08 - Django Templates: Variablen, Tags und Filter

Einleitung

- **Themen**: Einführung in die Django Template Language (DTL) als Mittel zur Erzeugung dynamischer HTML-Seiten. Behandlung der Kernkomponenten: Variablen, Tags und Filter.
- **Fokus**: Trennung von Logik (Python/Views) und Präsentation (HTML/Templates). Wie Daten aus Views an Templates übergeben und dort dynamisch dargestellt und formatiert werden.
- **Lernziele**:
  - Die grundlegende Syntax und Funktionsweise der Django Template Language verstehen.
  - Variablen aus dem Kontext in Templates einfügen und auf Attribute von Objekten zugreifen.
  - Steuerstrukturen mittels Template-Tags (`if`, `for`) anwenden.
  - Die Nützlichkeit von fortgeschritteneren Tags wie `{% url %}` und `{% with %}` erkennen.
  - Daten mit Template-Filtern für die Anzeige formatieren.
  - Die Konfiguration für Templates in Django-Projekten verstehen.

1. Grundlagen: Was sind Django Templates?

Django Templates sind Textdateien (meist HTML), die dynamische Inhalte mithilfe der Django Template Language (DTL) generieren. Die DTL ist keine vollwertige Programmiersprache, sondern bietet eine begrenzte, aber mächtige Syntax, um Präsentationslogik abzubilden. Das Hauptziel ist die Trennung von Design und Python-Logik.

2. Template-Konfiguration (Einrichtung)

Django muss wissen, wo es nach Template-Dateien suchen soll. Dies wird in der `settings.py`-Datei des Projekts konfiguriert.

- **`settings.py` - Die `TEMPLATES`-Einstellung**:

```
# settings.py
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [], # Liste von Verzeichnissen, in denen Django projektweit nach Templates
sucht
        'APP_DIRS': True, # Django sucht automatisch in einem 'templates'-Unterordner jeder
installierten App
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

  - `APP_DIRS: True`: Ist dies gesetzt, sucht Django in jeder App im `INSTALLED_APPS`-Array nach einem Unterordner namens `templates`. Innerhalb dieses Ordners sollte für eine bessere Namensraumtrennung ein weiterer Unterordner mit dem Namen der App erstellt werden (z.B. `myapp/templates/myapp/mein_template.html`).
  - `DIRS`: Eine Liste von Pfaden, in denen Django zusätzlich nach Templates suchen soll (z.B. ein projektweiter `templates`-Ordner im Hauptverzeichnis des Projekts).

3. Der Render-Prozess: View -> Kontext -> Template

Eine View-Funktion in Django ist dafür verantwortlich, Daten vorzubereiten und ein Template mit diesen Daten zu rendern.

- **Die `render()`-Funktion**: Der häufigste Weg, ein Template zu laden und mit Daten zu füllen, ist die `render()`-Shortcut-Funktion.

```python
# myapp/views.py
from django.shortcuts import render

def meine_view(request):
    context = {
        'name': 'Welt',
        'alter': 30,
        'items': ['Apfel', 'Banane', 'Kirsche']
    }
    return render(request, 'myapp/mein_template.html', context)
```

- `request`: Das HttpRequest-Objekt.
- `'myapp/mein_template.html'`: Der Pfad zum Template, relativ zu einem der konfigurierten Template-Verzeichnisse.
- `context`: Ein Python-Dictionary, dessen Schlüssel als Variablennamen im Template verfügbar gemacht werden.

4. Variablen in Templates

Variablen werden in doppelten geschweiften Klammern `{{ variable }}` in das Template eingefügt.

- **Einfacher Zugriff**:

```html
<h1>Hallo, {{ name }}!</h1>
<p>Alter: {{ alter }}</p>
```

- **Dot-Notation**: Für den Zugriff auf Attribute von Objekten oder Werte in Dictionaries.

```
<p>Name: {{ person.name }}</p> <p>Beruf: {{ person.job }}</p>
<p>Erste Frucht: {{ fruits.0 }}</p> ```

Django versucht nacheinander: Dictionary-Lookup, Attribut-Lookup, Listen-Index-Lookup.
```

5. Django Template Tags (Steuerstrukturen)

Tags stellen die Logik in Templates bereit und werden mit `{% tag %}` umschlossen.

- `{% if %}`, `{% elif %}`, `{% else %}`, `{% endif %}`: Für bedingte Anweisungen.

