상속을 통해서 템플릿 코드를 재사용할 수 있고 사이트의 룩 앤 필을 일관성 있게 보여줄 수 있다.
- 부모 템플릿에 템플릿의 뼈대를 만들고 {% block %} 태그를 통해 상속할 부분을 지정

부모 템플릿 templates₩temp₩base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="/static/css/style.css">
    <title>{% block title %}Base site{% endblock %}</title>
</head>
<body>
    <div id="sidebar">
        {% block sidebar %}
            <ul>
                <li><a href="/tempfilter/">Filter</a></li>
                <li><a href="/temptag/">Tag</a></li>
            </ul>
        {% endblock %}
    </div>
    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

statis폴더를 전역으로 사용할 경우 settings.py에서

STATIC_URL = 'static/'
STATICFILES_DIRS = [BASE_DIR / 'static'] 추가

STATIC_URL = 'static/' 만 사용할 경우 App하위 폴더의 static 으로 인식

4.3.6 템플릿 상속

- 자식 템플릿에서 상속받기 위해 {% extends %} 태그를 사용한다.

자식 템플릿 templates₩temp₩base.html

```
{% extends "temps/base.html" %}
{% block title %}Template Tag Test size{% endblock %}
{% block content %}
    <h3>{{ name }}</h3>

    {% for skill in skills %}
       <p>{{ forloop.counter0 }} : {{ skill.field }} - {{ skill.level }}</p>
    {% endfor %}

    <ul>
        {% for lang in languages %}
        <li>{{ forloop.counter }} : {{ lang }}</li>
        {% endfor %}
    </ul>

    {% if 'Django' in languages %}
        <p>Django makes it easier to build better web apps more quickly and with less code.</p>
    {% else %}
        <p>Let's learn Django framework!</p>
    {% endif %}

    … (생략) …

{% endblock %}
```

title 및 content는 자식 템플릿에서 오버라이딩하여 사용

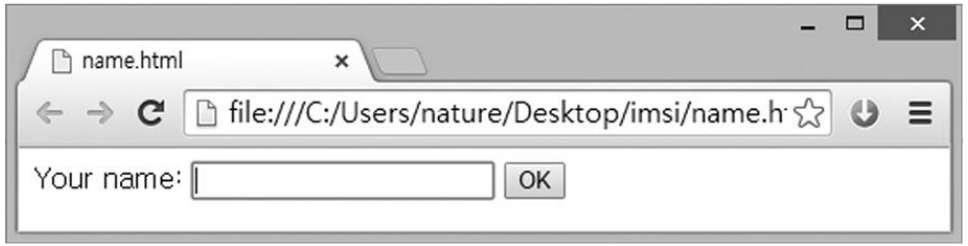
Sidebar는 오버라이딩 하지않고 그대로 부모 템플릿을 사용

4.4.1 HTML에서의 폼

- 우리는 웹 사이트를 개발할 때 사용자로부터 입력을 받기 위해서 폼을 사용
- HTML로 표현 하면 폼은 <form>...</form> 사이에 있는 엘리먼트들의 집합.
- 폼 데이터는 텍스트를 입력할 수도 있고, 항목을 선택(radio, checkbox 등...) 할 수도 있음

<form 태그 예시>

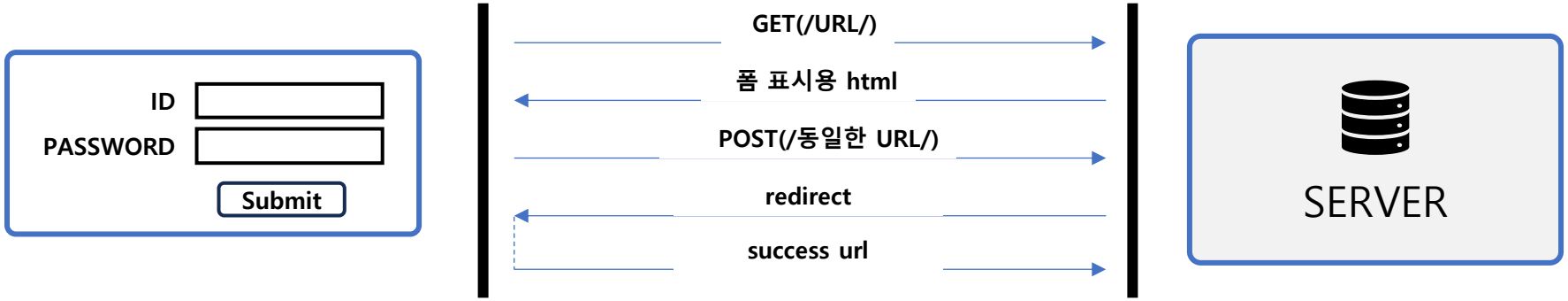
```
<form action="/url/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name">
    <input> type="submit" value="OK">
</form>
```



