

# 220831 Django 2

## 1. Namespace

- 개체를 구분할 수 있는 범위를 나타내는 namespace(이름공간)

### 1.1 Namespace의 필요성

- 두번째 app pages의 index 페이지를 작성 시 2가지 문제점 발생 (시험에 나왔음)
  - article app index 페이지에 작성한 두번째 앱 index로 이동하는 하이퍼 링크를 클릭 시 현재 페이지로 다시 이동
    - URL namespace
  - pages app의 index url로 직접 이동해도 articles app의 index 페이지가 출력됨
    - Template namespace

### 1.2 URL namespace

- Django templates의 기본 경로에 app과 같은 이름의 폴더를 생성해 폴더 구조를 `app_name/templates/app_name/` 형태로 변경
- `app_name attribute` 를 작성해 URL namespace를 설정

```
app_name = 'articles'  
urlpatterns = [  
    ...,  
]
```

- URL tag의 변화
  - `{% URL 'index' %}` → `{% url 'articles:index' %}`
  - app\_name을 지정한 이후에는 url 태그에서 반드시 app\_name : url\_name 형 태로만 사용
    - 그렇지 않으면 NoReverseMatch 에러 발생
- URL 참조
  - `:` 연산자를 사용하여 지정
  - app\_name이 articles이고 URL name이 index인 주소 참조는 `articles:index`

## 1.3 Template namespace

- Django는 기본적으로 `app_name/templates/` 경로에 있는 templates 파일들만 찾을 수 있으며, `settings.py`의 `INSTALLED_APPS`에 작성한 app 순서로 `template`을 검색 후 렌더링 함 (시험에 나왔음 서술형)
- 기본 경로에 app과 같은 이름의 폴더를 생성해 `app_name/templates/app_name/` 형태로 변경
  - 기본 경로는 바꿀수 없기 때문에 index앞에 물리적인 폴더를 하나 더 집어넣음
- 만일 단일 앱으로만 이루어진 프로젝트라면 상관없지만, 여러 앱이 되었을때는 겹치는 이름이 생기기 마련이라 고려할 것

## 2. Django Model

### 2.1 Database

- 개념
  - 체계화 된 데이터의 모임
  - 검색 및 구조화 같은 작업을 보다 쉽게 하기 위해 조직화된 데이터를 수집하는 저장 시스템
- Database 기본 구조
  - 스키마 (Schema)
  - 테이블 (Table)

#### 2.1.1 스키마

- 뼈대
- 데이터베이스에서 자료의 구조, 표현방법, 관계 등을 정의한 구조

column	datatype
id	INT
name	TEXT
age	INT
email	TEXT

## 2.1.2 테이블(Table)

- 필드와 레코드를 사용해 조직된 데이터 요소들의 집합
- 관계(Relation)라고도 부름
- 데이터가 작성되었다 → 레코드가 작성
- DB - table - record - field
- 필드는 데이터의 속성

**필드**

	A	B	C	D
1	id	name	age	email
2	1	hong	42	hong@gmail.com
3	2	kim	16	kim@naver.com
4	3	kang	29	kang@hotmail.com
5	4	chol	8	choi@hanmail.com

**테이블**

**레코드**

- **필드(filed)**
    - 속성 혹은 컬럼
    - 각 필드에는 고유한 데이터 형식을 지원
  - **레코드(record)**
    - 튜플, 행
    - 테이블의 데이터는 레코드에 저장됨
  - **PK (Primary Key)**
    - 기본 키
    - 각 레코드의 고유한 값(식별자로 사용)
    - 기술적으로 다른 항목과 절대로 중복될 수 없는 단일 값(unique)을 가짐
    - 데이터베이스 관리 및 테이블 간 관계 설정 시 주요하게 활용 됨
- ex) 주민등록번호

## 1.2.3 쿼리

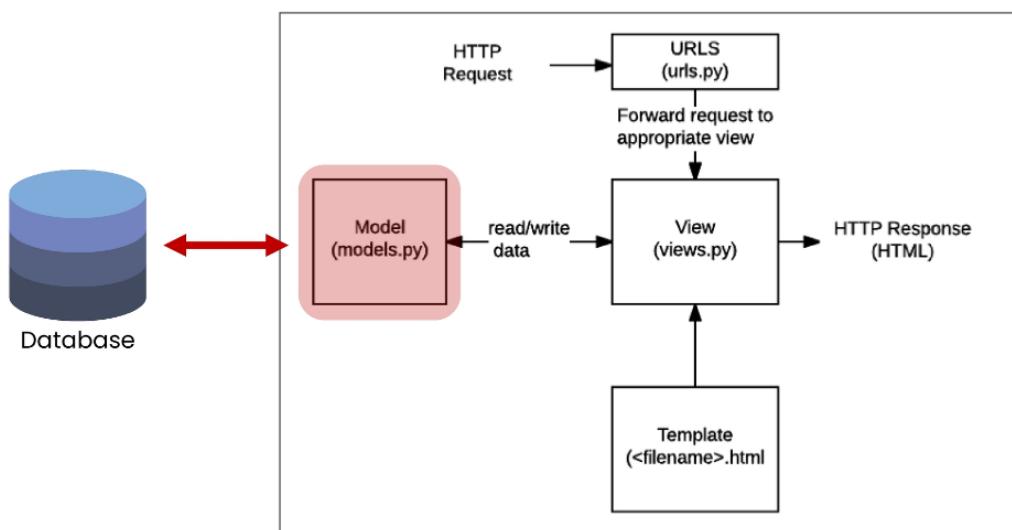
- 데이터를 조회하기 위한 명령어
- 조건에 맞는 데이터를 추출하거나 조작하는 명령어
  - (주로 테이블형 자료구조에서)
- Query를 날린다 → 데이터베이스를 조작한다

