13 - Django: Medienverwaltung & Statische Dateien Vertiefung

Einleitung

- **Themen**: Dieses Skript behandelt die Verwaltung von zwei fundamental unterschiedlichen Dateitypen in Django: **Statische Dateien** (die zum Code gehören) und **Mediendateien** (die von Benutzern hochgeladen werden). [cite: 53]
- **Fokus**: Eine klare, verständliche Trennung dieser beiden Konzepte. Der Schwerpunkt liegt auf der detaillierten Erklärung, was `MEDIA_URL` und `MEDIA_ROOT` sind, wie sie zusammenarbeiten und wie der komplette Prozess von der Konfiguration bis zum Upload und zur Anzeige von Benutzer-Bildern funktioniert.
- **Lernziele**:
  - Den Unterschied zwischen statischen Dateien und Mediendateien sicher erklären können.
  - Das Konzept von `MEDIA_URL` (der Web-Adresse) und `MEDIA_ROOT` (dem Speicherort im Dateisystem) vollständig verstehen.
  - Die notwendigen Einstellungen in `settings.py` und `urls.py` für Mediendateien vornehmen können.
  - Ein `ImageField` in einem Modell verwenden und die `Pillow`-Bibliothek installieren.
  - Einen vollständigen Upload-Prozess implementieren.
  - Die Bedeutung und Anwendung des `collectstatic`-Befehls für das Deployment verstehen.

Teil 1: Das Fundament - Statische vs. Mediendateien

- **Statische Dateien (`Static Files`)**:
  - **Was sie sind**: Dateien, die Teil deiner Anwendung sind. Sie ändern sich nicht durch Benutzerinteraktion.
  - **Beispiele**: Dein CSS für das Layout, dein JavaScript-Code, dein Logo, Icons.
  - **Verwaltung**: Sie sind Teil deines Quellcodes und werden für das Deployment mit `collectstatic` an einen zentralen Ort kopiert.
- **Mediendateien (`Media Files`)**:
  - **Was sie sind**: Dateien, die von den **Benutzern** deiner Webseite hochgeladen werden.
  - **Beispiele**: Profilbilder, Bilder zu Rezepten, hochgeladene PDFs.
  - **Verwaltung**: Sie sind *nicht* Teil deines Quellcodes. Ihre Verwaltung und Auslieferung im Produktivbetrieb übernimmt ein Webserver.

Teil 2: Die Ordnerstruktur in der Praxis

Um die Konzepte zu verdeutlichen, schauen wir uns eine beispielhafte Projektstruktur mit zwei Apps an: `recipes` und `users`.

2.1 Anschauungsbeispiel: Projekt-Ordnerstruktur

```
mein_projekt/           <-- Äußerer Projektordner (Git-Repository-Wurzel)
├── manage.py
├── mein_projekt/       <-- Innerer Projektordner (Konfiguration)
│   ├── settings.py
│   └── urls.py
│
├── recipes/            <-- App 1
│   ├── models.py
│   ├── views.py
│   └── static/         <-- STATIC: Spezifisch für die recipes-App
│       └── recipes/
│           ├── css/
│           │   └── recipe_style.css
│           └── images/
│               └── placeholder.png
│
├── users/              <-- App 2
│   ├── models.py
│   ├── views.py
```

```
│   └── static/          <-- STATIC: Spezifisch für die users-App
│       └── users/
│           └── css/
│               └── profile_style.css
│
├── static/              <-- STATIC: Projektweit (optional, für geteilte Dateien)
│   ├── css/
│   │   └── base.css
│   └── images/
│       └── logo.png
│
└── mediafiles/          <-- MEDIA: Ein zentraler Ordner für das gesamte Projekt
    ├── recipe_images/   <-- Unterordner wird von `upload_to` im Recipe-Modell erstellt
    │   ├── spaghetti.jpg
    │   └── kuchen.png
    └── profile_pics/    <-- Unterordner wird von `upload_to` im UserProfile-Modell erstellt
        ├── max_m.jpg
        └── anna_s.png
```

