11 - Django ModelForms: Effiziente Formularerstellung

Einleitung

- **Themen**: Dieses Skript stellt `ModelForms` vor, eine mächtige Abstraktion in Django, um Formulare direkt aus Modellen zu generieren.
- **Fokus**: Effiziente Erstellung von Formularen für Datenbankoperationen (Erstellen, Bearbeiten). Automatische Feldgenerierung, Übernahme von Validierungen aus dem Modell, die `save()`-Methode und Anpassungsmöglichkeiten von Feldern und Widgets.
- **Lernziele**:
  - Ein klares Modell als Basis für ein `ModelForm` definieren oder wiedererkennen.
  - Ein `ModelForm` erstellen und über die `class Meta` (`model`, `fields`, `widgets`, `labels` etc.) detailliert konfigurieren.
  - Verstehen, wie Formularfelder eines `ModelForm` über die `Meta`-Klasse oder durch direktes Überschreiben "ausgeschmückt" und angepasst werden.
  - Die `form.save()`-Methode nutzen, insbesondere `form.save(commit=False)`.
  - Bestehende Modellinstanzen mit einem `ModelForm` bearbeiten.

1. Was ist ein ModelForm?

Ein `ModelForm` ist eine Helferklasse, die eine Formular-Klasse direkt aus einem Django-Modell erstellt. Statt Formularfelder manuell zu definieren, generiert `ModelForm` die meisten Felder und deren Basisvalidierungen automatisch aus der Modelldefinition.

- **Vorteile**:
  - **DRY (Don't Repeat Yourself)**: Felddefinitionen und Validierungen aus dem Modell werden wiederverwendet.
  - **Zeitersparnis**: Deutlich weniger Code für Standard-CRUD-Formulare.
  - **Konsistenz**: Formularvalidierung ist eng an die Modellvalidierung gekoppelt.
  - **Einfaches Speichern**: Die `save()`-Methode des Formulars kann das Modellobjekt direkt erstellen oder aktualisieren.

2. Das zugrundeliegende Model (Basis)

Bevor wir ein `ModelForm` erstellen, benötigen wir ein Django-Modell. Die Definition dieses Modells in `models.py` bleibt unverändert.

Beispiel: Ein einfaches `Event`-Modell (`events/models.py`)

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

class Event(models.Model):
    title = models.CharField(max_length=200, verbose_name="Veranstaltungstitel")
    description = models.TextField(blank=True, help_text="Optionale Beschreibung der
Veranstaltung.")
    event_date = models.DateTimeField(default=timezone.now, verbose_name="Datum und Uhrzeit")
    location = models.CharField(max_length=150, blank=True)
    is_public = models.BooleanField(default=True, verbose_name="Öffentliche Veranstaltung?")

    def __str__(self):
        return self.title

    class Meta:
        ordering = ['-event_date']
```

Dieses Modell `Event` dient als Grundlage für die folgenden `ModelForm`-Beispiele.

3. Grundlegende Erstellung eines ModelForm

Ein `ModelForm` wird in `forms.py` definiert und erbt von `django.forms.ModelForm`.

- **Die `class Meta`**: Das Herzstück eines `ModelForm` ist die innere Klasse `Meta`. Hier wird Django mitgeteilt, welches Modell verwendet werden soll und welche Felder im Formular erscheinen sollen.

Beispiel für `polls/forms.py` (unter Verwendung des `Question`-Modells):

```
# events/forms.py
from django import forms
from .models import Event # Das zugehörige Modell importieren