## 2.2 Model

웹 애플리케이션의 데이터를 구조화하고 조작하기 위한 도구

### 2.2.1 개요

- Django는 Model을 통해 데이터를 접근하고 조작
- 사용하는 데이터들의 필수적인 필드들과 동작들을 포함
- 저장된 데이터베이스의 구조
- 일반적으로 각각의 모델은 하나의 데이터베이스 테이블에 맵핑
- 모델 클래스 1개 = 데이터베이스 테이블 1개
- Model을 통해 데이터 관리 (각각의 역할을 쓰라고 시험)



[참고] 맵핑

- 하나의 값을 다른 값으로 대응시키는 것

### 2.2.2 모델 작성하기

- 모델 클래스를 작성 하는 것은 데이터베이스 테이블의 스키마를 정의 하는 것
- 모델 클래스 = 테이블 스키마

```
# articles/models.py

class Article(models.Model):
    title = models.CharField(max_length=10)
    content = models.TextField()
```

- id 컬럼은 테이블 생성시 Django가 자동으로 생성

- 각 모델은 django.models.Model 클래스의 서브 클래스로 표현됨
  - 즉, 각 모델은 django.db.models 모듈의 Model 클래스를 상속받아 구성 됨
  - 클래스 상속 기반 형태의 장고 프레임워크 개발
- models 모듈을 통해 어떠한 타입의 DB 필드를 정의할 것인지 정의

```
# articles/models.py

class Article(models.Model):
    title = models.CharField(max_length=10)
    content = models.TextField()
```

- 클래스 변수 title과 content는 DB 필드를 나타냄
  - 클래스 변수(속성)명 : DB 필드의 이름
  - 클래스 변수 값 (models 모듈의 field 클래스) : DB 필드의 데이터 타입

## 2.2.3 Django Model Filed

- Django는 모델 필드를 통해 테이블의 필드(컬럼)에 저장할 데이터 유형을 정의
- CharField(max\_length=None, \*\*options)
  - 길이의 제한이 있는 문자열을 넣을 때 사용
  - max\_length
    - 필드의 최대 길이(문자)
    - CharField의 필수 인자
    - 데이터베이스와 Django의 유효성 검사(값을 검증하는것)에서 활용
- TextField(\*\*options)
  - 글자의 수가 많을 때 사용
  - max\_length 옵션 작성 시 사용자 입력 단계에서는 반영 되지만,  
모델과 데이터베이스 단계에는 적용되지 않음(CharFiled를 사용해야 함)
    - 실제로 저장될 때 길이에 대한 유효성을 검증하지 않음

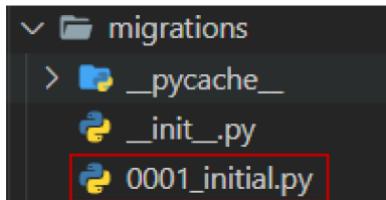
## 2.3 Migrations

- 지금까지 작성한 models.py는 다음과 같은 데이터베이스 스키마를 설계한 것
- 이제 데이터베이스에 테이블을 생성하기 위한 설계도 작성 이 필요함
- 모델에 대한 청사진을 만들고 이를 통해 테이블을 생성하는 일련의 과정
- Django가 모델에 생긴 변화(필드 추가, 수정 등)를 실제 DB에 반영하는 방법

