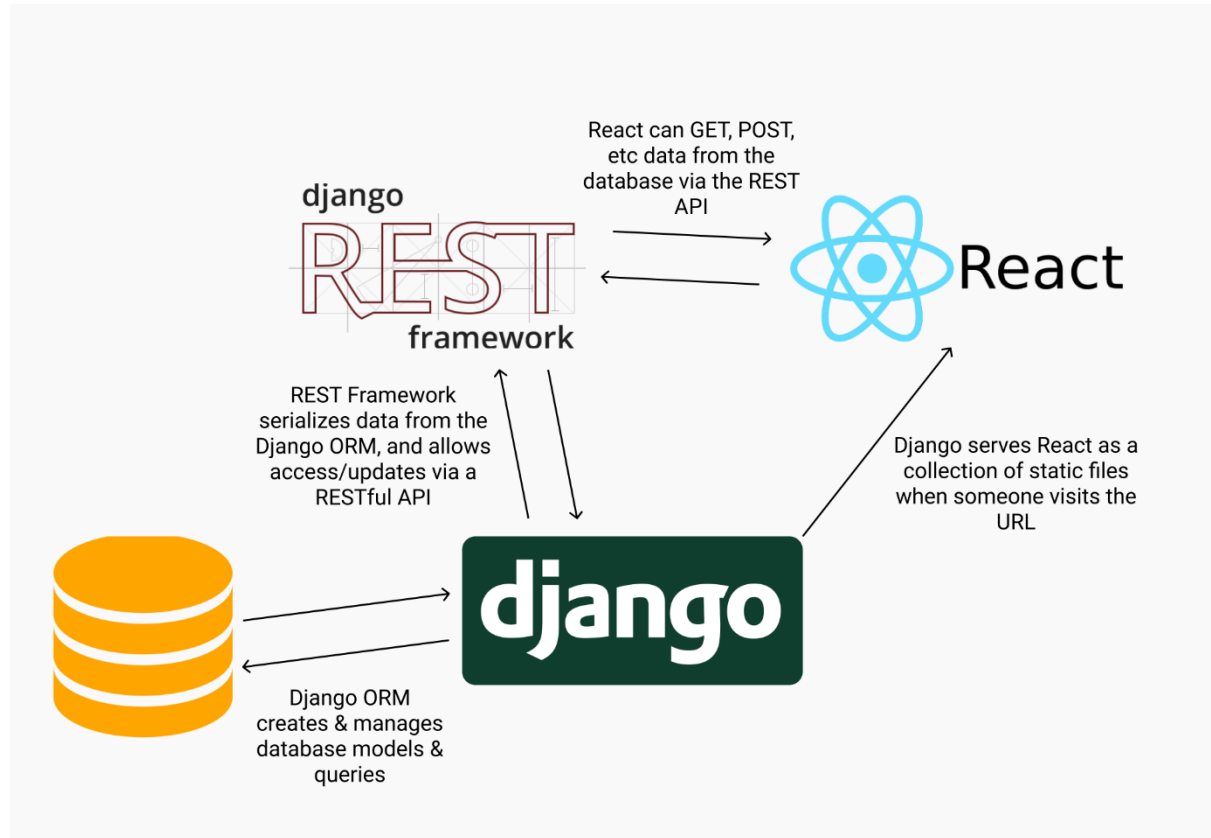


DJANGO

DJANGO 기능

django 기능

- Web Server의 구성과 동작방식



django 기능

- **MVC 패턴 기반 MVT**

- 장고는 Model-view-Controller 를 기반으로 한 프레임워크 이다.
- 장고에서 View를 Template
- Controller 를 View 라고 부른다.
- Model 은 DB에 액세스하는 컴포넌트 이다.
- Template 은 데이터를 사용자에게 보여주는 컴포넌트

- **ORM**

- 장고는 Object-Relational Mapping 방식을 이용해 DB에 데이터를 저장한다.
- 객체 관계 매핑 방식은 데이터 베이스 시스템과 데이터 모델 클래스를 연결시키는 다리 역할을 한다.
- SQL 문장을 사용하지 않고 테이블을 조작할수 있다.
- SQLite3, MySQL, PostgreSQL 등 엔진에 상관없이 사용할수 있다.

django 기능

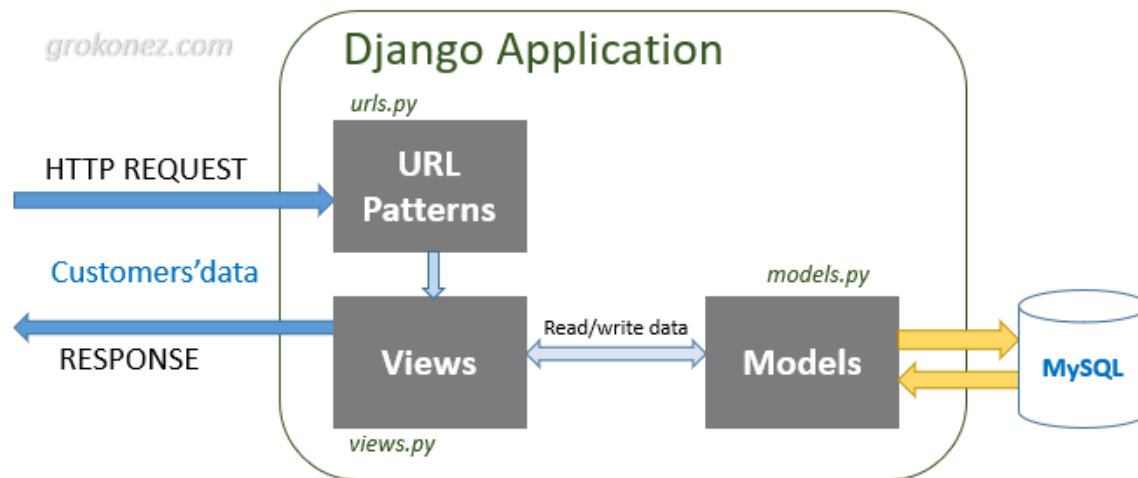
- **주요 기능**

- Function Based Views : HTTP 요청을 함수로 처리한다.
- Class Based Views : Class 로 function 기반 Views 를 만들수 있다.
- Templates language : Views 에서 처리된 결과를 Templates Language로 변환하여 HTML 페이지로 보여진다.

