07 - Django: Model-Beziehungen (`ForeignKey`, `ManyToManyField`, `OneToOneField`)

Einleitung

- **Themen**: Dieses Skript vertieft das Verständnis von Django Models, indem es die verschiedenen Arten von Beziehungen zwischen Modellen behandelt: `ForeignKey` (Viele-zu-Eins), `ManyToManyField` (Viele-zu-Viele) und `OneToOneField` (Eins-zu-Eins).
- **Fokus**: Die korrekte Modellierung komplexer Datenstrukturen, bei denen Daten in verschiedenen Tabellen miteinander in Beziehung stehen. Wichtige Aspekte wie das `on_delete`-Verhalten bei `ForeignKey` und das `through`-Argument bei `ManyToManyField` werden beleuchtet.
- **Lernziele**:
  - Die drei Haupttypen von Modell-Beziehungen in Django verstehen und anwenden können.
  - Die Bedeutung und Auswirkungen der `on_delete`-Optionen für `ForeignKey` und `OneToOneField` kennen.
  - Wissen, wann und wie ein `ManyToManyField` mit einem expliziten `through`-Modell verwendet wird.
  - Daten über diese Beziehungen hinweg abfragen und erstellen können.

1. `ForeignKey`: Die Viele-zu-Eins-Beziehung

Ein `ForeignKey` wird verwendet, um eine Viele-zu-Eins-Beziehung zu definieren. Das bedeutet, ein Datensatz in einem Modell kann mit *einem einzigen* Datensatz in einem anderen Modell verbunden sein, während das andere Modell mit *vielen* Datensätzen im ersten Modell verbunden sein kann. Klassische Beispiele sind ein Blog-Beitrag, der genau einen Autor hat (aber ein Autor kann viele Beiträge haben) oder eine Antwort (`Choice`) die zu genau einer Frage (`Question`) gehört.

- **Definition**:

```python
from django.db import models

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

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    # Jedes Buch hat einen Autor.
    # Wenn der Autor gelöscht wird, werden alle seine Bücher auch gelöscht
(models.CASCADE).
    author = models.ForeignKey(Author, on_delete=models.CASCADE)

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

  In diesem Beispiel kann ein `Author` viele `Book`-Instanzen haben, aber jedes `Book` gehört zu genau einem `Author`.

- **`on_delete`-Optionen**: Dieses Argument ist bei `ForeignKey` (und `OneToOneField`) zwingend erforderlich und definiert, was passieren soll, wenn das Objekt, auf das der `ForeignKey` zeigt, gelöscht wird.
  - `models.CASCADE`: Löscht auch das Objekt, das den `ForeignKey` enthält. Wenn ein Autor gelöscht wird, werden alle seine Bücher mitgelöscht.
  - `models.PROTECT`: Verhindert das Löschen des referenzierten Objekts, solange noch Objekte darauf verweisen. (Löschen des Autors würde einen Fehler auslösen, wenn er noch Bücher hat).
  - `models.SET_NULL`: Setzt den `ForeignKey` auf `NULL`. Dies erfordert, dass das Feld in der Datenbank `null=True` erlaubt. (Wenn der Autor gelöscht wird, wird das `author`-Feld der Bücher zu `NULL`).
  - `models.SET_DEFAULT`: Setzt den `ForeignKey` auf seinen Standardwert. Erfordert, dass ein `default` für das Feld definiert ist.
  - `models.SET()`: Setzt den `ForeignKey` auf den Wert, der dieser Funktion übergeben wird, oder das Ergebnis eines Aufrufs.
  - `models.DO_NOTHING`: Tut nichts. Dies kann zu Inkonsistenzen in der Datenbank führen, wenn nicht anderweitig sichergestellt wird, dass keine verwaisten Einträge entstehen.

- **Zugriff auf verknüpfte Objekte**:
    - Von einem `Book`-Objekt zum `Author`: `my_book.author`
    - Von einem `Author`-Objekt zu all seinen `Book`-Objekten: `my_author.book_set.all()`. Der Name `_set` wird automatisch von Django generiert.

2. `ManyToManyField`: Die Viele-zu-Viele-Beziehung

Eine `ManyToManyField`-Beziehung wird verwendet, wenn mehrere Datensätze eines Modells mit mehreren Datensätzen eines anderen Modells verbunden sein können. Django erstellt hierfür automatisch eine separate Zwischentabelle in der Datenbank. Ein Student kann viele Kurse besuchen, und ein Kurs kann viele Studenten haben.

- **Definition**:

