



Static/Media

📅 강의날짜	@2022/10/11
🕒 작성일시	@2022년 10월 11일 오전 11:39
🕒 편집일시	@2022년 10월 11일 오후 4:12
▼ 분야	django
▼ 공부유형	강의
☑ 복습	<input type="checkbox"/>
☰ 태그	Image Upload Managing static files QuerySet API Advanced

Managing static files

개요

- 개발자가 서버에 미리 준비한 혹은 사용자가 업로드한 정적파일을 클라이언트에게 제공하는 방법

Static files

정적 파일

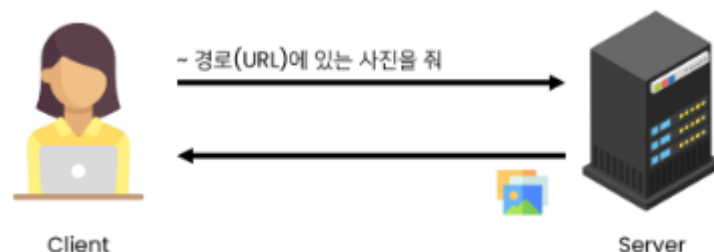
- 응답할 때 별도의 처리 없이 파일 내용을 그대로 보여주면 되는 파일
 - 사용자의 요청에 따라 내용이 바뀌는 것이 아니라 요청한 것을 그대로 보여주는 파일
- 파일 자체가 고정되어있고, 서비스 중에도 추가되거나 변경되지 않고 고정 되어있음

- 예를 들어, 웹 사이트는 일반적으로 이미지, 자바 스크립트 또는 CSS와 같은 미리 준비된 추가 파일(움직이지 않는)을 제공해야 함
- Django에서는 이러한 파일들을 **static file**이라 함
 - Django는 `staticfiles` 앱을 통해 정적 파일과 관련된 기능을 제공

Media File

- 미디어 파일
- 사용자가 웹에서 업로드하는 정적 파일(user-uploaded)
- 유저가 업로드한 모든 정적 파일

웹 서버와 정적파일



- 이는 **자원과 자원에 접근 가능한 주소가 있다**라는 의미
 - 예를 들어, 사진 파일은 자원이고 해당 **사진 파일을 얻기 위한 경로인 웹 주소(URL)가 존재함**
- 즉, 웹 서버는 요청 받은 URL로 서버에 존재하는 정적 자원(Static resource)를 제공함

Static files 구성하기

Django에서 정적파일을 구성하고 사용하기 위한 몇 가지 단계

1. `INSTALLED_APPS` 에 `django.contrib.staticfiles` 가 포함되어 있는지 확인하기
2. `settings.py` 에서 `STATIC_URL` 을 정의하기
3. 앱의 `static` 폴더에 정적 파일을 위치하기
 - a. 예시) `my_app/static/sample_img.jpg`
4. 템플릿에서 static 템플릿 태그를 사용하여 지정된 경로에 있는 정적 파일의 URL 만들기

```
{% load static %}

```

Django template tag

```
{% load %}
```

- load tag
- 특정 라이브러리, 패키지에 등록된 모든 템플릿 태그와 필터를 로드

```
{% static ' ' %}
```

- static tag
- `STATIC_ROOT` 에 저장된 정적 파일에 연결

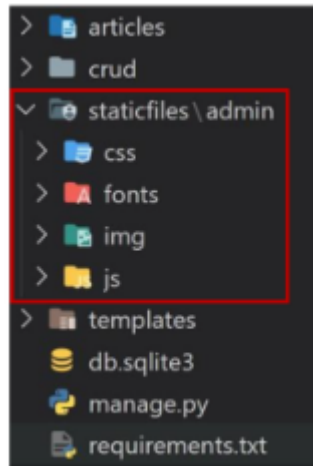
Static files 관련 Core Settings

1. `STATIC_ROOT`
 - a. Default : None
 - b. Django 프로젝트에서 사용하는 모든 정적 파일을 한곳에 모아 넣는 경로
 - c. `collectstatic` 이 배포를 위해 정적 파일을 수집하는 디렉토리의 절대 경로
 - d. 개발 과정에서 `setting.py` 의 `DEBUG` 값이 `True`로 설정되어 있으면 해당 값은 작용되지 않음
 - e. 실 서비스 환경(배포 환경)에서 Django의 모든 정적 파일을 다른 웹 서버가 직접 제공하기 위해 사용
 - f. 배포 환경에서는 Django를 직접 실행하는 것이 아니라, 다른 서버에 의해 실행되기 때문에 실행하는 다른 서버는 Django에 내장되어 있는 정적 파일들을 인식하지 못함(내장되어 있는 정적 파일들을 밖으로 꺼내는 이유)
- ✓ `collectstatic`
- `STATIC_ROOT` 에 Django 프로젝트의 모든 정적 파일을 수집

```
# settings.py
STATIC_ROOT = BASE_DIR / 'staticfiles'
```

```
$ python manage.py collectstatic
```

❖ 결과를 확인하고 수집된 정적파일을 모두 삭제한다.



2. `STATICFILES_DIRS`

- Default : [] (Empty list)
- `app/static/` 디렉토리 경로를 사용하는 것 (기본 경로)외에 추가적인 정적 파일 경로 목록을 정의하는 리스트
- 추가 파일 디렉토리에 대한 전체 경로를 포함하는 문자열 목록으로 작성되어야 함

```
# 작성 예시
STATICFILES_DIRS = [
    BASE_DIR / 'static',
]
```

3. `STATIC_URL`

- Default : None
- `STATIC_ROOT` 에 있는 정적 파일을 참조할 때 사용할 URL
- 개발 단계에서는 실제 정적 파일들이 저장되어 있는 `app/static/` 경로(기본 경로) 및 `STATICFILES_DIRS` 에 정의된 추가 경로들을 탐색
- 실제 파일이나 디렉토리가 아니며, URL로만 존재