## 2.3.1 Migrations 관련 주요 명령어

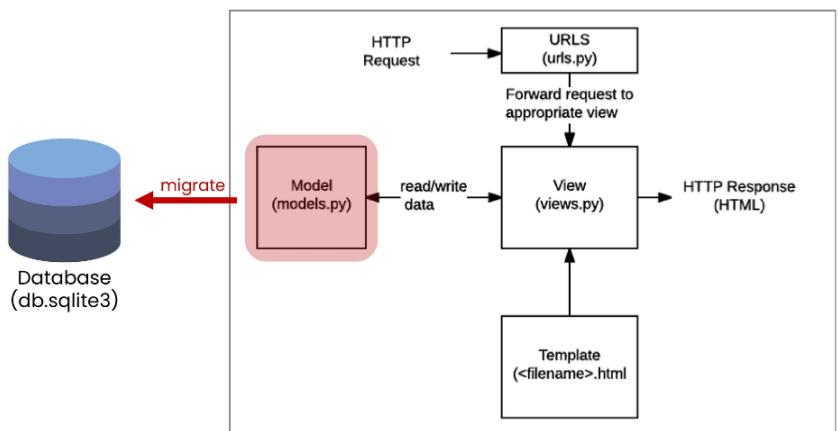
- `makemigrations`

- 모델을 작성 혹은 변경한 것에 기반한 새로운 migration을 만들 때 사용
- `$ python manage.py makemigrations`
- 파이썬으로 작성된 설계도



- 마이그레이션 파일은 앞에 0001 같은 숫자 4자리가 붙어있음
- 앱\_클래스 이름으로 데이터베이스 만듦

- `migrate`



- `makemigrations`로 만든 설계도를 실제 데이터베이스(db.sqlite3 DB)에 반영 하는 과정

- 결과적으로 모델에서의 변경사항들과 DB의 스키마가 동기화를 이룸  
→ 모델과 DB의 동기화

- `$ python manage.py migrate`

- 클래스 작성 > 설계도 생성(마이그레이션) > DB에 반영(마이그레이트)

- 기타 명령어

- `showmigrations`

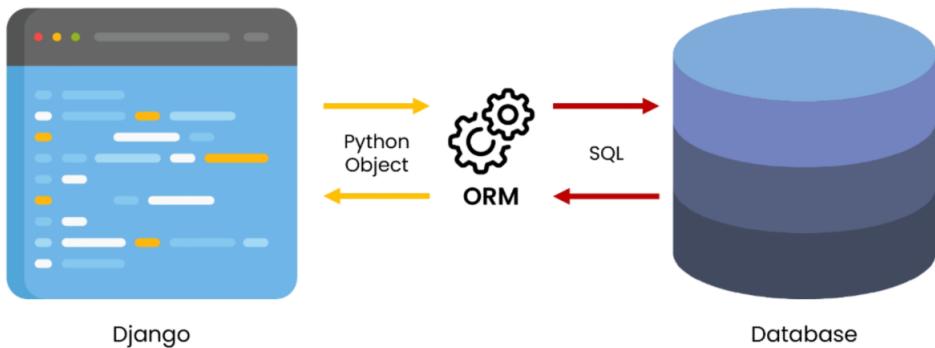
- migrations 파일들이 migrate 됐는지 안됐는지 여부를 확인하는 용도
- [X]표시가 있으면 migrate가 완료되었음을 의미
- `$ python manage.py showmigrations`

- `sqlmigrate`

- 해당 migrations 파일이 SQL 문으로 어떻게 해석 될 지 미리 확인할 수 있음
- `$ python manage.py sqlmigrate articles 0001`

- ORM (설계도 해석)
  - makemigrations로 인해 만들어진 설계도는 파이썬으로 작성
  - DB는 SQL만 알아들을 수 있기 때문에 이 과정에서 중간에 해석을 담당하는 것이 ORM

### 2.3.2 ORM(Object-Relational-Mapping)



- 객체 지향 프로그래밍에서 데이터베이스를 연동할 때, 데이터베이스와 객체 지향 프로그래밍 언어 간의 ( $\text{Django} \leftrightarrow \text{SQL}$ ) 호환되지 않는 데이터를 변환하는 프로그래밍 기법
- Django는 내장 Django ORM을 사용
- ORM 장단점
  - 장점
    - SQL을 잘 알지 못해도 객체지향 언어로 DB 조작이 가능
    - 객체 지향적 접근으로 인한 높은 생산성
  - 단점
    - ORM만으로 세밀한 데이터베이스 조작을 구현하기 어려운 경우가 있음
- ORM을 사용해야하는 이유
  - 생산성
  - 현시대 개발에서 가장 중요한 키워드는 바로 생산성
  - 우리는 DB를 객체(object)로 조작하기 위해 ORM을 사용할 것

## 2.4 추가필드정의

### 2.4.1 Model 변경사항 반영하기

- models.py에 변경 사항이 생겼다면 추가 모델 필드 작성 후 다시 한번 makemigrations 진행
- 새로운 설계도가 만들어 진 것을 확인 후 DB와 동기화를 진행해야 함
- 0002 설계도는 dependencies = [0001]
  - 0001번 설계도에 의존성이 있음(1번 설계도에 추가한거여서)
  - 변경사항을 쌓아나가는 느낌

## 2.4.2 migration 3 단계

- models.py에서 변경사항이 발생하면
- `makemigrations` → migration 생성
- `migrate` → DB 반영 (모델과 DB의 동기화)

