05 - Django: Models und ORM-Grundlagen

Einleitung

- **Themen**: Dieses Skript führt in Django Models als zentrale Datenstruktur ein. Es wird behandelt, wie Models als "Blueprint" für Datenbanktabellen dienen.
- **Fokus**: Das Verständnis, wie Daten in einer Django-Anwendung strukturiert werden. Schwerpunkte sind die Konfiguration von Modell-Feldern, die Abgrenzung zu Formular-Widgets und der Migrationsprozess.
- **Lernziele**:
  - Das Konzept eines Django Models als Python-Klasse verstehen.
  - Die wichtigsten Feldtypen und ihre **Feld-Optionen** (`null`, `blank`, `default` etc.) zur Feinabstimmung kennen.
  - Die `class Meta`-Klasse zur Konfiguration von Modell-Metadaten anwenden können.
  - Den zweistufigen Migrationsprozess und seine Funktionsweise im Hintergrund verstehen.

1. Was ist ein Django Model?

Ein Django Model ist die alleinige und endgültige Quelle der Wahrheit über Daten. Jedes Model ist eine Python-Klasse, die von `django.db.models.Model` erbt und eine einzelne Datenbanktabelle repräsentiert. Django nutzt ein **Object-Relational Mapping (ORM)**, um diese Python-Klassen mit Datenbanktabellen zu verbinden, sodass man mit Python statt mit SQL arbeiten kann.

2. Wichtige Modell-Felder

Jedes Attribut eines Models ist eine Instanz einer Feld-Klasse. Der Feldtyp bestimmt den Datentyp der Spalte in der Datenbank.

Feldtyp	Beschreibung	Wichtige Argumente
`CharField`	Zeichenkette	`max_length`: Maximale Länge (Pflichtargument).
`TextField`	Langer Text	Keine zusätzlichen Pflichtargumente.
`IntegerField`	Ganzzahl	`default`, `null`, `blank`
`FloatField`	Dezimalzahl	Wie bei `IntegerField`.
`DecimalField`	Dezimalzahl mit fester Präzision	`max_digits`, `decimal_places` (beide Pflicht).
`BooleanField`	Wahr/Falsch-Wert	`default`: Standardwert (z.B. `False`).
`DateField`	Datum	`auto_now`: Aktualisiert bei jeder Speicherung. `auto_now_add`: Setzt Datum nur bei Erstellung.
`DateTimeField`	Datum und Uhrzeit	Wie bei `DateField`.
`ImageField`	Bilddatei	`upload_to`: Ordner für den Upload. Benötigt `Pillow`.
`FileField`	Datei	`upload_to`: Speicherort der Datei.
`ForeignKey`	Beziehung zu einem anderen Modell (N:1)	`to`: Zielmodell. `on_delete`: Verhalten beim Löschen.

Beispiel zur Demonstration von Feldtypen:

```python
# Ein Modell, das verschiedene Feldtypen zeigt
from django.db import models

class Article(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    publication_date = models.DateTimeField(auto_now_add=True)
    is_published = models.BooleanField(default=False)
    views_count = models.IntegerField(default=0)
```

3. Feld-Optionen: Die Feinabstimmung

Jeder Feldtyp kann mit optionalen Argumenten konfiguriert werden.

- `null=True`: Erlaubt `NULL`-Werte in der Datenbank (DB-Ebene).
- `blank=True`: Erlaubt, dass das Feld in Formularen leer ist (Validierungs-Ebene).
- `default=...`: Setzt einen Standardwert für das Feld.
- `unique=True`: Stellt sicher, dass jeder Wert in dieser Spalte einzigartig ist.
- `choices=[...]`: Bietet eine Liste von Auswahlmöglichkeiten.
- `verbose_name="..."`: Ein menschenlesbarer Name für das Feld.
- `help_text="..."`: Zusätzlicher Hilfetext, der in Formularen angezeigt wird.

Beispiel zur Demonstration von Feld-Optionen:

```
# Ein Modell, das verschiedene Feld-Optionen nutzt
from django.contrib.auth.models import User

class Profile(models.Model):
    GENDER_CHOICES = [
        ('M', 'Männlich'),
        ('F', 'Weiblich'),
        ('D', 'Divers'),
    ]

    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(blank=True, help_text="Erzählen Sie etwas über sich.")
    location = models.CharField(max_length=100, blank=True, verbose_name="Wohnort")
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES, null=True, blank=True)
```

4. Die `class Meta`-Optionen: Metadaten des Modells

Innerhalb eines Models kann eine innere Klasse `Meta` definiert werden, um Konfigurationen festzulegen, die das gesamte Modell betreffen.

