04 - Django: Query-Parameter und MVT-Architektur

Einleitung

- Dieses Kapitel erweitert das Verständnis des URL-Routings um die Verwendung von Query-Parametern.
- Query-Parameter bieten eine flexible Möglichkeit, optionale oder zusätzliche Daten an Views zu übergeben.
- Zudem wird die Django-spezifische MVT-Architektur (Model-View-Template) detailliert beleuchtet.
- Die MVT-Architektur ist zentral für das Verständnis des Datenflusses und der Verantwortlichkeiten innerhalb einer Django-Anwendung.

---

1. URL-Routing mit Query-Parametern

Neben den URL-Parametern, die direkt im Pfad enthalten sind (wie in Skript 03 behandelt), gibt es eine weitere Methode, Daten an eine View zu übergeben: **Query-Parameter**. Diese sind besonders nützlich für optionale Daten wie Suchbegriffe, Filterkriterien oder Sortieroptionen.

1.1 Was sind Query-Parameter?

Query-Parameter sind Schlüssel-Wert-Paare, die an das Ende einer URL angehängt werden, beginnend mit einem Fragezeichen (`?`). Mehrere Parameter werden durch ein Ampersand (`&`) voneinander getrennt.

Beispiel-URL mit Query-Parametern: `http://example.com/search?q=django+tutorial&sort=date&page=2`

In diesem Beispiel:

- `q`: Suchbegriff (`django+tutorial`)
- `sort`: Sortierkriterium (`date`)
- `page`: Seitennummer (`2`)

Query-Parameter sind Teil des `Query`-Segments der URL-Struktur, wie in Skript 01 beschrieben[cite: 989].

1.2 Auslesen von Query-Parametern in Views (`request.GET`)

Django stellt Query-Parameter über das `request.GET`-Attribut des `HttpRequest`-Objekts zur Verfügung. `request.GET` ist ein `QueryDict`-Objekt, das wie ein Python-Dictionary funktioniert.

Schritt-für-Schritt-Anleitung:

1. **View-Funktion erstellen:** Definieren Sie eine View-Funktion, die die Query-Parameter ausliest.

```
# myapp/views.py
from django.http import HttpResponse

def search_results(request):
    # request.GET.get('key', default_value) ist die empfohlene Methode,
    # da sie einen Standardwert zurückgibt, wenn der Parameter nicht vorhanden ist.
    query = request.GET.get('q', 'kein Suchbegriff')
    category = request.GET.get('cat', 'alle Kategorien')
    sort_by = request.GET.get('sort', 'relevanz')

    response_text = f"Suchergebnisse für: '{query}' in Kategorie: '{category}', sortiert
nach: '{sort_by}'."
    return HttpResponse(response_text)
```

2. **App-spezifische URLs anpassen (`myapp/urls.py`):** Fügen Sie ein URL-Pattern für die View hinzu. Beachten Sie, dass Query-Parameter nicht direkt im `path()`-Muster definiert werden, da sie optional sind und an jede URL angehängt werden können.

```
# myapp/urls.py
from django.urls import path
```

```python
from . import views

urlpatterns = [
    # ... andere Routen
    path('search/', views.search_results, name='search_results'),
]
```

3. **Server starten und testen:**
    - Starten Sie den Entwicklungsserver: `python manage.py runserver`
    - Testen Sie im Browser die Funktionalität mit verschiedenen URLs:
        - `http://127.0.0.1:8000/app/search/` (Standardwerte)
        - `http://127.0.0.1:8000/app/search/?q=pizza`
        - `http://127.0.0.1:8000/app/search/?q=pizza&cat=italienisch`
        - `http://127.0.0.1:8000/app/search/?cat=dessert&sort=rating`

1.3 `request.GET.getlist()` für Mehrfachauswahl

Wenn ein Parameter mehrfach in der URL vorkommt (z.B. bei mehreren ausgewählten Checkboxen in einem Formular), kann `request.GET.getlist()` verwendet werden, um alle Werte als Liste zu erhalten.