```python
from django.db import models

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

    def __str__(self):
        return self.name

class Course(models.Model):
    title = models.CharField(max_length=100)
    # Ein Kurs kann viele Studenten haben, und ein Student viele Kurse.
    # Django erstellt eine Zwischentabelle.
    students = models.ManyToManyField(Student)

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

- **Zugriff auf verknüpfte Objekte**:
    - Von einem `Course`-Objekt zu seinen `Student`-Objekten: `my_course.students.all()`
    - Von einem `Student`-Objekt zu seinen `Course`-Objekten: `my_student.course_set.all()`

- **Das `through`-Argument für Zwischentabellen mit zusätzlichen Daten**: Manchmal möchte man zusätzliche Informationen über die Beziehung selbst speichern (z.B. das Anmeldedatum eines Studenten zu einem Kurs). Hierfür kann man ein explizites Zwischenmodell definieren und es mit dem `through`-Argument im `ManyToManyField` angeben.

```python
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    def __str__(self): return self.name

class Course(models.Model):
    title = models.CharField(max_length=100)
    # `through` verweist auf das explizite Zwischenmodell `Membership`
    students = models.ManyToManyField(Student, through='Membership')
    def __str__(self): return self.title

class Membership(models.Model):
    student = models.ForeignKey(Student, on_delete=models.CASCADE)
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    enrollment_date = models.DateField()
    grade = models.CharField(max_length=2, blank=True, null=True)

    def __str__(self):
        return f"{self.student.name} in {self.course.title}"
```

In diesem Fall würde man Objekte des `Membership`-Modells erstellen, um Studenten zu Kursen hinzuzufügen und dabei das `enrollment_date` festzuhalten.

3. `OneToOneField`: Die Eins-zu-Eins-Beziehung

Eine `OneToOneField`-Beziehung wird verwendet, wenn ein Datensatz in einem Modell genau mit einem Datensatz in einem anderen Modell verbunden ist (oder umgekehrt). Dies ist konzeptuell ähnlich zu einem `ForeignKey` mit dem Zusatz `unique=True`, hat aber eine etwas andere semantische Bedeutung und Zugriffsmuster. Es wird oft verwendet, um ein Modell zu erweitern, z.B. ein Benutzerprofil für Djangos eingebautes `User`-Modell.

- **Definition**:

```
from django.db import models
from django.contrib.auth.models import User # Djangos User-Modell

class UserProfile(models.Model):
    # Jedes UserProfile ist mit genau einem User-Objekt verbunden.
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(blank=True)
    profile_picture_url = models.URLField(blank=True)

    def __str__(self):
        return self.user.username
```

  Hier hat jeder `User` genau ein `UserProfile` (und umgekehrt).

- **`on_delete`**: Funktioniert wie bei `ForeignKey`.
- **Zugriff auf verknüpfte Objekte**:
  - Von einem `User`-Objekt zum `UserProfile`: `my_user.userprofile` (Beachte: kein `_set`)
  - Von einem `UserProfile`-Objekt zum `User`: `my_profile.user`

Fazit

- **`ForeignKey` (N:1)**: Die häufigste Beziehung, um Hierarchien oder Zugehörigkeiten abzubilden. `on_delete` ist entscheidend für die Datenintegrität.
- **`ManyToManyField` (N:M)**: Für komplexe Verbindungen, bei denen Objekte auf beiden Seiten mehrere Verknüpfungen haben können. Django verwaltet die Zwischentabelle meist automatisch, aber `through` erlaubt benutzerdefinierte Zwischenmodelle für zusätzliche Beziehungsdaten.
- **`OneToOneField` (1:1)**: Nützlich zur Erweiterung bestehender Modelle oder wenn zwei Entitäten immer paarweise auftreten.
- Das Verständnis dieser Beziehungen ist fundamental, um Daten in einer relationalen Datenbank korrekt zu strukturieren und effizient über das Django ORM darauf zuzugreifen.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) sind die Beziehungen bereits korrekt definiert (`Choice` hat einen `ForeignKey` zu `Question`).