- **Zweck**: Beeinflusst das Verhalten des Modells, z.B. die Sortierung von Abfragen oder die Darstellung im Admin-Bereich.
- **Wichtige Optionen**:
  - `ordering`: Legt die Standard-Sortierreihenfolge fest.
  - `verbose_name`: Ein menschenlesbarer Name für das Modell im Singular.
  - `verbose_name_plural`: Der Pluralname für das Modell.
  - `db_table`: Erlaubt die explizite Festlegung des Tabellennamens.

5. Exkurs: Models vs. Forms (Wo gehören Widgets hin?)

- **Models** definieren die **Datenstruktur** und wie Daten in der **Datenbank** gespeichert werden.
- **Forms** definieren, wie Daten vom Benutzer **eingegeben und validiert** werden. Sie sind für die **HTML-Darstellung** zuständig.

Ein **Widget** ist ein Konzept aus **Django Forms**. Es bestimmt, wie ein Formularfeld im HTML gerendert wird (z.B. als `<input type="text">` oder `<textarea>`). Dieses Thema wird detailliert in **Woche 2 und 3** behandelt.

6. Migrationen - Die Brücke zur Datenbank

Die Synchronisation von `models.py` mit der Datenbank erfolgt in zwei Schritten. Dies ist ein Sicherheitsmechanismus, der erlaubt, den Plan der Änderungen zu prüfen, bevor er ausgeführt wird.

1. `makemigrations` **- Der Planer**
   - **Was passiert?** Django vergleicht den aktuellen Zustand der `models.py`-Dateien mit den zuletzt erstellten Migrationsdateien. Aus den erkannten Unterschieden (z.B. ein neues Feld) generiert es eine neue, nummerierte Python-Datei im `migrations/`-Ordner.
   - **Im Hintergrund**: Diese Datei enthält eine "Blaupause" der Änderungen in Python-Code. Sie beschreibt die Operationen (z.B. `migrations.CreateModel`, `migrations.AddField`) auf eine datenbank-agnostische Weise. Man erstellt also einen Plan, ohne sich um die spezifische SQL-Syntax der Zieldatenbank kümmern zu müssen.

     ```
     python manage.py makemigrations
     ```

2. `migrate` **- Der Ausführer**
    - **Was passiert?** Dieser Befehl nimmt die noch nicht angewendeten Migrationsdateien und führt sie in der richtigen Reihenfolge aus.
    - **Im Hintergrund**: Django übersetzt die Python-Anweisungen aus den Migrationsdateien in konkrete SQL-Befehle (z.B. `CREATE TABLE ...`, `ALTER TABLE ... ADD COLUMN ...`), die auf die in `settings.py` konfigurierte Datenbank zugeschnitten sind. Diese SQL-Befehle werden dann ausgeführt, um die Datenbankstruktur physisch zu verändern.

```
python manage.py migrate
```

Fazit

- **Models als Blaupause**: Models definieren die Datenstruktur für die Datenbank.
- **Feld-Optionen & Meta**: Dienen der Feinabstimmung von Feldern und dem Verhalten des gesamten Modells.
- **Models vs. Widgets**: Models speichern Daten, Widgets (aus Forms) stellen sie im HTML dar.
- **Migrationen**: Ein sicherer, zweistufiger Prozess: Zuerst wird ein Plan erstellt (`makemigrations`), dann wird dieser Plan ausgeführt, um die Datenbank zu verändern (`migrate`).

Projekt-Anwendung (Leitfaden-Projekt)

Für das Umfrage-Projekt werden die Modelle für `Question` und `Choice` definiert.

```
# polls/models.py
from django.db import models
import datetime
from django.utils import timezone

class Question(models.Model):
    question_text = models.CharField(max_length=200, verbose_name="Fragetext")
    pub_date = models.DateTimeField(verbose_name="Veröffentlichungsdatum")

    class Meta:
        ordering = ['-pub_date']
        verbose_name = "Umfrage"
        verbose_name_plural = "Umfragen"

    def __str__(self):
        return self.question_text

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE, verbose_name="Zugehörige
Frage")
    choice_text = models.CharField(max_length=200, verbose_name="Antworttext")
    votes = models.IntegerField(default=0, verbose_name="Stimmen")

    def __str__(self):
        return self.choice_text
```

Cheat Sheet

Wichtige Modell-Felder & Argumente

Feldtyp	Beschreibung & Wichtige Argumente
`CharField`	Zeichenkette. `max_length` ist erforderlich.
`TextField`	Langer Text. `blank=True` erlaubt leere Eingaben.

