Modul 2: Modelle und Datenbankintegration

Ziele

- Verstehen der Grundlagen von Django-Modellen.
- Erstellen und Verwalten von Datenbanktabellen.
- Durchführung von Datenbankmigrationen.
- Arbeiten mit Django ORM: Abfragen, Hinzufügen, Bearbeiten und Löschen von Datenbankeinträgen.
- Vertiefung der Konzepte von ForeignKey, ManyToMany und OneToOne-Beziehungen.
- Vergleich der Methoden zur Generierung von Datenbankeinträgen.
- Übersicht über relevante Datenbankbefehle.
- Umsetzung eines größeren Projekts mit mehreren Modellen, Views und Formularen.

---

1. Einführung in Modelle

Was ist ein Modell?

Ein Modell in Django ist eine Python-Klasse, die eine Tabelle in der Datenbank repräsentiert. Django verwendet das **Object-Relational Mapping (ORM)**, um Python-Klassen und ihre Instanzen mit Datenbanktabellen und -zeilen zu verbinden.

Beispiel:

```python
from django.db import models

class BlogPost(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
```

2. Alle Felder für Django-Modelle (Erweiterte Tabelle)

Feldtyp	Beschreibung	Wichtige Argumente
`CharField`	Zeichenkette	`max_length`: Maximale Länge (Pflichtargument).
`TextField`	Langer Text	Keine zusätzlichen Pflichtargumente.
`IntegerField`	Ganzzahl	`default`: Standardwert. `null`: `True` erlaubt NULL-Werte. `blank`: `True` erlaubt leere Werte in Formularen.
`FloatField`	Dezimalzahl	Wie bei `IntegerField`.
`DecimalField`	Dezimalzahl mit Präzision	`max_digits`: Maximale Anzahl an Ziffern. `decimal_places`: Anzahl der Nachkommastellen.
`BooleanField`	Wahr/Falsch-Wert	`default`: Standardwert.

Feldtyp	Beschreibung	Wichtige Argumente
`DateField`	Datum	`auto_now`: Aktualisiert sich bei jeder Speicherung. `auto_now_add`: Setzt das Datum nur bei der Erstellung.
`DateTimeField`	Datum und Uhrzeit	Wie bei `DateField`.
`EmailField`	E-Mail-Adresse	`max_length`: Maximale Länge (optional).
`SlugField`	URL-sicherer Text	`max_length`, `unique`: Nur ein Datensatz mit diesem Slug erlaubt.
`URLField`	URL	`max_length`: Maximale Länge (optional).
`ImageField`	Bilddatei	`upload_to`: Ordner für den Upload (Pflicht).
`FileField`	Datei	`upload_to`: Speicherort der Datei.
`ForeignKey`	Beziehung zu einem anderen Modell	`to`: Zielmodell (Pflicht). `on_delete`: Wie der verknüpfte Datensatz behandelt wird (siehe unten).
`OneToOneField`	Eins-zu-Eins-Beziehung	`to`: Zielmodell. `on_delete`: Wie bei `ForeignKey`.
`ManyToManyField`	Viele-zu-Viele-Beziehung	`to`: Zielmodell. `through`: Zwischenmodell für zusätzliche Felder (optional).

3. Konzept von Beziehungen: ForeignKey, ManyToMany, OneToOne

a) ForeignKey (Viele-zu-Eins)

Ein ForeignKey wird verwendet, um eine **Viele-zu-Eins-Beziehung** zwischen zwei Modellen zu definieren. Dies bedeutet, dass ein Datensatz in einem Modell mit einem einzigen Datensatz in einem anderen Modell verbunden ist, während das andere Modell mit vielen Datensätzen im ersten Modell verbunden sein kann.

