14 - Django: Datenbankwechsel zu MySQL & Security Best Practices

- **Lernziele**:
  - Die Notwendigkeit eines Wechsels von SQLite zu einer Server-Datenbank wie MySQL für Produktionsumgebungen verstehen.
  - Eine MySQL-Datenbank-Instanz mit Docker und `docker-compose` einfach und reproduzierbar aufsetzen können.
  - Datenbank-Zugangsdaten sicher mit `python-dotenv` und Umgebungsvariablen verwalten, anstatt sie direkt in `settings.py` zu schreiben.
  - Den kompletten Prozess der Datenmigration mit `dumpdata` und `loaddata` beherrschen.

1. Warum von SQLite zu MySQL wechseln? (Der Kontext)

- **SQLite (Entwicklung)**: Perfekt für die Entwicklung. Es ist eine einzelne Datei, erfordert keine Installation oder Konfiguration und ist sofort einsatzbereit.
- **MySQL (Produktion)**: Wenn eine Webseite von vielen Benutzern gleichzeitig genutzt wird, stößt SQLite an seine Grenzen, insbesondere bei gleichzeitigen Schreibvorgängen. MySQL ist ein dedizierter Datenbank-Server, der darauf ausgelegt ist, viele Anfragen parallel zu verarbeiten, bietet erweiterte Features, bessere Performance und höhere Datensicherheit. Für jede Live-Anwendung ist ein Wechsel zu einem solchen System (wie MySQL oder PostgreSQL) unerlässlich.

2. Schritt 1: MySQL-Umgebung einrichten (Die empfohlene Methode: Docker)

Anstatt MySQL nativ auf jedem Betriebssystem (Windows, macOS, Linux) zu installieren, verwenden wir **Docker**. Docker erlaubt es uns, eine MySQL-Datenbank in einem isolierten Container zu betreiben, der auf jedem System exakt gleich läuft.

2.1 Docker installieren (je nach Betriebssystem)

Die Voraussetzung ist eine funktionierende Docker-Installation.

- **Windows**:
  - Der beste Weg ist die Installation von **Docker Desktop**.
  - Docker Desktop benötigt das **WSL 2 (Windows Subsystem for Linux)**. Der Installationsprozess von Docker Desktop ist sehr benutzerfreundlich und prüft, ob WSL 2 vorhanden ist. Falls nicht, bietet er an, es zu installieren und zu konfigurieren.
  - **Anleitung**: Lade Docker Desktop von der offiziellen Docker-Webseite herunter und folge den Installationsanweisungen.
- **macOS**:
  - Auch hier ist **Docker Desktop** die Standardlösung.
  - **Anleitung**: Lade Docker Desktop von der offiziellen Docker-Webseite herunter. Achte darauf, den korrekten Installer für deinen Prozessor auszuwählen (Apple Silicon M1/M2/M3 oder Intel).
- **Linux (z.B. Ubuntu/Debian)**:
  - Hier können die Docker-Komponenten direkt über den Paketmanager installiert werden.
  - **Anleitung**: Öffne ein Terminal und führe die folgenden Befehle aus:

```
# System aktualisieren
sudo apt update
# Docker und Docker Compose installieren
sudo apt install docker.io docker-compose -y
# Den aktuellen Benutzer zur Docker-Gruppe hinzufügen, um `sudo` zu vermeiden
sudo usermod -aG docker $USER
# Wichtig: Nach diesem Befehl neu anmelden oder das System neu starten,
# damit die Gruppenänderung wirksam wird.
```

2.2 Die `docker-compose.yml`-Datei erstellen

Erstelle im Hauptverzeichnis deines Django-Projekts (auf derselben Ebene wie `manage.py`) eine neue Datei namens `docker-compose.yml`.

```yaml
# docker-compose.yml
version: '3.8'

services:
    db:
    image: mysql:8.0  # Wir verwenden das offizielle MySQL-Image in Version 8.0
    container_name: mysql_db_container
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    environment:
        MYSQL_ROOT_PASSWORD: 'root_password_sehr_sicher' # Passwort für den root-User in MySQL
        MYSQL_DATABASE: 'recipes_db' # Name der Datenbank, die erstellt werden soll
        MYSQL_USER: 'recipes_user' # Benutzername, der erstellt werden soll
        MYSQL_PASSWORD: 'user_password_sehr_sicher' # Passwort für den neuen Benutzer
    ports:
        - "3307:3306" # Leitet den Port 3306 im Container auf Port 3307 auf deinem Computer um
    volumes:
        - mysql_data:/var/lib/mysql

volumes:
    mysql_data:
```

Diese Datei definiert einen Service namens `db`, der eine MySQL-Datenbank mit einer spezifischen Datenbank, einem Benutzer und Passwörtern erstellt.

