22 - Finale Integration & Deployment-Konzepte

Einleitung

- **Themen**: Dies ist das letzte Skript unseres Kurses. Wir führen unser Django REST Backend und das React Frontend zu einer funktionierenden Full-Stack-Anwendung zusammen. Anschließend werfen wir einen Blick auf die konzeptionellen und praktischen Schritte, um eine solche Anwendung online zu stellen (Deployment).
- **Fokus**: Die Lösung des CORS-Problems, das bei der lokalen Entwicklung auftritt. Verständnis des React Build-Prozesses. Ein konzeptioneller Überblick über eine professionelle Deployment-Architektur und als praktisches Beispiel eine Anleitung für den Einstieg mit PythonAnywhere.
- **Lernziele**:
  - Verstehen, was CORS ist und wie man es mit `django-cors-headers` für die lokale Entwicklung konfiguriert.
  - Eine produktionsreife Version einer React-Anwendung mit `npm run build` erstellen.
  - Die konzeptionellen Rollen von Nginx, Gunicorn und einem Datenbankserver im Produktivbetrieb verstehen.
  - Ein grundlegendes Deployment einer Django/React-Anwendung auf PythonAnywhere durchführen können.

1. Die letzte Meile: Backend und Frontend verbinden (CORS)

Wenn man den Django-Server (z.B. auf `localhost:8000`) und den React-Entwicklungsserver (z.B. auf `localhost:5173`) gleichzeitig laufen lässt, wird man im Browser auf einen Fehler stoßen: **CORS (Cross-Origin Resource Sharing)**.

- **Das Problem**: Aus Sicherheitsgründen verbieten Browser standardmäßig, dass eine Webseite (Origin 1: `http://localhost:5173`) per JavaScript Anfragen an eine andere Domain (Origin 2: `http://localhost:8000`) sendet.
- **Die Lösung**: Wir müssen unserem Django-Backend explizit erlauben, Anfragen von unserem React-Frontend anzunehmen. Die Standardlösung dafür ist die Bibliothek `django-cors-headers`.

Schritt-für-Schritt-Anleitung zur CORS-Konfiguration:

1. **Bibliothek installieren**:

```
pip install django-cors-headers
```

2. `settings.py` **anpassen:** `INSTALLED_APPS`: Füge `corsheaders` zur Liste hinzu, am besten weit oben.

```
INSTALLED_APPS = [
    # ...
    'corsheaders',
    # ...
]
```

3. `settings.py` **anpassen:** `MIDDLEWARE`: Füge die `CorsMiddleware` hinzu, am besten direkt vor der `CommonMiddleware`.

```
MIDDLEWARE = [
    # ...
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    # ...
]
```

4. `settings.py` **anpassen: Erlaubte Origins definieren**: Lege fest, welche Frontend-Domains auf deine API zugreifen dürfen.

```
# settings.py

# Für die lokale Entwicklung erlauben wir den React Dev-Server
CORS_ALLOWED_ORIGINS = [
```

```
    "http://localhost:5173",
    "http://127.0.0.1:5173",
]

# Optional: Um Anfragen mit Authentifizierungs-Token zu erlauben
# CORS_ALLOW_CREDENTIALS = True
```

Nach einem Neustart des Django-Servers sollte das React-Frontend erfolgreich API-Anfragen stellen können.

2. Der React Build-Prozess (`npm run build`)

Der Code im `src`-Ordner unseres React-Projekts ist für die Entwicklung optimiert, aber nicht für die Produktion. Er enthält JSX, unkomprimierten Code und viele einzelne Dateien.

- **Was der Befehl tut**:

  ```
  npm run build
  ```

  Dieser Befehl startet den Build-Prozess von Vite. Er nimmt den gesamten React-Code und wandelt ihn in ein paar wenige, hochoptimierte und statische HTML-, CSS- und JavaScript-Dateien um.
- **Das Ergebnis**: Es wird ein neuer Ordner namens `dist` im React-Projektverzeichnis erstellt. **Dieser `dist`-Ordner enthält die fertige Webseite**, die auf einem Server hochgeladen werden kann.