2.2 Erklärung der Ordnerstruktur

- **Statische Dateien (`static`)**:
  - **Pro App**: Jede App kann einen eigenen `static`-Ordner haben, um ihre eigenen Assets zu bündeln. Um Namenskonflikte zu vermeiden, wird darin ein weiterer Ordner mit dem App-Namen erstellt (z.B. `recipes/static/recipes/`). Django findet diese Ordner automatisch.
  - **Projektweit**: Ein optionaler `static`-Ordner im Projekt-Hauptverzeichnis kann für geteilte Dateien genutzt werden. Dieser Ordner muss in `settings.py` in der `STATICFILES_DIRS`-Liste eingetragen werden.
- **Mediendateien (`mediafiles`)**:
  - **Zentral**: Es gibt nur **einen zentralen Ordner** für das gesamte Projekt. Sein Ort wird durch `MEDIA_ROOT` in `settings.py` festgelegt.
  - **Keine `media`-Ordner in Apps**: Es ist eine falsche Annahme, dass Apps eigene `media`-Ordner haben.
  - **Unterordner**: Die Organisation *innerhalb* des zentralen Media-Ordners (z.B. `recipe_images/`) wird ausschließlich durch das `upload_to`-Argument im `ImageField` des jeweiligen Modells gesteuert.

Konfiguration von `STATICFILES_DIRS` in `settings.py`

Um Django mitzuteilen, dass es auch im projektweiten `static`-Ordner (dem Ordner, der auf derselben Ebene wie `manage.py` liegt) nach statischen Dateien suchen soll, fügt man diesen Pfad zur `STATICFILES_DIRS`-Liste hinzu.

```python
# mein_projekt/settings.py
import os

# ... (andere Einstellungen wie SECRET_KEY, DEBUG, etc.) ...

# BASE_DIR zeigt auf das Hauptverzeichnis des Projekts (wo manage.py liegt)
# Diese Zeile ist in der Regel bereits in settings.py vorhanden.
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))) # Oder eine ähnliche
Definition

# ...

STATIC_URL = '/static/'

# HIER wird der projektweite static-Ordner hinzugefügt
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static'),
]

# Optional: Konfiguration für das Deployment
```

```
# STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

Erklärung:

- `BASE_DIR`: Diese Variable, die von Django standardmäßig definiert wird, enthält den absoluten Pfad zum Projekt-Hauptverzeichnis.
- `os.path.join(BASE_DIR, 'static')`: Dieser Befehl baut einen betriebssystem-unabhängigen Pfad, der `BASE_DIR` und den Ordnernamen `static` korrekt verbindet (z.B. `/pfad/zu/mein_projekt/static/` auf Linux/macOS oder `C:\pfad\zu\mein_projekt\static\` auf Windows).
- Indem dieser Pfad zur `STATICFILES_DIRS`-Liste hinzugefügt wird, weiß der `collectstatic`-Befehl und der Entwicklungsserver, dass auch in diesem Ordner nach statischen Dateien gesucht werden soll.

Teil 3: Der komplette Workflow für Mediendateien

3.1 Das Kernkonzept: `MEDIA_URL` vs. `MEDIA_ROOT` (Die Bibliotheks-Analogie)

- **`MEDIA_ROOT`**: Dies ist die **physische Adresse** der Bibliothek im Dateisystem (z.B. `/home/user/mein_projekt/mediafiles/`). Hier stehen die echten Dateien im Regal. Dieser Pfad ist "privat" und nur dem Server bekannt.
- **`MEDIA_URL`**: Dies ist die **öffentliche Web-Adresse**, unter der die Bibliothek im Internet erreichbar ist (z.B. `/media/`). Das ist der Teil der URL, den der Browser verwendet, um eine Datei anzufordern.

Zusammenspiel: Django benutzt die `MEDIA_URL`, um im Template einen Link zu generieren (z.B. `<img src="/media/recipe_images/spaghetti.jpg">`). Wenn der Browser diesen Link anfordert, weiß der Server, dass er unter der `MEDIA_URL` (`/media/`) nachsieht und die Anfrage an den physischen Speicherort `MEDIA_ROOT` weiterleitet, um die Datei `spaghetti.jpg` im Unterordner `recipe_images` zu finden und auszuliefern.

3.2 Die Implementierung (Schritt-für-Schritt)

1. **`Pillow` installieren**:

   ```
   pip install Pillow
   ```

2. **`settings.py` konfigurieren**:

   ```
   # settings.py
   MEDIA_URL = '/media/'
   MEDIA_ROOT = BASE_DIR / 'mediafiles' # oder os.path.join(BASE_DIR, 'mediafiles')
   ```

3. **Haupt-`urls.py` für den Entwicklungsserver anpassen**:

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

   urlpatterns = [ ... ]

   if settings.DEBUG:
       urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
   ```