django 기능

• 전체적인 구조

- Client로부터 Request를 받으면 URLs.py 파일을 이용해 URL을 분석
- URLs.py 파일에 정의되어 요청을 처리할 Views를 결정한다.
- Views에 정의된 함수에 의해 로직이 처리되며 필요에 따라 Model을 참조한다.
- 처리된 결과를 Template로 전달하여 Template Language에 의해 HTML페이지를 생성한다.
- HTML 파일을 Client로 전달하여 응답한다.



django 기능

- **주요 기능**

- Function Based Views : HTTP 요청을 함수로 처리한다.
- Class Based Views : Class 로 function 기반 Views 를 만들수 있다.
- Templates language : Views 에서 처리된 결과를 Templates Language로 변환하여 HTML 페이지로 보여진다.

django 기능

- **URLconf**

- Client로부터 요청을 받으면 장고는 가장먼저 요청에 들어있는 URL을 분석한다.
- URL이 urls.py 파일에 정의된 URL 패턴과 매칭되는지를 분석
 - settings.py 파일의 ROOT_URLCONF 항목을 통해 urls.py 파일의 위치가 정의된다.
 - urlpatterns 변수에 지정되어 있는 URL 리스트를 확인한다.

```
path('pybo/<int:number>/', views.pybo_function),
```

- Request URL 이 **/pybo/3** 이면 View 함수를 **views.pybo_function(request, number=3)** 으로 호출한다.

django 기능

• 주요 기능

- Function Based Views : HTTP 요청을 함수로 처리한다.
- Class Based Views : Class 로 function 기반 Views 를 만들수 있다.
- Templates language : Views 에서 처리된 결과를 Templates Language로 변환하여 HTML 페이지로 보여진다.

django 기능

- **Views**

- View 함수는 첫 번째 인자로 HttpRequest 객체인 request 를 받는다.
- 필요한 처리 이후 HttpResponse 객체를 반환
- HttpResponse 이외 다른 객체들이 많이 존재 한다.

```
# views.py

from django.http import HttpResponse

def function_name(request):
    html = <html 에 전달할 내용>
    return HttpResponse(html)
```

Model

- **Template**

- Client 에게 반환하는 최종 응답은 HTML 텍스트
- 템플릿 파일은 *.html 확장자를 가지며, 장고 템플릿 문법에 맞게 작성한다.

```
# settings.py
TEMPLATES = [
{
'DIRS': [os.path.join(BASE_DIR, 'templates')]
}
]
```

django 기능

- **장고 앱**

- 현재 프로젝트의 기능을 다른 프로젝트에서도 사용하려면?
- APP Template 형태로 제작하여 재사용 가능

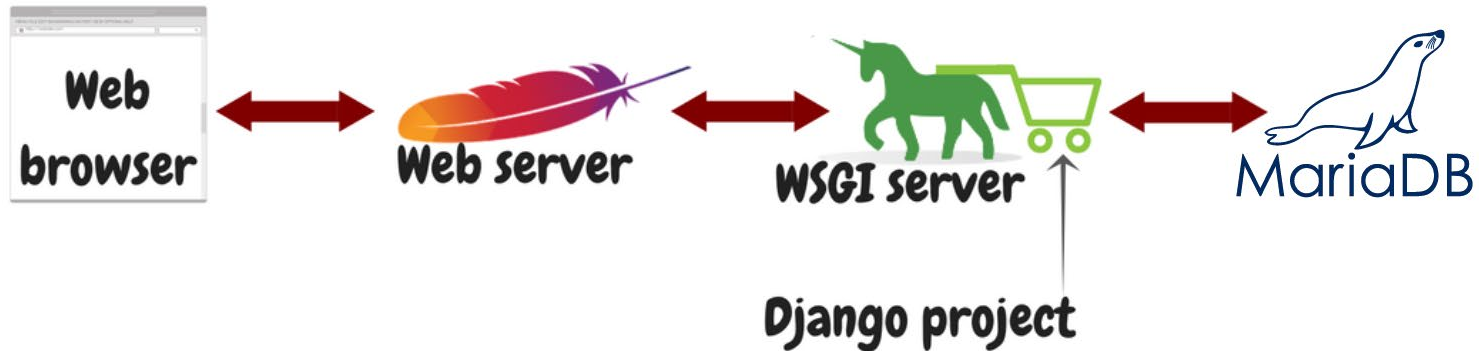
```
python manage.py startapp <appName>
```

- 프로젝트 상에서 유일해야한다.
- settings.INSTALLED_APPS 에 등록시켜 사용 가능

django 기능

- **Gole**

- Web의 동작방식을 이해한다.
- 3 tier architecture를 이해하고 구현한다.



django 기능

- **주요 기능**

- Function Based Views : HTTP 요청을 함수로 처리한다.
- Class Based Views : Class 로 function 기반 Views 를 만들수 있다.
- Templates language : Views 에서 처리된 결과를 Templates Language로 변환하여 HTML 페이지로 보여진다.

django 기능

- **Django 디렉터리 구조**

- `db.sqlite3` : SQLite3 데이터 베이스 파일
- `manage.py` : 장고 명령어 처리
- `urls.py` : 프로젝트 레벨/어플리케이션 레벨의 URL 패턴을 정의하는 URLconf 파일
- `wsgi.py` : 웹 서버와 WSGI 규격으로 연동하기 위한 파일
- `views.py` : 뷰 함수를 정의하는 파일. 함수형 뷰 및 클래스형 뷰 모두 이 파일에 정의한다.
- `templates` 디렉터리 : 템플릿 파일들이 들어있으며, 프로젝트/어플리케이션 레벨의 템플릿으로 구분된다.
- `static` 디렉터리 : CSS, Javascript 파일들이 들어있으며, 프로젝트/어플리케이션 레벨의 파일들로 구성되어있다.
- `logs` 디렉터리 : 로그 파일을 저장해 두는곳