class EventForm(forms.ModelForm):
    class Meta:
        model = Event  # Das Modell, auf dem das Formular basiert
        fields = '__all__' # Alle Felder des Event-Modells verwenden
        # Alternativ: fields = ['title', 'event_date', 'location'] # Nur spezifische Felder
```

- `model = Event`: Gibt an, dass dieses Formular für das `Event`-Modell ist.
- `fields = '__all__'`: Weist Django an, Formularfelder für alle Felder des `Event`-Modells zu erstellen.
- `fields = ['title', 'event_date', 'location']`: Man kann auch eine Liste von Feldnamen angeben, um nur bestimmte Felder im Formular aufzunehmen.

4. Formularfelder "Ausschmücken": Anpassung über `class Meta`

Die innere `class Meta` des `ModelForm` ist der primäre Ort, um die automatisch generierten Formularfelder anzupassen.

- **`model`**: (Pflicht) Das Modell, auf dem das Formular basiert.
- **`fields`**: Eine Liste von Feldnamen, die im Formular enthalten sein sollen. `fields = '__all__'` schließt alle Felder ein.
- **`fields` oder `exclude`**:
  - `fields = ['title', 'event_date', ... ]`: Liste der Felder, die im Formular erscheinen sollen.
  - `exclude = ['is_public', ... ]`: Liste der Felder, die *nicht* im Formular erscheinen sollen. (Entweder `fields` oder `exclude` verwenden).

```
class QuestionForm(forms.ModelForm):
    class Meta:
        model = Question
        exclude = ['votes'] # Angenommen, 'votes' ist ein Feld im Question-Modell
```

- **`widgets`**: Ein Dictionary, um die Standard-HTML-Widgets für bestimmte Modellfelder im Formular zu überschreiben.

```
# events/forms.py
class EventFormWithWidgets(forms.ModelForm):
    class Meta:
        model = Event
        fields = ['title', 'description', 'event_date', 'is_public']
        widgets = {
            'description': forms.Textarea(attrs={'rows': 4, 'placeholder': 'Was passiert
bei der Veranstaltung?'}),
            'event_date': forms.DateTimeInput(attrs={'type': 'datetime-local'}, format='%Y-
%m-%dT%H:%M'),
            'is_public': forms.CheckboxInput(attrs={'class': 'custom-checkbox'}),
        }
```

- **`labels`**: Ein Dictionary, um die angezeigten Beschriftungen der Felder zu ändern.

```
# events/forms.py
class EventFormWithLabels(forms.ModelForm):
    class Meta:
        model = Event
        fields = ['title', 'event_date']
        labels = {
```

```
            'title': 'Name der Veranstaltung',
            'event_date': 'Wann findet es statt?',
        }
```

- `help_texts`: Ein Dictionary, um Hilfetexte für Felder zu definieren/überschreiben.

```
# events/forms.py
class EventFormWithHelpTexts(forms.ModelForm):
    class Meta:
        model = Event
        fields = ['title', 'location']
        help_texts = {
            'location': 'Bitte geben Sie Stadt und Straße an.',
        }
```

- `error_messages`: Anpassung von Standard-Fehlermeldungen.

```
# events/forms.py
class EventFormWithErrors(forms.ModelForm):
    class Meta:
        model = Event
        fields = ['title']
        error_messages = {
            'title': {
                'max_length': "Dieser Titel ist zu lang, bitte kürzen.",
                'required': "Ein Titel muss angegeben werden.",
            },
        }
```

5. Felder im ModelForm direkt überschreiben oder erweitern

Wenn die `Meta`-Optionen nicht ausreichen, kann man Felder direkt in der `ModelForm`-Klasse definieren. Dies überschreibt das automatisch generierte Feld oder fügt ein neues hinzu.

- **Modellfeld-Verhalten im Formular ändern**: Man kann z.B. ein Feld im Formular als nicht-benötigt (`required=False`) markieren, auch wenn es im Modell ein Pflichtfeld ist (was dann vor dem Speichern in der View Logik erfordert).

```
# events/forms.py
class EventFormOverride(forms.ModelForm):
    # Überschreibt das 'location'-Feld aus dem Modell für das Formular
    location = forms.CharField(
        max_length=200,
        required=False,
        label="Ort (optional)",
        help_text="Wo findet die Veranstaltung statt?"
    )

    class Meta:
        model = Event
        fields = ['title', 'description', 'event_date', 'location', 'is_public']
```

- **Zusätzliche, nicht-modellgebundene Felder hinzufügen**: Nützlich für Bestätigungs-Checkboxen, zusätzliche Validierungen etc.