- 폼에서 사용할 수 있는 HTTP 메소드는 GET과 POST만 사용
  - GET요청 : 조회, POST요청 : 생성, 추가, 변경

4.4.2 장고의 폼 기능

- 폼 처리 단계 요약



구분	처리단계	상세설명
사전 준비	Form 클래스 정의	(1) Form 클래스 정의
GET처리	Form 객체 생성 HTML <form>태그 생성	(2) Form 객체 생성 (3) 템플릿에 처리 위임 (4) 템플릿에서 <form>태그 생성 (5) 클라이언트에게 응답
웹 브라우저	화면에 폼 출력 데이터 입력 후 제출	(6) 화면에 폼 출력 (7) 데이터 입력 후 제출
POST 처리	유효성 검사 후 DB저장 Redirect 응답 (새로운 요청과 응답 처리)	(8) 유효성 검사 (9) 데이터 처리 (10) Redirect 응답 (11) 새로운 요청과 응답 처리

4.4.2 장고의 폼 기능

- (1) Form 클래스 정의
  - Form 클래스를 통해 폼처리를 하므로 Form클래스를 사전에 정의, forms.py 파일에 코딩

```
class NameForm(forms.Form):
    your_name = forms.CharField(label="Your name", max_length=100)
```

- (2) Form 객체 생성
  - 장고의 폼 처리는 GET요청을 수신한 시점에 시작, views.py파일의 뷰에서 폼 변수에 폼 객체를 할당

```
form = NameForm() 또는 form = NameForm(data)
```

- (3) 템플릿에 처리 위임
  - 템플릿 엔진에 <form> 태그와 관련된 HTML을 생성하도록 요청

```
render(request, 'name.html', {'form':form})
```

4.4.2 장고의 폼 기능

(4) 템플릿에서 <form>태그 생성

- 개발자가 아래와 같은 코드를 작성하면 템플릿 엔진은 자동으로 <form>태그와 관련된 HTML 생성

```
<form action="/url/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="OK">
</form>
```

- 변환 후 코드

```
<form action="/url/" method="post">
    <input type="hidden" value="xxxx1234xxxxxxxx">
    <label for="your_name>Your name: </label>
    <input id="your_name" type="text" name="your_name" maxlength="100">
    <input type="submit" value="OK">
</form>
```

form 기능에 의해서 자동으로 입력된 csrf_token

4.4.2 장고의 폼 기능

- (5) 클라이언트에게 응답
  - 템플릿 HTML 파일이 완성되면 장고는 클라이언트에게 HTML을 전송, HTML내에는 <form>태그 포함
- (6) 화면에 폼 출력
- (7) 데이터 입력 후 제출
  - POST방식으로 데이터를 서버로 전송
- (8) 유효성 검사
  - is_valid() 함수로 유효성 검사를 수행 (수신 데이터의 데이터 타입)
- (9) 데이터 처리
  - 유효성 검사를 통과한 데이터는 cleaned_data 변수에 담기고 뷰는 이 데이터를 사용하여 로직 처리
- (10) Redirect응답
- (11) 새로운 요청과 응답처리

4.4.3 Form 클래스로 폼 정의

- 장고는 데이터베이스 테이블을 Model 클래스로 매핑해서 처리하는 것과 유사한 방식으로 HTML의 <form>태그를 Form 클래스로 매핑해서 처리
- 모든 폼 클래스는 Django.forms.Form의 하위 클래스로 정의

```
from django import forms
from .validators import validate_com

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField(validators=[validate_com])
    cc_myself = forms.BooleanField(required=False)
```

- validate_com은 이메일 형식 유효성 검사 속성
- 필드 타입마다 디폴트 위젯이 미리 정의되어 있으며, 변경 가능함

4.4.4 ModelFrom 클래스로 폼 생성

- 폼 생성을 ModelFrom클래스를 사용하여 Model 클래스와 연계하여 처리 할 수 있다.

```
from django.db import models
from .validators import validate_com


class ContactForm(models.Model):
    subject = models.CharField(max_length=100)
    message = models.CharField(max_length=200)
    sender = models.EmailField(validators=[validate_com])
    cc_myself = models.BooleanField(blank=True, null=True)
```

- 내부 클래스 Meta를 이용하여 폼의 설정 정보를 구성

```
from django.forms import ModelForm
from .models import Contact

class ContactForm(ModelForm):
    class Meta:
        model = Contact
        fields = ['subject', 'message', 'sender']
        # exclude = ['cc_myself']
        # fields = '__all__'
```

4.4.5 뷰에서 폼 클래스 처리

- 폼을 처리하는 뷰 기능은 GET방식과 POST방식을 하나의 뷰로 통합하여 폼을 처리하는 것을 권장함

뷰에서 폼클래스를 처리하는 방식

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from Names.forms import NameForm

def get_name(request):
    if request.method == "POST":
        form = NameForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['name']
            return HttpResponseRedirect('/thanks/')
    else:
        form = NameForm()