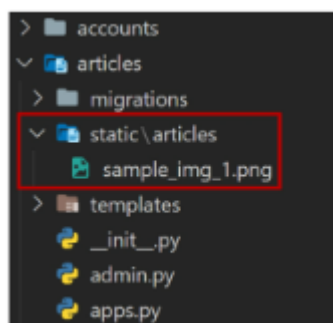
e. 비어있지 않은 값으로 설정한다면 반드시 slash(/)로 끝나야 함

```
# 작성 예시
STATIC_URL = '/static/'
```

Static files 사용하기

기본 경로에 있는 static file 가져오기

1. `articles/static/articles` 경로에 이미지 파일 배치하기



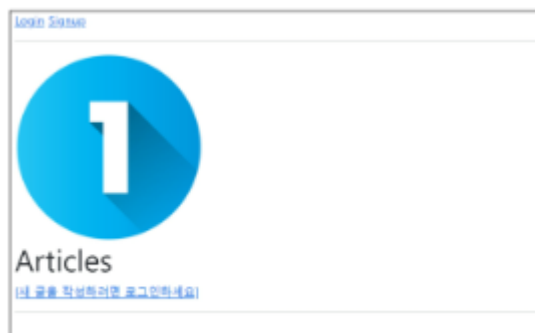
2. static tag를 사용해 이미지 파일 출력하기

```
<!-- articles/index.html -->
{% extends 'base.html' %}
{% load static %}
{% block content %}

<h1>Articles</h1>
...

```

3. 이미지 출력 확인

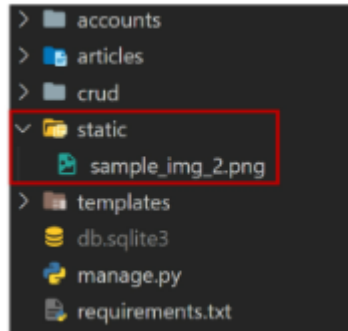


추가 경로에 있는 static file 가져오기

1. 추가 경로 작성

```
# setting.py
STATICFILES_DIRS = [
    BASE_DIR / 'static',
]
```

2. `static/` 경로에 이미지 파일 배치하기



3. static tag를 사용해 이미지 파일 출력하기

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

{% block content %}
    
    
    <h1>Articles</h1>
...

```

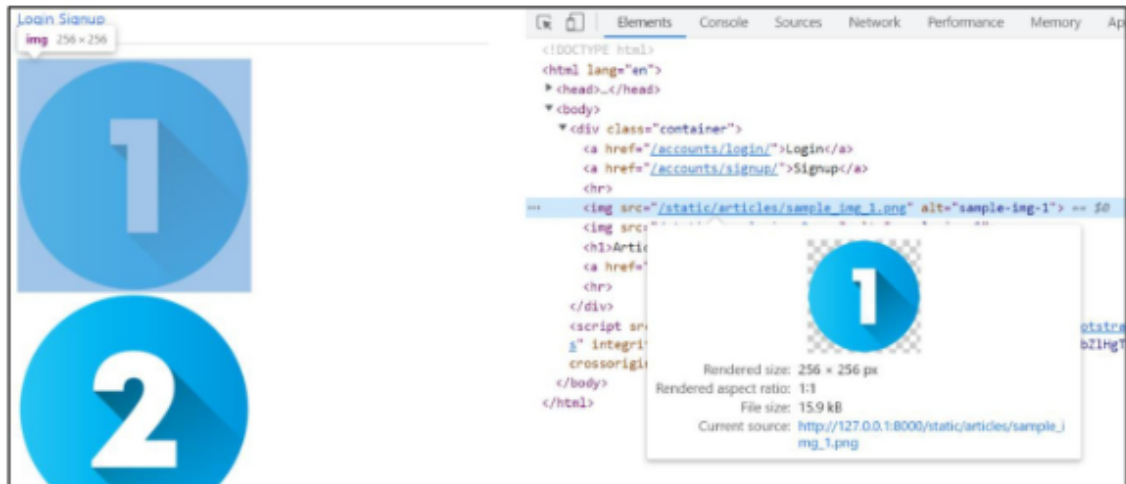
4. 이미지 출력 확인



STATIC_URL 확인하기

- Django가 해당 이미지를 클라이언트에게 응답하기 위해 만든 image url 확인하기

- 개발자 도구- inspect 버튼을 통해 확인
- **STATIC_URL + static file 경로로 설정됨**
 - `http://127.0.0.1:8000 /static/ articles/sample_img_1.png`



- 개발자 도구 - Network에서 Request URL 확인해보기
 - 클라이언트에게 이미지를 응답하기 위한 요청 URL을 만든 것

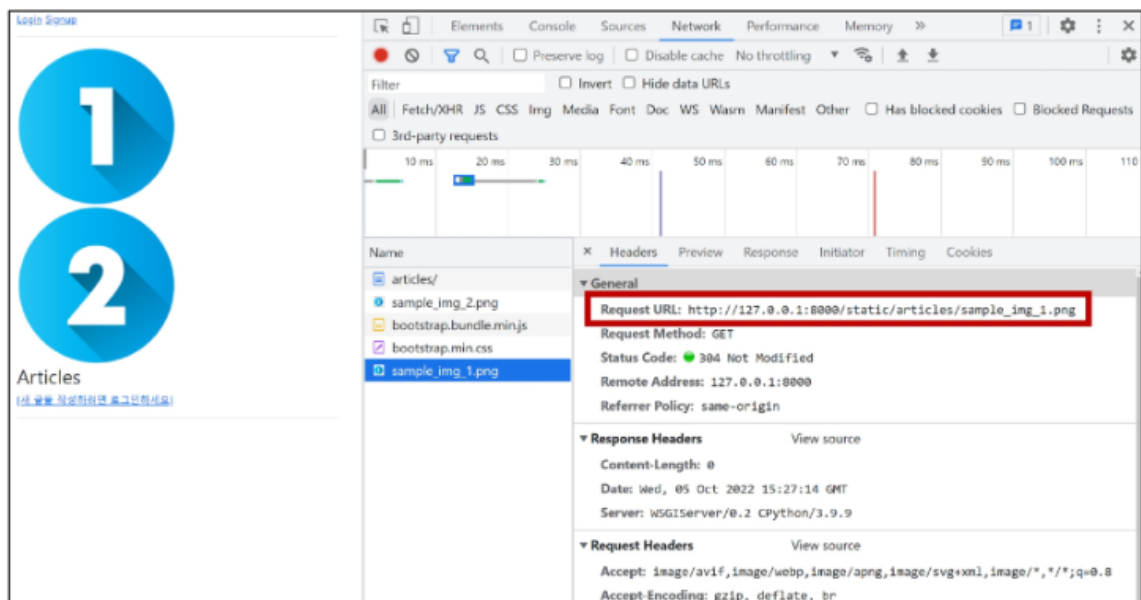


Image Upload

개념

- Django ImageField를 사용해 사용자가 업로드한 정적 파일(미디어 파일) 관리하기

ImageField

ImageField()

- 이미지 업로드에 사용하는 모델 필드
- FileField를 상속받는 서브 클래스이기 때문에 FileField의 모든 속성 및 메서드를 사용 가능
- 더해서 사용자에게 의해 업로드된 객체가 유효한 이미지인지 검사
- ImageField 인스턴스는 최대 길이가 100자인 문자열로 DB에 생성되며, `max_length` 인자를 사용하여 최대 길이를 변경할 수 있음

FileField()

- `FileField(upload_to='', storage=None, max_length=100, **options)`
- 파일 업로드에 사용하는 모델 필드
- 2개의 선택인자를 가지고 있음
 - `upload_to`
 - `storage`

FileField / ImageField를 사용하기 위한 단계

1. `setting.py` 에 `MEDIA_ROOT`, `MEDIA_URL` 설정
2. `upload_to` 속성을 정의하여 업로드된 파일에 사용할 `MEDIA_ROOT`의 하위 경로를 지정(선택사항)

MEDIA_ROOT

- Default: "" (Empty string)
- 사용자가 업로드한 파일(미디어 파일)들을 보관할 디렉토리의 절대 경로
- Django는 성능을 위해 업로드 파일은 데이터 베이스에 저장하지 않음
 - 데이터베이스에 저장되는 것은 **파일 경로**
- `MEDIA_ROOT` 는 `STATIC_ROOT` 와 반드시 다른 경로로 지정해야함