3. Allgemeine Deployment-Konzepte (Der "Industriestandard")

Eine professionelle Django-Anwendung im Produktivbetrieb besteht typischerweise aus mehreren Komponenten:

- **Webserver (z.B. Nginx)**: Nimmt alle Anfragen von außen entgegen. Liefert statische und Mediendateien extrem schnell direkt aus. Leitet dynamische Anfragen an den Applikationsserver weiter.
- **Applikationsserver (z.B. Gunicorn)**: Ein Python-spezifischer Server, der die Django-Anwendung ausführt und mehrere Anfragen parallel verarbeiten kann – viel robuster als der `runserver`-Befehl.
- **Datenbankserver (z.B. PostgreSQL, MySQL)**: Eine dedizierte, für Performance optimierte Datenbank.
- **Server/VPS**: Der virtuelle oder physische Server, auf dem all diese Komponenten laufen.

Fazit

- **CORS ist ein Muss**: Für die lokale Entwicklung einer entkoppelten Anwendung ist die Konfiguration von `django-cors-headers` unerlässlich.
- **`npm run build`**: Erstellt die produktionsreife, statische Version deiner React-Anwendung im `dist`-Ordner.
- **Deployment ist vielschichtig**: Eine professionelle Produktionsumgebung besteht aus mehreren spezialisierten Komponenten (Webserver, Applikationsserver, DB-Server).
- **Einfacher Einstieg**: Plattformen wie PythonAnywhere abstrahieren viele dieser Details und bieten einen einfacheren, geführten Weg, um eine Anwendung online zu stellen.

Cheat Sheet

CORS-Konfiguration

1. ```
   pip install django-cors-headers
   ```
2. `'corsheaders'` zu `INSTALLED_APPS` hinzufügen.
3. `'corsheaders.middleware.CorsMiddleware'` zu `MIDDLEWARE` hinzufügen.
4. In `settings.py`: `CORS_ALLOWED_ORIGINS = ["http://dein-frontend.de"]`.

React Build & Django Static

- **React-App bauen**:
   ```

```
npm run build
```

- **Statische Dateien für Django sammeln**:

```
python manage.py collectstatic
```

Übungsaufgaben

Die Hauptübung ist die finale Projektarbeit und eine abschließende Code-Review.

1. **Finale Code-Review**:
   - Überprüfe dein gesamtes Projekt (Django & React) anhand der Best-Practice-Checkliste aus Skript 15.
   - Sind alle `console.log`-Anweisungen entfernt? Ist der Code sauber formatiert?
2. **Vorbereitung für das Deployment**:
   - Erstelle eine `requirements.txt`-Datei für dein Django-Projekt:

     ```
     pip freeze > requirements.txt
     ```

   - Führe den `npm run build`-Befehl in deinem React-Projekt aus und inspiziere den Inhalt des `dist`-Ordners.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Es ist an der Zeit, alles zusammenzufügen.

Aufgabe:

1. **CORS konfigurieren**: Installiere `django-cors-headers` und konfiguriere deine Django `settings.py`, um Anfragen von deinem React-Entwicklungsserver (`http://localhost:5173`) zu erlauben.
2. **Funktionalität finalisieren**: Stelle sicher, dass der gesamte CRUD-Zyklus inklusive Authentifizierung zwischen deinem React-Frontend und Django-Backend reibungslos funktioniert.
3. **Code-Qualität prüfen**: Führe eine letzte Überprüfung deines Codes durch. Refaktoriere unsaubere Stellen und füge Kommentare hinzu, wo nötig.
4. **Deployment vorbereiten**: Erstelle eine `requirements.txt`-Datei und führe `npm run build` aus. Du hast nun eine Anwendung, die bereit für das Deployment ist.

Absolut. Das ist ein sehr wichtiger Punkt. Der Deployment-Prozess für eine reine Django-Anwendung, bei der Django auch das Frontend über Templates rendert, unterscheidet sich vom Deployment einer entkoppelten Anwendung mit React.

Ich habe ein neues, eigenständiges Kapitel erstellt, das du in das letzte Skript (Skript 22) einfügen kannst, zum Beispiel vor dem Anhang zu PythonAnywhere. Es konzentriert sich ausschließlich auf die Vorbereitung und die Konzepte für das Deployment einer traditionellen Django-Anwendung.