Beispiel: `http://example.com/products?color=red&color=blue&size=M`

```python
# myapp/views.py
def product_filter(request):
    colors = request.GET.getlist('color') # ['red', 'blue']
    size = request.GET.get('size')        # 'M'
    response_text = f"Produkte filtern nach Farben: {', '.join(colors)} und Größe: {size}."
    return HttpResponse(response_text)
```

---

2. Django's MVT-Architektur (Model-View-Template)

Django verwendet eine Variante des klassischen MVC-Musters, bekannt als **MVT-Architektur (Model-View-Template)** . Das Verständnis dieser Architektur ist entscheidend für die Strukturierung und den Datenfluss in Django-Anwendungen.

2.1 Überblick über MVC (Model-View-Controller)

MVC ist ein weit verbreitetes Architekturmuster in der Softwareentwicklung, das eine Anwendung in drei logische Komponenten unterteilt[cite: 113, 122]:

- **Model:** Repräsentiert die Datenstruktur und Geschäftslogik einer Anwendung. Es steuert die Interaktionen mit der Datenbank[cite: 114].
- **View:** Die Präsentationsschicht, die die Benutzeroberfläche anzeigt und oft Logik zur Präsentation enthält[cite: 115].
- **Controller:** Steuert die Verbindungen zwischen Model und View. Er interpretiert Benutzeraktionen, manipuliert das Modell und aktualisiert die Ansicht[cite: 116].

2.2 Das Django-spezifische MVT-Muster

Django implementiert eine eigene Variante des MVC-Musters namens MVT[cite: 123]. Der Hauptunterschied besteht darin, dass Django einen Großteil der Controller-Logik "im Hintergrund" übernimmt und Entwickler sich mehr auf die Definition von Models, Views und Templates konzentrieren können.

- **Model:** Bleibt unverändert im Vergleich zu MVC. Es stellt die Datenstruktur der Anwendung dar und definiert, wie Daten gespeichert und abgerufen werden. In Django sind Model-Klassen Python-Klassen, die Tabellen in der Datenbank repräsentieren[cite: 117, 125, 126].
    - **Verantwortlichkeit:** Datenmanagement, Datenbankinteraktion, Geschäftslogik der Daten.
    - **Dateien:** Hauptsächlich in `models.py`.

- **View:** In Django entspricht die View-Komponente der **Controller-Logik** im klassischen MVC. Sie verarbeitet Anfragen (`HttpRequest`), interagiert mit dem Model, um Daten zu erhalten oder zu speichern, und wählt das passende Template aus, um eine Antwort zu generieren[cite: 118, 127]. Die View ist der Ort, an dem die Anwendung ihre Hauptlogik steuert.
    - **Verantwortlichkeit:** Verarbeitung von Anfragen, Interaktion mit Models, Auswahl und Übergabe von Daten an Templates, Generierung der Antwort.
    - **Dateien:** Hauptsächlich in `views.py`.
- **Template:** Dies ist die **Präsentationsschicht** in Django. Templates sind HTML-Dateien, die Platzhalter für dynamische Inhalte enthalten und zusammen mit der Django Template Language (DTL) zur Darstellung von Daten genutzt werden[cite: 119, 129, 225]. Sie sind für die Struktur und das Aussehen der Benutzeroberfläche zuständig.
    - **Verantwortlichkeit:** Darstellung der Daten für den Benutzer.
    - **Dateien:** Typischerweise in `templates/` Verzeichnissen innerhalb der Apps.

2.3 Datenfluss in der MVT-Architektur

Der Datenfluss in einer Django-Anwendung folgt einem klaren Muster:

1. **Request kommt an:** Ein Client (Webbrowser) sendet eine HTTP-Anfrage an den Django-Server.
2. **URL-Dispatcher:** Django's URL-Dispatcher (`urls.py`) empfängt die Anfrage und versucht, die URL einem passenden Pattern zuzuordnen.
3. **View wird aufgerufen:** Wenn ein passendes URL-Pattern gefunden wird, ruft der Dispatcher die zugehörige View-Funktion auf und übergibt das `HttpRequest`-Objekt.
4. **View-Logik:** Die View-Funktion führt die Geschäftslogik aus:
    - Sie kann mit dem **Model** interagieren, um Daten aus der Datenbank abzurufen oder zu speichern.
    - Sie verarbeitet die Daten.
    - Sie bereitet ein Kontext-Dictionary mit den Daten vor, die im Template benötigt werden.
    - Sie wählt ein **Template** aus.
5. **Template-Rendering:** Die View übergibt das Kontext-Dictionary und das ausgewählte Template an Django's Template-Engine. Die Engine füllt die Platzhalter im Template mit den Daten aus dem Kontext.
6. **Response geht zurück:** Das gerenderte HTML (oder eine andere Antwort) wird als `HttpResponse`-Objekt an den Client zurückgesendet.