    return render(request, "name.html", {"form":form})
```

GET요청이 도착하면 빈 폼 객체를 생성

폼 데이터가 유효하면 cleaned_data로 복사

새로운 URL로 리다이렉션
(폼 생성, 저장 완료 메시지)

GET요청에 따라, 빈 폼 객체를 렌더링

4.4.5 폼 클래스를 템플릿으로 변환

- {{ form }} 구문은 HTML의 <label>과 <input> 엘리먼트 쌍으로 렌더링 되며, 이러한 변환 시 {{ form }} 이외에도 몇가지 옵션이 더 있다.
  - {{ form.as_table }} : <tr> 태그로 감싸서 테이블 셀로 렌더링, {{form}}과 동일함
  - {{ form.as_p }} : <p> 태그로 감싸도록 렌더링
  - {{ form.as_ul }} : <li> 태그로 감싸도록 렌더링
  - {{ form.as_div }} : 태그로 감싸도록 렌더링

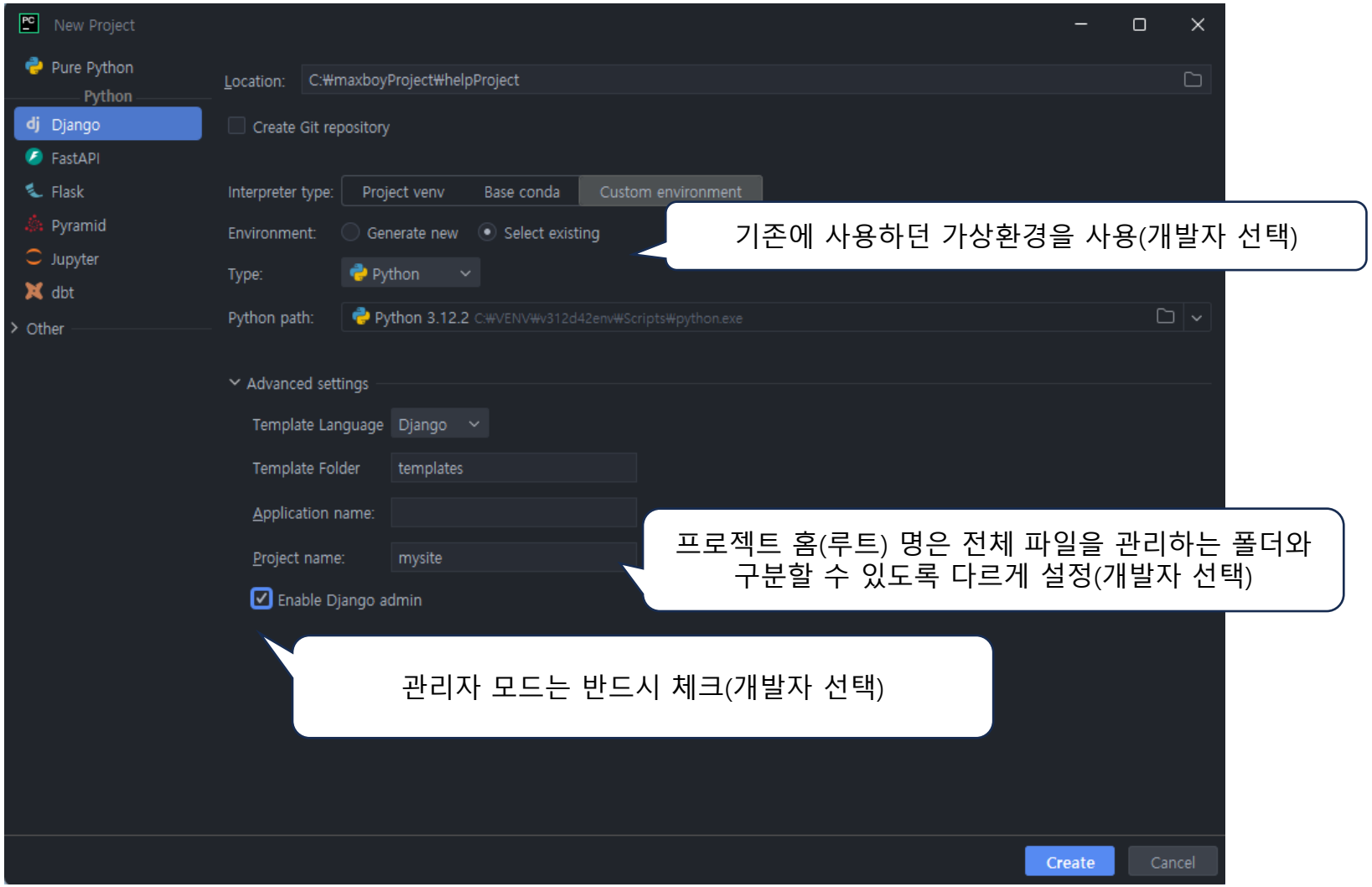
```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

```
<p><label for="id_subject">Subject:</label>
   <input id="id_subject" type="text" name="subject" maxlength="100"></p>
<p><label for="id_message">Message:</label>
   <input id="id_message" type="text" name="message"></p>
<p><label for="id_sender">Sender:</label>
   <input id="id_sender" type="email" name="sender"></p>
<p><label for="id_cc_myself">CC myself:</label>
   <input id="id_cc_myself" type=checkbox" name="cc_myself"></p>
```