## 2.4.3 `DateTimeField()`

- python의 `datetime.datetime` 인스턴스로 표시되는 날짜 및 시간을 값으로 사용하는 필드
- `DateField`를 상속받는 클래스
- 선택인자 (문제.....!)
  - `auto_now_add`
    - 최초 생성 일자
    - 데이터가 실제로 만들어질 때 현재 날짜와 시간으로 자동으로 초기화 되도록 함
  - `auto_now`
    - 최종 수정 일자
    - 데이터가 수정될 때마다 현재 날짜와 시간을 자동으로 갱신되도록 함

## 3. QuerySet API

- Django shell
  - ORM 관련 구문 연습을 위해 파이썬 쉘 환경을 사용
  - `python manage.py shell_plus`

```
$ pip install ipython
$ pip install django-extensions
# setting.py에 django_extensions 등록
```

- IPython & django-extensions
  - iPython
    - 파이썬 기본 쉘보다 더 강력한 파이썬 쉘
    - django-extensions
  - django-extensions
    - Django 확장 프로그램 모음
    - shell\_plus, graph model 등 다양핚 확장 기능 제공

- Shell
  - 운영체제 상에서 다양한 기능과 서비스를 구현하는 인터페이스를 제공하는 프로그램
  - 셸(껍데기)은 사용자와 운영 체제의 내부 사이의 인터페이스를 감싸는 층
  - 사용자 ↔ 셸 ↔ 운영체제
- Python Shell
  - 파이썬 코드를 실행해주는 인터프리터
    - 인터프리터 : 코드를 한 줄 씩 읽어 내려가며 실행하는 프로그램
  - 인터렉티브 혹은 대화형 shell이라고 부름
  - Python 명령어를 실행하여 그 결과를 바로 제공

### 3.1 Database API

- Django가 기본적으로 ORM을 제공함에 따른 것으로 DB를 편하게 조작할 수 있게 도움
- Model을 만들면 Django는 객체들을 만들고 읽고 수정하고 지울 수 있는 DB API를 자동으로 생성
- Database API 구문

**Article.objects.all()**

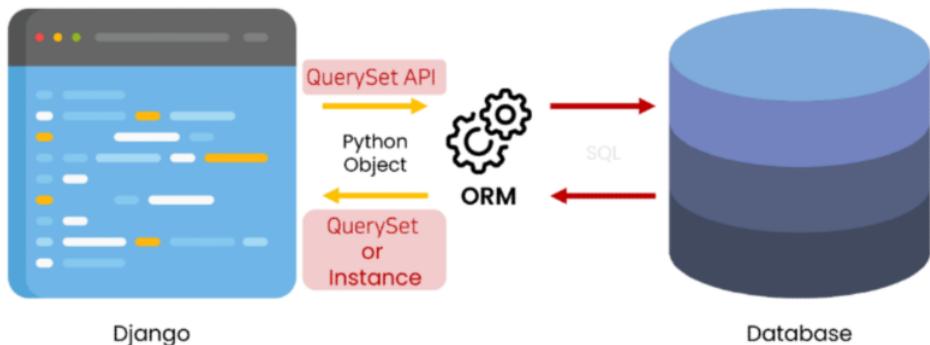
Model class

Manager

Queryset API

- `objects.manager`
  - Django 모델이 데이터베이스 쿼리 작업을 가능하게 하는 인터페이스
  - Django는 기본적으로 모든 모델 클래스에 대해 `objects`라는 Manager 객체를 자동으로 추가
  - Manager(objects)를 통해 특정 데이터를 조작(메서드)할 수 있음
  - DB를 Python class로 조작할 수 있도록 여러 메서드를 제공하는 `manager`
- `Query`
  - 데이터베이스에 특정한 데이터를 보여 달라는 요청
    - 쿼리문을 작성한다  
→ 원하는 데이터를 얻기 위해 데이터베이스에 요청을 보낼 코드를 작성한다
  - 이때, 파이썬으로 작성한 코드가 ORM에 의해 SQL로 변환되어 데이터베이스에 전달되며, 데이터베이스의 응답 데이터를 ORM이 `Queryset`이라는 자료 형태로 변환하여 우리에게 전달

- **QuerySet**
  - 데이터베이스에게서 전달 받은 객체 목록(데이터 모음)
    - 순회가 가능한 데이터로써 1개 이상의 데이터를 불러와 사용할 수 있음
  - Django ORM을 통해 만들어진 자료형이며, 필터를 걸거나 정렬 등을 수행 가능
  - 복수의 데이터를 가져오는 queryset method를 사용할 때 반환되는 객체
  - **단일한 객체를 반환** 할 때는 QuerySet이 아닌 **모델(class)의 인스턴스**로 반환 (시험)
- **QuerySet API**
  - QuerySet과 상호작용하기 위해 사용하는 도구 (메서드, 연산자 등)