URL

URL

- `django.urls.path()`
- `path()` 함수는 `route`, `view` 2개의 필수 인자와 `kwargs`, `name` 2개의 선택 인자를 받는다.
 - `route` : URL 패턴을 표현하는 문자열. URL 스트링 이라고도 불린다.
 - `View` : URL 스트링이 매칭되면 호출되는 View 함수. `HttpRequest` 객체와 URL 스트링에서 추출된 항목이 뷰 함수의 인자로 전달 된다.
 - `kwargs` : URL 스트링에서 추출된 항목 외에 **추가적인 인자를 View 함수에 전달할 때**, 파이썬 Dict 타입으로 인자를 정의 한다.
 - `name` : 각 URL 패턴 별로 이름을 붙여준다. 여기서 정해진 이름은 템플릿 파일에 많이 사용된다.

URL

```
path('polls', views.index, name='index')
```

- 요청 URL 이 /polls/ 이라면 views.index(request) 함수가 호출된다.
- URL 패턴의 이름은 index

```
path('polls/<int:question_id>'), views.detail, name='detail'
```

- URL 패턴의 이름은 detail
- 요청 URL이 /polls/7/
- 추출된 7은 python int 타입으로 변환
- views.detail(request, question_id=7) 로 대입

URL

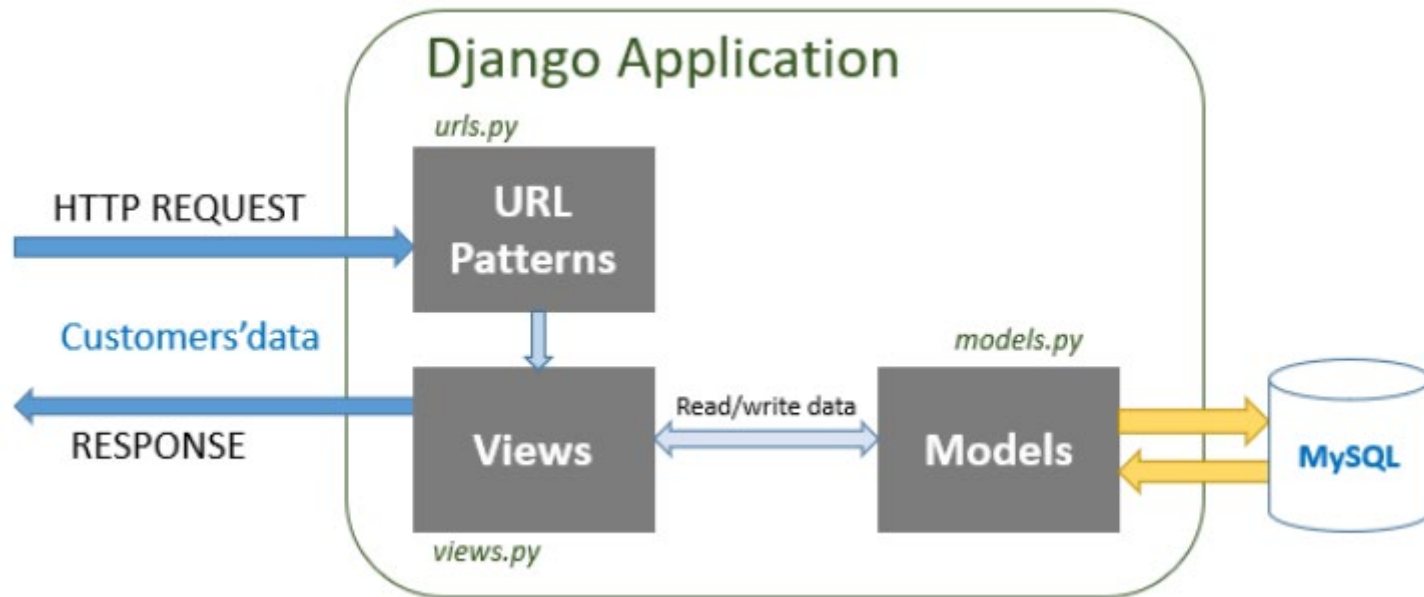
- **URL Name**

- app_name 변수는 URL 패턴의 이름이 충돌나는 것을 방지하기위한 Namespace 역할을 한다.
 - polls APP 의 detail 은 polls:detail
 - blog APP 의 detail 은 blog:detail
- {% url %} 템플릿 태그나 reverse() 함수에서 자주 사용된다.

FORM

FORM

- HTML 폼은 웹 페이지 상에서 한개 이상의 필드나 위젯들의 묶음을 의미한다.
- 사용자로부터 정보를 수집하여 서버에 제출



FORM

- 정보수집

- 텍스트 박스
- 체크 박스
- 라디오 버튼
- 날짜 선택기

- 교차 사이트 위조 방지(CSRF)

- cross-site request forgery protection 와 함께 POST 요청으로 데이터를 보낼수 있도록 지원한다.

```
MIDDLEWARE = [  
    ...  
    'django.middleware.csrf.CsrfViewMiddleware',  
    ...  
]
```

FORM

- **URL Name**

- app_name 변수는 URL 패턴의 이름이 충돌나는 것을 방지하기위한 Namespace 역할을 한다.
 - polls APP 의 detail 은 polls:detail
 - blog APP 의 detail 은 blog:detail
- {% url %} 템플릿 태그나 reverse() 함수에서 자주 사용된다.

