5) 클래스형 뷰

뷰는 요청을 받아서 응답을 반환해주는 호출 가능한 객체임. 장고에서는 뷰를 함수로도 작성할 수 있고 클래스로도 작성할 수 있음.

함수형 뷰보다 클래스형 뷰가 장점이 많음.

클래스형 뷰를 사용하면 상속과 믹스인 기능을 사용해서 코드를 재사용할 수 있고, 뷰를 체계적으로 구성할 수도 있음.

간단한 경우에는 함수형 뷰로 신속하게 개발

로직이 복잡해지고 프로젝트가 커질수록 클래스형 뷰의 장점이 효과를 발휘할 것

1. 클래스형 뷰를 사용하기 위해 가장 먼저 URLconf에서 함수형 뷰가 아니라 클래스형 뷰를 사용한다는 점을 표시해줘야 함.

예를 들어 MyView라는 클래스형 뷰를 사용한다면 URLconf는 다음과 같은 모습

<클래스형 뷰의 진입 메소드 as\_view()>

#urls.py

from django.urls import path

from myapp.views import MyView

urlpatterns = [

path(‘about/’, MyView.as\_view()),

]

장고의 URL 해석기는 요청과 관련된 파라미터들을 클래스가 아니라 함수에 전달하기 때문에, 클래스형 뷰는 클래스로 진입하기 위한 as\_view() 클래스 메소드 제공

as\_view(): 진입 메소드

클래스의 인스턴스를 생성하고 그 인스턴스의 dispatch() 메소드 호출

dispatch(): 요청을 검사해서 GET, POST 등의 어떤 HTTP 메소드로 요청되었는지를 알아낸 다음, 인스턴스 내에서 해당 이름을 갖는 메소드로 요청을 중계해줌.

만일 해당 메소드가 정의되어 있지 않으면 HttpResponseNotAllowed 익셉션 발생

MyView 클래스 정의

#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()와 dispatch(): 상속받는 View 클래스에 기본적으로 정의되어 있음->따로 정의 필요X

\*\*클래스형 뷰의 장점: 상속 기능

2. 클래스형 뷰의 장점-효율적인 메소드 구분

-GET, POST 등의 HTTP 메소드에 따른 처리 기능을 코딩할 때, IF 함수를 사용하지 않고 메소드명으로 구분할 수 있으므로 코드의 구조가 깔끔해짐

-다중 상속과 같은 객체 지향 기술이 가능하므로 클래스형 제네릭 뷰 및 믹스인 클래스 등을 사용할 수 있고, 이는 코드의 재사용성이나 개발 생산성을 획기적으로 높여줌.

함수형 뷰에서 GET 메소드를 처리하는 로직

from django.http import HttpResponse

def my\_view(request):

if request.method == ‘GET’:

#뷰 로직 작성

return HttpResponse(‘result’)

요청 request 객체의 method 속성 체크 로직이 필요->항상 if 조건의 체크 문장이 필요

클래스형 뷰로 HTTP GET 메소드 코딩

#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’)

클래스형 뷰에서는 HTTP 메소드 이름으로 클래스 내에 메소드를 정의하면 됨. 단, 메소드명은 소문자로.. get(), post(), head()…

클래스형 뷰에는 내부적으로 dispatch() 메소드가 있기 때문에 dispatch() 메소드가 GET, POST 등의 어떤 HTTP 메소드로 요청되었는지를 알아낸 다음 해당 이름을 갖는 메소드로 요청을 중계해주므로 가능

아래 예제에서 HEAD HTTP 메소드는 서점에 방문한 직후에 새롭게 출간된 책이 있는지를 서버에 문의하는 용도(최근 발간 책이 없는데도 책 리스트를 서버로부터 받아온다면 네트워크 대역폭이 낭비되므로 이를 방지하기 위해 HEAD 메소드 사용)

HEAD 요청에 대한 응답은 바디 없이 헤더만 보내주면 됨

<클래스형 뷰로 HTTP HEAD 메소드 코딩>

#views.py

from django.http import HttpResponse

from django.views.generic import ListView

from books.models import Book

class BookListView(ListView):

model = Book

def head(self, \*args, \*\*kwargs):

last\_book = self.get\_queryset().latest(‘publication\_date’)

response = HttpResponse(‘’)

#RFC 1123 date 포맷

response[‘Last-Modified’] = last\_book.publication\_date.strftime(‘%a, %d %b %Y

%H:%M:%S GMT’)

return response

3. 클래스형 뷰의 장점-상속 기능 가능

웹 애플리케이션을 개발하는 경우 공통적으로 반복되는 과정들이 발생

장고에서는 특히 모델, 뷰, 템플릿 개발 과정에서의 이러한 단순 반복 작업을 많이 없애주었는데 제네릭 뷰가 바로 뷰 개발 과정에서 개발자의 단순 반복 작업을 덜어주는 기능.