- **Wichtig**: Der Port wird auf `3307` umgeleitet, um Konflikte mit einer eventuell lokal installierten MySQL-Instanz auf dem Standardport `3306` zu vermeiden.

2.3 Datenbank-Container verwalten

- **Starten**: Um den in der Datei definierten MySQL-Container zu starten, führe im Terminal (im selben Ordner wie die `docker-compose.yml`) aus:

  ```
  docker-compose up -d
  ```

  - `up`: Startet die Services.
  - `-d` (detached): Lässt die Container im Hintergrund laufen.

- **Status prüfen**: Um zu sehen, welche Container laufen, benutze:

  ```
  docker-compose ps
  ```

- **Herunterfahren**: Um die Container zu stoppen und zu entfernen, benutze:

  ```
  docker-compose down
  ```

  - Dies stoppt und entfernt die Container, aber die Daten bleiben im `mysql_data`-Volume erhalten. Beim nächsten `docker-compose up` ist die Datenbank also wieder im selben Zustand.

3. Schritt 2: Django-Projekt konfigurieren

3.1 Die unsichere Methode: Zugangsdaten in `settings.py`

Zuerst installieren wir den Datenbank-Treiber:

```
pip install pymysql
```

Und dann in settings.py (am besten ganz oben):

```
#settings.py oben
import pymysql
pymysql.install_as_MySQLdb()
```

Dann tragen wir die Zugangsdaten direkt in `settings.py` ein. **Dies ist eine schlechte Praxis und dient nur zur Demonstration.**

```
# settings.py (schlechte Variante - nicht nachmachen!)
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'recipes_db',              # Aus docker-compose.yml
        'USER': 'recipes_user',            # Aus docker-compose.yml
        'PASSWORD': 'user_password_sehr_sicher', # Aus docker-compose.yml
        'HOST': '127.0.0.1',               # oder 'localhost'
        'PORT': '3307',                    # Der Port, den wir nach außen freigegeben haben
    }
}
```

Problem: Die Zugangsdaten stehen im Klartext im Code und würden in Git landen. Das ist ein großes Sicherheitsrisiko.

Ja, hier ist das überarbeitete Kapitel 3.2, das sowohl die Variante mit `python-dotenv` als auch die mit `django-environ` zeigt und die beiden Methoden vergleicht.

3.2 Die sichere Methode: Umgebungsvariablen verwalten

Zugangsdaten wie Passwörter oder `SECRET_KEY` gehören niemals direkt in den Code oder in ein Versionierungssystem wie Git. Die beste Methode ist, sie in Umgebungsvariablen auszulagern. Für die lokale Entwicklung nutzen wir eine `.env`-Datei, um diese Variablen zu simulieren. Hierfür gibt es zwei beliebte Bibliotheken.

Methode A: `python-dotenv` (Der einfache, universelle Weg)

Diese Bibliothek ist ein einfacher Standard, um Variablen aus einer `.env`-Datei in die Umgebung (`os.environ`) zu laden.

1. **Bibliothek installieren**:

   ```
   pip install python-dotenv
   ```

2. **`.env`-Datei erstellen** (im Projekt-Hauptverzeichnis, auf derselben Ebene wie `manage.py`):

   ```
   # .env
   # Diese Datei gehört in die .gitignore!
   SECRET_KEY="django-insecure-dein-zufälliger-key-hier"
   DEBUG=True
   DB_NAME=recipes_db
   DB_USER=recipes_user
   DB_PASSWORD=user_password_sehr_sicher
   DB_HOST=127.0.0.1
   DB_PORT=3307
   ```

3. **`.gitignore`-Datei anpassen**: Füge die Zeile `.env` hinzu, um die Datei vor Git zu schützen.

4. **`settings.py` anpassen**:

```
# settings.py
import os
from dotenv import load_dotenv

# Lädt die Variablen aus der .env-Datei in die Umgebung
load_dotenv()

# ...

# Liest den Secret Key aus der Umgebung
SECRET_KEY = os.environ.get('SECRET_KEY')

# Wichtig: os.environ.get() gibt immer einen String zurück!
# Für einen Boolean muss man den String manuell vergleichen.
DEBUG = os.environ.get('DEBUG') == 'True'

# ...

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.environ.get('DB_NAME'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASSWORD'),
        'HOST': os.environ.get('DB_HOST'),
        'PORT': os.environ.get('DB_PORT'),
    }
}
```

Methode B: `django-environ` (Der bessere, Django-spezifische Weg)

Diese Bibliothek wurde speziell für Django entwickelt. Sie nutzt intern ebenfalls `python-dotenv`, bietet aber viele Komfortfunktionen wie automatisches Type-Casting und das Parsen von Datenbank-URLs.