Anhang A: Deployment einer reinen Django-Anwendung (mit Templates)

Dieses Kapitel behandelt die notwendigen Schritte und Konfigurationen, um eine Django-Anwendung, bei der das Frontend direkt mit Django-Templates gerendert wird (wie in Woche 1-3 erstellt), für eine Produktionsumgebung vorzubereiten.

1. Die `settings.py`: Checkliste für die Produktion

Der Wechsel von einer Entwicklungs- zu einer Produktionsumgebung erfordert mehrere wichtige Änderungen in der `settings.py`-Datei. Das Ziel ist es, die Anwendung sicherer und performanter zu machen.

1.1 `DEBUG = False` (Der wichtigste Schalter)

Dies ist die absolut wichtigste Änderung für den Produktivbetrieb.

- **Was es tut**:
  - **Sicherheit**: Schaltet die detaillierten, gelben Fehlerseiten von Django ab. Diese Fehlerseiten enthalten sensible Konfigurationsdetails (wie alle `settings.py`-Werte) und dürfen niemals öffentlich sichtbar sein.
  - **Performance**: Django verhält sich bei `DEBUG=False` anders. Es hört zum Beispiel auf, statische Dateien selbst auszuliefern, da dies die Aufgabe eines dedizierten Webservers ist.
- **Konfiguration**:

```
# settings.py
# In Produktion IMMER auf False setzen!
# Kann über eine Umgebungsvariable gesteuert werden
DEBUG = os.environ.get('DEBUG') == 'True' # Standard ist False
```

1.2 `ALLOWED_HOSTS` (Pflicht bei `DEBUG=False`)

- **Was es tut**: Aus Sicherheitsgründen muss man Django mitteilen, unter welchen Domain-Namen die Anwendung erreichbar sein darf. Dies schützt vor "HTTP Host Header"-Angriffen.
- **Konfiguration**: `ALLOWED_HOSTS` ist eine Liste von Strings. Wenn ein Browser eine Anfrage stellt, prüft Django den `Host`-Header der Anfrage. Wenn dieser nicht in der `ALLOWED_HOSTS`-Liste enthalten ist, wird die Anfrage mit einem Fehler (Statuscode 400) abgewiesen.

```
# settings.py
ALLOWED_HOSTS = ['www.meine-rezept-plattform.de', 'meine-rezept-plattform.de']
```

1.3 `SECRET_KEY` (Muss geheim bleiben)

- **Was es tut**: Dies ist ein Schlüssel, der für kryptografische Signaturen (z.B. für Sessions und Passwort-Reset-Links) verwendet wird.
- **Konfiguration**: Er sollte, wie im vorherigen Skript besprochen, unbedingt aus einer Umgebungsvariable geladen und niemals im Code oder in Git gespeichert werden.

```
# settings.py
SECRET_KEY = os.environ.get('SECRET_KEY')
```

1.4 Datenbank-Konfiguration

- **Was zu tun ist**: Sicherstellen, dass die `DATABASES`-Einstellung auf die Produktionsdatenbank (z.B. MySQL oder PostgreSQL) zeigt und die Zugangsdaten ebenfalls aus Umgebungsvariablen geladen werden. Die Verwendung von `db.sqlite3` ist im Produktivbetrieb nicht zu empfehlen.

1.5 Konfiguration für Statische Dateien (`STATIC_ROOT`)

Wenn `DEBUG=False` ist, liefert Django keine statischen Dateien mehr selbst aus. Dies ist nun die Aufgabe des Webservers (z.B. Nginx). Wir müssen Django aber anweisen, alle statischen Dateien an einem zentralen Ort zu sammeln.

- `STATIC_ROOT`: Dies ist der Pfad zu einem **einzigen Verzeichnis**, in das alle statischen Dateien deines Projekts (aus allen Apps und `STATICFILES_DIRS`) für das Deployment kopiert werden.

```
# settings.py
STATIC_URL = '/static/'

# Der Ordner, in den `collectstatic` alle statischen Dateien kopiert.
STATIC_ROOT = BASE_DIR / 'staticfiles'
```

- **Der `collectstatic`-Befehl**: Dieser Befehl muss während des Deployments auf dem Server ausgeführt werden.

  ```
  python manage.py collectstatic
  ```

  Er sammelt alle CSS-, JS- und Bild-Dateien aus deiner Anwendung und legt sie in dem in `STATIC_ROOT` definierten Ordner ab. Der Webserver wird dann so konfiguriert, dass er Anfragen an `/static/` direkt aus diesem Ordner bedient.