즉 제네릭 뷰란 뷰 개발 과정에서 공통적으로 사용할 수 있는 기능들을 추상화하고 이를 장고에서 미리 만들어 기본적으로 제공해주는 클래스형 뷰를 말함.

따라서 클래스형 뷰를 작성할 때는 제네릭 뷰를 상속받아 작성하는 것이 대부분

뷰가 클래스이기 때문에 상속 가능, 우리가 작성한 클래스형 뷰를 상속받아서 또 다른 클래스형 뷰를 작성할 수 있으므로 확장성도 증가함.

아래 예제는 만일 도움말 화면처럼 /about/이라는 URL로 웹 요청이 들어오면, 단순하게 about.html 템플릿을 보여준다고 가정하고 코딩한 것

<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”

urls.py 파일에서는 클래스형 뷰의 as\_view() 메소드를 호출하도록 하였음

views.py 파일의 소스를 보면 단 2줄로 로직을 완성. (TemplateView라는 제네릭 뷰를 상속받아 사용하고 있기 때문)

요청 request 객체를 분석하고, 템플릿 시스템에 넘겨줄 컨텍스트 변수를 구성하는 것은 모두 TemplateView 제네릭 뷰에서 처리. 우리는 간단하게 about.html 템플릿 파일을 사용하라고만 알려주면 됨.

조금 다른 방식으로 제네릭 뷰 사용 가능. urls.py 파일에서 about.html 템플릿 파일을 지정해주면 views.py 파일에 클래스형 뷰를 작성하는 것도 불필요해짐

<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”)),

]

매우매우 간단함->장고의 강력한 기능

TemplateView는 뷰에 특별한 로직이 없고 URL에 맞춰 해당 템플릿 파일의 내용만 보여줄 때 사용하는 제네릭 뷰

\*\*위 두 개의 예제에는 상속 기능과 더불어 중요한 기능으로 손꼽히는 오버라이딩 기능이 포함되어 있다는 것임. 예를 들어 template\_name은 TemplateView 클래스에 정의되어 있는 클래스 속성임. 이런 클래스 속성을 필요에 따라 오버라이딩하여 사용할 수 있다는 것임. 위 예제에서는 하위 클래스인 AboutView 클래스에서 오버라이딩할 수도 있고, as\_view() 메소드를 통해서도 오버라이딩할 수 있다는 것을 보여줌.

4) 클래스형 제네릭 뷰

장고에서는 URL 패턴으로부터 파라미터를 추출하고 그 파라미터로 데이터베이스를 검색하여 해당 데이터를 템플릿 시스템에서 렌더링하는 기능처럼, 웹 프로그램 개발 시 공통적으로 사용할 수 있는 로직을 이미 개발해 놓고 기본 클래스로 제공하고 있기 때문에 이를 상속받아 사용하면 됨. 장고에서는 이렇게 공통된 로직을 미리 개발해 놓고 제공하는 뷰를 제네릭 뷰라고 부름. 제네릭 뷰는 클래스형 뷰로 구성되어 있음.

장고에서 제공하는 제네릭 뷰

-Base View: 뷰 클래스를 생성하고 다른 제네릭 뷰의 부모 클래스를 제공하는 기본 제네릭 뷰

-Generic Display View: 객체의 리스트를 보여주거나 특정 객체의 상세 정보를 보여줌

-Generic Edit View: 폼을 통해 객체를 생성, 수정, 삭제하는 기능 제공

-Generic Date View: 날짜 기반 객체의 연/월/일 페이지로 구분해서 보여줌

장고의 제네릭 뷰 리스트(일부)

|  |  |  |
| --- | --- | --- |
| 제네릭 뷰 분류 | 제네릭 뷰 이름 | 뷰의 기능 또는 역할 |
| Base View | View  TemplateView  RedirectView | 가장 기본이 되는 최상위 제네릭 뷰  다른 모든 제네릭 뷰는 View의 하위 클래스  템플릿이 주어지면 해당 템플릿을 렌더링해줌  URL이 주어지면 해당 URL로 리다이렉트시켜줌 |
| Generic Display View | ListView  DetailView | 조건에 맞는 여러 개의 객체를 보여줌  객체 하나에 대한 상세한 정보를 보여줌 |
| Generic Edit View | FormView  CreateView  UpdateView  DeleteView | 폼이 주어지면 해당 폼을 보여줌.  객체를 생성하는 폼을 보여줌.  기존 객체를 수정하는 폼을 보여줌.  기존 객체를 삭제하는 폼을 보여줌. |
| Generic Date View | ArchiveIndexView  YearArchiveView  MonthArchiveView  WeekArchiveView  DayArchiveView  TodayArchiveView  DateDetailView | 조건에 맞는 여러 개의 객체 및 그 객체들에 대한 날짜 정보 보여줌  연도가 주어지면 그 연도에 해당하는 객체 보여줌  연, 월이 주어지면 그에 해당하는 객체들을 보여줌  연도와 주차가 주어지면 그에 해당하는 객체들을 보여줌  연, 월, 일이 주어지면 그 날짜에 해당하는 객체들을 보여줌  오늘 날짜에 해당하는 객체들을 보여줌  연, 월, 일, 기본키(또는 슬러그)가 주어지면 그에 해당하는 특정 객체 하나에 대한 상세 정보 보여줌 |