```
# settings.py
MEDIA_ROOT = BASE_DIR / 'media'
```


MEDIA_URL

- Default: "" (Empty string)
- `MEDIA_ROOT` 에서 제공되는 미디어 파일을 처리하는 URL
- 업로드된 파일의 주소(URL)를 만들어 주는 역할
 - 웹 서버 사용자가 사용하는 public URL
- 비어 있지 않은 값으로 설정한다면 반드시 slash(/)로 끝나야함
- `MEDIA_URL` 은 `STATIC_URL` 과 반드시 다른 경로로 지정해야함

```
# settings.py
MEDIA_URL = '/media/'
```

개발 단계에서 사용자가 업로드한 미디어 파일 제공하기

```
# crud/urls.py
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('articles/', include('articles.urls')),
    path('accounts/', include('accounts.urls')),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- 사용자로부터 업로드된 파일이 프로젝트에 업로드 되고나서, 실제로 사용자에게 제공하기 위해서는 업로드된 파일의 URL이 필요한
 - 업로드 된 파일의 URL == `settings.MEDIA_URL`
 - 위 URL을 통해 참조하는 파일의 실제 위치 == `settings.MEDIA_ROOT`

CREATE

ImageField 작성

```
# articles/models.py

class Article(models.Model):
    title = models.CharField(max_length=20)
    content = models.TextField()
    image = models.ImageField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

❖ 기존 컬럼 사이에 작성해도 실제 테이블에 추가 될 때는 가장 우측(뒤)에 추가됨

Model field option

- Model field option 중 아래 2가지 사항 알아보기

1. `blank`
2. `null`

blank

- Default : False
- True인 경우 필드를 비워 둘 수 있음
 - 이럴 경우 DB에는 "(빈 문자열)"이 저장됨
- 유효성 검사에서 사용됨 (is_valid)
 - Validation-related**
 - 필드에 `blank=True` 가 있으면 form 유효성 검사에서 빈 값을 입력할 수 있음

null

- Default : False
- True인 경우 Django는 빈 값을 DB에 NULL로 저장함
 - Database-related**

null 관련 주의사항

- CharField, TextField와 같은 문자열 기반 필드에서 null 옵션 사용을 피해야 함
 - 문자열 기반 필드에 `null=True` 로 설정시 데이터없음에 대한 표현에 빈 문자열과 NULL 2가지 모두 가능하게 됨
 - 데이터 없음에 대한 표현에 두 개의 가능한 값을 갖는 것은 좋지 않음
 - Django는 문자열 기반 필드에서 NULL이 아닌 빈 문자열을 사용하는 것이 규칙

Migrations

- ImageField를 사용하려면 반드시 Pillow 라이브러리가 필요
 - Pillow 설치 없이는 makemigrations 실행 불가

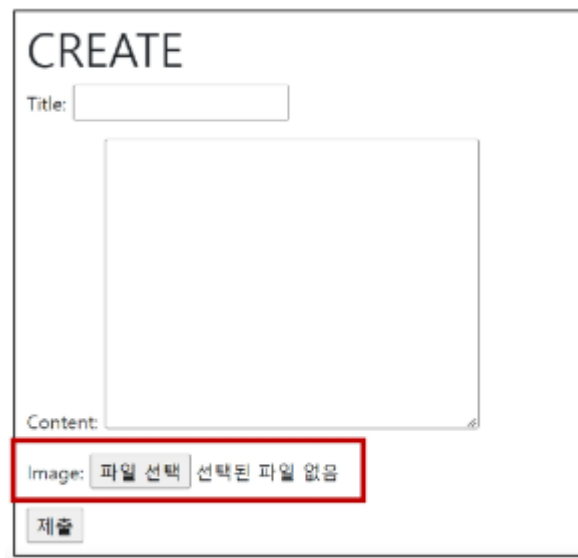
```
$ pip install Pillow  
  