4. **Modell anpassen (`models.py`)**:

   ```
   class Recipe(models.Model):
       # ... andere Felder
       # Das 'upload_to'-Argument gibt ein Unterverzeichnis innerhalb von MEDIA_ROOT an.
   ```

```
    # Es können auch dynamische Pfade erstellt werden.
    image = models.ImageField(upload_to='recipes/images/', blank=True, null=True,
verbose_name="Rezeptbild")
```

Wenn ein Benutzer ein Bild hochlädt, wird es unter `MEDIA_ROOT/recipes/images/` gespeichert.

5. **Formular-Template anpassen (`.html`)**:

```
<form method="post" enctype="multipart/form-data">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Speichern</button>
</form>
```

6. **View anpassen (`views.py`)**: Hochgeladene Dateien befinden sich nicht in `request.POST`, sondern in `request.FILES`. Man muss `request.FILES` an die Formular-Instanz übergeben.

```
# views.py
def add_recipe_view(request):
    if request.method == 'POST':
        # request.FILES wird als zweites Argument übergeben
        form = RecipeForm(request.POST, request.FILES)
        if form.is_valid():
            form.save()
            return redirect('success_url')
    else:
        form = RecipeForm()
    return render(request, 'recipe_form.html', {'form': form})
```

7. **Datei im Template anzeigen (`.html`)**: Um ein hochgeladenes Bild anzuzeigen, verwendet man die `.url`-Eigenschaft des Feldobjekts.

```
{% if recipe.image %}
    <img src="{{ recipe.image.url }}" alt="{{ recipe.title }}">
{% else %}
    <p>Kein Bild vorhanden.</p>
{% endif %}
```

Django konstruiert hier automatisch die vollständige URL (z.B. `/media/recipes/images/mein_bild.jpg`).

Teil 4: Exkurs - Mediendateien im Produktivbetrieb (Production)

Die bisher gezeigte Methode, Mediendateien über eine Regel in `urls.py` mit dem Django-Entwicklungsserver auszuliefern, ist **ausschließlich für die lokale Entwicklung gedacht**. Sie ist ineffizient und unsicher und darf niemals in einer Live-Umgebung verwendet werden.

4.1 Warum die Entwicklungsmethode nicht ausreicht

- **Ineffizienz**: Der Django-Entwicklungsserver ist darauf optimiert, Python-Code auszuführen, nicht aber, statische oder Mediendateien schnell auszuliefern. Ein dedizierter Webserver wie Nginx oder Apache kann dies um ein Vielfaches schneller und mit weniger Ressourcen.
- **Sicherheit**: Das Serven von Dateien direkt über Django im `DEBUG=False`-Modus (der für Produktion zwingend ist) ist ein Sicherheitsrisiko und wird von Django standardmäßig unterbunden.
- **Skalierbarkeit**: Der Entwicklungsserver ist nicht dafür gebaut, viele gleichzeitige Anfragen zu bewältigen.

4.2 Das Produktions-Konzept: Der Webserver übernimmt die Arbeit

Im Produktivbetrieb steht vor der Django-Anwendung (die meist über einen Applikationsserver wie Gunicorn läuft) ein **Webserver** (z.B. Nginx). Die Aufgaben werden klar verteilt:

- **Django/Gunicorn**: Verarbeitet die dynamischen Anfragen (z.B. das Rendern von HTML-Templates, Datenbankabfragen).
- **Nginx (Webserver)**: Nimmt alle Anfragen von außen entgegen. Er leitet die dynamischen Anfragen an Django weiter, aber Anfragen für statische und Mediendateien bearbeitet er **selbst** und direkt.