## 3.2 QuerySet API 익히기

- **CRUD**
  - Create / Read / Update / Delete
    - 생성 / 조회 / 수정 / 삭제
  - 대부분의 컴퓨터 소프트웨어가 가지는 기본적인 데이터 처리 가능 4가지를 묶어서 일컫는 말

### 3.2.1 CREATE

#### 1. 데이터 생성 첫번째 방법

- **article = Article()**
  - 클래스를 통한 인스턴스 생성
- **article.title**
  - 클래스 변수명과 같은 이름의 인스턴스 변수를 생성 후 값을 할당
- **article.save**
  - 인스턴스로 save 메소드 호출

```

In [2]: article = Article() # Article(class)로 부터
article(instance)
In [3]: article
Out[3]: <Article: Article object (None)>

In [4]: article.title = 'first'
In [5]: article.content = 'django!'

# save를 하지 않으면 아직 DB에 값이 저장되지 않음
In [6]: article.save()

In [7]: article
Out[7]: <Article: Article object (1)>
        # save 될 때 id 부여

```

## 2. 인스턴스 생성 시 초기 값을 함께 작성하여 생성

- `aritcle = Article(title=title, content=content)`
- `article.save()`

## 3. QuerySet API 중 `create()` 활용

- `Article.objects.create(title='third'`  
`content='django!')`
- `.save()`
  - 객체를 데이터베이스에 저장함
  - 데이터 생성 시 save를 호출하기 전에는 객체의 id 값은 None
    - id 값은 Django가 아니라 데이터베이스에서 계산되기 때문에
  - 단순히 모델 클래스를 통해 인스턴스를 생성하는 것은 DB에 영향을 미치지 않기 때문에 반드시 save를 호출해야 테이블에 레코드가 생성됨
  - 1번 삭제후 2,3번이 남아있는 상태에서 값을 저장하면 4번에 저장

### 3.2.2 READ

- 개요
  - QuerySet API method를 사용해 데이터를 다양하게 조회하기
  - QuerySet API method는 크게 2가지로 분류됨
    - Method that 'return new querysets'
    - Method that 'do not return querysets'

- `all()`
  - `Article.objects.all()`
  - QuerySet return
  - 전체 데이터 조회
- `get()`
  - `Article.objects.get(pk=1)`
  - 단일 데이터 조회
  - 객체를 찾을 수 없으면 DoesNotExist 예외를 발생시키고,  
둘 이상의 객체를 찾으면 MultipleObjectsReturned 예외를 발생시킴
  - primary key와 같이 고유성(uniqueness)을 보장하는 조회에서 사용
- `filter()`
  - `Article.objects.filter(content='django!')`
  - 지정된 조회 매개 변수와 일치하는 객체를 포함하는 새 QuerySet을 반환
  - 조회된 객체가 없거나 1개여도 QuerySet 반환
  - pk 값을 조회할 때 적합하지 않음
    - 쿼리셋으로 줘서 한 겹 더 벗겨내야함
    - 값이 존재하지 않아도 예외를 발생시키지 않고 빈 쿼리셋을 반환
- Field lookups
  - `Article.objects.filter(content__contains='dj')`
  - 특정 레코드에 대한 조건을 설정하는 방법
  - QuerySet 메서드 filter(), exclude() 및 get()에 대한 키워드 인자로 지정됨

### 3.2.3 UPDATE

- 수정하고자 하는 article 인스턴스 객체를 조회 후 변환 값을 저장
  - 먼저 조회를 한 후 수정해야함!
- article 인스턴스 객체의 인스턴스 변수 값을 새로운 값으로 할당
- save() 인스턴스 메서드 호출

```
article = Article.objects.get(pk=1)      # 인스턴스 객체 조회

article.title = 'byebye'      # 인스턴스 변수를 변경
article.save()                # 저장
```

### 3.2.4 DELETE

- 삭제하고자 하는 article 인스턴스 객체를 조회 후 반환 값을 저장
- delete() 인스턴스 메서드 호출

```
article = Article.objects.get(pk=1)      # 인스턴스 객체 조회  
article.delete()                      # 인스턴스 객체 삭제
```

[참고] `__str__()`

- 표준 파이썬 클래스의 메서드인 str()을 정의하여 각각의 object가 사람이 읽을 수 있는 문자열을 반환(return)하도록 할 수 있음
- DB설계도에 아무런 변경 사항도 일으키지 않음 어떠한 구조에도 영향X

```
# models.py  
  
def __str__(self):  
    return self.title
```

## 4. CRUD with view functions

- 사전 준비
  - bootstrap CDN 및 템플릿 추가 경로 작성

```
<!-- templates/base.html -->  
  
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta http-equiv="X-UA-Compatible" content="IE=edge">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <!-- bootstrap CSS CDN -->  
    <title>Document</title>  
</head>  
<body>  
    <div class="container">  
        {<% block content %>  
        {<% endblock content %>}  
    </div>  
    <!-- bootstrap JS CDN -->  
    </body>  
</html>
```

```
# settings.py  
  
'TEMPLATES' = [  
    {  
        ...  
        'DIRS': [BASE_DIR / 'templates'],  
        ...  
    }  
]
```