Demonstration der Abfrage über Beziehungen:

1. **Django Shell starten**:

```
python manage.py shell
```

2. **Modelle importieren und Objekte abrufen**:

```
from polls.models import Question, Choice
```

```python
# Eine spezifische Frage holen
q = Question.objects.get(id=1) # Annahme: Frage mit ID 1 existiert

# Alle Choices für diese Frage abrufen
# Django erstellt automatisch das Attribut 'choice_set' auf der Question-Instanz
choices_for_q = q.choice_set.all()
print(f"Antwortmöglichkeiten für Frage '{q.question_text}':")
for choice in choices_for_q:
    print(f"- {choice.choice_text} (Stimmen: {choice.votes})")

# Eine neue Choice für diese Frage erstellen
new_choice = q.choice_set.create(choice_text="Neue Antwortmöglichkeit", votes=0)
print(f"Neue Antwort '{new_choice.choice_text}' hinzugefügt.")

# Die Frage zu einer bestimmten Choice finden
c = Choice.objects.get(id=1) # Annahme: Choice mit ID 1 existiert
print(f"Die Antwort '{c.choice_text}' gehört zur Frage: '{c.question.question_text}'")
```

Cheat Sheet

- **`ForeignKey(ZielModell, on_delete=models.AKTION)`**:
  - `AKTION` kann sein: `CASCADE`, `PROTECT`, `SET_NULL` (benötigt `null=True`), `SET_DEFAULT` (benötigt `default=...`), `DO_NOTHING`.
  - Zugriff vom "viele"-Objekt zum "eins"-Objekt: `obj.foreignkey_feldname`
  - Zugriff vom "eins"-Objekt zum "viele"-Set: `obj.modellname_set.all()` (z.B. `author.book_set.all()`)
- **`ManyToManyField(ZielModell, through='ZwischenModellOptional')`**:
  - Zugriff vom Objekt zum Set der verknüpften Objekte: `obj.manytomany_feldname.all()`
  - Hinzufügen: `obj.manytomany_feldname.add(anderes_obj)`
  - Entfernen: `obj.manytomany_feldname.remove(anderes_obj)`
  - Erstellen und hinzufügen: `obj.manytomany_feldname.create(...)`
- **`OneToOneField(ZielModell, on_delete=models.AKTION)`**:
  - Zugriff vom Objekt zum verknüpften Objekt: `obj.onetoone_feldname` oder `obj.zielmodellname` (wenn `related_name` nicht gesetzt ist und die Beziehung vom anderen Modell aus definiert wurde).

Übungsaufgaben

1. **`Author`/`Book`-Beziehung (ForeignKey)**:
   - Erstelle die Modelle `Author` (mit Feld `name`) und `Book` (mit Feldern `title`, `publication_year`).
   - Füge dem `Book`-Modell einen `ForeignKey` zum `Author`-Modell hinzu. Wähle eine sinnvolle `on_delete`-Option.
   - Erstelle in der Django Shell einige Autoren und Bücher und verknüpfe sie.
   - Gib alle Bücher eines bestimmten Autors aus.
   - Gib den Autor eines bestimmten Buches aus.
2. **`Student`/`Course`-Beziehung (ManyToManyField)**:
   - Erstelle die Modelle `Student` (mit Feld `name`) und `Course` (mit Feld `title`).
   - Verbinde sie mit einem `ManyToManyField`.
   - Erstelle in der Shell Studenten und Kurse.
   - Füge Studenten zu Kursen hinzu und umgekehrt.
   - Gib alle Studenten eines Kurses und alle Kurse eines Studenten aus.
3. **`User`/`Profile`-Beziehung (OneToOneField)**:
   - Importiere Djangos `User`-Modell (`from django.contrib.auth.models import User`).
   - Erstelle ein `UserProfile`-Modell mit Feldern wie `bio` (TextField) und `birth_date` (DateField, optional).
   - Verbinde `UserProfile` mit einem `OneToOneField` zum `User`-Modell (`on_delete=models.CASCADE`).

