10 - Django Forms: Einführung (Normal Forms)

Einleitung

- **Themen**: Dieses Skript führt in die Grundlagen von Django Forms ein. Es wird behandelt, wie Formulare definiert, in Templates gerendert und wie deren Daten in Views verarbeitet und validiert werden. Der Fokus liegt hier auf "normalen" Forms, die nicht direkt an ein Django-Modell gebunden sind.
- **Fokus**: Verarbeitung von Benutzereingaben, die nicht unbedingt direkt in einem Datenbankmodell gespeichert werden müssen (z.B. Kontaktformulare, Suchformulare). Erlernen des Zyklus: Formular-Definition -> Rendern im Template -> Datenübermittlung -> Validierung und Verarbeitung in der View.
- **Lernziele**:
  - Eine `forms.Form`-Klasse definieren und verschiedene Feldtypen (`CharField`, `EmailField`, `ChoiceField` etc.) mit ihren jeweiligen Optionen und Widgets verstehen.
  - Ein Formular in einem Django-Template mit `{{ form.as_p }}`, `{{ form.as_ul }}`, `{{ form.as_table }}` oder durch manuelles Rendern der Felder darstellen.
  - Den `{% csrf_token %}`-Tag verstehen und anwenden.
  - Daten aus einem gesendeten Formular in einer View entgegennehmen (`request.POST`).
  - Die `form.is_valid()`-Methode zur Validierung der Formulardaten nutzen.
  - Auf validierte Daten über `form.cleaned_data` zugreifen.

1. Grundlagen von Django Forms

Django bietet ein robustes System zur Handhabung von HTML-Formularen. Es kümmert sich um das Rendern der Formularfelder, die Validierung der Benutzereingaben und die Konvertierung der Daten in Python-Datentypen.

- **Zwei Arten von Formularen**:
  1. `forms.Form`: Für Formulare, die nicht direkt mit einem Django-Modell verbunden sind.
  2. `forms.ModelForm`: Generiert Formularfelder automatisch basierend auf einem Django-Modell (Skript 11).
- **Der Formular-Workflow**:
  1. **Definition**: Eine Python-Klasse, die von `forms.Form` erbt, definiert die Felder des Formulars.
  2. **Instanziierung**: In der View wird eine Instanz dieser Formular-Klasse erstellt.
  3. **Rendern**: Die Formular-Instanz wird an das Template übergeben und dort als HTML gerendert.
  4. **Übermittlung**: Der Benutzer füllt das Formular aus und sendet es (meist per `POST`-Request) an den Server.
  5. **Binding & Validierung**: In der View wird eine neue Formular-Instanz mit den übermittelten Daten (`request.POST`) erstellt ("gebunden"). Die Methode `form.is_valid()` prüft, ob alle Daten den Validierungsregeln der Felder entsprechen.
  6. **Verarbeitung**: Wenn das Formular gültig ist, sind die bereinigten Daten im Dictionary `form.cleaned_data` verfügbar und können weiterverarbeitet werden.
  7. **Feedback**: Wenn das Formular ungültig ist, wird es mit den Fehlermeldungen erneut im Template angezeigt.

2. Formular-Klassen definieren (`forms.Form`)

Formulare werden als Klassen definiert, die von `django.forms.Form` erben. Die Felder des Formulars werden als Klassenattribute dieser Klasse deklariert.

- **Beispiel einer einfachen Formular-Klasse in `myapp/forms.py`**:

```python
# myapp/forms.py
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100, label="Ihr Name", help_text="Maximal 100
Zeichen.")
    email = forms.EmailField(label="Ihre E-Mail-Adresse")
    message = forms.CharField(widget=forms.Textarea, label="Ihre Nachricht")
    send_copy = forms.BooleanField(required=False, label="Kopie an mich senden?")
```

3. Wichtige Formular-Felder (`forms.Field`)

Django bietet eine Vielzahl von Feldtypen, die denen der Modelle ähneln, aber spezifisch für Formulare sind.