- DIRS : [BASE\_DIR / 'templates'] 이거 시험 나왔었음
- URL 분리 및 연결

```
# articles/urls.py  
  
from django.urls import path  
  
app_name = 'articles'  
urlpatterns = [  
]
```

```
# crud/urls.py  
  
from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('articles/', include('articles.urls')),  
]
```

- index 페이지 작성

```
# articles/urls.py

from django.urls import path
from . import views

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

```
# articles/views.py

def index(request):
    return render(request, 'articles/index.html')
```

```
<!-- templates/articles/index.html -->

{% extends 'base.html' %}

{% block content %}
<h1>Articles</h1>
{% endblock content %}
```

## 4.1 READ 1 (index page)

- index 페이지에서는 전체 게시글을 조회해서 출력

```
# articles/views.py

from .models import Article

def index(request):
    articles = Article.objects.all()
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index.html', context)
```

```
<!--templates/articles/index.html-->

{% extends 'base.html' %}

{% block content %}
<h1>Articles</h1>
<hr>
{% for article in articles %}
<p>글 번호: {{ article.pk }}</p>
<p>글 제목: {{ article.title }}</p>
<p>글 내용: {{ article.content }}</p>
<hr>
{% endfor %}
{% endblock content %}
```

## 4.2 CREATE

- CREATE 로직을 구현하기 위해서는 2개의 함수가 필요

- **new** 사용자의 입력을 받을 페이지를 렌더링 하는 함수
- **create** 사용자가 입력한 데이터를 전송 받아 DB에 저장하는 함수

### 4.2.1 new

```
# articles/urls.py

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

```
# articles/views.py

def new(request):
    return render(request, 'articles/new.html')
```

```
<!-- templates/articles/new.html -->

{% extends 'base.html' %}

{% block content %}
<h1>NEW</h1>
<form action="#" method="GET">
<label for="title">Title: </label>
<input type="text" name="title"><br>
<label for="content">Content: </label>
<textarea name="content"></textarea><br>
<input type="submit">
</form>
<hr>
<a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

- index에서 new 페이지로 이동할 수 있는 하이퍼링크 작성

```
<!-- templates/articles/index.html -->

{% extends 'base.html' %}

{% block content %}
    <h1>Articles</h1>
    <a href="{% url 'articles:new' %}">NEW</a>
    <hr>
    ...
{% endblock content %}
```

#### 4.2.2 create

- index 작성 및 데이터 생성

```
# articles/urls.py

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

```
def create(request):
    title = request.GET.get('title')
    content = request.GET.get('content')

    # 1.
    # article = Article()
    # article.title = title
    # article.content = content
    # article.save()

    # 2.
    article = Article(title=title, content=content)
    article.save()

    # 3.
    # Article.objects.create(title=title, content=content)

    return render(request, 'articles/create.html')
```

- 2번째 생성 방식을 사용하는 이유

- create 메서드가 간단해 보이지만 추후 데이터가 저장되기 전에 유효성 검사 과정을 거치게 될 예정
- 유효성 검사가 진행된 후에 save 메서드가 호출되는 구조를 택하기 위함

- 게시글 작성 후 확인

```
<!-- templates/articles/new.html -->

{% extends 'base.html' %}

{% block content %}
<h1>NEW</h1>
<form action="{% url 'articles:create' %}" method="GET">
<label for="title">Title: </label>
<input type="text" name="title"><br>
<label for="content">Content: </label>
<textarea name="content" cols="30" rows="5"></textarea><br>
<input type="submit">
</form>
<hr>
<a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

- 게시글 작성 후 index 페이지로 돌아가도록 함

```
# articles/views.py

def create(request):
...
    return render(request, 'articles/index.html')
```

- 2가지 문제적 발생

- 게시글 작성 후 index 페이지가 출력되지만 게시글을 조회되지 않음
  - create 함수에서 index.html 문서를 렌더링할 때 context 데이터와 함께 렌더링 하지 않았기 때문에
  - index 페이지 url로 다시 요청을 보내면 정상적으로 출력됨
- 게시글 작성 후 URL은 여전히 create에 머물러 있음
  - index view 함수를 통해 렌더링 된 것이 아니기 때문에
  - index view 함수의 반환 값이 아닌 단순히 index 페이지만 render 되었을 뿐

- `redirect()`

- 인자에 작성된 곳으로 요청을 보냄
- 사용 가능한 인자
  - view name (URL pattern name)