```
# events/forms.py
class EventFormWithExtraField(forms.ModelForm):
    confirm_terms = forms.BooleanField(
        label="Ich akzeptiere die Teilnahmebedingungen.",
```

```
            required=True
        )

        class Meta:
            model = Event
            fields = ['title', 'description', 'event_date']
            # confirm_terms ist nur im Formular, nicht im Event-Modell
```

Der Wert von `confirm_terms` ist dann in `form.cleaned_data` verfügbar, wird aber nicht automatisch mit `form.save()` verarbeitet.

6. Die `save()`-Methode

Ein großer Vorteil von `ModelForm` ist die eingebaute `save()`-Methode.

- **`form.save()`**: Wenn das Formular mit Daten gebunden und valide ist (`form.is_valid() == True`), erstellt oder aktualisiert `form.save()` das zugehörige Modellobjekt in der Datenbank und gibt die Instanz zurück.
    - Wenn das Formular beim Instanziieren **keine** `instance` übergeben bekommen hat, erstellt `save()` ein neues Objekt.
    - Wenn das Formular mit einer `instance` (einem bestehenden Modellobjekt) initialisiert wurde, aktualisiert `save()` dieses Objekt.
- **`form.save(commit=False)`**: Diese Variante ist extrem nützlich. Sie erstellt das Modellobjekt im Speicher, speichert es aber **noch nicht** in der Datenbank. Man erhält die Objektinstanz zurück und kann diese modifizieren, bevor sie endgültig gespeichert wird.

Beispiel: Erstellen eines neuen Datenbankeintrags (Event-Beispiel)

Angenommen, wir haben das `Event`-Modell und das `EventForm` (wie zuvor im Skript definiert). Eine View-Funktion zum Erstellen eines neuen Events könnte so aussehen:

```
# events/views.py
from django.shortcuts import render, redirect
from .forms import EventForm # Annahme: EventForm ist in forms.py definiert
# from .models import Event # Wird hier nicht direkt benötigt, da ModelForm das Modell kennt

def add_event(request):
    if request.method == 'POST':
        form = EventForm(request.POST) # Formular mit den gesendeten Daten binden
        if form.is_valid():
            # Das Formular ist gültig, die Daten sind in form.cleaned_data
            # form.save() erstellt eine neue Event-Instanz und speichert sie in der DB
            new_event = form.save()

            # Weiterleitung zu einer Erfolgsseite oder zur Detailansicht des neuen Events
            # Annahme: Es gibt eine URL namens 'event_list'
            return redirect('event_list_url_name')
    else:
        # GET-Request: Ein leeres Formular erstellen
        form = EventForm()

    return render(request, 'events/event_form_template.html', {'form': form, 'form_title': 'Neue
Veranstaltung erstellen'})

# Annahme: 'events/event_form_template.html' ist ein Template, das das Formular anzeigt
# z.B. <form method="post">{% csrf_token %}{{ form.as_p }}<button
type="submit">Speichern</button></form>
```

In dieser `add_event`-View:

1. Wenn die Anfrage ein `POST` ist (das Formular wurde abgeschickt), wird eine `EventForm`-Instanz mit den `request.POST`-Daten erstellt.
2. `form.is_valid()` prüft die Daten.

3. Wenn gültig, ruft `form.save()` auf. Da kein `instance`-Argument beim Erstellen des `EventForm` übergeben wurde, weiß `save()`, dass ein neues `Event`-Objekt erstellt werden soll. Dieses wird direkt in der Datenbank gespeichert.
4. Anschließend erfolgt eine Weiterleitung.
5. Wenn die Anfrage ein `GET` ist (Seite wird zum ersten Mal aufgerufen) oder das Formular im `POST`-Fall ungültig war, wird ein leeres bzw. fehlerbehaftetes Formularobjekt erstellt und das Template damit gerendert.

Anwendungsfälle für `commit=False`:

- Daten hinzufügen, die nicht Teil des Formulars sind (z.B. den aktuell angemeldeten Benutzer als Autor setzen).
- Werte basierend auf anderen Daten berechnen, bevor gespeichert wird.
- Viele-zu-Viele-Beziehungen speichern, was oft einen separaten Schritt nach dem Speichern des Hauptobjekts erfordert (mit `form.save_m2m()`).

Beispiel mit `commit=False` (unter Verwendung eines fiktiven `Question`-Modells und `QuestionForm` zur Illustration des Prinzips):