- <input> <label>태그의 id는 각 필드의 필드명을 사용하여 id_필드명 형식으로 자동으로 생성

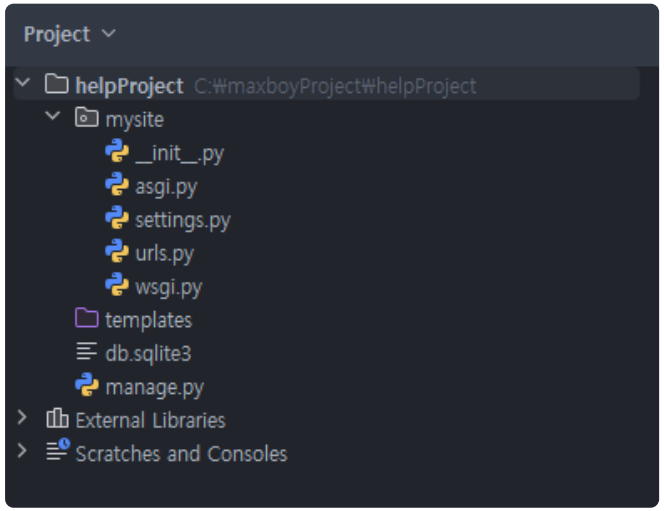
4.4+.1 새 프로젝트 생성 및 테스트

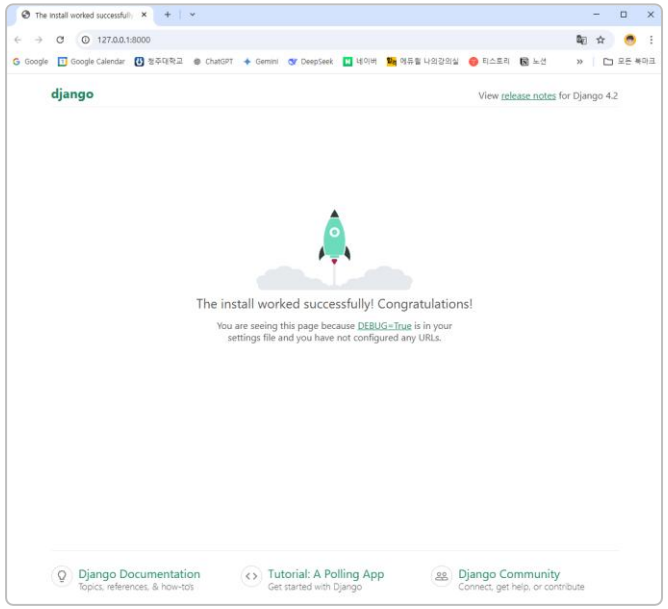
- helpProject (변경가능) 로 프로젝트 전체 파일을 관리하는 폴더를 생성한다.



4.4+.1 새 프로젝트 생성 및 테스트

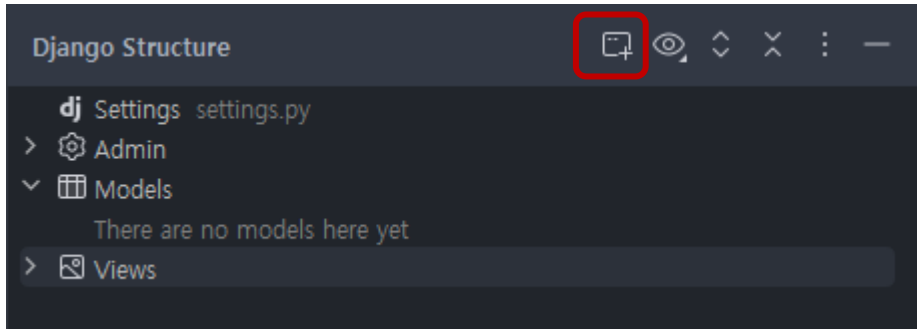
- 생성된 프로젝트 뼈대 확인
- mysite.settings.py 파일의 세팅 내용을 확인
  - language_code 설정 (ko-kr)
  - timezone 설정 (Asia/Seoul)
  - 기타 필요 사항 확인
- 127.0.0.1:8000 에서 서버 동작 상태 확인