```
{% if alter >= 18 %}
    <p>Diese Person ist volljährig.</p>
{% elif alter >= 16 %}
    <p>Diese Person ist jugendlich.</p>
{% else %}
    <p>Diese Person ist minderjährig.</p>
{% endif %}
```

- `{% for item in liste %}`, `{% empty %}`, `{% endfor %}`: Iteriert über jedes Element in einer Sequenz.

```
<ul>
    {% for item in items %}
        <li>{{ item }}</li>
    {% empty %}
        <li>Keine Items vorhanden.</li>
    {% endfor %}
</ul>
```

Innerhalb einer `for`-Schleife stehen spezielle `forloop`-Variablen zur Verfügung:

  - `forloop.counter`: Der aktuelle Stand des Zählers (1-basiert).
  - `forloop.counter0`: Der aktuelle Stand des Zählers (0-basiert).
  - `forloop.first`: `True`, wenn es der erste Durchlauf ist.
  - `forloop.last`: `True`, wenn es der letzte Durchlauf ist.

- **`{% with %}`**: Weist das Ergebnis eines komplexen Ausdrucks einer einfacheren Variable zu, um die Lesbarkeit zu erhöhen oder um teure Operationen nicht mehrmals auszuführen.

```
{% with anzahl_items=items|length %}
    {% if anzahl_items > 0 %}
        <p>Es gibt {{ anzahl_items }} Item(s).</p>
    {% endif %}
{% endwith %}
```

- **`{% url 'url_name' arg1 arg2 %}`**: Erzeugt dynamisch eine URL basierend auf dem Namen, der in `urls.py` für ein URL-Pattern vergeben wurde. Dies ist die bevorzugte Methode, um URLs in Templates zu referenzieren, da sie robust gegenüber Änderungen in den URL-Pfaden ist.

```
# myapp/urls.py
# path('article/<int:article_id>/', views.article_detail, name='article-detail')
```

```
<a href="{% url 'article-detail' article.id %}">Zum Artikel: {{ article.title }}</a>
```

- **`{% csrf_token %}`**: Ein kritischer Security-Tag, der in jedes Django-Formular eingefügt werden muss, das per `POST`-Methode gesendet wird, um Cross-Site Request Forgery-Angriffe zu verhindern. Wird später bei Formularen wichtig.

6. Django Template Filter (Datenformatierung)

Filter verändern die Darstellung von Variablen und werden mit einem Pipe-Symbol `|` angewendet: `{{ variable|filtername:argument }}`.

- **`upper`** / **`lower`** / **`title`**: Ändert die Groß-/Kleinschreibung.

```
<p>{{ name|upper }}</p> <p>{{ "SOME Text"|lower }}</p> <p>{{ name|title }}</p> ```
```

- **`length`**: Gibt die Länge einer Liste oder eines Strings zurück.

```
<p>Anzahl Items: {{ items|length }}</p>
```

- **`date`**: Formatiert Datums- oder Zeitobjekte. Benötigt einen Formatstring.

```
<p>Veröffentlicht am: {{ pub_date|date:"d. M. Y" }}</p> <p>Zeit: {{ pub_date|date:"H:i" }}
</p> ```
```

- **`default`**: Gibt einen Standardwert aus, falls die Variable nicht existiert, `False` oder leer ist.

```
<p>Beschreibung: {{ product.description|default:"Keine Beschreibung verfügbar." }}</p>
```

- **`truncatewords:ZAHL`** / **`truncatechars:ZAHL`**: Kürzt einen String auf eine bestimmte Anzahl von Wörtern oder Zeichen.