제네릭 뷰에 대한 전체 리스트 및 자세한 설명은

<https://docs.djangoproject.com/en/2.1/ref/class-based-views/>

5) 클래스형 뷰에서 폼 처리

폼 처리 과정

-최초의 GET: 사용자에게 처음으로 폼(빈 폼 또는 초기 데이터로 채워진 폼)을 보여줌

-유효한 데이터를 가진 POST: 데이터를 처리함, 주로 리다이렉트 처리됨

-유효하지 않은 데이터를 가진 POST: 보통은 에러 메시지와 함께 폼이 다시 출력됨

이러한 폼 처리 과정을 함수형 뷰와 클래스형 뷰로 코딩하는 경우를 통해 비교.

또한 장고는 이러한 폼 처리 과정을 FormView라는 제네릭 뷰로 제공

<함수형 뷰로 폼을 처리>

from django.http import HttpResponseRedirect

from django.shortcuts import render

from .forms import MyForm

def myview(request):

if request.method == “POST”:

form = MyForm(request.POST)

if form.is\_valid():

#cleaned\_data로 관련 로직 처리

return HttpResponseRedirect(‘/success/’)

else:

form = MyForm(initial={‘key’: ‘value’})

return render(request, ‘form\_template.html’, {‘form’: form})

IF 문장으로 HTTP의 GET 방식과 POST 방식을 구분하여 뷰가 GET 방식으로 요청을 받은 경우에는 사용자에게 처음으로 폼을 보여주도록 처리하고, 뷰가 POST 방식으로 요청을 받은 경우에는 데이터가 담긴 제출된 폼으로 간주하여 처리하는 방식임.

동일한 로직을 클래스형 뷰로 코딩하면 다음과 같음

<클래스형 뷰로 폼을 처리>

from django.http import HttpResponseRedirect

from django.shortcuts import render

from django.views.generic import View

from .forms import MyForm

class MyFormView(View):

form\_class = MyForm

initial = {‘key’: ‘value’}

template\_name = ‘form\_template.html’

def get(self, request, \*args, \*\*kwargs): #최초의 GET

form = self.form\_class(initial=self.initial)

return render(request, self.template\_name, {‘form’: form})

def post(self, request, \*args, \*\*kwargs):

form = self.form\_class(request.POST)

if form.is\_valid():

#cleaned\_data로 관련된 로직 처리

return HttpResponseRedirect(‘/success/’) #유효한 데이터를 가진 POST

return render(request, self.template\_name, {‘form’: form}) #유효하지 않은 데이터를 가진 POST

클래스형 뷰에서는 GET 방식과 POST 방식을 메소드로 구분하여 처리함으로써 코드 구조가 깔끔해지고 읽기도 편해짐. 또한 이해하기 쉽게 코드에 표현됨

<FormView 제네릭 뷰로 폼을 처리>

from .forms import MyForm

from django.views.generic.edit import FormView

class MyFormView(FormView):

form\_class = MyForm

template\_name = ‘form\_template.html’

success\_url = ‘/thanks/’

def form\_valid(self, form):

#cleaned\_data로 관련 로직 처리

return super(MyFormView, self).form\_valid(form)

FormView 제네릭 뷰를 사용하면 FormView 클래스에 이미 정의되어 있기 때문에 클래스 내에 get(), post() 메소드 정의도 불필요하게 됨. 개발자가 코딩할 내용이 더 단순해졌는데 다음 4가지 사항 유의하여 코딩

-form\_class: 사용자에게 보여줄 폼을 정의한 forms.py 파일 내의 클래스명

-template\_name: 폼을 포함하여 렌더링할 템플릿 파일 이름

-success\_url: MyFormView 처리가 정상적으로 완료되었을 때 리다이렉트시킬 URL

-form\_valid() 함수: 유효한 폼 데이터로 처리할 로직 코딩, super() 함수를 사용하면, success\_url로 지정된 URL로 리다이렉션 처리됨.