2. Die Produktions-Server-Architektur (Gunicorn & Nginx)

Der `python manage.py runserver` ist nur für die Entwicklung. Im Produktivbetrieb wird eine Kombination aus einem Applikationsserver und einem Webserver verwendet.

- **Gunicorn (Applikationsserver)**: Ein Python-Programm, das deine Django-Anwendung ausführt. Es kann mehrere "Worker"-Prozesse starten, um mehrere Anfragen gleichzeitig zu bearbeiten, und ist deutlich robuster als der Entwicklungsserver. Gunicorn lauscht auf einem internen Port (z.B. `localhost:8000`).
- **Nginx (Webserver)**: Der Server, der direkt mit dem Internet verbunden ist. Er nimmt alle Anfragen auf Port 80 (HTTP) oder 443 (HTTPS) entgegen und verteilt sie:
  1. **Anfragen für statische Dateien** (URL beginnt mit `/static/`): Nginx greift direkt auf den `STATIC_ROOT`-Ordner zu und liefert die Datei aus. **Django wird nicht kontaktiert.**
  2. **Anfragen für Mediendateien** (URL beginnt mit `/media/`): Nginx greift direkt auf den `MEDIA_ROOT`-Ordner zu und liefert die Datei aus. **Django wird nicht kontaktiert.**
  3. **Alle anderen Anfragen** (dynamische Seiten): Nginx leitet die Anfrage an Gunicorn weiter, der sie von Django verarbeiten lässt. Django rendert das Template und sendet die HTML-Antwort über Gunicorn zurück an Nginx, der sie an den Benutzer ausliefert.

3. Der Deployment-Prozess im Überblick

Ein typischer Deployment-Prozess für eine reine Django-Anwendung sieht grob so aus:

1. **Code bereitstellen**: Den finalen Code auf den Server bringen (z.B. über `git pull`).
2. **Abhängigkeiten installieren**: Python-Pakete in einer virtuellen Umgebung installieren (`pip install -r requirements.txt`).
3. **Umgebungsvariablen setzen**: Die `.env`-Datei (oder ein anderes System) mit den Produktionswerten für `SECRET_KEY`, `DEBUG=False`, `ALLOWED_HOSTS` und die Datenbank-Zugangsdaten auf dem Server anlegen.
4. **Datenbank-Migrationen anwenden**:

   ```
   python manage.py migrate
   ```

5. **Statische Dateien sammeln**:

   ```
   python manage.py collectstatic
   ```

6. **Server-Prozesse starten/neustarten**: Die Gunicorn- und Nginx-Dienste auf dem Server starten oder neu laden, damit sie die Code-Änderungen übernehmen.

Anhang B: Gunicorn & WhiteNoise – Die praktischen Werkzeuge für das Deployment

1. Gunicorn (Der Applikationsserver - Nahezu immer benötigt)

- **Was es ist:** Gunicorn ("Green Unicorn") ist ein **WSGI HTTP Server**. Er ist die Brücke zwischen einem Webserver (wie Nginx) und deiner Django-Anwendung.
- **Warum es benötigt wird:** Der Django-Entwicklungsserver (`runserver`) ist **nicht** für den Produktivbetrieb geeignet. Er ist langsam, unsicher und kann nur eine Anfrage auf einmal bearbeiten. Gunicorn ist ein robuster Server, der dafür gebaut ist, deine Python-Anwendung unter Last zu betreiben. Er startet mehrere "Worker"-Prozesse, um viele Anfragen gleichzeitig zu bewältigen.

- **Wie es funktioniert:**
    1. **Installation:**

        ```
        pip install gunicorn
        ```

    2. **Verwendung:** Im Produktivbetrieb startest du Django nicht mehr mit `runserver`, sondern mit Gunicorn und verweist auf die `wsgi.py`-Datei deines Projekts.

        ```
        # Beispielhafter Befehl zum Starten
        gunicorn --workers 3 mein_projekt.wsgi
        ```

        Ein Webserver wie Nginx wird dann so konfiguriert, dass er Anfragen an den Gunicorn-Prozess weiterleitet.