Der Ablauf für eine Mediendatei in Produktion:

1. Ein Browser fordert ein Bild an: `https://deine-domain.de/media/recipes/kuchen.png`
2. **Nginx** empfängt diese Anfrage.
3. Nginx ist so konfiguriert, dass es erkennt: "Aha, die URL beginnt mit `/media/`. Diese Anfrage leite ich **nicht** an Django weiter."
4. Stattdessen schaut Nginx direkt ins Dateisystem. Es hat eine Regel, die den Web-Pfad (`MEDIA_URL`) auf den Dateisystem-Pfad (`MEDIA_ROOT`) abbildet.
5. Nginx findet die Datei unter `/pfad/zu/deinem/projekt/mediafiles/recipes/kuchen.png` und schickt sie extrem schnell direkt an den Browser zurück.
6. Django wird für diesen Vorgang nicht beansprucht und kann sich auf die Verarbeitung der Anwendungslogik konzentrieren.

4.3 Beispiel: Nginx-Konfiguration für Mediendateien

Die Django-Einstellungen (`MEDIA_URL` und `MEDIA_ROOT` in `settings.py`) bleiben exakt gleich. Sie liefern die Information, die Nginx für seine Konfiguration benötigt.

Hier ist ein typischer Konfigurations-Block in einer Nginx-Konfigurationsdatei (`nginx.conf` oder in einer Site-spezifischen Datei), der die Auslieferung von Mediendateien regelt:

```
# Beispiel Nginx-Konfiguration

server {
    listen 80;
    server_name deine-domain.de;

    # ... andere Einstellungen wie SSL, etc. ...

    # Regel für Mediendateien
    location /media/ {
        # 'alias' ist der Befehl, der den URL-Pfad auf einen Dateisystem-Pfad abbildet.
        # Hier wird der Wert aus deiner Django settings.py MEDIA_ROOT eingetragen.
        alias /pfad/zu/deinem/projekt/mediafiles/;

        # Optional: Setzt Header, um Caching im Browser zu ermöglichen
        expires 7d;
    }

    # Regel für statische Dateien (funktioniert nach dem gleichen Prinzip)
    location /static/ {
        alias /pfad/zu/deinem/projekt/staticfiles/;
        expires 7d;
    }

    # Alle anderen Anfragen werden an die Django-Anwendung weitergeleitet
    location / {
        proxy_pass http://localhost:8000; # Annahme: Gunicorn läuft auf Port 8000
        # ... weitere proxy-Einstellungen ...
    }
}
```

Zusammengefasst:

- Die Django-Einstellungen bleiben unverändert.
- Die `urls.py`-Anpassung für den Entwicklungsserver wird im Produktivbetrieb unwirksam, da `DEBUG=False` ist.
- Die gesamte Verantwortung für die Auslieferung von Mediendateien wird an den Webserver (z.B. Nginx) übergeben. Dessen Konfiguration sorgt für die korrekte Zuordnung von URL zu Dateipfad.

Fazit

- **Klare Trennung**: Der wichtigste Schritt ist das Verständnis, dass `MEDIA_URL` eine **Web-Adresse für den Browser** ist und `MEDIA_ROOT` ein **Speicherort im Dateisystem für den Server**.
- **Struktur**: Statische Dateien können pro App organisiert werden, Mediendateien haben einen einzigen, zentralen Speicherort.
- **Konfigurations-Kette**: Alle Schritte müssen korrekt konfiguriert sein, damit der Upload funktioniert: `settings.py` -> `urls.py` (für Dev) -> `models.py` -> Template mit `enctype` -> View mit `request.FILES`.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) wird ein optionaler Bild-Upload für `Question`-Objekte implementiert.