FORM

- **폼 개발**

- Form HTML 작성 → 서버로 입력된 데이터의 유효성 검증 → 유효하지 않는 데이터는 사용자가 알 수 있도록 에러 표시
- Form HTML 작성 → 서버로 입력된 데이터의 유효성 검증 → 유효한 데이터 → 처리 → 반환

- **Form TAG**

- action : 폼이 Submit 될 때 처리가 필요한 데이터를 전달받는 곳의 주소
- method : 데이터 전송시 사용할 HTTP 메소드 (GET or POST)

- POST
- GET

```
<form action="/team_name_url/" method="post">
  <label for="team_name">Enter name: </label>
  <input id="team_name" type="text" name="name_field" value="Default"
  <input type="submit" value="OK">
</form>
```


FORM

- **Form 작성하기**

- Form 클래스는 form내 field들, field 배치, 디스플레이 widget, 라벨, 초기값, 유효한 값과 (**유효성 체크** 이후에) 비 유효 field에 관련된 에러메시지를 결정
- Form 클래스는 또한 미리 정의된 포맷(테이블, 리스트 등등) 의 템플릿으로 그 자신을 렌더링하는 method 등을 제공한다.
 - 미리 정해 둔 포맷 → method → 렌더링
- Form 을 선언하는 문법은 model을 선언하는 것과 비슷하다.
 - 같은 필드타입 사용
 - 매개변수 유사
 - 유효성 규칙 검사 확인

FORM

- **form 선언**

- Form 을 선언하는 문법은 model을 선언하는것과 비슷하다.
 - 같은 필드타입 사용
 - 매개변수 유사
 - 유효성 규칙 검사 확인

```
# app/forms.py

from django import forms
class <formname>(forms.Form):
    <data_name> = forms.<FieldType>()
```

FORM











- **form 필드의 종류**

- BooleanField, CharField, ChoiceField, TypedChoiceField, DateField, DateTimeField, DecimalField, DurationField, EmailField, FileField, FilePathField, FloatField, ImageField, IntegerField, GenericIPAddressField, MultipleChoiceField, TypedMultipleChoiceField, NullBooleanField, RegexField, SlugField, TimeField, URLField, UUIDField, ComboField, MultiValueField, SplitDateTimeField, ModelMultipleChoiceField, ModelChoiceField ..

MODEL

Model

- APP 생성 후에는 APP에서 사용해야한 DB가 필요하다
- Django 에서는 DB를 관리하기 위해 각 APP마다 models.py 파일을 생성해 DB 객체를 생성한다.
- 장고에서 지원되는 DB

 stable/3.1.x	django / django / db / backends /	Go to file
This branch is 395 commits ahead, 1399 commits behind main.		 Contribute
 feliixm [3.1.x] Fixed #32403 -- Fixed re-raising DatabaseErrors when usi... on 3 Feb  History		
..		
 base	[3.1.x] Bumped minimum isort version to 5.1.0.	9 months ago
 dummy	Made DatabaseFeatures.uses_savepoints default to Tr...	3 years ago
 mysql	[3.1.x] Fixed #31965 -- Adjusted multi-table fast-delet...	10 months ago
 oracle	[3.1.x] Fixed #31836 -- Dropped support for JSONFiel...	11 months ago
 postgresql	[3.1.x] Fixed #32403 -- Fixed re-raising DatabaseError...	5 months ago
 sqlite3	[3.1.x] Fixed #32224 -- Avoided suppressing connecti...	7 months ago

Model

- 기초 작업

- `models.py` 파일에 모델 Class 를 정의해줌으로서 데이터 설계
- DataBase Table과 Python class 를 1:1 로 맵핑하여 실행한다.

```
# models.py

from django.db import models

class Question(models.Model):
    title = model.CharField(max_length=50)
    ...
```

Model

- **model Field Type**

- Primary Key : AutoField, BigAutoField
- 문자열 : CharField, TextField, SlugField
- 날짜/시간 : DateField, TimeField, DateTimeField, DurationField
- 참/거짓 : BooleanField, NullBooleanField
- 숫자 : IntegerField, SmallIntegerField, positiveIntegerField, PositiveSmallIntegerField, BigIntegerField, DecimalField, FloatField
- 파일 : NainaryField, FileField, ImageField, FilePathField
- 이메일 : EmailField
- URL : URLField

Model

- App Model 작성

```
# mysite/pybo/models.py

from django.db import models

class Question(models.Model):
    subject = models.CharField(max_length=200)
    content = models.TextField()
    create_date = models.DateTimeField()

# ----- [edit] ----- #
class Answer(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    content = models.TextField()
    create_date = models.DateTimeField()
```


Model

- **App Model**

- `models.ForeignKey` 테이블의 필드 중에서 다른 테이블의 행과 식별할수 있는키
- `ForeignKey`는 테이블과 테이블을 연결하기 위한키
- `Answer`는 `Question` 에 대한 답변이므로 `Question` 을 속성으로 가져야 한다 (`ForeignKey`가 필요하다)
- `on_delete=models.CASCADE`는 답변에 연결된 질문이 삭제되면 답변도 함께 삭제하라는 의미이다.

Model

- **Model 관리 순서**

1. Model Class 작성

2. Model Class로 부터 Migration 파일 생성

```
python3 manage.py makemigrations
```

3. Migration 파일을 DB에 적용

```
python3 manage.py migrate
```

4. Table 생성

1. Table 명은 "AppName_ModelName" 의 방식으로 형성된다.

2. pybo 앱

Question 모델 → "pybo_Question" Table

Model

- Admin page 에 등록

- admin 페이지에서 Model 을 관리하기 위해서는 별도의 등록 코드가 필요하다.
 - admin.site.register(<Class 명>) : 해당 Class 를 admin site에 등록해준다.
 - Class QuestionAdmin

```
# pybo/admin.py

from django.contrib import admin
# ----- [edit] ----- #
from .models import Question

class QuestionAdmin(admin.ModelAdmin):
    search_fields = ['subject']

admin.site.register(Question)
```

Model

- **Key**

- PK(primary Key)는 클래스에 지정해주지 않아도 , 장고는 항상 PK에 대한 속성을 Not Null및 Autoincrement로, 이름은 id로 해서 자동으로 만들어 줍니다 .
- FK(Foreign Key)는 항상 다른 테이블의 PK에 연결되므로 , Question 클래스의 id 변수까지 지정할 필요 없이 Question 클래스만 지정하면 됩니다 . 실제 테이블에서 FK로 지정 된 컬럼은 _id 접미사가 붙는다는 점도 알아두기 바랍니다 .