Vorteile von MVT in Django:

- **Klare Trennung der Logik:** Das MVT-Pattern strukturiert den Code sauber, da Views und Templates klar getrennt sind[cite: 120].
- **Effizienz:** Django übernimmt automatisch viele Aufgaben, die im klassischen MVC ein Controller erledigen müsste (z.B. das Routing), was Entwicklungszeit spart[cite: 121].
- **Wartbarkeit und Skalierbarkeit:** Durch die klare Trennung der Verantwortlichkeiten sind Django-Anwendungen leichter zu warten, zu testen und zu skalieren.

Fazit

- Query-Parameter bieten eine flexible Methode zur Übergabe optionaler Daten an Django Views über die URL, die mittels `request.GET` ausgelesen werden.
- Django's MVT-Architektur (Model-View-Template) ist eine effiziente Adaption des MVC-Prinzips.
- Im MVT-Muster ist das Model für Daten, die View für die Logik (Controller-Teil) und das Template für die Darstellung verantwortlich.
- Das Verständnis des Datenflusses durch URL-Dispatcher, View, Model und Template-Engine ist grundlegend für die Entwicklung robuster Django-Anwendungen.

Cheat Sheet: Skript 04 - Django Query-Parameter und MVT-Architektur

Query-Parameter (URL)

- **Definition:** Schlüssel-Wert-Paare nach `?` in der URL (z.B. `?key=value&key2=value2`).
- **Auslesen in View:** `request.GET.get('parameter_name', 'Standardwert')`
- **Mehrere Werte auslesen:** `request.GET.getlist('parameter_name')`
- **Einsatzbereich:** Optionale Filter, Suchen, Sortierungen.

Django MVT-Architektur

- **Model:**
  - **Verantwortlichkeit:** Datenstruktur, Datenbank-Interaktion, Geschäftslogik der Daten.
  - **Dateien:** `models.py`.
  - **Entspricht in MVC:** Model.
- **View:**
  - **Verantwortlichkeit:** Verarbeitung von HTTP-Anfragen, Interaktion mit Models, Auswahl/Aufbereitung von Daten für Templates, Generierung der `HttpResponse`.
  - **Dateien:** `views.py`.
  - **Entspricht in MVC:** Controller-Logik.
- **Template:**
  - **Verantwortlichkeit:** Präsentation der Daten für den Benutzer (HTML mit Platzhaltern).
  - **Dateien:** `templates/` Verzeichnisse.
  - **Entspricht in MVC:** View.

Datenfluss in MVT

1. **Request** (Client)
2. **URL-Dispatcher** (`urls.py`)
3. **View** (Verarbeitet Request, interagiert mit Model, wählt Template)
4. **Model** (Interagiert mit DB)
5. **Template-Engine** (Rendert Template mit Daten)
6. **Response** (Server an Client)

Übungsaufgaben zu Skript 04

Ziel der Übungen: Anwendung von Query-Parametern und Festigung des Verständnisses für den Datenfluss in der MVT-Architektur.

1. **Produktsuche mit Query-Parametern:**
   - **Aufgabe:**
     1. Erstellen Sie in einer Ihrer Django-Apps (z.B. `homepage` oder einer neuen Test-App) eine View-Funktion namens `product_search`.
     2. Diese View soll zwei Query-Parameter entgegennehmen: `q` für den Suchbegriff und `category` für eine optionale Kategorie.
     3. Wenn `q` nicht vorhanden ist, soll der Standardwert "Alle Produkte" verwendet werden. Wenn `category` nicht vorhanden ist, soll "alle Kategorien" verwendet werden.
     4. Die View soll eine `HttpResponse` zurückgeben, die die ausgelesenen Parameter anzeigt, z.B.: "Suche nach: 'Laptop' in Kategorie: 'Elektronik'".
     5. Definieren Sie ein entsprechendes URL-Pattern in der App-`urls.py`, das den Pfad `'products/search/'` der `product_search`-View zuordnet.
     6. Testen Sie die Funktion mit URLs wie:
        - `http://127.0.0.1:8000/app/products/search/`
        - `http://127.0.0.1:8000/app/products/search/?q=keyboard`
        - `http://127.0.0.1:8000/app/products/search/?q=mouse&category=peripherals`
2. **Filterung nach mehreren Kriterien (konzeptionell):**
   - **Aufgabe:**
     1. Stellen Sie sich vor, Sie haben eine URL wie `http://example.com/filter?size=S&size=M&color=blue`.
     2. Beschreiben Sie, wie Sie in einer Django View auf die Werte für `size` und `color` zugreifen würden, um alle ausgewählten Größen und die Farbe zu erhalten.
     3. Zeigen Sie den entsprechenden Python-Code-Ausschnitt für die View-Funktion.
3. **MVT-Rollenspiel:**