Feldtyp	Beschreibung	Wichtige Argumente (`label`, `required`, `widget`, `help_text` sind fast immer verfügbar)
`CharField`	Für Texteingaben.	`max_length`, `min_length`
`EmailField`	Validiert, ob die Eingabe eine E-Mail-Adresse ist.	`max_length`, `min_length`
`URLField`	Validiert, ob die Eingabe eine URL ist.	`max_length`, `min_length`
`IntegerField`	Für ganze Zahlen.	`min_value`, `max_value`
`FloatField`	Für Fließkommazahlen.	`min_value`, `max_value`
`DecimalField`	Für Dezimalzahlen mit fester Genauigkeit.	`max_digits`, `decimal_places`, `min_value`, `max_value`
`BooleanField`	Für Wahr/Falsch-Werte (oft als Checkbox gerendert).	`required` (oft `False` gesetzt, da nicht angekreuzt auch ein valider Zustand ist).
`ChoiceField`	Für eine Auswahl aus vordefinierten Optionen.	`choices` (eine Liste von Tupeln, z.B. `[('wert1', 'Anzeige1'), ...]`)
`MultipleChoiceField`	Für eine Mehrfachauswahl.	`choices`
`DateField`	Für Datumseingaben.	`input_formats`
`DateTimeField`	Für Datums- und Zeiteingaben.	`input_formats`
`FileField`	Für Datei-Uploads.	`max_length`, `allow_empty_file`
`ImageField`	Für Bild-Uploads (benötigt `Pillow`).	Wie `FileField`.

Komplexeres Beispiel verschiedener Feldtypen und Widgets:

```
# myapp/forms.py
from django import forms

class PizzaOrderForm(forms.Form):
    name = forms.CharField(
        max_length=100,
        label="Ihr Name",
        widget=forms.TextInput(attrs={'placeholder': 'Max Mustermann'})
    )
    email = forms.EmailField(
        label="Ihre E-Mail",
        widget=forms.EmailInput(attrs={'placeholder': 'max@example.com'})
    )

    SIZE_CHOICES = [
        ('small', 'Klein (26cm)'),
        ('medium', 'Mittel (32cm)'),
        ('large', 'Groß (40cm)'),
    ]
    size = forms.ChoiceField(
        choices=SIZE_CHOICES,
        widget=forms.RadioSelect, # Darstellung als Radio-Buttons
        label="Größe"
    )

    TOPPING_CHOICES = [
        ('pepperoni', 'Pepperoni'),
```

```
        ('mushrooms', 'Pilze'),
        ('onions', 'Zwiebeln'),
        ('sausage', 'Wurst'),
        ('bacon', 'Speck'),
        ('extra_cheese', 'Extra Käse'),
    ]
    toppings = forms.MultipleChoiceField(
        choices=TOPPING_CHOICES,
        widget=forms.CheckboxSelectMultiple, # Darstellung als Checkboxen
        label="Beläge",
        required=False # Mehrfachauswahl ist oft optional
    )

    special_requests = forms.CharField(
        widget=forms.Textarea(attrs={'rows': 3, 'placeholder': 'Sonderwünsche?'}),
        required=False,
        label="Sonderwünsche"
    )
```

4. Widgets: Die HTML-Darstellung der Felder

Jedes Formularfeld hat ein Standard-Widget, das seine HTML-Darstellung bestimmt (z.B. `CharField` -> `<input type="text">`). Man kann dieses Widget jedoch mit dem `widget`-Argument anpassen.

- **Beispiele für Widgets**:
  - `forms.TextInput(attrs={'placeholder': 'Ihr Name hier'})`: Standard für `CharField`. `attrs` erlaubt das Setzen von HTML-Attributen.
  - `forms.EmailInput()`: Standard für `EmailField`.
  - `forms.Textarea(attrs={'rows': 5, 'cols': 40})`: Für mehrzeilige Texteingaben (oft mit `CharField` verwendet, nicht nur `TextField` im Sinne von Modellen).
  - `forms.PasswordInput(render_value=False)`: Für Passworteingaben (versteckt die Zeichen). `render_value=False` verhindert, dass das Passwort bei erneuter Anzeige des Formulars (z.B. nach Validierungsfehler) wieder im Feld steht.
  - `forms.CheckboxInput()`: Standard für `BooleanField`.
  - `forms.Select(choices=...)`: Dropdown-Menü (oft für `ChoiceField`).
  - `forms.RadioSelect(choices=...)`: Radio-Buttons.
  - `forms.CheckboxSelectMultiple(choices=...)`: Checkboxen für Mehrfachauswahl.
  - `forms.DateInput(attrs={'type': 'date'})`: Nutzt das native HTML5-Datumsauswahlfeld.
  - `forms.ClearableFileInput()`: Für Datei-Uploads mit Möglichkeit zum Löschen.