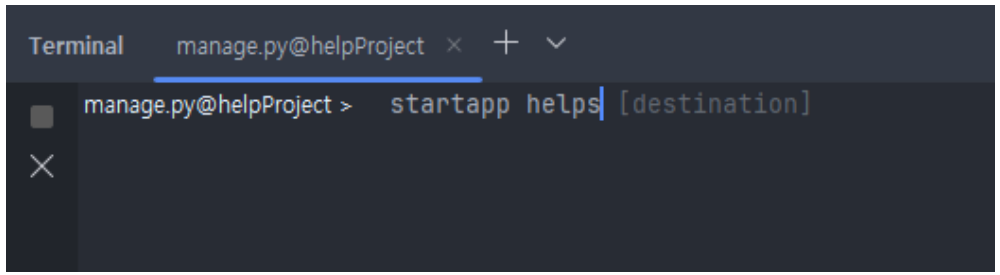


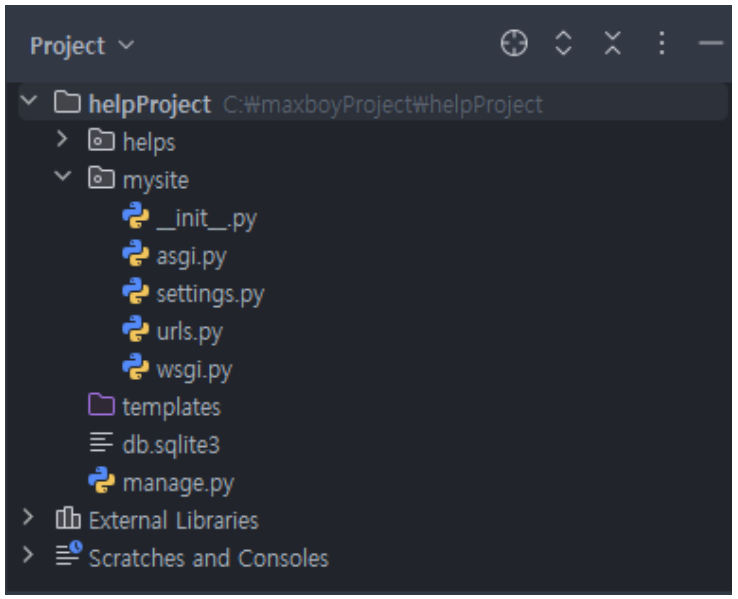
4.4+.2 helps 어플리케이션 등록

- 아래와 같이 Django Structure에서 new Django App 등록 버튼 클릭



- helps 이름으로 App을 등록하면 우측 화면과 같이 장고프로젝트 구조가 생성





4.4+.3 데이터 테이블 생성

- Help 테이블 생성

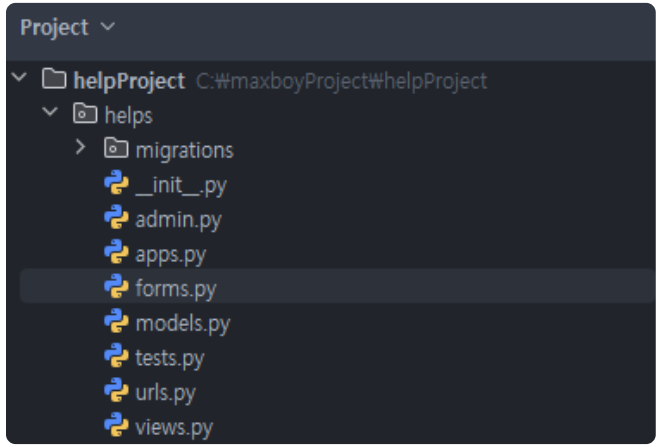
helpProject₩helps₩models.py

```
from django.db import models

# Create your models here.
class Help(models.Model):
    subject = models.CharField(max_length=200)
    name = models.CharField(max_length=100)
    email = models.EmailField()
    message = models.TextField()

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

- 아래 명령으로 DB에 데이터 적용
  - python manage.py makemigrations
  - python manage.py migrate

4.4+.4 Form 클래스로 HelpForm 정의

- helps 폴더에 form.py 파일 생성



- forms.Form을 상속하는 클래스로 정의

helpProject₩helps₩forms.py

```
from django import forms

class HelpForm(forms.Form):
    subject = forms.CharField(label="제목", max_length=200)
    name = forms.CharField(label="이름", max_length=100)
    email = forms.EmailField(label="이메일")
    message = forms.CharField(label="메시지", widget=forms.Textarea)
```

4.4+.5 URLconf 수정

- mysite의 urls.py 수정

helpProject₩msite₩urls.py

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

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

- helps의 urls.py 수정

helpProject₩helps₩urls.py

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


app_name = 'helps'
urlpatterns = [
    path('', views.help_view, name='help_view'),
]
```

4.4+.6 뷰 함수 help_view( ) 및 템플릿 코딩

- helps에 뷰 함수 수정

helpProject₩helps₩views.py

```
from django.shortcuts import render
from helps.forms import HelpForm

# Create your views here.
def help_view(request):
    if request.method == "POST":
        form = HelpForm(request.POST)
        if form.is_valid():
            subject = form.cleaned_data['subject']
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']

            return render(request,"helps/success.html", {"name":name})
    else:
        form = HelpForm()

    return render(request,"helps/helps.html", {"form":form})
```

4.4+.6 뷰 함수 help_view( ) 및 템플릿 코딩

- helps.html 작성

templates₩helps₩helps.html

```
<form action="{% url 'helps:help_view' %}" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="OK">
</form>
```

- success.html 작성

templates₩helps₩success.html

```
<h1>문의가 성공적으로 접수되었습니다. </h1>
<p>{{ name }}님, 감사합니다.</p>
```

4.4+.7 DB저장을 위한 뷰 함수 help_view( ) 수정

helps₩views.py

```
from django.shortcuts import render
from helps.forms import HelpForm
from .models import Help

def help_view(request):
    if request.method == "POST":
        form = HelpForm(request.POST)
        if form.is_valid():
            subject = form.cleaned_data['subject']
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']

            Help.objects.create(
                subject=form.cleaned_data['subject'],
                name=form.cleaned_data['name'],
                email=form.cleaned_data['email'],
                message=form.cleaned_data['message'],
            )

            return render(request,"helps/success.html", {"name":name})
    else:
        form = HelpForm()

    return render(request,"helps/helps.html", {"form":form})
```