Feldtyp	Beschreibung & Wichtige Argumente
`IntegerField`	Ganzzahl. `default=0` setzt einen Standardwert.
`BooleanField`	Wahr/Falsch-Wert. `default=False` ist üblich.
`DateField`	Datum. `auto_now_add=True` für Erstellungsdatum.
`DateTimeField`	Datum und Zeit. `auto_now=True` für Änderungsdatum.
`ForeignKey`	Beziehung. `to=Modell`, `on_delete=models.CASCADE`.

Wichtige Feld-Optionen (für alle Felder)

- `null=True`: Erlaubt `NULL` in der DB.
- `blank=True`: Erlaubt leere Eingabe im Formular.
- `default=...`: Setzt einen Standardwert.
- `unique=True`: Stellt Einzigartigkeit sicher.
- `verbose_name="..."`: Menschenlesbarer Name für Formulare/Admin.

Wichtige `class Meta`-Optionen

- `ordering = ['-feldname']`: Standard-Sortierung (Minus für absteigend).
- `verbose_name = "Name"` / `verbose_name_plural = "Namen"`

Migrationen

1. **Vorbereiten**:

```
python manage.py makemigrations
```

2. **Anwenden**:

```
python manage.py migrate
```

Übungsaufgaben

1. **Modell `Product` erstellen**:
   - In einer Test-App ein Modell namens `Product` erstellen.
   - Das Modell soll folgende Felder haben: `name` (`CharField`), `description` (`TextField`), `price` (`DecimalField` mit `max_digits=10`, `decimal_places=2`), `stock_quantity` (`IntegerField` mit `default=0`) und `is_available` (`BooleanField` mit `default=True`).
2. **Modell `Book` erweitern**:
   - Ein Modell `Book` erstellen und eine `Meta`-Klasse hinzufügen, die eine Standard-Sortierung nach dem `publication_year` festlegt.
3. **Migrationen für beide Übungen durchführen**:
   - Nach der Definition der Modelle den Befehl `makemigrations` ausführen.
   - Danach den Befehl `migrate` ausführen.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Für die "Community Recipe Sharing Platform" müssen nun die grundlegenden Modelle erstellt werden, die das Herzstück der Anwendung bilden.

Aufgabe:

1. **Models definieren**: In der `recipes/models.py`-Datei sind folgende Modelle zu erstellen:

- **Recipe**:
  - `title`: `CharField` mit `max_length=200` und `verbose_name="Rezepttitel"`.
  - `description`: `TextField` mit `help_text="Eine kurze Zusammenfassung des Rezepts"`.
  - `created_at`: `DateTimeField` mit der Option `auto_now_add=True`.
  - `updated_at`: `DateTimeField` mit der Option `auto_now=True`.
- **Ingredient**:
  - `name`: `CharField` mit `max_length=100`, `unique=True` und `verbose_name="Zutat"`.
- **Step**:
  - `step_number`: `IntegerField` mit `verbose_name="Schrittnummer"`.
  - `description`: `TextField` mit `verbose_name="Beschreibung des Schritts"`.

2. **Meta-Klassen hinzufügen**:
   - Dem `Recipe`-Modell eine `Meta`-Klasse hinzufügen, die eine Standard-Sortierung nach dem `updated_at`-Feld (neueste zuerst) festlegt (`ordering = ['-updated_at']`).
   - Dem `Step`-Modell eine `Meta`-Klasse hinzufügen, die eine Standard-Sortierung nach `step_number` festlegt.
3. **Migrationen durchführen**:
   - Den Befehl ausführen, um die Migrationsdatei für die `recipes`-App zu erstellen:

     ```
     python manage.py makemigrations recipes
     ```

   - Anschließend die Migrationen auf die Datenbank anwenden:

     ```
     python manage.py migrate
     ```

   - Optional: Den `migrations/`-Ordner untersuchen, um die generierte Datei zu verstehen.