$ python manage.py makemigrations  
$ python manage.py migrate  
  
$ pip freeze > requirements.txt
```

✓ Pillow

- 광범위한 파일 형식 지원, 효율적이고 강력한 이미지 처리 기능을 제공하는 라이브러리
- 이미지 처리 도구를 위한 견고한 기반을 제공

ArticleForm에서 image 필드 출력 확인

- 확인 후 이미지를 첨부하여 게시글 작성 시도



- 하지만 이미지가 업로드 되지 않음
- 파일 또는 이미지 업로드시에는 form 태그에 `enctype` 속성을 다음과 같이 변경해야 함

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

{% extends 'base.html' %}

{% block content %}
<h1>CREATE</h1>
<form action="{% url 'articles:create' %}" method="POST" enctype="multipart/form-data">
  {% csrf_token %}
  {{ form.as_p }}
  <input type="submit">
</form>
...

```

✓ form 태그의 `enctype(인코딩)` 속성 값

1. `application/x-www-form-urlencoded`
 - a. 기본값
 - b. 모든 문자 인코딩
2. `multipart/form-data`
 - a. 파일/이밋 업로드 시에 반드시 사용해야 함
 - b. 전송되는 데이터의 형식을 지정
 - c. `<input type="file">` 을 사용할 경우 사용
3. `text/plain`

request.FILES

- 파일 및 이미지는 request의 POST 속성값으로 넘어가지 않고 FILES 속성값에 담겨 넘어감

```
# articles/views.py

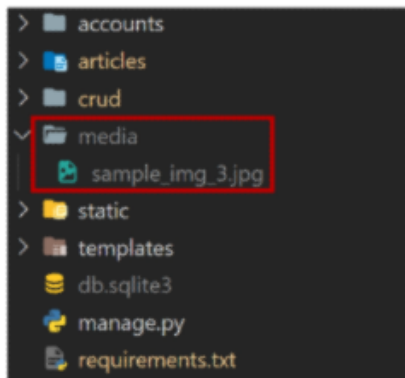
@login_required
@require_http_methods(['GET', 'POST'])
def create(request):
    if request.method == 'POST':
        form = ArticleForm(request.POST, request.FILES)
    ...

```

이미지 첨부하기

- 이미지를 첨부해서 한번, 첨부하지 않고 한번 게시글 작성해보기

- 이미지를 첨부하지 않으면 blank=True 속성으로 인해 빈 문자열이 저장되고, 이미지를 첨부한 경우는 MEDIA_ROOT 경로에 이미지가 업로드됨

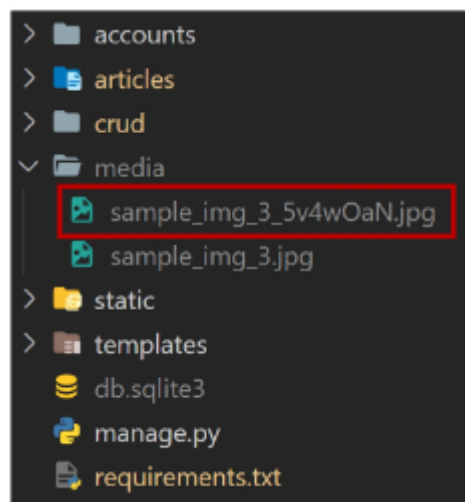


id	title	content	created_at	updated_at	user_id	image
1	이미지	올려보기	2022-10-05 15:41:46.363005	2022-10-05 15:41:46.363005	1	sample_img_3.jpg
2	이미지 없이	작성해보기	2022-10-05 15:43:40.937291	2022-10-05 15:43:40.937291	1	

❖ 파일 자체가 아닌 "경로"가 저장 된다는 것을 잊지 말 것

53

- 만약 같은 이름의 파일을 업로드한다면 Django는 파일 이름 끝에 임의의 난수 값을 붙여 저장함



READ

업로드 이미지 출력하기

- 업로드 된 파일의 상대 URL은 Django가 제공하는 url 속성을 통해 얻을 수 있음

```

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

{% extends 'base.html' %}

{% block content %}

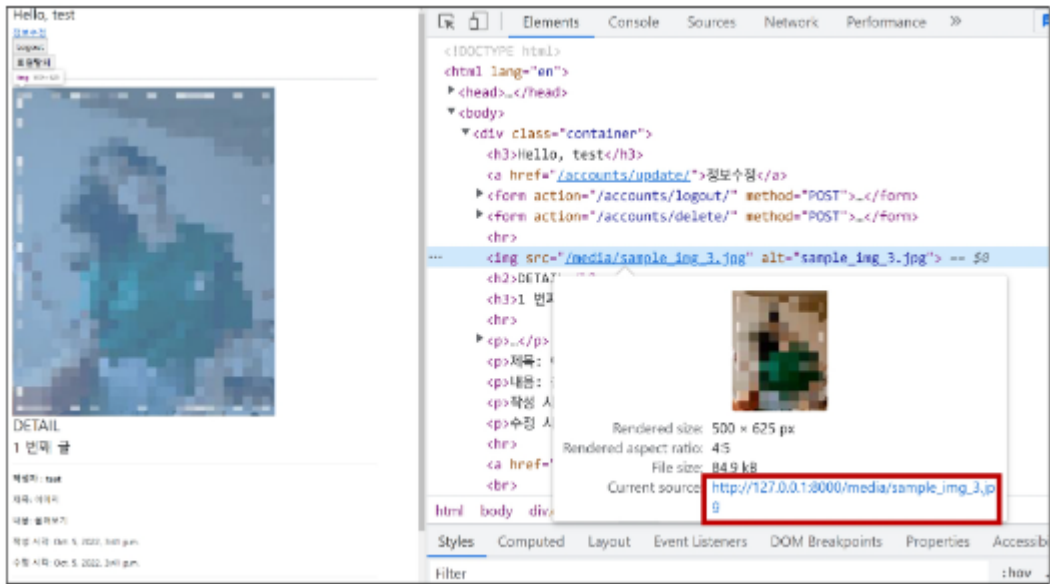
<h2>DETAIL</h2>
...