Beispiel für Widget-Anpassung:

```
# myapp/forms.py
class FeedbackForm(forms.Form):
    subject = forms.CharField(
        max_length=100,
        widget=forms.TextInput(attrs={'class': 'form-control', 'placeholder': 'Betreff'})
    )
    message = forms.CharField(
        widget=forms.Textarea(attrs={'class': 'form-control', 'rows': 5, 'placeholder': 'Ihre
Nachricht...'}),
        help_text="Bitte beschreiben Sie Ihr Anliegen."
    )
    rating = forms.ChoiceField(
        choices=[(i, str(i)) for i in range(1, 6)], # ( (1,'1'), (2,'2'), ... )
        widget=forms.RadioSelect,
        label="Bewertung (1-5)"
    )
```

5. Formulare in Templates rendern

Eine Formular-Instanz kann auf verschiedene Weisen im Template gerendert werden:

- `{{ form.as_p }}`: Rendert jedes Feld in einem `<p>`-Tag, mit Label und Feld.
- `{{ form.as_ul }}`: Rendert jedes Feld in einem `<li>`-Tag. Die `<ul>`-Tags müssen manuell darum gesetzt werden.
- `{{ form.as_table }}`: Rendert jedes Feld in einer Tabellenzeile (`<tr>`). Die `<table>`-Tags müssen manuell darum gesetzt werden.
- **Manuelles Rendern**: Bietet volle Kontrolle über das HTML-Layout.
  - `{{ form.feldname.label_tag }}`: Das `<label>`.
  - `{{ form.feldname }}`: Das Widget des Feldes.
  - `{{ form.feldname.errors }}`: Eine Liste von Validierungsfehlern für dieses Feld (als `<ul>`).
  - `{{ form.non_field_errors }}`: Fehler, die nicht einem spezifischen Feld zugeordnet sind.
  - `{{ form.feldname.help_text }}`: Der Hilfetext des Feldes.

Wichtig: `{% csrf_token %}` Jedes Formular, das mit der `POST`-Methode gesendet wird, muss den `{% csrf_token %}`-Tag innerhalb des `<form>`-Elements enthalten. Dies schützt vor Cross-Site Request Forgery Angriffen.

Beispiel-Template (`myapp/templates/myapp/contact_form.html`):

```
{# myapp/templates/myapp/contact_form.html #}
<h1>Kontaktieren Sie uns</h1>
<form method="post">
    {% csrf_token %}

    {{ form.as_p }}

    <button type="submit">Senden</button>
</form>
```

6. Formularverarbeitung in Views

Die View, die das Formular anzeigt (meist bei einem `GET`-Request), muss auch die gesendeten Daten (bei einem `POST`-Request) verarbeiten.

```
# myapp/views.py
from django.shortcuts import render, redirect
from django.http import HttpResponse
from .forms import ContactForm # Die oben definierte Formular-Klasse

def contact_view(request):
    if request.method == 'POST':
        # Formular mit den gesendeten Daten binden
        form = ContactForm(request.POST)
        if form.is_valid():
            # Daten sind gültig, Zugriff über cleaned_data
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']

            # Hier würde die Logik zur Verarbeitung stehen (z.B. E-Mail senden)
            # print(f"Name: {name}, Email: {email}, Nachricht: {message}")

            # Optional: Weiterleitung nach erfolgreicher Verarbeitung
            # return redirect('myapp:success_page_name')
            return HttpResponse(f"Danke, {name}! Ihre Nachricht wurde empfangen.")
        # Wenn das Formular nicht gültig ist, wird es mit Fehlern unten erneut gerendert
    else:
        # GET-Request: Leeres Formular erstellen und anzeigen
        form = ContactForm()

    return render(request, 'myapp/contact_form.html', {'form': form})
```

- `request.method == 'POST'`: Prüft, ob das Formular gesendet wurde.
- `form = ContactForm(request.POST)`: Erstellt eine Formularinstanz und bindet die gesendeten POST-Daten daran.
- `form.is_valid()`: Führt alle Validierungsregeln für jedes Feld aus. Gibt `True` zurück, wenn alle Daten gültig sind. Wenn `False`, enthält das `form`-Objekt die Fehlerinformationen, die im Template angezeigt werden können.
- `form.cleaned_data`: Ein Dictionary, das die validierten und in Python-Datentypen konvertierten Daten enthält. Zugriff erfolgt über die Feldnamen als Schlüssel.