- Erstelle in der Shell einen User (falls noch keiner existiert oder über den Admin) und ein zugehöriges Profil.
- Greife vom User-Objekt auf das Profil zu und umgekehrt.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die bisherigen Modelle (`Recipe`, `Ingredient`, `Step`) müssen nun miteinander in Beziehung gesetzt werden.

Aufgabe:

1. **Beziehungen in `recipes/models.py` definieren**:
   - **Recipe und User (ForeignKey)**: Ein Rezept wird von einem Benutzer erstellt.
     - Importiere Djangos `User`-Modell: `from django.contrib.auth.models import User`
     - Füge dem `Recipe`-Modell ein Feld `author` hinzu, das ein `ForeignKey` zum `User`-Modell ist.
     - Wähle eine sinnvolle `on_delete`-Option (z.B. `models.CASCADE`, wenn Rezepte gelöscht werden sollen, wenn der Autor gelöscht wird, oder `models.SET_NULL` und `null=True`, wenn Rezepte erhalten bleiben sollen).
   - **Recipe und Step (ForeignKey)**: Ein Rezept besteht aus vielen Schritten, aber jeder Schritt gehört zu genau einem Rezept.
     - Füge dem `Step`-Modell ein Feld `recipe` hinzu, das ein `ForeignKey` zum `Recipe`-Modell ist (`on_delete=models.CASCADE` ist hier sinnvoll).
   - **Recipe und Ingredient (ManyToManyField mit `through`)**: Ein Rezept kann viele Zutaten haben, und eine Zutat kann in vielen Rezepten vorkommen. Wir benötigen zusätzliche Informationen (Menge, Einheit) für jede Zutat *in einem bestimmten Rezept*.
     - Erstelle ein neues Modell namens `RecipeIngredient`.
     - `RecipeIngredient` soll folgende Felder haben:
       - `recipe`: `ForeignKey` zum `Recipe`-Modell (`on_delete=models.CASCADE`).
       - `ingredient`: `ForeignKey` zum `Ingredient`-Modell (`on_delete=models.CASCADE`).
       - `amount`: `CharField` (z.B. "200", "1/2 Tasse", "nach Geschmack") mit `max_length=50`.
       - `unit`: `CharField` (z.B. "g", "ml", "Stk.") mit `max_length=50`, `blank=True`, `null=True` (optional).
     - Füge dem `Recipe`-Modell ein Feld `ingredients` hinzu, das ein `ManyToManyField` zum `Ingredient`-Modell ist. Nutze das `through='RecipeIngredient'` Argument.
     - Optional: Überlege, ob das `Ingredient`-Modell selbst (mit nur `name`) ausreicht, oder ob es sinnvoll wäre, gängige Einheiten etc. direkt dort zu speichern (für spätere Erweiterungen). Fürs Erste reicht die einfache Struktur.

2. **Migrationen durchführen**:
   - Nachdem alle Beziehungen definiert wurden:

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

   - Prüfe die generierte Migrationsdatei.
   - Anschließend:

     ```
     python manage.py migrate
     ```

3. **(Optional) Testen im Django Shell**:
   - Versuche, einige Instanzen deiner Modelle zu erstellen und sie über die neuen Beziehungen zu verknüpfen. Erstelle z.B. ein `Recipe`, dann einige `Step`-Objekte, die auf dieses Rezept verweisen. Erstelle `Ingredient`-Objekte und dann `RecipeIngredient`-Objekte, um sie mit Mengenangaben einem Rezept zuzuordnen.