```

- `article.image.url` - 업로드 파일의 경로
- `article.image` - 업로드 파일의 파일 이름
- 출력 확인하기



- `MEDIA_URL` 확인하기



- 이미지를 업로드하지 않은 게시물은 detail 템플릿을 출력할 수 없는 문제 해결하기
 - 이미지 데이터가 있는 경우만 이미지 출력할 수 있도록 처리

```

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

{% extends 'base.html' %}

{% block content %}
  {% if article.image %}
    
  {% endif %}
  <h2>DETAIL</h2>
...

```

UPDATE

개요

- 이미지는 바이너리 데이터이기 때문에 텍스트처럼 일부만 수정하는 것은 불가능
- 때문에 새로운 사진으로 대체하는 방식을 사용

업로드 이미지 수정하기

- `enctype` 속성값 추가

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

{% block content %}
<h1>UPDATE</h1>
<form action="{% url 'articles:update' article.pk %}" method="POST" enctype="multipart/form-data">
...

```

- 이미지 파일이 담겨있는 `request.FILES` 추가 작성

```
# articles/views.py

@login_required
@require_http_methods(['GET', 'POST'])
def update(request, pk):
    article = Article.objects.get(pk=pk)
    if request.user == article.user:
        if request.method == 'POST':
            form = ArticleForm(request.POST, request.FILES, instance=article)
...

```

`upload_to` argument

사용자 지정 업로드 경로와 파일 이름 설정하기

- ImageField는 업로드 디렉토리와 파일 이름을 설정하는 2가지 방법을 제공
 1. 문자열 값이나 경로 지정 방법
 2. 함수 호출 방법
- 1. 문자열 값이나 경로 지정 방법
 - `upload_to` 인자에 새로운 이미지 저장 경로를 추가 후 migration 과정 진행

```
# articles/models.py

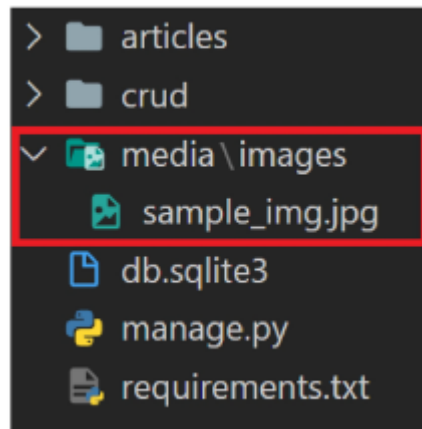
class Article(models.Model):
    title = models.CharField(max_length=20)
    content = models.TextField()
    # image = models.ImageField(blank=True)
    image = models.ImageField(blank=True, upload_to='images/ ')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

```

```
$ python manage.py makemigrations
$ python manage.py migrate

```


- 이미지 업로드 후 변경된 업로드 경로 확인
- `MEDIA_ROOT` 이후 경로가 추가되는것



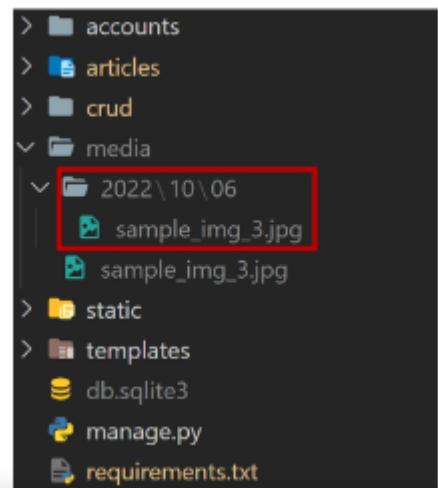
- 단순 문자열 뿐만 아니라 파이썬 time 모듈의 `strftime()` 형식도 포함될 수 있으며, 이는 파일 업로드 날짜/시간으로 대체됨

```
# articles/models.py

class Article(models.Model):
    title = models.CharField(max_length=20)
    content = models.TextField()
    # image = models.ImageField(blank=True)
    # image = models.ImageField(blank=True, upload_to='images/ ')
    image = models.ImageField(blank=True, upload_to='%Y/%m/%d/ ')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

- migration 과정 진행 후 이미지 업로드 결과 확인하기

```
$ python manage.py makemigrations
$ python manage.py migrate
```



2. 함수 호출 방법

- `upload_to` 는 독특하게 함수처럼 호출이 가능하며 해당 함수가 호출되면서 반드시 2개의 인자를 받음

```
# articles/models.py

def articles_image_path(instance, filename):
    return f'images/{instance.user.username}/{filename}'

class Article(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=10)
    content = models.TextField()
    # image = models.ImageField(blank=True)
    # image = models.ImageField(blank=True, upload_to='%Y/%m/%d/')
    image = models.ImageField(blank=True, upload_to=articles_image_path)
```

1. `instance`

- a. `FileField`가 정의된 모델의 인스턴스
- b. 대부분 이 객체는 아직 데이터베이스에 저장되기 전이므로 아직 PK 값이 없을 수 있으니 주의

2. `filename`

- a. 기존 파일 이름
- migration 과정 진행 후 이미지 업로드 결과 확인하기
 - `username`이 `test`인 회원이 업로드한 결과

```
$ python manage.py makemigrations
$ python manage.py migrate
```

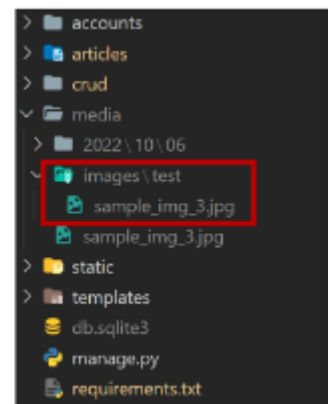


Image Resizing

개요

- 실제 원본 이미지를 서버에 그대로 로드하는 것은 여러 이유로 부담이 큼
- HTML 태그에서 직접 사이즈를 조정할 수도 있지만, 업로드 될 때 이미지 자체를 resizing하는 것을 사용해 볼 것