1. **Bibliothek installieren**:

```
pip install django-environ
```

2. **`.env`-Datei erstellen**: Die `.env`-Datei kann hierfür vereinfacht werden, indem eine Datenbank-URL verwendet wird.

```
# .env
# Diese Datei gehört in die .gitignore!
SECRET_KEY="django-insecure-dein-zufälliger-key-hier"
DEBUG=True

# Alle DB-Infos in einer einzigen URL, Format: db_type://user:password@host:port/db_name
DATABASE_URL=mysql://recipes_user:user_password_sehr_sicher@127.0.0.1:3307/recipes_db
```

3. **`.gitignore`-Datei anpassen**: Auch hier muss `.env` in die `.gitignore`.

4. **`settings.py` anpassen**:

```
# settings.py
import environ
import os

# Initialisiere environ
env = environ.Env(
    # set casting, default value
```

```
    DEBUG=(bool, False)
)

# Lese die .env-Datei (BASE_DIR muss korrekt definiert sein)
environ.Env.read_env(os.path.join(BASE_DIR, '.env'))

# ...

# Variablen mit automatischem Type-Casting auslesen
SECRET_KEY = env('SECRET_KEY')
DEBUG = env('DEBUG') # env.bool('DEBUG') funktioniert auch

# ...

# Die Datenbank-URL wird automatisch in die korrekte Dictionary-Struktur geparst
DATABASES = {
    'default': env.db(),
}
```

Fazit: `python-dotenv` vs. `django-environ` – Was ist besser?

Für Django-Projekte ist **`django-environ` die klar bessere und empfohlene Methode**.

- **`python-dotenv`** ist einfach und leichtgewichtig. Der große Nachteil ist, dass es **kein automatisches Type-Casting** durchführt. Wie im Beispiel gezeigt, wird `DEBUG=True` aus der `.env`-Datei als String `"True"` gelesen. Man muss dies manuell mit `os.environ.get('DEBUG') == 'True'` in einen Boolean umwandeln. Das ist fehleranfällig.
- **`django-environ`** löst dieses Problem elegant. `env('DEBUG')` oder `env.bool('DEBUG')` wandelt den Wert automatisch in den korrekten Boolean-Typ (`True`/`False`) um. Die größte Stärke ist jedoch das Parsen von Datenbank-URLs mit `env.db()`. Eine einzige, saubere `DATABASE_URL` in der `.env`-Datei zu haben, ist eine gängige Best Practice (siehe "12-Factor App"-Prinzipien) und macht die Konfiguration in `settings.py` deutlich übersichtlicher und weniger fehleranfällig.

Empfehlung: Nutze `django-environ` für deine Django-Projekte. Es ist die robustere, sicherere und modernere Lösung.

4. Schritt 3: Der eigentliche Datenumzug

1. **Daten aus SQLite exportieren (`dumpdata`)**: Stelle sicher, dass in `settings.py` noch die **alte SQLite-Datenbank** konfiguriert ist. Führe dann aus:

   ```
   python manage.py dumpdata > data.json
   ```

2. **Datenbankschema in neuer DB erstellen (`migrate`)**: Ändere jetzt die `settings.py` auf die **neue MySQL-Datenbank**. Führe dann `migrate` aus, um die Tabellenstruktur in der leeren MySQL-Datenbank zu erstellen.

   ```
   python manage.py migrate
   ```

3. **Daten in MySQL importieren (`loaddata`)**: Importiere nun die gesicherten Daten in die neuen Tabellen.

   ```
   python manage.py loaddata data.json
   ```

5. Schritt 4: Testen und Abschluss

Starte den Entwicklungsserver. Deine Anwendung sollte jetzt mit der MySQL-Datenbank verbunden sein und alle alten Daten enthalten.

```
python manage.py runserver
```

Fazit

- **Docker für Konsistenz**: Die Verwendung von Docker und `docker-compose` ist der moderne Standard, um Entwicklungsumgebungen systemunabhängig und reproduzierbar zu machen.
- **Sicherheit zuerst**: Zugangsdaten gehören niemals direkt in den Quellcode. Die Auslagerung in Umgebungsvariablen ist eine grundlegende Sicherheitspraxis.
- **Der Migrations-Prozess**: Der Datenumzug ist ein klar definierter Prozess: `dumpdata` (alte DB) -> `settings.py` ändern -> `migrate` (neue DB) -> `loaddata` (neue DB).