Help클래스를 호출

Help클래스를 이용하여 DB에 직접 생성
form.save() 메서드가 Form클래스에서 지원 되지 않음

4.4+.8 ModelForm 클래스로 폼 정의

helps₩views.py

```
from django.shortcuts import render
from helps.forms import HelpForm
from .models import Help

def help_view(request):
    if request.method == "POST":
        form = HelpForm(request.POST)
        if form.is_valid():
            subject = form.cleaned_data['subject']
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']

            Help.objects.create(
                subject=form.cleaned_data['subject'],
                name=form.cleaned_data['name'],
                email=form.cleaned_data['email'],
                message=form.cleaned_data['message'],
            )

            return render(request,"helps/success.html", {"name":name})
    else:
        form = HelpForm()

    return render(request,"helps/helps.html", {"form":form})
```

Help클래스를 호출해서 DB에 직접 생성

4.4+.8 ModelForm 클래스로 폼 정의

- forms.py를 ModelForm클래스로 변경

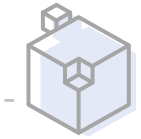
helps₩forms.py

```
# from django import forms
#
# class HelpForm(forms.Form):
#     subject = forms.CharField(label="제목", max_length=200)
#     name = forms.CharField(label="이름", max_length=100)
#     email = forms.EmailField(label="이메일")
#     message = forms.CharField(label="메시지", widget=forms.Textarea)

from django.forms import ModelForm
from .models import Help

class HelpForm(ModelForm):
    class Meta:
        model = Help
        fields = ['subject', 'name', 'email', 'message']
        labels = {
            'subject': '제목',
            'name': '이름',
            'email': '이메일',
            'message': '메시지',
        }
```

4.4+.8 ModelForm 클래스로 폼 정의

helps₩views.py

```
from django.shortcuts import render
from helps.forms import HelpForm
from .models import Help

def help_view(request):
    if request.method == "POST":
        form = HelpForm(request.POST)
        if form.is_valid():
            subject = form.cleaned_data['subject']
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']

            # Form클래스를 상속했을 때
            # Help.objects.create(
            #     subject=form.cleaned_data['subject'],
            #     name=form.cleaned_data['name'],
            #     email=form.cleaned_data['email'],
            #     message=form.cleaned_data['message'],
            # )
            form.save() #폼 클래스가 모델폼을 상속했을 때 사용가능
            return render(request,"helps/success.html", {"name":name})
    else:
        form = HelpForm()

    return render(request,"helps/helps.html", {"form":form})
```



class Meta: 란?

- 장고에서 Meta클래스는 메타정보(부가정보)를 설정할 수 있는 내부 클래스이다. 외부 클래스에 대해 추가 설정이나 연결 정보를 제공한다.

- 클래스 내부에 클래스 구조(메서드와 같은)를 갖는 이유는?
  - Meta 클래스를 외부에서 분리해 두는 것보다 한 클래스 안에 내장되어 있는 것이 가독성과 유지보수 측면에서 유리
  - 추상 메서드처럼 반드시 오버라이딩 해야 하는 것은 아니지만, 장고 내부에서 약속된 패턴으로 설계해서 의도적으로 오버라이딩 가능 한 구조를 제공

4.5 함수형 뷰와 클래스형 뷰

- FBV(Function-Based View)와 CBV(Class-Based View)는 장고에서 웹 요청을 처리하는 두 가지 방식으로 각각의 방식은 장단점이 있으며, 상황에 따라 적절하게 선택하여 사용
- FBV 방식 : 함수로 뷰를 정의
  - 장점 : 간단하고 직관적, 코드의 흐름이 눈에 잘 들어오기 때문에 디버깅이 쉬움
  - 단점 : 복잡한 로직이나 HTTP메서드를 분기 처리 하다 보면 코드가 길어져 유지보수가 어려움

```
from django.http import HttpResponse

def hello_view(request):
    return HttpResponse("Hello, world!")
```

- CBV 방식 : 클래스로 뷰를 정의
  - 장점 : 클래스 내의 메서드를 오버라이딩하여 요청 메서드(GET, POST등)를 처리하며 제네릭 뷰를 활용하여 CRUD을 간단하게 처리
  - 단점 : 처음 접하는 사람은 구조가 생소하여 가독성이 떨어짐, 상속(추상화)구조가 많아 디버깅이 어려움

```
from django.http import HttpResponse
from django.views import View

class HelloView(View):
    def get(self, request):
        return HttpResponse("Hello, class-based world!")
```

4.5.1 클래스형 뷰의 시작점

- 클래스형 뷰를 사용하기에 앞서 우선 URLconf에 함수형 뷰가 아니라 클래스형 뷰를 사용한다는 것으로 표현

클래스형 뷰의 진입 메소드 as_view() myapp₩urls.py

```
from Django.urls import path
from myapp.views import MyView

urlpatterns = [
    path('about/', MyView.as_view()),
]
```

- as_view() 메서드
  - URLconf는 요청과 관련된 파라미터들을 클래스가 아니라 함수에 전달, 클래스형 뷰를 사용하기 위해서는 클래스로 진입하기 위한 as_view() 클래스 메서드를 제공, 진입 메소드
- dispatch() 메서드
  - 생성된 인스턴스에서 요청확인 (GET 또는 POST)
  - GET요청 시 get()메서드 호출, POST요청 시 post()메서도 호출

