Django-React Deploy Tutorial

Wir werden folgende, bewährte und kostenlose Dienste nutzen:

- **Für das Django Backend: Render** als Platform-as-a-Service (PaaS). Es ist ein moderner Heroku-Nachfolger und bietet einen kostenlosen Plan für den Webserver und eine kostenlose PostgreSQL-Datenbank.
- **Für das React Frontend: Netlify**. Einer der besten Dienste für das Hosting von statischen Seiten (was ein gebautes React-Projekt ist). Der kostenlose Plan ist extrem großzügig und der Prozess ist super einfach.
- **Für den Code: GitHub**. Wir gehen davon aus, dass dein Code in einem GitHub-Repository liegt. Sowohl Render als auch Netlify lassen sich perfekt damit verbinden.

Canvas: Detaillierte Anleitung zum Deployment von Django & React

Voraussetzungen

1. Dein Django-Projekt und dein React-Projekt sind in einem oder zwei GitHub-Repositories. (Ein einziges Repo mit "frontend" und "backend" Ordnern ist in Ordnung).
2. Du hast einen kostenlosen Account bei GitHub, Render und Netlify.
3. Du hast `git` auf deinem Computer installiert und kennst die Grundlagen (`add`, `commit`, `push`).

Teil A: Vorbereitung des Django-Backends für die Produktion

Dein lokaler Entwicklungsserver (`manage.py runserver`) ist **nicht** für die Produktion geeignet. Wir müssen das Projekt so konfigurieren, dass es auf einem echten Webserver laufen kann.

Schritt 1: Produktions-Abhängigkeiten installieren

Füge die folgenden Python-Pakete zu deinem Projekt hinzu.

```
# In deiner virtuellen Umgebung im Django-Projekt-Ordner:
pip install gunicorn psycopg2-binary dj-database-url whitenoise python-dotenv
```

- `gunicorn`: Ein produktionsreifer WSGI HTTP Server, der deine Django-Anwendung ausführt.
- `psycopg2-binary`: Der Adapter, damit Django mit einer PostgreSQL-Datenbank sprechen kann.
- `dj-database-url`: Hilft uns, die Datenbankverbindung über eine einfache URL-Umgebungsvariable zu konfigurieren.
- `whitenoise`: Ermöglicht es deiner Django-Anwendung, ihre eigenen statischen Dateien (CSS, JS, Bilder aus dem `static` Ordner) effizient auszuliefern.
- `python-dotenv`: Zum Laden von Umgebungsvariablen aus einer `.env`-Datei während der lokalen Entwicklung.

Schritt 2: `requirements.txt` aktualisieren

Erstelle oder aktualisiere deine `requirements.txt`-Datei.

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

Diese Datei wird Render mitteilen, welche Pakete installiert werden müssen.

Schritt 3: `settings.py` für die Produktion anpassen

Öffne die `settings.py`-Datei deines Django-Projekts.

```
# settings.py

import os
import dj_database_url
from pathlib import Path
```

```python
from dotenv import load_dotenv # Am Anfang der Datei hinzufügen

# ...

# Lade Umgebungsvariablen aus .env (nur für lokale Entwicklung)
load_dotenv()

# SECRET_KEY sollte niemals im Code stehen!
# Wir laden sie aus einer Umgebungsvariable.
SECRET_KEY = os.environ.get('SECRET_KEY')

# DEBUG wird in der Produktion auf False gesetzt.
# Wir benutzen 'RENDER' als Indikator, dass wir auf Render laufen.
# os.environ.get('RENDER') wird in der Render-Umgebung automatisch gesetzt.
DEBUG = 'RENDER' not in os.environ

# Deine zukünftigen Live-URLs.
# Die Render-URL kommt später hinzu. 'localhost' für lokale Entwicklung.
ALLOWED_HOSTS = ['localhost', '127.0.0.1']

RENDER_EXTERNAL_HOSTNAME = os.environ.get('RENDER_EXTERNAL_HOSTNAME')
if RENDER_EXTERNAL_HOSTNAME:
    ALLOWED_HOSTS.append(RENDER_EXTERNAL_HOSTNAME)


# ... (deine INSTALLED_APPS)

INSTALLED_APPS = [
    'django.contrib.admin',
    # ... andere Apps
    'corsheaders', # Sollte schon da sein
    'whitenoise.runserver_nostatic', # Wichtig: über django.contrib.staticfiles
    'django.contrib.staticfiles',
    # ...
]

# ... (deine MIDDLEWARE)

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware', # Wichtig: direkt nach SecurityMiddleware
    'django.contrib.sessions.middleware.SessionMiddleware',
    'corsheaders.middleware.CorsMiddleware', # Sollte schon da sein
    'django.middleware.common.CommonMiddleware',
    # ...
]

# ...

# Datenbank-Konfiguration
# Lokal wird weiterhin sqlite benutzt, wenn keine DATABASE_URL gesetzt ist.
# In Produktion auf Render wird die DATABASE_URL automatisch gesetzt.
DATABASES = {
    'default': dj_database_url.config(
        default='sqlite:///db.sqlite3',
        conn_max_age=600,
        ssl_require=True if 'RENDER' in os.environ else False # SSL für Render DB erzwingen
    )
}


# ...

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.2/howto/static-files/

STATIC_URL = '/static/'
```

```
# Dieser Pfad ist für die gebauten Static-Dateien in der Produktion
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

# Konfiguration für Whitenoise
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

Schritt 4: CORS für die Live-URL vorbereiten

In deiner `settings.py`, bereite die CORS-Einstellung vor. Wir kennen die React-URL noch nicht, aber wir können schon mal einen Platzhalter einfügen und sie später aktualisieren.