```
return redirect('articlesIndex')
```

- absolute or relative URL

```
return redirect('/articles/')
```

- 동작원리

- 클라이언트가 create url로 요청을 보냄
- create view 함수의 redirect 함수가 302 status code를 응답
- 응답 받은 브라우저는 redirect 인자에 담긴 주소(index)로 사용자를 이동시키기 위해 index url로 Django에 재요청
- index page를 정상적으로 응답 받음(200 status code)

- [참고] 302 Found
  - HTTP response status code 중 하나
  - 해당 상태 코드를 응답 받으면 브라우저는 사용자를 해당 URL 페이지로 이동시킴

### 4.2.3 HTTP response status code

- 클라이언트에게 특정 HTTP 요청이 성공적으로 완료되었는지 여부를 알려줌
- 응답은 5개의 그룹으로 나뉘어짐
  - Informational response (1xx)
  - Successful responses (2xx)
  - Redirection messages (3xx)
  - Client error responses (4xx)
  - Server error responses (5xx)

### 4.2.4 HTTP request method

- **GET**
  - 특정 리소스를 가져오도록 요청할 때 사용
  - 반드시 데이터를 가져올 때만 사용해야 함
  - DB에 영향을 주지 않음
  - CRUD에서 R 역할을 함
- **POST**
  - 서버로 데이터를 전송할 때 사용
  - 서버에 변경사항을 만듦
  - 리소스를 생성/변경하기 위해 데이터를 HTTP body에 담아 전송
  - GET의 쿼리 스트링 파라미터와 다르게 URL로 보내지지 않음
  - CRUD에서 C/U/D 역할을 담당
- 그럼 왜 검색에서는 GET을 사용할까?
  - 검색은 서버에 영향을 미치는 것이 아닌 특정 데이터를 조회만 하는 요청이기 때문
  - 특정 페이지를 조회하는 요청을 보내는 HTML의 a tag 또한 GET을 사용

→ GET은 단순히 조회하려는 경우 & POST는 서버나 DB에 변경을 요청하는 경우

### 4.2.5 CSRF(Cross Site REquest Forgery)

- 사이트 간 요청 위조
- 사용자가 자신의 의지와 무관하게 공격자가 의도한 행동을 하여 특정 웹페이지를 보안에 취약하게하거나 수정, 삭제 등의 작업을 하게 만드는 공격 방법

- CSRF 공격 방어
  - Security Token 사용 방식 (CSRF Token)
  - 사용자의 데이터에 임의의 난수 값(token)을 부여해 매 요청마다 해당 난수 값을 포함시켜 전송
  - 이후 서버에서 요청을 받을 때마다 전달되어 token 값이 유효한지 검증
  - 일반적으로 데이터 변경 가능한 POST, PATCH, DELETE Method 등에 적용
  - Django에서는 DTL에서 csrf\_token 템플릿 태그를 제공
- {% csrf\_token %}
  - 해당 태그가 없다면 Django 서버는 요청에 대해 403 forbidden으로 응답
  - 템플릿에서 내부 URL로 향하는 Post form을 사용하는 경우에 사용
    - 외부 URL로 향하는 POST form에 대해서는 CSRF 토큰이 유출되어 취약성을 유발할 수 있기 때문에 사용해서는 안됨
  - [참고] 403 Forbidden
    - 서버에 요청이 전달되었지만, 권한 때문에 거절되었다는 것을 의미
    - 서버에 요청은 도달했으나 서버가 접근을 거부할 때 반환
    - 모델(DB)을 조작하는 것은 단순 조회와 달리 최소한의 신원 확인이 필요

## 4.3 READ 2 (detail page)

- 개별 게시글 상세 페이지 제작
- 모든 게시글마다 뷰 함수와 템플릿 파일을 만들수는 없음
  - 글의 번호(pk)를 활용해서 하나의 뷰 함수와 템플릿 파일로 대응
- 무엇을 활용할 수 있을까?
  - Variable Routing

### 4.3.1 작성

- urls
  - URL로 특정 게시글을 조회할 수 있는 번호를 받음

```
# articles/urls.py

urlpatterns = [
    ...
    path('<int:pk>', views.detail, name='detail'),
```

- templates

```
<!-- templates/articles/detail.html -->
{% extends 'base.html' %}

{% block content %}
    <h2>DETAIL</h2>
    <h3>{{ article.pk }} 번째 글</h3>
    <hr>
    <p>제목: {{ article.title }}</p>
    <p>내용: {{ article.content }}</p>
    <p>작성 시각: {{ article.created_at }}</p>
    <p>수정 시각: {{ article.updated_at }}</p>
    <hr>
    <a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

```
<!-- templates/articles/index.html -->
{% extends 'base.html' %}

{% block content %}
    <h1>Articles</h1>
    <a href="{% url 'articles:new' %}">[new]</a>
    <hr>
    {% for article in articles %}
        ...
        <a href="{% url 'articles:detail' article.pk %}">[detail]</a>
        <hr>
    {% endfor %}
    {% endblock content %}
```

- views.py

- redirect 인자 변경

```
# articles/views.py

def create(request):
    ...
    return redirect('articles:detail', article.pk)
```