4.5.1 클래스형 뷰의 시작점

- 클래스형 뷰는 함수형 뷰와 동일하게 views.py파일에 코딩

클래스형 뷰의 MyView 클래스 정의 myapp₩views.py

```
from django.http import HttpResponse
from django.views.generic import View


class MyView(View):
    def get(self, request):
        # 뷰 로직 작성
        return HttpResponse('result')
```

- as_view() 및 dispatch()메소드는 View 클래스 메서드에 정의 되어 있다.
- MyView 클래스는 View를 상속하면 별도로 메서드를 정의할 필요 없음

4.5.2 클래스형 뷰의 장점 – 효율적인 메소드 구분

- 클래스형 뷰의 장점
  - GET, POST 등의 HTTP 메소드에 따른 처리 기능구현 시 if문을 사용하지 않고 메서드로 처리 가능, 코드의 구조가 명확해짐
  - 다중 상속과 같은 객체 지향 기술이 가능하므로, 클래스형 제네릭 뷰 및 믹스인 클래스 등을 사용할 수 있고, 이는 코드의 재사용성이나 개발 생산성을 획기적으로 높여 줌

함수형 뷰로 HTTP GET 및 POST 메소드 코딩 myapp₩views.py

```
from django.http import HttpResponse

def my_view(request):
    if request.method == "GET":
        # GET 방식 처리
        return HttpResponse("GET 요청 처리 결과")
    else:
        # POST 방식 처리
        return HttpResponse("POST 요청 처리 결과")
```

클래스형 뷰로 HTTP GET 및 POST 메소드 코딩 myapp₩views.py

```
from django.http import HttpResponse
from django.views.generic import View

class MyView(View):
    def get(self, request):
        # GET 방식 처리
        return HttpResponse("GET 요청 처리 결과")

    def post(self, request):
        # POST 방식 처리
        return HttpResponse("POST 요청 처리 결과")
```

오버라이딩

4.5.3 클래스형 뷰의 장점 – 상속 기능 가능

- 제네릭 뷰란?
  - 뷰 개발 과정에서 공통적으로 사용할 수 있는 기능을 추상화하고 이를 미리 만들어 장고에서 기본으로 제공하는 클래스형 뷰를 의미한다. 클래스형 뷰 대부분은 이 제네릭 뷰를 상속받는다.

클래스형 뷰 작성 – TemplateView 상속 some_app₩urls.py

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

urlpatterns = [
    path('about/', AboutView.as_view()),
]
```

some_app₩views.py

```
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

오버라이딩

  - AboutView클래스는 템플릿 파일만 지정 함으로써 템플릿 시스템에 넘겨줄 컨텍스트 변수를 구성하고 렌더링 결과를 반환하는 일은 TemplateView 내부(메서드 : get_context_data(**kwargs))에서 처리된다.

4.5.3 클래스형 뷰의 장점 – 상속 기능 가능

- 아래와 같이 urls.py파일에 TemplateView 클래스와 메서드에 about.html 템플릿 파일 속성을 지정하면 views.py 파일에 클래스형 뷰를 작성하지 않아도 동작 하도록 구현 할 수 있다.

클래스형 뷰 작성 – URLconf에 TemplateView 지정 some_app₩urls.py

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

urlpatterns = [
    path('about/', TemplateView.as_view(template_name="about.html")),
]
```

4.5.4 클래스형 제네릭 뷰

- Base View : 뷰 클래스를 생성하고 다른 제네릭 뷰의 부모 클래스를 제공하는 기본 제네릭 뷰
- Generic Display View : 객체의 리스트를 보여 주거나 특정 객체의 상세 정보를 보여줌
- Generic Edit View : 폼을 통해 객체를 생성, 수정, 삭제하는 기능을 제공
- Generic Date View : 날짜 기반 객체를 연/월/일 단위로 구분해서 보여줌

표 **4-3** 장고의 제네릭 뷰 리스트(일부)

제네릭 뷰 분류	제네릭 뷰 이름	뷰의 기능 또는 역할
Base View	View	가장 기본이 되는 최상위 제너릭 뷰, 다른 모든 제네릭 뷰는 하위 클래스
	TemplateView	템플릿이 주어지면 해당 템플릿을 렌더링
	RedirectView	URL이 주어지면 해당 URL로 리다이렉트
Generic Display View	Listview	조건에 맞는 여러 개의 객체 리스트를 출력
	DetailView	객체 하나에 대한 상세한 정보를 출력
Generic Edit View	FormView	폼을 보여 주고 폼의 데이터를 처리
	CreateView	폼을 보여 주고 폼 데이터로 객체를 생성
	UpdateView	폼을 보여 주고 폼 데이터로 객체를 수정
	DeleteView	폼을 보여 주고 폼 데이터로 객체를 삭제
Generic Date View	ArchiveIndexView	조건에 맞는 여러 개의 객체 및 객체들에 관한 날짜 정보 출력
	YearArchiveView	연도가 주어지면 그 연도에 해당하는 객체들을 출력
	MonthArchiveView	연, 월이 주어지면 그에 해당하는 객체들을 출력
	WeekArchiveView	연도와 주차(Week)가 주어지면 그에 해당하는 객체들을 출력
	DayArchiveView	연, 월,일이 주어지면 그 날짜에 해당하는 객체들을 출력
	TodayArchiveView	오늘 날짜에 해당하는 객체들을 출력
	DateDetailView	연, 월, 일, 기본키(또는 슬러그)가 주어지면 그에 해당하는 객체의 상세정보 출력

4.5.5 클래스형 뷰에서 폼 처리

- 앞에 Help예제에서 폼을 처리하는 과정을 살펴보았다. GET, POST 메서드를 구분하여 폼 처리를 진행하였으며, POST데이터를 처리하는 과정에서 유효성 검사를 포함하여 폼 처리과정을 3단계로 구분된다.

  - **최초의 GET** : 사용자에게 처음으로 폼(빈 폼 또는 초기 데이터로 채워진 폼)을 보여줌
  - **유효한 데이터를 가진 POST** : 데이터를 처리, 처리 후에는 리다이렉트로 응답을 처리
  - **유효하지 않은 데이터를 가진 POST** : 보통은 폼에 데이터를 입력 할 수 있도록 다시 출력

- Help 애플리케이션 예제를 이용하여 폼 처리 로직을 2가지 클래스형 뷰로 변경해 본다.
  - 가장 상위 클래스인 View 제네릭 뷰를 상속받아 구현
  - 폼 처리용 제네릭 뷰인 FormView를 상속 받아 구현

4.5.5 클래스형 뷰에서 폼 처리

- 아래는 Help 애플리케이션 예제에서 함수형 뷰로 폼을 처리하는 코드이다.

함수형 뷰로 폼을 처리 helps₩views.py