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) wird der vollständige Datenbankwechsel exemplarisch durchgeführt.

1. **Docker-Setup**: Eine `docker-compose.yml` für eine `polls_db` erstellen und den Container starten.
2. **Sichere Konfiguration**: Eine `.env`-Datei mit den Datenbank-Zugangsdaten erstellen und die `settings.py` anpassen, um `python-dotenv` zu verwenden.
3. **Datenmigration**:
    - `dumpdata` aus der bestehenden SQLite-Datenbank ausführen.
    - `settings.py` auf MySQL umstellen.
    - `migrate` auf der leeren MySQL-Datenbank ausführen.
    - `loaddata` ausführen, um die alten Umfragedaten zu importieren.
4. **Testen**: Die Anwendung starten und prüfen, ob alle alten Umfragen angezeigt werden.

Cheat Sheet

Docker Compose Befehle

- **Starten (im Hintergrund)**:

```
docker-compose up -d
```

- **Status prüfen**:

```
docker-compose ps
```

- **Stoppen & Entfernen**:

```
docker-compose down
```

Django Management Befehle

- **Daten exportieren**:

```
python manage.py dumpdata > data.json
```

- **Daten importieren**:

```
python manage.py loaddata data.json
```

Sichere `settings.py`-Konfiguration

```
```

1. ```
   pip install python-dotenv
   ```
2. `.env`-Datei erstellen und in `.gitignore` eintragen.
3. In `settings.py`: `load_dotenv()` aufrufen und mit `os.environ.get('KEY')` auf Variablen zugreifen.

Übungsaufgaben

1. **Testprojekt umstellen**:
   - Ein kleines Testprojekt (z.B. das aus früheren Übungen mit `Product`- und `Author`/`Book`-Modellen) nehmen.
   - Einige Beispieldaten in der SQLite-Datenbank erstellen.
   - Eine `docker-compose.yml` und eine `.env`-Datei für eine neue Test-Datenbank (`test_db`) erstellen.
   - Den vollständigen Migrationsprozess durchführen: `dumpdata`, `settings.py` ändern, `migrate`, `loaddata`.
   - Prüfen, ob alle Daten in der neuen MySQL-Datenbank vorhanden sind.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die "Community Recipe Sharing Platform" wird nun von SQLite auf eine robustere MySQL-Datenbank umgestellt.

Aufgabe:

1. **Docker-Umgebung für das Projekt einrichten**:
   - Eine `docker-compose.yml`-Datei im Hauptverzeichnis des `recipe_platform`-Projekts erstellen. Die Konfiguration kann aus dem Skript-Beispiel übernommen werden (Datenbankname, Benutzer, Passwörter können gleich bleiben oder angepasst werden).
   - Den Datenbank-Container mit `docker-compose up -d` starten.
2. **Sichere Konfiguration der Zugangsdaten**:
   - `python-dotenv` in der virtuellen Umgebung installieren.
   - Eine `.env`-Datei im Projekt-Hauptverzeichnis erstellen und die Zugangsdaten für die MySQL-Datenbank (Name, User, Passwort, Host, Port) dort ablegen.
   - Die `.env`-Datei zur `.gitignore`-Datei hinzufügen.
   - Die `settings.py` des Projekts so anpassen, dass sie die `.env`-Datei lädt und die Datenbank-Zugangsdaten aus den Umgebungsvariablen liest.
3. **Daten migrieren**:
   - Sicherstellen, dass die `settings.py` noch auf die alte `db.sqlite3`-Datei zeigt. Alle bisher erstellten Rezepte mit `python manage.py dumpdata recipes > recipes_data.json` in eine JSON-Datei exportieren. (Nur die `recipes`-App exportieren, um Konflikte zu minimieren).
   - Die `settings.py` auf die neue MySQL-Datenbankkonfiguration umstellen.
   - `python manage.py migrate` ausführen, um alle Tabellen in der neuen, leeren MySQL-Datenbank zu erstellen.
   - Die gesicherten Rezeptdaten mit `python manage.py loaddata recipes_data.json` importieren.
4. **Funktionalität überprüfen**:
   - Den Entwicklungsserver starten.
   - Die Webseite aufrufen und prüfen, ob alle zuvor erstellten Rezepte, Benutzer und anderen Daten korrekt angezeigt werden und die Anwendung wie erwartet funktioniert.
   - Im Admin-Bereich nachsehen, ob die Daten vorhanden sind.
   ```