사전 준비

- `django-imagekit` 모듈 설치 및 등록

```
$ pip install django-imagekit
$ pip freeze > requirements.txt
```

```
# settings.py

INSTALLED_APPS = [
    'articles',
    'accounts',
    'django_extensions',
    'imagekit',
    ...
]
```

썸네일 만들기

- 2가지 방식으로 썸네일 만들기를 진행
 1. 원본 이미지 저장 X
 2. 원본 이미지 저장 O
- 1. 원본 이미지 저장 X
 - `ProcessedImageField()`의 parameter로 작성된 값들은 `makemigrations` 후에 변경이 되더라도 다시 `makemigrations`를 해줄 필요없이 즉시 반영됨

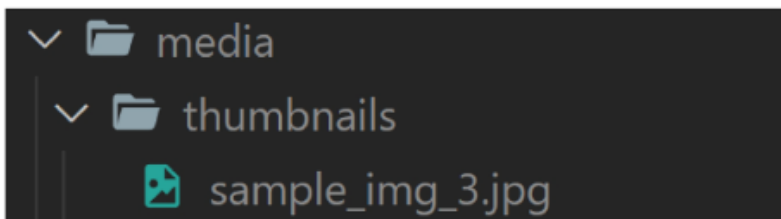
```
# articles/models.py

from imagekit.processors import Thumbnail
from imagekit.models import ProcessedImageField

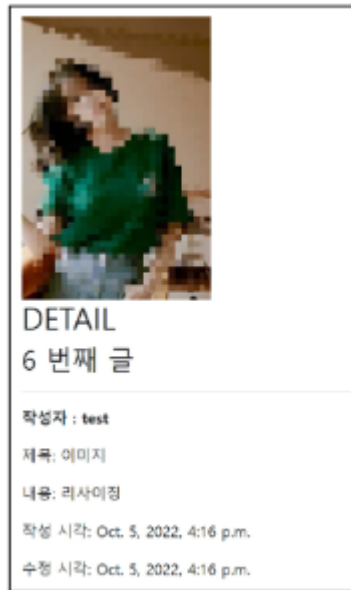
class Article(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=10)
    content = models.TextField()
    # image = models.ImageField(blank=True)
    # image = models.ImageField(blank=True, upload_to='%Y/%m/%d/')
    # image = models.ImageField(blank=True, upload_to=articles_image_path)
    image = ProcessedImageField(
        blank=True,
        upload_to='thumbnails/',
        processors=[Thumbnail(200,300)],
        format='JPEG',
        options={'quality': 80},
    )
    ...
```

- Migration 진행 후 이미지 업로드

```
$ python manage.py makemigrations
$ python manage.py migrate
```



- 작아진 이미지 사이즈 확인



2. 원본 이미지 저장 O

```
# articles/models.py

from imagekit.processors import Thumbnail
from imagekit.models import ProcessedImageField, ImageSpecField
from django.db import models
from django.conf import settings

class Article(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=10)
    content = models.TextField()
    image = models.ImageField(blank=True)
    image_thumbnail = ImageSpecField(
        source='image',
        processors=[Thumbnail(200,300)],
        format='JPEG',
        options={'quality': 80},
    )
```

- Migration 진행 후 이미지 업로드

```
$ python manage.py makemigrations
$ python manage.py migrate
```

- 확인해보면 기본적으로 원본 이미지가 업로드되고 출력됨
- 하지만 다음과 같이 입력 후 detail 페이지에서 다시 새로고침을 진행해보기

```

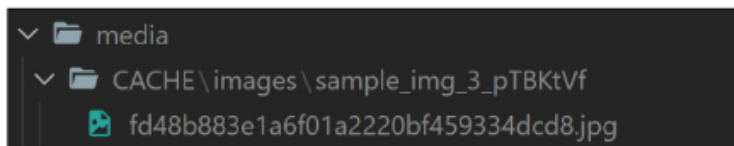
<!-- articles/detail.html -->

{% extends 'base.html' %}

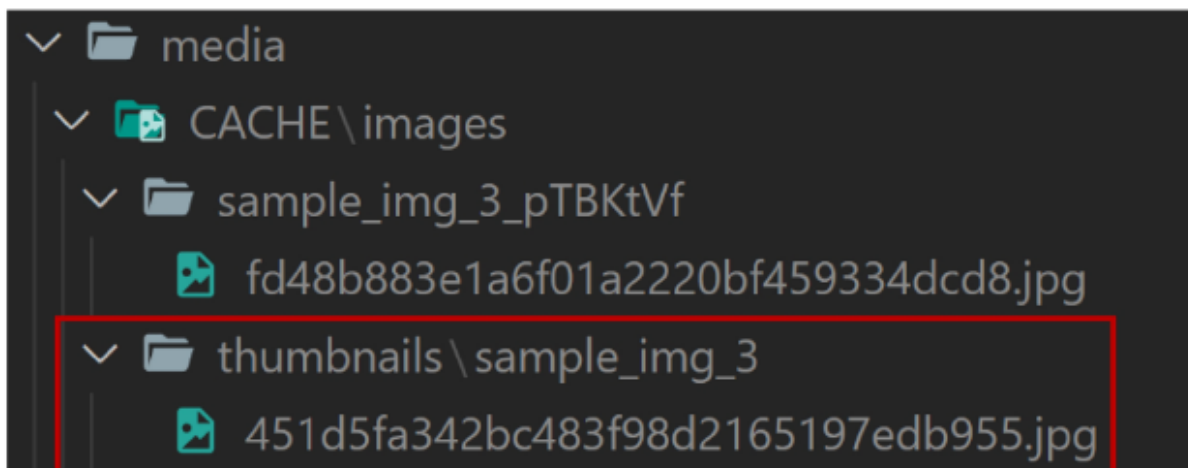
{% block content %}
    {% if article.image %}
        
        
    {% endif %}
...