## 4.4 DELETE

### 4.4.1 작성

- urls

- 모든 글을 삭제하는 것이 아니라 삭제하고자 하는 특정 글을 조회 후 삭제해야 함

```
# articles/urls.py

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

- views

```
# articles/views.py

def delete(request, pk):
    article = Article.objects.get(pk=pk)
    article.delete()
    return redirect('articles:index')
```

- templates

- Detail 페이지에 작성하여 DB에 영향을 미치기 때문에 POST method 사용

```
<!-- articles/detail.html -->
{% extends 'base.html' %}

{% block content %}
    ...
    <form action="{% url 'articles:delete' article.pk %}" method="POST">
        {% csrf_token %}
        <input type="submit" value="DELETE">
    </form>
    <a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

## 4.5 UPDATE

- 수정은 CREATE 로직과 마찬가지로 2개의 view 함수가 필요
  - `edit` 사용자의 입력을 받을 페이지를 렌더링 하는 함수
  - `update` 사용자가 입력한 데이터를 전송 받아 DB에 저장하는 함수

### 4.5.1 Edit

- urls & views

```
# articles/urls.py
urlpatterns = [
    ...
    path('<int:pk>/edit/', views.edit, name='edit'),
]
```

```
# articles/views.py
def edit(request, pk):
    article = Article.objects.get(pk=pk)
    context = {
        'article': article,
    }
    return render(request, 'articles/edit.html', context)
```

- templates

- html 태그의 value 속성을 사용해 기존에 입력 되어 있던 데이터를 출력

```
<!-- articles/edit.html -->

{% extends 'base.html' %}

{% block content %}
    <h1>EDIT</h1>
    <form action="#" method="POST">
        {% csrf_token %}
        <label for="title">Title: </label>
        <input type="text" name="title" value="{{ article.title }}"><br>
        <label for="content">Content: </label>
        <textarea name="content" cols="30" rows="5">{{ article.content }}</textarea><br>
        <input type="submit">
    </form>
    <hr>
    <a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

- textarea 태그는 value 속성이 없으므로 태그 내부 값으로 작성해야 함
- Edit 페이지로 이동하기 위한 하이퍼링크 작성

```
<!-- articles/detail.html -->

{% extends 'base.html' %}

{% block content %}
    <h2>DETAIL</h2>
    <h3>{{ article.pk }} 번째 글</h3>
    <hr>
    <p>제목: {{ article.title }}</p>
    <p>내용: {{ article.content }}</p>
    <p>작성 시각: {{ article.created_at }}</p>
    <p>수정 시각: {{ article.updated_at }}</p>
    <hr>
    <a href="{% url 'articles:edit' article.pk %}">EDIT</a><br>
    <form action="{% url 'articles:delete' article.pk %}" method="POST">
        {% csrf_token %}
        <input type="submit" value="DELETE">
    </form>
    <a href="{% url 'articles:index' %}">[back]</a>
{% endblock %}
```

## 4.5.2 Update 로직 작성

```
# articles/urls.py
urlpatterns = [
    ...
    path('<int:pk>/update/', views.update, name='update'),
]
```

```
# articles/views.py
def update(request, pk):
    article = Article.objects.get(pk=pk)
    article.title = request.POST.get('title')
    article.content = request.POST.get('content')
    article.save()
    return redirect('articles:detail', article.pk)
```

```
<!-- articles/edit.html -->
{% extends 'base.html' %}

{% block content %}
<h1>EDIT</h1>
<form action="{% url 'articles:update' article.pk %}" method="POST">
    {% csrf_token %}
    ...
    <a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

170

## 5. Admin site

- Django의 가장 강력한 기능 중 하나인 automatic admin interface 알아보기
- 관리자 페이지
  - 사용자가 아닌 서버의 관리자가 활용하기 위한 페이지
  - 모델 class를 admin.py에 등록하고 관리
  - 레코드 생성 여부 확인에 매우 유용하며 직접 레코드를 삽입할 수도 있음

### 5.1 admin 계정 생성

- `$ python manage.py createsuperuser`
- username과 password를 입력해 관리자 계정을 생성

### 5.2 admin에 모델 클래스 등록

- 모델의 record를 보기 위해서는 admin.py에 등록 필요
- `admin.site.register(Article)` 시험!!

```
# articles/admin.py

from django.contrib import admin
from .models import Article

admin.site.register(Article)
```

## 6. 마무리

- Model
  - Django는 Model을 통해 데이터에 접속하고 관리
- ORM
  - 객체지향 프로그래밍을 이용한 DB 조작
- Migrations
  - 모델에 생긴 변화(필드 추가, 모델 삭제 등)를 DB에 반영하는 방법(과정)
- HTTP request & response
  - 요청에 행동을 표현하는 HTTP request method
  - 요청에 대한 성공 여부 응답을 숫자로 표현하는 HTTP response status codes