```
<p>{{ article.content|truncatewords:10 }}</p> ```
```

- `join:", "`: Verbindet die Elemente einer Liste mit einem String.
- `first` / `last`: Gibt das erste/letzte Element einer Liste zurück.
- `slice:":3"`: Gibt einen Teil einer Liste zurück (hier die ersten 3 Elemente).

7. Kommentare in Templates

- Einzeilige Kommentare: `{# Das ist ein Kommentar #}`
- Mehrzeilige Kommentare: `{% comment %}` Mehrzeiliger Text `{% endcomment %}`

Fazit

- **DTL als Werkzeug**: Die Django Template Language ist ein mächtiges Werkzeug zur dynamischen Generierung von HTML.
- **Variablen, Tags, Filter**: Die Grundbausteine. Variablen zeigen Daten, Tags steuern die Logik, Filter formatieren die Ausgabe.
- **Konfiguration**: Die `settings.py` bestimmt, wo Django Templates findet.
- **Kontext**: Das Dictionary, das von der View an das Template übergeben wird, macht Python-Variablen im Template verfügbar.
- **Best Practices**: Die Verwendung von `{% url %}` für Verlinkungen erhöht die Wartbarkeit.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) werden nun Templates erstellt, um die Umfragenliste und eine Detailansicht dynamisch darzustellen.

1. View-Funktionen in `polls/views.py` anpassen/erstellen:

```
# polls/views.py
from django.shortcuts import render, get_object_or_404
from .models import Question

def question_list(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/question_list.html', context)

def question_detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    context = {'question': question}
    return render(request, 'polls/question_detail.html', context)
```

2. URL-Konfiguration in `polls/urls.py`:

```
# polls/urls.py
from django.urls import path
from . import views

app_name = 'polls' # Wichtig für den {% url %} Tag
urlpatterns = [
    path('', views.question_list, name='list'),
    path('<int:question_id>/', views.question_detail, name='detail'),
]
```

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

```
{# polls/templates/polls/question_list.html #}
{% if latest_question_list %}
    <h1>Alle Umfragen</h1>
```

```
    <ul>
    {% for question in latest_question_list %}
        <li>
            <a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a>
            (Veröffentlicht: {{ question.pub_date|date:"d. F Y" }})
        </li>
    {% endfor %}
    </ul>
{% else %}
    <p>Keine Umfragen verfügbar.</p>
{% endif %}
```

4. Template `polls/templates/polls/question_detail.html` erstellen:

```
{# polls/templates/polls/question_detail.html #}
<h1>{{ question.question_text }}</h1>
<p>Veröffentlicht am: {{ question.pub_date|date:"D, d. M. Y, H:i" }} Uhr</p>

{% if question.choice_set.all %}
    <ul>
    {% for choice in question.choice_set.all %}
        <li>{{ choice.choice_text }} -- {{ choice.votes }} Stimme(n)</li>
    {% endfor %}
    </ul>
{% else %}
    <p>Für diese Frage gibt es noch keine Antwortmöglichkeiten.</p>
{% endif %}