```

- 처음에는 원본만 사용하며 썸네일이 사용되었을 때만 resizing한 이미지를 생성



- 이미지가 출력되는 다른 detail 페이지에 이동할 때마다 썸네일이 생성됨



QuerySet API Advanced

사전 준비

1. 가상 환경 생성 및 활성화
2. 패키지 목록 설치
3. migrate 진행

```
$ python manage.py migrate
```

4. sqlite3에서 csv 데이터 import 하기

```
$ sqlite3 db.sqlite3
```

```
sqlite > .mode csv
sqlite > .import users.csv users_user
sqlite > .exit
```

- 테이블 확인

The screenshot shows a SQLite Explorer interface. On the left, a tree view lists tables including 'auth_group', 'auth_group_permissions', 'auth_permission', 'auth_user', 'auth_user_groups', 'auth_user_user_permissions', 'django_admin_log', 'django_content_type', 'django_migrations', 'django_session', and 'users_user'. The 'users_user' table is selected. On the right, the 'SQL' view displays the data for 'users_user' in a table format with columns: id, first_name, last_name, age, country, phone, and balance. The data contains 16 rows of user information.

id	first_name	last_name	age	country	phone	balance
1	정호	부	40	전라북도	938-7288-2835	320
2	영희	이	36	경상남도	811-9854-5133	5988
3	철자	구	37	전라남도	911-4117-4178	2988
4	태경	한	40	충청남도	811-9079-4413	250888
5	영환	차	38	충청북도	811-2021-4204	720
6	서준	이	26	충청북도	82-8881-7361	530
7	추종	한	18	강기도	811-2325-1978	388
8	채진	김	33	충청북도	938-5123-5187	3788
9	서현	김	23	세종특별자치도	938-6839-1186	63888
10	서윤	오	23	충청남도	811-8053-6452	48888
11	서영	김	15	세종특별자치도	938-3846-5822	648888
12	태경	홍	22	충청남도	938-4338-4736	52888
13	희운	남	32	전라북도	938-8544-1498	35888
14	영일	서	25	전라남도	911-4448-4198	720
15	지현	박	24	경상북도	82-3383-1183	35888
16	복자	김	18	경상남도	811-3038-5964	720

- `shell_plus` 실행

```
$ python manage.py shell_plus
```

CRUD 기본

- 모든 user 레코드 조회

```
User.objects.all()
```

- user 레코드 생성

```
User.objects.create(  
    first_name='길동',  
    last_name='홍',  
    age=100,  
    country='제주도',  
    phone='010-1234-4567',  
    balance=10000,  
)
```

- 101번 user 레코드 조회

```
User.objects.get(pk=101)
```

- 101번 user 레코드의 last_name을 김으로 수정

```
user = User.objects.get(pk=101)  
user.last_name = '김'  
user.save()  
  
# 확인  
user.last_name
```


- 101번 user 레코드 삭제

```
user = User.objects.get(pk=101)
user.delete()

# 확인
User.objects.get(pk=101)
```

- 전체 인원 수 조회

```
# 1
User.objects.count()

# 2
len(User.objects.all())
```

`.count()`

- QuerySet과 일치하는 데이터베이스의 개체 수를 나타내는 정수를 반환
- `.all()` 을 사용하지 않아도 됨

sorting data

- 나이가 어린 순으로 이름과 나이 조회하기

```
User.objects.order_by('age').values('first_name', 'age')
```

`order_by()`

- `.order_by(*fields)`
- QuerySet의 정렬을 재정의

- 기본적으로 오름차순으로 정렬하며 필드명에 '-'(하이픈)을 작성하면 내림차순으로 정렬
- 인자로 '?'을 입력하면 랜덤으로 정렬

values()

- `.values(*fields, **expressions)`
- 모델 인스턴스가 아닌 딕셔너리 요소들을 가진 QuerySet을 반환
- *fields는 선택인자이며 조회하고자 하는 필드명을 가변인자로 입력받음
 - 필드를 지정하면 각 딕셔너리에는 지정한 필드에 대한 key와 value만을 출력
 - 입력하지 않을 경우 각 딕셔너리에는 레코드의 모든 필드에 대한 key와 value를 출력
- values 사용 여부에 따른 출력 비교

```
# 미사용
User.objects.filter(age=30)
<QuerySet [(<User: User object (5)>), (<User: User object (57)>), (<User: User object (60)>)]>

# 사용
User.objects.filter(age=30).values('first_name')
<QuerySet [{"first_name": '영환'}, {"first_name": '보람'}, {"first_name": '은영'}]>
```

- 이름과 나이를 나이가 많은 순서대로 조회하기

```
User.objects.order_by('-age').values('first_name', 'age')
```

- 이름, 나이, 계좌 잔고를 나이가 어린 순으로, 만약 같은 나이라면 계좌 잔고가 많은 순으로 정렬해서 조회하기

```
User.objects.order_by('age', '-balance').values('first_name', 'age', 'balance')
```

✓ order_by 주의사항

- 다음과 같이 작성할 경우 앞의 호출은 모두 지워지고 마지막 호출만 적용됨

```
User.objects.order_by('balance').order_by('-age')

# 결국 User.objects.order_by('-age') 와 같다.
```

Filtering data

- 중복 없이 모든 지역 조회하기

```
User.objects.distinct().values('country')
```

- 지역 순으로 오름차순 정렬하여 중복없이 모든 지역 조회하기

```
User.objects.distinct().values('country').order_by('country')
```

- 이름과 지역이 중복 없이 모든 이름과 지역 조회하기

```
User.objects.distinct().values('first_name', 'country')
```

- 이름과 지역 중복 없이 지역 순으로 오름차순 정렬하여 모든 이름과 지역 조회하기

```
User.objects.distinct().values('first_name', 'country').order_by('country')
```