Model

- **RDBMS**
 - 1:1 OneToOneField
 - 1:n ForeignKey
 - m:n ManyToManyField

Model

- **1:1 관계**

- <https://docs.djangoproject.com/en/3.2/ref/models/fields/#django.db.models.OneToOneField>
- 축구장 : 주심
- 역 참조시 단일 객체를 리턴하는 점에서 1:n 관계와 차이가 있다.
- 각각은 AUTH_USER_MODEL 과 1:1 관계를 형성하고 있다.

```
class OneToOneField(to, on_delete, parent_link=False, **options)
```

Model

- 특이사항

- 1:1:1 → 1:2(역참조 이므로 1:n의 관계 형성)

```
from django.conf import settings
from django.db import models

class MySpecialUser(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
    supervisor = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='supervisor_of',
    )
```

Model

- **1:N 관계(ForeignKey)**

- <https://docs.djangoproject.com/en/3.2/ref/models/fields/#django.db.models.ForeignKey>
- 축구장 : 선수
- 주심 : 선수
- 1개의 포스팅에 N개의 댓글
- 1개의 유저에 N개의 포스팅

```
class ForeignKey(to, on_delete, **options)
```


Model

- 대상을 직접 지정하는 방식

```
from django.db import models

class Car(models.Model):
    manufacturer = models.ForeignKey(
        'Manufacturer',
        on_delete=models.CASCADE,
    )

class Manufacturer(models.Model):
    # ...
    pass
```

Model

- 자기 자신을 참조하는 방식(재귀적 참조)

```
models.ForeignKey('self', on_delete=models.CASCADE)
```

Model

- **M:N(many)**

- 축구 선수 : 축구 선수
- 1개의 Post 에는 다수의 Tag, 1개의 Tag에는 다수의 Post
- 피자과 토핑의 관계
 - 피자는 다수의 토핑을 가진다면, 토핑또한 다수의 피자에 있을수 있다.

```
class ManyToManyField(to, **options)
```

Model

- 자기 자신을 참조하는 방식(재귀적 참조)

```
from django.db import models

class Topping(models.Model):
    name = models.CharField(max_length=50)
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

Model

- 재귀 관계에 따른 다대다 모델
 - 나는 너의 친구고 너는 나의 친구다

```
from django.db import models

class Person(models.Model):
    friends = models.ManyToManyField("self")
```

- **중간 경유 모델**

- 예를 들어 뮤지션이 속한 음악 그룹을 추적하는 애플리케이션
- MemberShip 은 다음과 같은 두개의 관계를 정의한다.

- PerSon
- Group

- 개인과 구성원이 속한 그룹간에 다 대다 관계가 있으므로 ManyToManyField를 사용하여이 관계를 나타낼 수 있지만, 그 사람이 그룹에 가입 한 날짜와 같이 수집 할 수있는 멤버십에 대한 많은 세부 정보가 필요로함

- PerSon : 뮤지션
- Group : 뮤지션이 속할 그룹
- Membership : 뮤지션과 그룹간의 관계를 정의
 - 날짜
 - 참여이유

Model

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=50)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(
        Person,
        through='Membership',
        through_fields=('group', 'person'),
    )

class Membership(models.Model):
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    inviter = models.ForeignKey(
        Person,
        on_delete=models.CASCADE,
        related_name="membership_invites",
    )
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

- `through_fields` 는 2-tuple (`field1`, `field2`) 를 허용
- `field1`은 `ManyToManyField`가 선언된 모델 (`group`)에 대한 Foreign 키의 이름 `field2`는 대상 모델(`person`)에 대한 foreign키의 이름이다.

Model

- 다대다 관계에 참여하는 모델중 어느 하나(또는 둘 모두)의 중간 모델에 두 개 이상의 foreign 키가 있다면, through_fields 를 지정

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>]>
>>> ringo.group_set.all()
<QuerySet [<Group: The Beatles>]>
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>]>
```


TEMPLATE

Template

- **docs**

- <https://docs.djangoproject.com/en/3.2/ref/templates/>

- 랜더링 : 템플릿 코드 -> 템플릿 파일

- **템플릿 태그**

- {% tag %} 인 기본형을 갖고 있다.

Template

- 형식

```
# views.py

def fun_1(request)
    ...
    value = <전달 내용>
    context = {"variable":value}
    return render(request, <index_name>.html, <context>)
```

```
# <index_name>.html

{{ variable }}
```

Template

- Template 문법에서 도트(.)를 만나면 다음과 같은 순서로 문장을 찾는다.
- `foo.bar`라는 템플릿 변수가 있다면 다음과 같이 해석한다.
 - `foo`가 사전 타입인지 확인한다. 그렇다면 `foo['bar']` 로 해석한다.
 - 그 다음은 `foo` 속성을 찾는다. `bar` 라는 속성이 있으면 `foo.bar`로 해석한다.
 - 그것도 아니면 `foo`가 리스트인지 확인한다. 그렇다면 `foo[bar]`로 해석한다.