<a href="{% url 'polls:list' %}">Zurück zur Übersicht</a>
```

Cheat Sheet

Wichtige Template-Konstrukte

- **Variable**: `{{ variable.attribut|filter:"argument" }}`
- **Tags**:
  - `{% if bedingung %}` ... `{% elif ... %}` ... `{% else %}` ... `{% endif %}`
  - `{% for item in liste %}` ... `{% empty %}` ... `{% endfor %}` (Helfer: `forloop.counter`)
  - `{% with alias=variable %}` ... `{% endwith %}`
  - `{% url 'url_name' arg1 %}` (benötigt `app_name` in `app/urls.py`)
  - `{% csrf_token %}` (für POST-Formulare)
- **Filter**:
  - `|length`: Länge.
  - `|upper`, `|lower`, `|title`: Groß-/Kleinschreibung.
  - `|date:"D d M Y"`: Datumsformatierung (siehe Python `strftime`-Format).
  - `|default:"fallback"`: Standardwert.
  - `|truncatewords:10`: Auf X Wörter kürzen.
- **Kommentare**: `{# Kommentar #}` oder `{% comment %}` ... `{% endcomment %}`

Übungsaufgaben

1. **Daten im View vorbereiten**:
   - Erstelle in einer `views.py` eine View-Funktion.
   - In dieser Funktion ein Dictionary als Kontext erstellen, das Folgendes enthält:
     - `page_title` (String, z.B. "Meine tolle Produktliste")
     - `products` (eine Liste von Dictionaries, wobei jedes Produkt-Dictionary Schlüssel wie `name`, `price` und `is_available` (boolean) hat). Mindestens 3 Produkte definieren.
     - `current_user_name` (String, z.B. "Gast" oder ein Name)

2. **Template erstellen**:
    - Ein HTML-Template erstellen, das diesen Kontext verwendet.
    - Den `page_title` im `<title>`-Tag und als `<h1>`-Überschrift ausgeben.
    - Mit einer `for`-Schleife über die `products`-Liste iterieren.
    - Für jedes Produkt den Namen und den Preis anzeigen.
    - Mit einer `if`-Bedingung prüfen, ob `is_available` `True` ist. Wenn ja, "Auf Lager" anzeigen, sonst "Nicht verfügbar".
    - Den `current_user_name` anzeigen. Wenn er "Gast" ist, einen Link zur (noch nicht existenten) Login-Seite anbieten, sonst eine persönliche Begrüßung.
    - Den Preis mit einem Filter formatieren (falls relevant, z.B. wenn es eine komplexere Zahl wäre, hier eher zum Üben des Filters).
    - Die Anzahl der Produkte in der Liste mit dem `length`-Filter anzeigen.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die "Community Recipe Sharing Platform" erhält nun ihre ersten Templates, um Rezepte anzuzeigen. Zunächst arbeiten wir mit Dummy-Daten aus den Views.

Aufgabe:

1. **App-Template-Ordner erstellen**: Sicherstellen, dass die Ordnerstruktur `recipes/templates/recipes/` existiert.
2. **View für Rezeptliste (`recipe_list_view`) in `recipes/views.py` erstellen**:
    - Diese View soll vorerst eine **statische Liste von Dictionaries** als `recipes_data` erstellen. Jedes Dictionary repräsentiert ein Rezept und sollte Schlüssel wie `id`, `title` (String), `description` (String) und `author_name` (String) haben. Mindestens 3 Beispielrezepte definieren.
    - Einen Kontext erstellen, der diese `recipes_data` unter dem Schlüssel `recipes` an das Template übergibt.
    - Das Template `recipes/recipe_list.html` rendern.
3. **Template `recipes/recipe_list.html` erstellen**:
    - Eine Überschrift "Alle Rezepte" anzeigen.
    - Mit einer `{% for recipe in recipes %}`-Schleife über die Rezepte iterieren.
    - Für jedes Rezept den `title` und den `author_name` anzeigen. Den `title` mit dem `|title`-Filter formatieren.
    - Jeden Rezepttitel zu einer Detailseite verlinken (z.B. `{% url 'recipes:detail' recipe.id %}`). Dieser Link wird erst im nächsten Schritt funktionieren, aber der Tag kann schon vorbereitet werden.
    - Wenn die `recipes`-Liste leer ist (`{% empty %}`), eine Nachricht "Keine Rezepte vorhanden." anzeigen.
4. **URL-Konfiguration für die Rezeptliste**:
    - In `recipes/urls.py` einen Pfad (z.B. leer `''` oder `list/`) für die `recipe_list_view` definieren und ihm den Namen `list` geben.
    - Die `recipes/urls.py` in die Haupt-`urls.py` einbinden, falls noch nicht geschehen (z.B. unter dem Präfix `rezepte/`).
5. **Testen**: Die Seite im Browser aufrufen und prüfen, ob die Dummy-Rezepte korrekt angezeigt werden.
6. **(Optional) View und Template für Rezeptdetails (`recipe_detail_view` und `recipe_detail.html`)**:
    - Analog zur Listenansicht eine Detailansicht für ein einzelnes (Dummy-)Rezept erstellen. Die View soll eine `recipe_id` aus der URL entgegennehmen und ein einzelnes (Dummy-)Rezept-Dictionary an das Template `recipes/recipe_detail.html` übergeben.
    - Im Template `recipe_detail.html` alle Details des Rezepts anzeigen (Titel, Beschreibung, Autor).
    - Eine entsprechende URL in `recipes/urls.py` definieren (z.B. `<int:recipe_id>/` mit dem Namen `detail`).