- 나이가 30인 사람들의 이름 조회

```
User.objects.filter(age=30).values('first_name')
```

- 나이가 30살 이상인 사람들의 이름과 나이 조회하기

```
User.objects.filter(age__gte=30).values('first_name', 'age')
```

Field lookups

- SQL WHERE 절의 상세한 조건을 지정하는 방법
- QuerySet 메서드 `filter()`, `exclude()` 및 `get()` 에 대한 키워드 인자로 사용됨
- 문법 규칙
 - 필드명 뒤에 **double-underscore** 이후 작성함

field__lookuptype=value

- 나이가 30살 이상이고 계좌 잔고가 50만원 초과인 사람들의 이름, 나이, 계좌 잔고 조회하기

```
User.objects.filter(age__gte=30, balance__gt=500000).values('first_name', 'age', 'balance')
```

- 이름에 '호'가 포함되는 사람들의 이름과 성 조회하기

```
User.objects.filter(first_name__contains='호').values('first_name', 'last_name')
```

- 핸드폰 번호가 011로 시작하는 사람들의 이름과 핸드폰 번호 조회

```
User.objects.filter(phone__startswith='011-').values('first_name', 'phone')
```

1. SQL에서의 '%' 와일드 카드와 같음
 2. '_'(under score)는 별도로 정규 표현식을 사용해야함
- 이름이 '준'으로 끝나는 사람들의 이름 조회하기

```
User.objects.filter(first_name__endswith='준').values('first_name')
```

- 경기도 혹은 강원도에 사는 사람들의 이름과 지역 조회하기

```
User.objects.filter(country__in=['경기도', '강원도']).values('first_name', 'country')
```

- 경기도 혹은 강원도에 살지 않는 사람들의 이름과 지역 조회하기

```
User.objects.exclude(country__in=['경기도', '강원도']).values('first_name', 'country')
```

exclude()

1. exclude(**kwargs)
 2. 주어진 매개변수와 일치하지 않는 객체를 포함하는 QuerySet 반환
- 나이가 가장 어린 10명의 이름과 나이 조회하기

```
User.objects.order_by('age').values('first_name', 'age')[:10]
```

- 나이가 30이거나 성이 김씨인 사람들 조회

```
# shell_plus에서는 import 문 생략 가능
from django.db.models import Q

User.objects.filter(Q(age=30) | Q(last_name='김'))
```

‘Q’ object

- 기본적으로 `filter()` 와 같은 메서드의 키워드 인자는 AND statement를 따름
- 만약 더 복잡한 쿼리를 실행해야 하는 경우가 있다면 Q 객체가 필요함
 - 예를 들어 OR statement 같은 경우

```
# 예시

from django.db.models import Q

Q(question__startswith='What')
```

- ‘&’ 및 ‘|’을 사용하여 Q 객체를 결합할 수 있음

```
# 예시

Q(question__startswith='Who') | Q(question__startswith='What')
```

- 조회를 하면서 여러 Q 객체를 제공할 수도 있음

```
# 예시

Article.objects.get(
    Q(title__startswith='Who'),
    Q(created_at=date(2005, 5, 2)) | Q(created_at=date(2005, 5, 6))
)
```

Aggregation(Grouping data)

aggregate()

- **Aggregate calculates values for the entire queryset.**
- 전체 queryset
- 특정 필드 전체의 합, 평균, 개수 등을 계산할 때 사용
- 딕셔너리를 반환
- **Aggregation functions**
 - Avg, Count, Max, Min, Sum 등
- 나이가 30살 이상인 사람들의 평균 나이 조회하기

```
# shell_plus 에서는 import하지 않아도 된다.

from django.db.models import Avg

User.objects.filter(age__gte=30).aggregate(Avg('age'))
=> {'age__avg': 37.65909090909091}
```

```
# 딕셔너리 key 이름을 수정할 수도 있다.

User.objects.filter(age__gte=30).aggregate(avg_value=Avg('age'))
=> {'avg_value': 37.65909090909091}
```

- 가장 높은 계좌 잔액 조회하기

```
from django.db.models import Max

User.objects.aggregate(Max('balance'))
=> {'balance__max': 1000000}
```

- 모든 계좌 잔액 총액 조회하기

```
from django.db.models import Sum

User.objects.aggregate(Sum('balance'))
=> {'balance__sum': 14435040}
```

annotate()

- 쿼리의 각 항목에 대한 요약값을 계산
- SQL의 `GROUP BY`에 해당
- 주석을 달다라는 사전적 의미를 가지고 있음
- 각지역별로 몇 명씩 살고 있는지 조회하기

```
from django.db.models import Count

User.objects.values('country').annotate(Count('country'))

=> <QuerySet [{'country': '강원도', 'country__count': 14},
{'country': '경기도', 'country__count': 9},
{'country': '경상남도', 'country__count': 9},]...>
```

aggregate와 마찬가지로 딕셔너리의 key 값을 변경할 수 있다.

```
User.objects.values('country').annotate(num_of_country=Count('country'))

=> <QuerySet [{'country': '강원도', 'num_of_country': 14},
{'country': '경기도', 'num_of_country': 9},
{'country': '경상남도', 'num_of_country': 9},]...>
```

- 각 지역별로 몇 명씩 살고 있는지 + 지역별 계좌 잔액 평균 조회하기
 - 한번에 여러 값을 계산해 조회할 수 있음

```
User.objects.values('country').annotate(Count('country'), avg_balance=Avg('balance'))
```

- 각 성씨가 몇 명씩 있는지 조회하기

```
User.objects.values('last_name').annotate(Count('last_name'))
```

N:1 예시

- 만약 Comment - Article 관계가 N:1인 경우 다음과 같은 참조도 가능

```
# 예시

Article.objects.annotate(
    number_of_comment=Count('comment'),
    pub_date=Count('comment', filter=Q(comment__created_at__lte='2000-01-01'))
)
```

- 전체 게시글을 조회하면서(Article.objects.all()) annotate로 각 게시글의 댓글 개수 (number_of_comment)와 2000-01-01 보다 나중에 작성된 댓글의 개수(pub_date)를 함께 조회하는 것