Fazit

- **`forms.Form`**: Die Basis für Formulare, die nicht direkt an Modelle gebunden sind. Erfordert manuelle Definition der Felder.
- **Felder & Widgets**: Felder definieren Datentypen und Validierungsregeln; Widgets steuern die HTML-Darstellung. Viele Anpassungsmöglichkeiten.
- **CSRF-Schutz**: `{% csrf_token %}` ist unerlässlich für POST-Formulare.
- **Validierungszyklus**: `request.POST` -> `form.is_valid()` -> `form.cleaned_data`. Dieser Zyklus ist zentral für die sichere Verarbeitung von Benutzereingaben.
- **Trennung**: Django Forms helfen, die Logik der Formularverarbeitung sauber von der Präsentation im Template zu trennen.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) wird ein einfaches Feedback-Formular erstellt, das nicht an ein Modell gebunden ist.

1. `polls/forms.py` erstellen:

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

class FeedbackPollsForm(forms.Form):
    RATING_CHOICES = [
        (1, '1 - Sehr schlecht'),
        (2, '2 - Schlecht'),
        (3, '3 - Mittelmäßig'),
        (4, '4 - Gut'),
        (5, '5 - Sehr gut'),
    ]
    email = forms.EmailField(label="Ihre E-Mail (optional)", required=False)
    rating = forms.ChoiceField(choices=RATING_CHOICES, widget=forms.RadioSelect, label="Wie
bewerten Sie unsere Polls-App?")
    comments = forms.CharField(widget=forms.Textarea, label="Ihre Kommentare", required=False)
```

2. View in `polls/views.py` erstellen:

```
# polls/views.py
# ... (andere imports und views)
from .forms import FeedbackPollsForm
from django.http import HttpResponse # Für die einfache Antwort

def feedback_view(request):
    if request.method == 'POST':
        form = FeedbackPollsForm(request.POST)
        if form.is_valid():
            # Normalerweise hier E-Mail senden oder Daten anderweitig verarbeiten
            print("Feedback erhalten:", form.cleaned_data)
            return HttpResponse("Vielen Dank für Ihr Feedback!")
    else:
        form = FeedbackPollsForm()
    return render(request, 'polls/feedback_form.html', {'form': form})
```

3. Template `polls/templates/polls/feedback_form.html` erstellen:

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

{% block title %}Feedback - Polls App{% endblock %}

{% block content %}
    <h2>Geben Sie uns Feedback</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Feedback senden</button>
    </form>
{% endblock %}
```

4. URL in `polls/urls.py` hinzufügen:

```
# polls/urls.py
# ...
path('feedback/', views.feedback_view, name='feedback'),
```

Cheat Sheet

Formular-Definition (`forms.py`)

```
from django import forms

class MyForm(forms.Form):
    name = forms.CharField(max_length=50, label="Benutzername")
    age = forms.IntegerField(min_value=18, help_text="Mindestalter 18")
    comment = forms.CharField(widget=forms.Textarea)
```

Formular-Rendering im Template

- `{% csrf_token %}` (innerhalb `<form method="post">`)
- `{{ form.as_p }}` (jedes Feld als `<p>`)
- `{{ form.as_ul }}` (jedes Feld als `<li>`)
- `{{ form.as_table }}` (jedes Feld als `<tr>`)
- Manuell: `{{ form.feldname.label_tag }}`, `{{ form.feldname }}`, `{{ form.feldname.errors }}`, `{{ form.feldname.help_text }}`

Formularverarbeitung in der View (`views.py`)

```
from .forms import MyForm
from django.shortcuts import render

def my_view(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            data = form.cleaned_data # Zugriff auf validierte Daten
            # ... Daten verarbeiten ...
            # return redirect_or_success_response
    else:
        form = MyForm() # Leeres Formular für GET
    return render(request, 'template.html', {'form': form})
```