Fazit zu Gunicorn: Für fast jedes professionelle Django-Deployment auf einem Linux-Server ist Gunicorn (oder ein ähnlicher WSGI-Server wie uWSGI) ein unverzichtbarer Baustein.

2. WhiteNoise (Der Static-File-Helfer - Eine nützliche, optionale Vereinfachung)

Jetzt wird es interessant, denn WhiteNoise verändert, wie wir über statische Dateien im Produktivbetrieb denken.

- **Was es ist:** WhiteNoise ist eine Python-Bibliothek, die es deiner Django-Anwendung erlaubt, ihre eigenen statischen Dateien **effizient und sicher** auszuliefern, auch wenn `DEBUG=False` ist. Es fügt Caching- und Komprimierungs-Funktionen hinzu, die weit über das hinausgehen, was der Django-Entwicklungsserver kann.
- **Wann und warum wird es benötigt? Die zwei Strategien:**
    - **Strategie 1: Der klassische Weg mit Nginx (OHNE WhiteNoise)**
        - In dieser Konfiguration ist Nginx für **alles** zuständig, was statisch ist.
        - Du führst `python manage.py collectstatic` aus.
        - Du konfigurierst Nginx so, dass es Anfragen an `/static/` direkt aus dem `STATIC_ROOT`-Ordner beantwortet. Django wird für diese Anfragen gar nicht erst kontaktiert.
        - **Vorteil:** Maximale Performance, da der hochoptimierte Webserver die Dateien ausliefert.
        - **Nachteil:** Erfordert mehr Konfigurationsaufwand auf dem Server (Nginx muss konfiguriert werden).
    - **Strategie 2: Der vereinfachte Weg mit WhiteNoise (MIT WhiteNoise)**
        - In dieser Konfiguration lässt du deine Django-Anwendung (die über Gunicorn läuft) die statischen Dateien selbst ausliefern. WhiteNoise sorgt dafür, dass dies auf eine für die Produktion geeignete Weise geschieht.
        - Du führst ebenfalls `python manage.py collectstatic` aus.
        - Du musst **keine** spezielle Konfiguration für statische Dateien in Nginx vornehmen. Nginx leitet einfach alle Anfragen an Gunicorn/Django weiter. Wenn eine Anfrage für eine statische Datei kommt, fängt WhiteNoise sie ab und liefert die korrekte Datei aus `STATIC_ROOT` aus.
        - **Vorteil:** Deutlich einfachere Konfiguration. Ideal für Platform-as-a-Service (PaaS) Anbieter wie Heroku, wo man oft keinen Zugriff auf die Nginx-Konfiguration hat.
        - **Nachteil:** Potenziell minimal weniger performant als die Auslieferung direkt durch Nginx, aber für die meisten Anwendungen ist der Unterschied vernachlässigbar.
- **Wie WhiteNoise konfiguriert wird:**
    1. **Installation:**

        ```
        pip install whitenoise
        ```

    2. **`settings.py` anpassen (`MIDDLEWARE`)**: WhiteNoise wird als Middleware hinzugefügt, am besten direkt nach der `SecurityMiddleware`.

        ```
        ```

```
# settings.py
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware', # WhiteNoise hier einfügen
    'django.contrib.sessions.middleware.SessionMiddleware',
    # ...
]
```

3. `settings.py` **anpassen (**`STATIC_FILES_STORAGE`**)** (Optional, aber empfohlen für Komprimierung):

```
# settings.py
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

Zusammenfassung: Welche Strategie ist die beste?

- **Gunicorn** ist fast immer dein Freund und die Basis für den Betrieb der Django-Anwendung.
- Die Wahl zwischen **Nginx für Static Files** vs. **WhiteNoise** hängt von deiner Hosting-Umgebung und deinen Präferenzen ab:
  - **Du hast volle Kontrolle über einen Server und willst maximale Performance?** -> Konfiguriere Nginx, um statische Dateien direkt auszuliefern (Strategie 1).
  - **Du willst ein möglichst einfaches Deployment oder nutzt eine Plattform wie Heroku?** -> Verwende WhiteNoise (Strategie 2).

Tutorial für Python Django App auf PythonAnywhere.com

> Django Deployment Tutorial