```
# Angenommen, Question hat ein Feld 'author', das nicht im Formular ist
# und in der View gesetzt werden soll (z.B. request.user).
# Und QuestionForm ist ein ModelForm für das Question-Modell.

if request.method == 'POST':
    form = QuestionForm(request.POST) # QuestionForm wäre hier ein ModelForm für Question
    if form.is_valid():
        question_instance = form.save(commit=False) # Objekt erstellen, aber nicht in DB
speichern
        question_instance.author = request.user     # Zusätzliche Daten setzen
        question_instance.save()                    # Jetzt endgültig in DB speichern
        # Optional: form.save_m2m() falls ManyToMany-Felder im Question-Modell involviert sind
        return redirect('success_url')
```

Dieses zweite Beispiel zeigt das Prinzip von `commit=False`. Im Kontext unseres `Event`-Beispiels könnten wir `commit=False` verwenden, wenn das `Event`-Modell ein Feld wie `created_by` hätte, das wir in der View mit `request.user` füllen wollten, bevor das Event endgültig gespeichert wird.

7. ModelForm für das Bearbeiten von Objekten (Update)

Um ein bestehendes Objekt zu bearbeiten, übergibt man die Modellinstanz beim Erstellen des `ModelForm` an das `instance`-Argument.

```
# views.py (Bearbeitungs-View)
from django.shortcuts import render, redirect, get_object_or_404
from .models import Event
from .forms import EventForm

def edit_event(request, event_id):
    event_instance = get_object_or_404(Event, pk=event_id)

    if request.method == 'POST':
        form = EventForm(request.POST, instance=event_instance) # Instanz übergeben
        if form.is_valid():
            form.save() # Aktualisiert die bestehende Instanz
            return redirect('polls:detail', pk=event_id) # Zur Detailseite weiterleiten
    else:
        form = EventForm(instance=event_instance) # Formular mit Daten der Instanz füllen

    return render(request, '...', {'form': form, 'event': event_instance})
```

Das Formular wird dann mit den Daten der `event_instance` vorausgefüllt. Ein Aufruf von `form.save()` aktualisiert diese Instanz.

Fazit

- `ModelForm` **als Brücke**: Vereinfacht die Erstellung von Formularen, die eng mit Datenbankmodellen verknüpft sind.
- **Automatismen**: Felder, Widgets und Basisvalidierungen werden oft automatisch vom Modell übernommen.
- `class Meta`: Das zentrale Konfigurationselement für `ModelForm` (`model`, `fields`, `widgets` etc.).
- `.save(commit=False)`: Ein mächtiges Werkzeug, um Objekte vor dem endgültigen Speichern anzupassen oder mit zusätzlichen Daten anzureichern.
- **Bearbeiten mit** `instance`: Durch Übergabe einer Modellinstanz wird ein `ModelForm` leicht zu einem Bearbeitungsformular.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) wird ein `ModelForm` erstellt, um neue Fragen hinzuzufügen.

1. `polls/forms.py` **erstellen oder anpassen:**

```
# polls/forms.py
from django import forms
from .models import Question

class QuestionModelForm(forms.ModelForm):
    class Meta:
        model = Question
        fields = ['question_text', 'pub_date'] # Alle Felder, die der Benutzer eingeben soll
        widgets = {
            'question_text': forms.Textarea(attrs={'rows': 3}),
            'pub_date': forms.DateTimeInput(attrs={'type': 'datetime-local'}, format='%Y-%m-
%dT%H:%M')
        }
        labels = {
            'question_text': 'Ihre Frage',
            'pub_date': 'Veröffentlichungsdatum und -zeit'
        }
```

2. View in `polls/views.py` **zum Erstellen einer Frage:**