Beispiel:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

- **Beziehung:** Ein Autor kann viele Bücher haben, aber jedes Buch hat nur einen Autor.
- **Verwendung in Abfragen:**

```
# Alle Bücher eines Autors abrufen
author = Author.objects.get(id=1)
books = author.book_set.all()
```

Bedeutung von `on_delete`:

Das Argument `on_delete` definiert, wie sich Django verhalten soll, wenn der verknüpfte Datensatz gelöscht wird:

- `CASCADE`: Löscht alle verbundenen Einträge.
- `SET_NULL`: Setzt den Wert des ForeignKey auf `NULL` (erfordert `null=True`).
- `SET_DEFAULT`: Setzt den Wert auf einen Standardwert (erfordert `default`).
- `PROTECT`: Verhindert das Löschen des verbundenen Datensatzes.
- `DO_NOTHING`: Ignoriert das Löschen, was Inkonsistenzen verursachen kann.

---

b) ManyToMany (Viele-zu-Viele)

Eine ManyToMany-Beziehung wird verwendet, wenn mehrere Datensätze eines Modells mit mehreren Datensätzen eines anderen Modells verbunden sein können.

Beispiel:

```
class Student(models.Model):
    name = models.CharField(max_length=100)

class Course(models.Model):
    title = models.CharField(max_length=100)
    students = models.ManyToManyField(Student)
```

- **Beziehung:** Ein Kurs kann von mehreren Studenten besucht werden, und ein Student kann an mehreren Kursen teilnehmen.
- **Verwendung in Abfragen:**

  ```
  # Studenten in einem Kurs abrufen
  course = Course.objects.get(id=1)
  students = course.students.all()

  # Kurse eines Studenten abrufen
  student = Student.objects.get(id=1)
  courses = student.course_set.all()
  ```

Optionales Argument: `through`

Django erlaubt die Verwendung eines **Zwischenmodells** für zusätzliche Felder in der ManyToMany-Beziehung:

```
class Membership(models.Model):
    student = models.ForeignKey(Student, on_delete=models.CASCADE)
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    enrollment_date = models.DateField()

class Course(models.Model):
```

```
    title = models.CharField(max_length=100)
    students = models.ManyToManyField(Student, through='Membership')
```

---

c) OneToOne (Eins-zu-Eins)

Eine OneToOne-Beziehung wird verwendet, wenn ein Datensatz in einem Modell genau mit einem Datensatz in einem anderen Modell verbunden ist.

Beispiel:

```
class User(models.Model):
    username = models.CharField(max_length=100)

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField()
```

- **Beziehung:** Ein Benutzer hat ein Profil, und ein Profil gehört zu genau einem Benutzer.
- **Verwendung in Abfragen:**

  ```
  # Zugriff auf das Profil eines Benutzers
  user = User.objects.get(id=1)
  profile = user.profile

  # Zugriff auf den Benutzer eines Profils
  profile = Profile.objects.get(id=1)
  user = profile.user
  ```

Praktische Verwendung:

OneToOne-Beziehungen sind oft nützlich, um zusätzliche Daten zu einem bestehenden Modell hinzuzufügen, ohne die ursprüngliche Tabelle zu überladen (z. B. `User` und `Profile`).

---

4. Generierung von Datenbankeinträgen: `objects.create` vs. Objektinstanzierung

Methode	Beschreibung	Vorteile	Nachteile
`objects.create()`	Kombiniert die Instanziierung und das Speichern in einem Schritt.	Kürzer, wenn keine komplexen Operationen.	Keine Kontrolle über Zwischenschritte.
Objektinstanzierung	Erstellt ein Objekt, das später mit `.save()` gespeichert wird.	Mehr Kontrolle (z. B. Validierungen).	Längerer Code.

---

5. Übersicht über Datenbankbefehle

Befehl	Beschreibung	Beispiel
`objects.create()`	Erstellt und speichert einen neuen Eintrag.	`BlogPost.objects.create(title="Beispiel", content="Text")`
`.save()`	Speichert ein Objekt.	`post.save()`
`objects.all()`	Gibt alle Einträge zurück.	`BlogPost.objects.all()`
`objects.get()`	Gibt einen Eintrag zurück.	`BlogPost.objects.get(id=1)`
`objects.filter()`	Filtert nach Kriterien.	`BlogPost.objects.filter(title__icontains="Beispiel")`
`.update()`	Aktualisiert mehrere Einträge.	`BlogPost.objects.filter(id=1).update(title="Neu")`
`.delete()`	Löscht Einträge.	`post.delete()`