```
from django.shortcuts import render
from helps.forms import HelpForm

def help_view(request):
    if request.method == "POST":
        form = HelpForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['name']
            form.save()
            return render(request,"helps/success.html", {"name":name})
    else:
        form = HelpForm()

    return render(request,"helps/helps.html", {"form":form})
```

- form.is_valid() 메서드는 폼 객체의 모든 필드에 대한 유효성을 검사하며 모든 필드가 유효하다면 is_valid() 메서드는 다음과 같은 2가지일을 처리함
  - True를 반환
  - form.cleaned_data 속성에 폼 데이터를 저장

4.5.5 클래스형 뷰에서 폼 처리

- 함수형 뷰로 처리한 로직을 클래스형 뷰(View 클래스)로 코딩하면 아래와 같다.

클래스형 뷰로 폼을 처리 helps₩views.py

```
from django.shortcuts import render
from django.views.generic import View
from helps.forms import HelpForm

class HelpView(View):
    # noinspection PyMethodMayBeStatic
    def get(self, request):
        form = HelpForm()
        return render(request,"helps/helps.html", {"form":form})

    # noinspection PyMethodMayBeStatic
    def post(self, request):
        form = HelpForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['name']
            form.save()
            return render(request,"helps/success.html", {"name":name})
        return render(request, "helps/helps.html", {"form": form})
```

최초의 GET처리

유효한 데이터를 가진 POST처리

유효하지 않은 데이터를 가진 POST처리

4.5.5 클래스형 뷰에서 폼 처리

- View클래스 사용 시 URLconf 경로 수정

클래스형 뷰로 폼을 처리 – URLconf 수정 helps₩urls.py

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

app_name = 'helps'
urlpatterns = [
    path('', HelpView.as_view(), name='help_view'),
]
```

템플릿 폼 action에 이 패턴명으로 작성해 놓았기 때문에 변경하지 않음

4.5.5 클래스형 뷰에서 폼 처리

- 함수형 뷰로 처리한 로직을 클래스형 뷰(FormView 클래스)로 코딩하면 아래와 같다.

클래스형 뷰로 폼을 처리 helps₩views.py

```
from django.shortcuts import render
from django.views.generic.edit import FormView
from helps.forms import HelpForm

class HelpFormView(FormView):
    form_class = HelpForm
    template_name = "helps/helps.html"

    def form_valid(self, form):
        name = form.cleaned_data['name']
        form.save()
        return render(self.request, "helps/success.html", {"name": name})
```

- form_class 속성 : 사용자에게 보여 줄 폼을 정한 forms.py 파일 내의 클래스명을 지정
- template_name 속성 : 폼을 포함하여 렌더링할 템플릿 파일 이름을 지정
- success_url 속성 : HelpFormView처리가 정상적으로 완료되었을 때 리다이렉트 시킬 URL을 지정
- form_valid() 메서드 : 폼 데이터가 유효한 경우 처리할 로직 코딩, super().form_valid(form)으로 처리하면 success_url경로로 리다이렉션 처리됨

4.5.5 클래스형 뷰에서 폼 처리

- FormView클래스 사용 시 URLconf 경로 수정

클래스형 뷰로 폼을 처리 – URLconf 수정 helps₩urls.py

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

app_name = 'helps'
urlpatterns = [
    path('', HelpFormView.as_view(), name='help_view'),
]
```

템플릿 폼 action에 이 패턴명으로 작성해 놓았기 때문에 변경하지 않음