Übungsaufgaben

1. **Einfaches Suchformular erstellen**:
    - Erstelle eine `SearchForm` in einer `forms.py` deiner App. Diese soll nur ein Feld `query` (`CharField`) haben.
    - Erstelle eine View, die dieses Formular bei einem `GET`-Request anzeigt.
    - Wenn das Formular mit `GET` abgeschickt wird (HTML-Formulare können auch `GET` als `method` haben), soll die View den `query`-Wert aus `request.GET` auslesen (nicht `form.cleaned_data` in diesem speziellen Fall, da `GET`-Formulare nicht auf die gleiche Weise "gebunden" werden) und anzeigen.
    - Erstelle ein Template, das dieses Suchformular darstellt.
2. **Pizza-Bestellformular (Validierung üben)**:
    - Erstelle ein `PizzaOrderForm` (`03_Django_Forms.pdf` kann als Inspiration dienen).
    - Felder: `name` (CharField, required), `email` (EmailField, required), `size` (ChoiceField mit `small`, `medium`, `large`), `toppings` (MultipleChoiceField mit einigen Optionen).
    - Erstelle eine View und ein Template, um dieses Formular anzuzeigen und zu verarbeiten (wie im `contact_view`-Beispiel).
    - Bei erfolgreicher Validierung soll eine Bestätigungsnachricht mit den Bestelldaten ausgegeben werden.
    - Teste die Validierung, indem ungültige Daten eingegeben werden (z.B. keine E-Mail, kein Name).

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die "Community Recipe Sharing Platform" soll ein allgemeines Suchformular für Rezepte erhalten. Da wir noch keine Datenbankabfragen für die Suche implementieren, wird das Formular zunächst nur die Suchanfrage entgegennehmen und anzeigen.

Aufgabe:

1. **`forms.py` in der `recipes`-App erstellen**:
    - Falls noch nicht vorhanden, eine `recipes/forms.py` anlegen.
    - Darin eine Klasse `RecipeSearchForm` definieren, die von `forms.Form` erbt.
2. **Felder im `RecipeSearchForm` definieren**:
    - `search_term`: Ein `CharField`, das den Suchbegriff aufnimmt. Setze ein `label="Suche nach Rezepten"` und mache es nicht zwingend erforderlich (`required=False`). Gib ihm ein `TextInput`-Widget mit einem `placeholder`.
    - `cuisine_type`: Ein `CharField` (optional), um nach Küchenart zu filtern (z.B. "Italienisch", "Asiatisch"). Auch `required=False` und ein passender `placeholder`.
3. **View `recipe_search_view` in `recipes/views.py` erstellen**:
    - Diese View soll sowohl `GET`- als auch `POST`-Requests für das Formular handhaben können (obwohl eine reine `GET`-Suche hier üblicher wäre, üben wir den `POST`-Zyklus).
    - Wenn `request.method == 'POST'`:
        - Das `RecipeSearchForm` mit `request.POST` instanziieren.
        - Prüfen, ob `form.is_valid()`.
        - Wenn ja, die Werte aus `form.cleaned_data` auslesen und z.B. als `HttpResponse` ausgeben: `"Suche nach '{search_term_value}' in Küche '{cuisine_type_value}'."`.
    - Wenn `GET` (oder das Formular ungültig ist):
        - Ein leeres (oder fehlerbehaftetes) `RecipeSearchForm` instanziieren.
        - Das Template `recipes/recipe_search.html` mit dem Formular im Kontext rendern.
4. **Template `recipes/templates/recipes/recipe_search.html` erstellen**:
    - Dieses Template soll von `base_recipes.html` erben.
    - Den `title`-Block passend setzen.
    - Im `content`-Block ein `<form method="post">`-Tag erstellen.
    - Den `{% csrf_token %}` einfügen.
    - Das Formular mit `{{ form.as_p }}` rendern.
    - Einen Submit-Button hinzufügen.
5. **URL für die Suche in `recipes/urls.py` definieren**:
    - Einen Pfad (z.B. `suche/`) für die `recipe_search_view` anlegen und ihm den Namen `search` geben.
6. **Testen**:
    - Die Suchseite aufrufen.
    - Das Formular absenden (mit und ohne Eingaben) und die Reaktion der View überprüfen.
    - Prüfen, ob Validierungsfehler (falls welche definiert wären und ausgelöst würden) korrekt angezeigt werden, wenn man z.B. Felder manuell rendert und `{{ form.feldname.errors }}` einbaut.