1. **`Pillow` installieren**.
2. **`settings.py`** und **Haupt-`urls.py`** für Media-Dateien konfigurieren.
3. **`Question`-Modell** in `polls/models.py` um ein `ImageField` erweitern:

   ```
   image = models.ImageField(upload_to='polls/images/', blank=True, null=True,
   verbose_name="Zugehöriges Bild")
   ```

4. **Migrationen** durchführen.
5. **`QuestionModelForm`** in `polls/forms.py` um das `image`-Feld erweitern.
6. **Formular-Template** mit `enctype="multipart/form-data"` versehen.
7. **View** (`add_question_modelform` und eine neue `edit_question_modelform`) anpassen, um `request.FILES` zu übergeben.
8. **Detail-Template** (`question_detail.html`) anpassen, um das Bild anzuzeigen.

Cheat Sheet

Einstellungen (`settings.py`)

- `STATIC_URL = '/static/'`: URL für statische Dateien.
- `STATIC_ROOT = BASE_DIR / 'staticfiles'`: Zielordner für `collectstatic`.
- `MEDIA_URL = '/media/'`: URL für Mediendateien.
- `MEDIA_ROOT = BASE_DIR / 'mediafiles'`: Physischer Speicherort für hochgeladene Dateien.

Befehle

- ```
  pip install Pillow
  ```
- ```
  python manage.py collectstatic
  ```

Upload-Workflow

- **Modell**: `image = models.ImageField(upload_to='unterordner/')`
- **Template**: `<form method="post" enctype="multipart/form-data">`
- **View**: `form = MyForm(request.POST, request.FILES)`
- **Anzeige**: `<img src="{{ object.image_field.url }}">`

Übungsaufgaben

1. **Bildergalerie-Modell erstellen**:
   - Eine neue App `gallery` erstellen.

    - Ein Modell `GalleryImage` mit den Feldern `title` (`CharField`), `image` (`ImageField` mit `upload_to='gallery/'`) und `uploaded_at` (`DateTimeField` mit `auto_now_add=True`) erstellen.
2. **Upload-Formular erstellen**:
    - Ein `ModelForm` für `GalleryImage` erstellen.
    - Eine View und ein Template erstellen, um neue Bilder hochzuladen und den vollständigen Upload-Prozess zu implementieren.
3. **Galerie-Ansicht erstellen**:
    - Eine View erstellen, die alle `GalleryImage`-Objekte aus der Datenbank holt.
    - Ein Template erstellen, das über alle Objekte iteriert und jedes Bild zusammen mit seinem Titel anzeigt.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die "Community Recipe Sharing Platform" wird nun um die Möglichkeit erweitert, Bilder zu Rezepten hochzuladen.

Aufgabe:

1. `Pillow` **installieren**.
2. **Projekt konfigurieren**: Die `settings.py` und die Haupt-`urls.py` um die `MEDIA_URL`, `MEDIA_ROOT` und die URL-Konfiguration für den Entwicklungsserver erweitern.
3. `Recipe`**-Modell erweitern**: Das `Recipe`-Modell in `recipes/models.py` um ein `ImageField` erweitern:

    ```
    image = models.ImageField(upload_to='recipe_images/', blank=True, null=True,
    verbose_name="Rezeptbild")
    ```

4. **Migrationen erstellen und anwenden**.
5. `RecipeForm` **anpassen**: Das neue `image`-Feld zur `fields`-Liste im `RecipeForm` hinzufügen.
6. **Formular-Template anpassen**: Sicherstellen, dass das `<form>`-Tag `enctype="multipart/form-data"` enthält.
7. **Views (**`add_recipe_view`**,** `edit_recipe_view`**) anpassen**, um `request.FILES` zu übergeben.
8. **Templates zur Anzeige anpassen**: Im `recipe_detail.html` das Bild des Rezepts anzeigen. Optional im `recipe_list.html` Vorschaubilder anzeigen.
9. **Testen**: Den vollständigen Prozess testen: Ein Rezept bearbeiten und ein Bild hinzufügen. Ein neues Rezept mit Bild erstellen. Sicherstellen, dass die Bilder korrekt angezeigt werden.