```
# settings.py

# Die URL deines React-Frontends (wird später von Netlify vergeben)
# und die URL deines Backends zu Entwicklungszwecken
CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",
    "http://127.0.0.1:3000",
]

# Später fügen wir hier die Netlify-URL hinzu, z.B. "https://mein-super-projekt.netlify.app"
# Du kannst auch eine Umgebungsvariable dafür verwenden.
REACT_APP_URL = os.environ.get('REACT_APP_URL')
if REACT_APP_URL:
    CORS_ALLOWED_ORIGINS.append(REACT_APP_URL)
```

Schritt 5: `.env`-Datei für lokale Entwicklung

Erstelle eine Datei namens `.env` im Stammverzeichnis deines Django-Projekts (dort, wo `manage.py` liegt). Füge sie **unbedingt** zu deiner `.gitignore`-Datei hinzu!

```
# .env - Diese Datei NICHT auf GitHub hochladen!

# Generiere einen neuen Key, z.B. hier: https://djecrety.ir/
SECRET_KEY='dein-super-geheimer-lokaler-key'

# Für lokale Entwicklung können wir die sqlite DB nutzen,
# daher muss DATABASE_URL hier nicht gesetzt sein.
```

Schritt 6: Code auf GitHub hochladen

Stelle sicher, dass alle deine Änderungen (neue Pakete in `requirements.txt`, angepasste `settings.py`, neue `.gitignore`-Regel) auf GitHub hochgeladen sind.

```
git add .
git commit -m "Configure Django for production deployment"
git push
```

Teil B: Deployment des Django-Backends auf Render

Schritt 1: Neue PostgreSQL-Datenbank auf Render erstellen

1. Logge dich bei Render ein und gehe zum Dashboard.
2. Klicke auf **"New +"** und wähle **"PostgreSQL"**.
3. Gib einen Namen ein (z.B. `mein-projekt-db`).
4. Wähle die **"Free"**-Instanz.
5. Klicke auf **"Create Database"**.

6. Nachdem die Datenbank erstellt wurde, kopiere dir die **"Internal Connection URL"**. Wir brauchen sie gleich.

Schritt 2: Neuen Web-Service auf Render erstellen

1. Klicke im Render-Dashboard auf **"New +"** und wähle **"Web Service"**.
2. Verbinde dein GitHub-Konto und wähle das Repository deines Projekts aus.
3. Render wird deinen Code analysieren. Gib dem Service einen eindeutigen Namen (z.B. `mein-projekt-backend`).
4. Konfiguriere die folgenden Einstellungen:
    - **Environment:** `Python 3`
    - **Region:** Wähle `Frankfurt (Germany)` für die beste Latenz.
    - **Branch:** Wähle deinen Haupt-Branch (z.B. `main` oder `master`).
    - **Build Command:** `pip install -r requirements.txt && python manage.py collectstatic --noinput`
        - Dies installiert die Pakete UND sammelt alle statischen Dateien in den `STATIC_ROOT`-Ordner.
    - **Start Command:** `gunicorn dein_projekt_name.wsgi:application`
        - **Wichtig:** Ersetze `dein_projekt_name` durch den Namen des Ordners in deinem Projekt, der die `wsgi.py`-Datei enthält.
5. Wähle den **"Free"**-Plan.
6. Klicke auf **"Advanced Settings"**, um die Umgebungsvariablen hinzuzufügen.
7. **Umgebungsvariablen hinzufügen:**
    - Klicke auf **"Add Environment Variable"**.
    - **Key:** `DATABASE_URL`
        - **Value:** Füge hier die **"Internal Connection URL"** deiner Render-PostgreSQL-Datenbank ein.
    - **Key:** `SECRET_KEY`
        - **Value:** Generiere einen **neuen, sicheren Secret Key** (z.B. mit `openssl rand -hex 32` im Terminal oder online) und füge ihn hier ein. **Niemals deinen lokalen Key verwenden!**
    - **Key:** `PYTHON_VERSION`
        - **Value:** Die Python-Version, die du verwendest (z.B. `3.11.4`).
    - **Key:** `REACT_APP_URL`
        - **Value:** Dies lassen wir vorerst leer. Wir füllen es, sobald wir die Netlify-URL haben.
8. Klicke auf **"Create Web Service"**.

Render wird nun dein Projekt zum ersten Mal bauen und deployen. Das kann einige Minuten dauern. Du kannst den Fortschritt im "Logs"-Tab verfolgen. Wenn alles gut geht, siehst du am Ende eine Meldung wie "Your service is live 🎉 ". Oben auf der Seite findest du die URL deines Backends (z.B. `https://mein-projekt-backend.onrender.com`).

Teil C: Vorbereitung des React-Frontends für die Produktion

Schritt 1: API-URL dynamisch machen

In deinem React-Code solltest du nicht fest `http://localhost:8000/api/...` verwenden. Wir nutzen Umgebungsvariablen.

1. Erstelle eine Datei `.env.production` im Stammverzeichnis deines React-Projekts.

    ```
    # .env.production
    # Diese URL wird von Netlify beim Build-Prozess verwendet.
    REACT_APP_API_URL=https://dein-backend.onrender.com
    ```

    Ersetze `dein-backend.onrender.com` mit deiner echten Render-URL.

2. Erstelle auch eine `.env.development`-Datei für die lokale Entwicklung.

    ```
    # .env.development
    REACT_APP_API_URL=http://localhost:8000
    ```

3. Passe deinen Code an, um diese Variable zu verwenden. Wenn du z.B. Axios nutzt:

```
// z.B. in einer api.js oder direkt in deinen Komponenten
import axios from 'axios';

const apiClient = axios.create({
  baseURL: process.env.REACT_APP_API_URL
});

// Beispiel-Aufruf
// apiClient.get('/api/posts/');
```

Schritt 2: Code auf GitHub hochladen

Pushe diese Änderungen in dein GitHub-Repository.

```
git add .
git commit -m "Configure React for production API URL"
git push
```

Teil D: Deployment des React-Frontends auf Netlify

Schritt 1: Neue Seite auf Netlify erstellen

1. Logge dich bei Netlify ein und gehe zum "Sites"-Tab.
2. Klicke auf **"Add new site"** und wähle **"Import an existing project"**.
3. Verbinde dein GitHub-Konto und wähle das Repository deines Projekts.

Schritt 2: Build-Einstellungen konfigurieren

Netlify erkennt in der Regel React-Projekte und schlägt die richtigen Einstellungen vor. Überprüfe sie:

- **Branch to deploy:** Dein Haupt-Branch (`main` oder `master`).
- **Base directory:** (Optional) Wenn dein React-Code in einem Unterordner wie `frontend/` liegt, gib hier `frontend` an. Sonst leer lassen.
- **Build command:** `npm run build` (oder `yarn build`).
- **Publish directory:** `build` (oder `dist`, je nachdem, was dein Build-Befehl erzeugt).

Schritt 3: Umgebungsvariable hinzufügen

1. Klicke auf **"Show advanced"** und dann auf **"New variable"**.
2. **Key:** `REACT_APP_API_URL`
3. **Value:** Gib hier die URL deines live Django-Backends auf Render ein (z.B. `https://mein-projekt-backend.onrender.com`).

Schritt 4: Deployment starten

1. Klicke auf **"Deploy site"**.

Netlify wird nun dein Projekt bauen und deployen. Das dauert nur ein oder zwei Minuten. Danach erhältst du eine zufällige URL wie `https://whimsical-biscuit-12345.netlify.app`. Das ist die Live-URL deines Frontends!

Teil E: Finale Verbindung und Abschluss

Wir müssen Django noch die genaue URL unseres Frontends mitteilen, damit CORS-Anfragen erlaubt werden.

Schritt 1: CORS in Django finalisieren

1. Gehe zurück zum Dashboard deines Web-Services auf **Render**.

2. Gehe zum **"Environment"**-Tab.
3. Finde die `REACT_APP_URL`-Variable, die wir vorher leer gelassen haben, oder füge sie neu hinzu.
4. **Key:** `REACT_APP_URL`
5. **Value:** Füge hier deine vollständige Netlify-URL ein (z.B. `https://whimsical-biscuit-12345.netlify.app`).
6. Speichere die Änderungen. Render wird dein Backend automatisch neu starten.

Schritt 2: Alles testen!

Öffne deine Netlify-URL im Browser. Das React-Frontend sollte laden. Teste eine Funktion, die Daten vom Backend abruft (z.B. ein Login oder das Laden einer Liste von Posts). Überprüfe die Entwicklerkonsole deines Browsers (F12) auf Fehler, insbesondere auf CORS-Fehler. Wenn keine auftreten, hast du es geschafft!

Glückwunsch! Du hast jetzt ein voll funktionsfähiges React/Django-Projekt in der Produktion, gehostet auf einer robusten und kostenlosen Infrastruktur.