```
# polls/views.py
from django.shortcuts import render, redirect
from .forms import QuestionModelForm
# ... (andere imports)

def add_question_modelform(request):
    if request.method == 'POST':
        form = QuestionModelForm(request.POST)
        if form.is_valid():
            # Beispiel für commit=False, falls man z.B. einen Autor setzen müsste
            # question = form.save(commit=False)
            # question.author = request.user # (Benötigt ein 'author'-Feld im Question-Modell)
            # question.save()
            form.save() # Speichert das neue Question-Objekt direkt
            return redirect('polls:list') # Zurück zur Fragenliste
    else:
        form = QuestionModelForm()
    return render(request, 'polls/add_question_form.html', {'form': form, 'form_title': 'Neue
Frage hinzufügen (ModelForm)'})
```

3. Template `polls/templates/polls/add_question_form.html` **erstellen:**

```
{# polls/templates/polls/add_question_form.html #}
{% extends "polls/base_polls.html" %}

{% block title %}{{ form_title }}{% endblock %}
```

```
{% block content %}
    <h2>{{ form_title }}</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Frage speichern</button>
    </form>
{% endblock %}
```

4. URL in `polls/urls.py` hinzufügen:

```
# polls/urls.py
# ...
path('add-mf/', views.add_question_modelform, name='add_question_modelform'),
```

Cheat Sheet

ModelForm-Definition (`forms.py`)

```
from django import forms
from .models import MyModel

class MyModelForm(forms.ModelForm):
    extra_field = forms.CharField(required=False) # Zusätzliches Feld

    class Meta:
        model = MyModel
        fields = ['model_field1', 'model_field2'] # oder '__all__'
        # exclude = ['model_field_to_hide']
        widgets = {
            'model_field1': forms.Textarea(attrs={'rows':3}),
        }
        labels = {
            'model_field2': 'Benutzerdefiniertes Label',
        }
```

ModelForm-Verarbeitung in der View (`views.py`)

- **Neues Objekt erstellen**:

  ```
  if request.method == 'POST':
      form = MyModelForm(request.POST, request.FILES) # request.FILES bei Datei-Uploads
      if form.is_valid():
          instance = form.save(commit=False)
          # instance.some_other_field = request.user (oder andere Logik)
          instance.save()
          # form.save_m2m() # falls ManyToMany-Felder
          return redirect('success_url')
  else:
      form = MyModelForm()
  ```

- **Bestehendes Objekt bearbeiten**:

  ```
  from django.shortcuts import get_object_or_404

  obj_instance = get_object_or_404(MyModel, pk=obj_id)
  if request.method == 'POST':
  ```