Template

- **for**

```
{% for %}
```

```
{% endfor %}
```

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

Template

- **if**
 - 변수를 평가하여 True이면 바로 아래 문장 실행
 - 중첩if문 {% if %} {% elif %} {% else %} {% endif %} 가능
 - {% if %} 태그에 필터와 연산자 사용 가능
 - 주의할 점은, 필터가 스트링을 반환 시 산술연산 안됨 (단, length 필터는 가능)
 - and, or, not, and not, >, >=, <, <=, ==, !=, in, not in 연산도 가능

```
{% if %} {% endif %}
```

Template

- **csrf**

- CSRF 공격 방지 위한 태그이며, 장고는 내부적으로 CSRF 토큰값의 유효성을 검증함
- <form> 태그 바로 첫 줄에 넣어줌

```
<form>
{% csrf_token %}
...
</form>
```

Template

- URL 태그

- 하드코딩 방지용
- namespace : urls.py 의 app_name 변수에 정의한 namespace
- view-name : urls.py 파일에서 정의한 URL 패턴 이름
- argN : view 함수에서 사용하는 인자로, 없을 수도 있고 여러 개인 경우 빈칸으로 구분

```
# appname/index.html
{% url 'namespae:view-name' arg1 arg2 %}
```


Template

- 예

```
# urls.py

from django.urls import path
from . import views

app_name = 'polls'
urlpatterns = [path('<int:question_id>/', views.detail, name='detail'),
path('<int:question_id>/results/', views.results, name='results'),
]
```

```
# index.html
<a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

Template

- **Load 태그**

- 사용자 정의 태그 및 필터를 로딩
- file1.py 파일 및 package/file2.py 파일에 정의된 사용자 정의 태그 및 필터 로딩

```
{% load file1 package.file2 %}
```

Template

- 폼 클래스를 템플릿으로 변환

- {{ form }} : HTML 의 <Label> 과 <input> 엘리먼트 쌍으로 렌더링
- {{ form.as_table }} : <tr> 태그로 감싸서 테이블 Cell로 렌더링 된다.
- {{ form.as_p }} : <p> 태그로 감싸도록 렌더링
- {{ form.as_ul }} : 태그로 감싸도록 렌더링

```
<form method="POST">
    {{ form.as_p }}
    <input type="submit" value="저장"/>
</form>
```

Template

- **템플릿 상속(template-inheritance)**
 - HTML 문서중 공통으로 사용되는 코드를 저장해 두고 상속하여 사용
- **기본 템플릿은 base.html 으로 지정한다.**
 - 기본형
 - templates위치 지정

```
'DIRS': [os.path.join(BASE_DIR, '<ProjectName>/templates')]
```

Template

- base.html 생성

```
# <ProjectName>/templates/base.html
<!DOCTYPE html>
<html>
<tag> {% block <name> %} 상속할 내용 {% endblock %} </tag>
...
</html>
```

- 대상 Template 에 base.html 확장

```
{% extends 'base.html' %}

{% block <name> %}
상속받을 내용이 적용 되는곳
{% endblock %}
```

Template

• 예

```
# <ProjectName>/templates/base.html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="style.css">
  <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
  <div id="sidebar">
    {% block sidebar %}
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/blog/">Blog</a></li>
    </ul>
    {% endblock %}
  </div>

  <div id="content">
    {% block content %}{% endblock %}
  </div>
</body>
</html>
```

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
  <h2>{{ entry.title }}</h2>
  <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="style.css">
  <title>My amazing blog</title>
</head>

<body>
  <div id="sidebar">
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/blog/">Blog</a></li>
    </ul>
  </div>

  <div id="content">
    <h2>Entry one</h2>
    <p>This is my first entry.</p>

    <h2>Entry two</h2>
    <p>This is my second entry.</p>
  </div>
</body>
</html>
```

Template

- 자식 템플릿이 **sidebar**블록을 정의하지 않았기 때문에 부모 템플릿의 값이 대신 사용됩니다
- title, content 는 중복 되기 때문에 적용되지 않음

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">
    <title>My amazing blog</title>
</head>
<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

VIEWS

views

- **views**

- HTTP 요청에 대해 View 가 호출되어 동작을 수행한다.
- click → HTTPRequest 객체 생성 → urls.py → views.py 내부의 함수 호출 → HTTPRequest 처리 → HTTPResponse 객체 전달

```
path('polls/<int:question_id>'), views.detail, name='detail'
```

views

- 종류
 - View 의 종류
 - Function Based View
 - Class Based View

- 자주 사용되는 객체

- Request 객체

- request 요청시 `django.core.handlers.wsgi.WSGIRequest` 객체 생성
 - WSGIRequest 객체는 `HttpRequest` Class 를 상속받아 생성 된다.

```
class HttpRequest:
    """A basic HTTP request."""

    # The encoding used in GET/POST dicts. None means use default setting.
    _encoding = None
    _upload_handlers = []

    def __init__(self):
        # WARNING: The `WSGIRequest` subclass doesn't call `super`.
        # Any variable assignment made here should also happen in
        # `WSGIRequest.__init__`.

        self.GET = QueryDict(mutable=True)
        self.POST = QueryDict(mutable=True)
        self.COOKIE = {}
        self.META = {}
        self.FILES = MultiValueDict()

        self.path = ''
        self.path_info = ''
        self.method = None
        self.resolver_match = None
        self.content_type = None
        self.content_params = None

    def __repr__(self):
        if self.method is None or not self.get_full_path():
            return '<%s>' % self.__class__.__name__
        return '<%s: %s %r>' % (self.__class__.__name__, self.method, self.get_full_path())
```

views

```
class HttpRequest:
    """A basic HTTP request."""

    # The encoding used in GET/POST dicts. None means use default setting.
    _encoding = None
    _upload_handlers = []

    def __init__(self):
        # WARNING: The `WSGIRequest` subclass doesn't call `super`.
        # Any variable assignment made here should also happen in
        # `WSGIRequest.__init__`.

        self.GET = QueryDict(mutable=True)
        self.POST = QueryDict(mutable=True)
        self.COOKIES = {}
        self.META = {}
        self.FILES = MultiValueDict()

        self.path = ''
        self.path_info = ''
        self.method = None
        self.resolver_match = None
        self.content_type = None
        self.content_params = None

    def __repr__(self):
        if self.method is None or not self.get_full_path():
            return '<%s>' % self.__class__.__name__
        return '<%s: %s %r>' % (self.__class__.__name__, self.method, self.get_full_path())

    ..|
```