```
        form = MyModelForm(request.POST, request.FILES, instance=obj_instance)
        if form.is_valid():
            form.save()
            return redirect('success_url')
    else:
        form = MyModelForm(instance=obj_instance) # Formular mit Daten füllen
```

Übungsaufgaben

1. **`ProductModelForm` erstellen**:
    - Für das `Product`-Modell aus früheren Übungen ein `ProductModelForm` erstellen.
    - Über die `Meta`-Klasse nur die Felder `name`, `description` und `price` einschließen.
    - Das Widget für `description` auf `forms.Textarea` setzen.
    - Ein Label für `price` auf "Preis (EUR)" ändern.
2. **View zum Hinzufügen von Produkten**:
    - Eine View erstellen, die das `ProductModelForm` verwendet, um neue Produkte hinzuzufügen.
    - Bei erfolgreicher Speicherung zu einer (noch zu erstellenden) Erfolgsseite oder Produktliste weiterleiten.
    - Ein Template erstellen, um das Formular anzuzeigen.
3. **View zum Bearbeiten von Produkten**:
    - Eine View erstellen, die ein bestehendes Produkt über seine ID lädt und das `ProductModelForm` mit der `instance` des Produkts initialisiert.
    - Die View soll das Bearbeiten und Speichern des Produkts ermöglichen.
    - Das gleiche Template wie für das Hinzufügen kann wiederverwendet oder angepasst werden.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die "Community Recipe Sharing Platform" soll nun Formulare erhalten, um Rezepte hinzuzufügen und zu bearbeiten. Hierfür sind `ModelForms` ideal.

Aufgabe:

1. **`RecipeForm` in `recipes/forms.py` erstellen**:

    - Eine Klasse `RecipeForm` erstellen, die von `forms.ModelForm` erbt.
    - In der `class Meta` das `Recipe`-Modell zuweisen.
    - Über `fields` die Felder `title`, `description` auswählen (den `author` werden wir in der View setzen, `created_at` und `updated_at` werden automatisch verwaltet).
    - Passe die Widgets an: `description` soll eine `forms.Textarea` sein.
    - Definiere benutzerfreundliche `labels` für `title` und `description`.

2. **View zum Hinzufügen neuer Rezepte (`add_recipe_view`) in `recipes/views.py`**:

    - Diese View muss mit dem `@login_required`-Decorator geschützt werden (Import: `from django.contrib.auth.decorators import login_required`), da nur angemeldete Benutzer Rezepte erstellen sollen.
    - Wenn `request.method == 'POST'`:
        - `RecipeForm` mit `request.POST` instanziieren.
        - Wenn `form.is_valid()`:
            - Das Rezept mit `recipe = form.save(commit=False)` erstellen, aber noch nicht in der Datenbank speichern.
            - Den `author` des Rezepts auf `request.user` setzen: `recipe.author = request.user`.
            - Das Rezept endgültig speichern: `recipe.save()`.
            - Weiterleitung zu einer Erfolgsseite oder zur (später zu erstellenden) Detailseite des neuen Rezepts.
    - Wenn `GET`-Request:
        - Ein leeres `RecipeForm` instanziieren.
    - Das Template `recipes/recipe_form.html` mit dem Formular im Kontext rendern.

3. **View zum Bearbeiten bestehender Rezepte (`edit_recipe_view`) in `recipes/views.py`**:

- Diese View muss ebenfalls mit `@login_required` geschützt werden.
- Sie soll eine `recipe_id` aus der URL entgegennehmen.
- Das zu bearbeitende `Recipe`-Objekt mit `get_object_or_404(Recipe, pk=recipe_id)` holen.
- **Wichtig**: Sicherstellen, dass nur der Autor des Rezepts dieses bearbeiten darf (z.B. `if recipe_instance.author != request.user: return HttpResponseForbidden()`).
- Wenn `request.method == 'POST'`:
  - `RecipeForm` mit `request.POST` UND `instance=recipe_instance` instanziieren.
  - Wenn `form.is_valid()`, das Formular speichern (`form.save()`).
  - Weiterleitung zur Detailseite des bearbeiteten Rezepts.
- Wenn `GET`-Request:
  - `RecipeForm` mit `instance=recipe_instance` instanziieren, um es mit den Daten des Rezepts vorzufüllen.
- Das Template `recipes/recipe_form.html` mit dem Formular im Kontext rendern (kann das gleiche Template wie für das Hinzufügen sein).

4. **Template `recipes/templates/recipes/recipe_form.html` erstellen**:
   - Soll von `base_recipes.html` erben.
   - Einen passenden `title`-Block und eine Überschrift setzen (z.B. "Neues Rezept erstellen" oder "Rezept bearbeiten").
   - Das Formular mit `<form method="post">`, `{% csrf_token %}`, `{{ form.as_p }}` und einem Submit-Button rendern.

5. **URLs in `recipes/urls.py` definieren**:
   - Einen Pfad für `add_recipe_view` (z.B. `neu/`) mit dem Namen `add_recipe`.
   - Einen Pfad für `edit_recipe_view` (z.B. `<int:recipe_id>/bearbeiten/`) mit dem Namen `edit_recipe`.

6. **Testen**:
   - Versuchen, als nicht angemeldeter Benutzer auf die Formularseiten zuzugreifen (sollte zur Login-Seite führen).
   - Rezepte erstellen und bearbeiten.
   - Versuchen, ein Rezept eines anderen Benutzers zu bearbeiten (sollte fehlschlagen oder verboten sein).