`self.body` # request의 body 객체
`self.headers` # request의 headers 객체
`self.COOKIES` # 모든 쿠키를 담고 있는 딕셔너리 객체
`self.method` # request의 메소드 타입
`self.GET` # GET 파라미터를 담고 있는 딕셔너리 같은 객체
`self.POST` # POST 파라미터를 담고 있는 딕셔너리 같은 객체

views

- HttpResponse class
- 다양한 객체로 변경하여 응답 가능, 주로 HTML 을 반환한다.
 - 엑셀파일
 - 이미지

```
HttpResponse(data, content_type)
```

```
# string 전달
HttpResponse("Here's the text of the Web page.")

# html 태그 전달
response = HttpResponse() #객체 생성
>>> response.write("<p>Here's the text of the Web page.</p>") #content
```

- **render()**

- 템플릿 파일과 컨텍스트 사전을 인자로 받은 후 렌더링 처리한 후, **HttpResponse** 객체를 반환
- request, template_name 을 제외한 나머지 인자는 필수인자가 아니다.

- **필수 인수**

- **request**이 응답을 생성하는 데 사용되는 요청 개체
- **template_name** 사용할 템플릿의 전체 이름 또는 일련의 템플릿 이름입니다. 시퀀스가 제공되면 존재하는 첫 번째 템플릿이 사용됩니다. 템플릿을 찾는 방법에 대한 자세한 내용은 템플릿로드 설명서 를 참조하십시오.

- **get_object_or_404()**
 - **get_object_or_404(*klass* , * *args* , ** *kwargs*)**
 - klass 모델에 해당하는 테이블에서 args 또는 kargs 조건에 맞는 레코드를 검색한다. 있으면 해당 레코드를 반환하고, 없으면 Http404 익셉션을 발생시킨다.

views

- 예제

- **MyModel**에서 기본 키가 1인 객체를 찾아서 반환한다.

```
from django.shortcuts import get_object_or_404

def my_view(request):
    obj = get_object_or_404(MyModel, pk=1)
```

- 이는 다음과 같은 문장을 단축했다.

```
from django.http import Http404

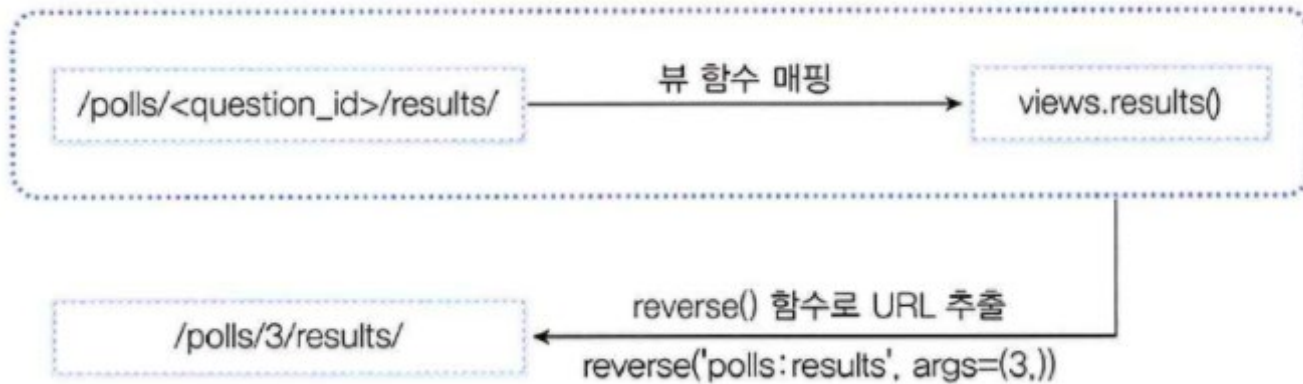
def my_view(request):
    try:
        obj = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("No MyModel matches the given query.")
```


views

- reverse()

```
reverse(viewname, urlconf=None, args=None, kwargs=None, current_app=None)
```

URLconf에 정의된 URL 패턴명 = polls:results



views

- Class View
 - [Django Class-Based-View Inspector -- Classy CBV \(ccbv.co.uk\)](http://ccbv.co.uk)

- Base View : 뷰 클래스를 생성하고 다른, 제너릭 뷰의 부모 클래스가 되는 기본 제너릭 뷰
 - View: 최상위 부모 제너릭 뷰 클래스
 - TemplateView: 주어진 템플릿으로 렌더링
 - RedirectView: 주어진 URL로 리다이렉트
- Generic Display View : 객체의 목록 또는 하나의 객체 상세 정보를 보여주는 뷰
 - DetailView: 조건에 맞는 하나의 객체 출력
 - ListView: 조건에 맞는 객체 목록 출력

- Generic Edit View : 폼을 통해 객체를 생성, 수정, 삭제하는 기능을 제공하는 뷰
 - FormView: 폼이 주어지면 해당 폼을 출력
 - CreateView: 객체를 생성하는 폼 출력
 - UpdateView: 기존 객체를 수정하는 폼을 출력
 - DeleteView: 기존 객체를 삭제하는 폼을 출력
- Generic Date View : 날짜 기반 객체의 연/월/일 페이지로 구분해 보여주는 뷰
 - YearArchiveView: 주어진 연도에 해당하는 객체 출력
 - MonthArchiveView: 주어진 월에 해당하는 객체 출력
 - DayArchiveView: 주어진 날짜에 해당하는 객체 출력
 - TodayArchiveView: 오늘 날짜에 해당하는 객체 출력
 - DateDetailView: 주어진 연, 월, 일 PK(또는 슬러그)에 해당하는 객체 출력

■ 변수

- model : 기본 뷰(View, Template, RedirectView) 3개를 제외하고 모든 제너릭 뷰에서 사용한다.
- queryset : 기본 뷰(View, Template, RedirectView) 3개를 제외하고 모든 제너릭 뷰에서 사용한다. queryset을 사용하면 model 속성은 무시된다.
- template_name : TemplateView를 포함한 모든 제너릭 뷰에서 사용한다. 템플릿 파일명을 문자열로 지정한다.
- context_object_name : 뷰에서 템플릿 파일에 전달하는 컨텍스트 변수명을 지정한다.
- paginate_by : ListView와 날짜 기반 뷰(예, YearArchiveView)에서 사용한다. 페이징 기능이 활성화 된 경우 페이지당 출력 항목 수를 정수로 지정한다.
- date_field : 날짜 기반 뷰(예, YearArchiveView)에서 사용한다. 이 필드의 타입은 DateField 또는 DateTimeField이다.
- form_class : FormView, CreateView, UpdateView에서 폼을 만드는데 사용할 클래스를 지정한다.
- success_url : FormView, CreateView, UpdateView, DeleteView에서 폼에 대한 처리가 성공한 후 리디렉트할 URL 주소이다.

WSGI

WSGI

- **WSGI (Web Server Gateway Interface)**

- Python Web Application Server 에서 준수하는 규격
- 파이썬 웹 프레임 워크와 웹 서버를 연동할때 사용
- WSGI 모듈이 CGI 모듈을 포함하고 있으므로 CGI 모듈을 단독적으로 사용하지 않는다.

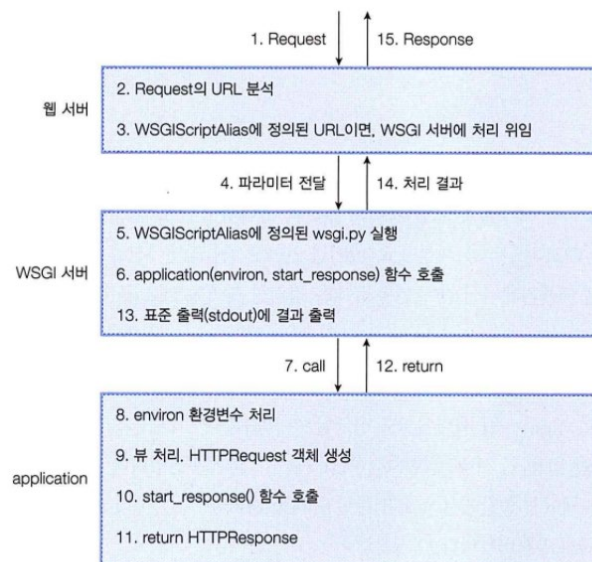
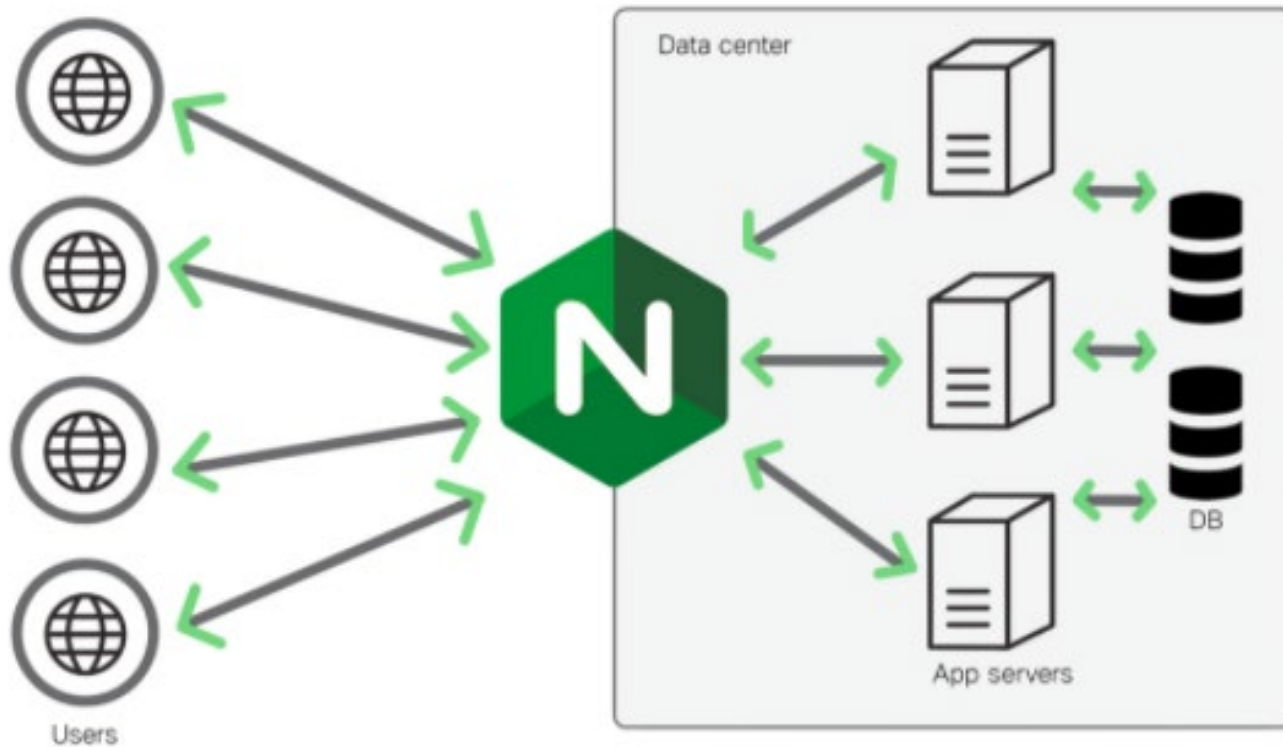


그림 2-12 WSGI 애플리케이션의 처리 순서

WSGI

- Proxy



WSGI

- Static 파일 연동