- **Aufgabe:**
    1. Wählen Sie eine einfache Webanwendung (z.B. ein Blog, ein Online-Shop, ein Wetterberichtsdienst).
    2. Beschreiben Sie für diese Anwendung einen typischen Datenfluss (z.B. "Benutzer ruft Blogbeitrag auf"), und erklären Sie dabei die genauen Rollen von **Model, View und Template** im Kontext von Django's MVT-Architektur. Was würde jede Komponente tun?

📚 Leitfaden-Projekt (Demonstration): Online Umfragesystem

Konzept für den heutigen Tag: Heute wird demonstriert, wie die Umfrage-App im "Online Umfragesystem" erste dynamische Funktionalitäten mittels URL-Parametern und Query-Parametern erhält. Dies bildet die Grundlage für spätere Interaktionen.

Aufgabe (Tag 4 für den Dozenten):

1. **View für spezifische Umfrage-Details (mit URL-Parameter):**
    - Zeigen Sie in der `views.py` Ihrer `questions` (oder `polls`)-App eine `detail_view`-Funktion.
    - Diese Funktion soll einen `question_id` (Integer-Typ) als URL-Parameter entgegennehmen.
    - Als `HttpResponse` soll ein Text wie "Details der Umfrage mit ID: [question_id] werden hier angezeigt." zurückgegeben werden.
    - **Demonstration:** Zeigen Sie, wie eine URL wie `http://127.0.0.1:8000/polls/5/` die `question_id` 5 ausliest.
2. **Simulierte Suchfunktion für Umfragen (mit Query-Parametern):**
    - Erstellen Sie eine neue View-Funktion namens `search_polls` in der `views.py`.
    - Diese View soll einen optionalen Query-Parameter `q` (für den Suchbegriff) entgegennehmen. Wenn `q` nicht vorhanden ist, soll ein Standardtext angezeigt werden (z.B. "Geben Sie einen Suchbegriff ein.").
    - Als `HttpResponse` soll der Text "Suche nach Umfragen mit Begriff: '[Suchbegriff]'" zurückgegeben werden.
    - **Demonstration:** Zeigen Sie, wie URLs wie `http://127.0.0.1:8000/polls/search/` und `http://127.0.0.1:8000/polls/search/?q=Sport` funktionieren.
3. **Analyse des Datenflusses im Polls-Projekt (MVT):**
    - Erklären Sie anhand des bisherigen Polls-Projekts (Setup, Views, URLs), wie eine Anfrage für `http://127.0.0.1:8000/polls/5/` die verschiedenen Komponenten der MVT-Architektur durchläuft:
        - **Client:** Sendet Request.
        - **`urls.py` (Projekt-Level):** Leitet an App-URLs weiter.
        - **`urls.py` (App-Level):** Matcht den `question_id`-Parameter und ruft die `detail_view` auf.
        - **`views.py` (`detail_view`):** Empfängt den `request` und `question_id`, generiert `HttpResponse`.
        - **Response:** Wird zurück an den Client gesendet.

🚀 Schüler-Projekt: Aufbau der "Community Recipe Sharing Platform"

Konzept für den heutigen Tag: Die "Community Recipe Sharing Platform" erhält heute eine simulierte Suchfunktionalität. Die Schüler wenden die Konzepte der Query-Parameter an, um Filter- und Suchanfragen an ihre noch statische Rezeptliste zu senden. Dies ist ein wichtiger Schritt, um die spätere dynamische Suche vorzubereiten.

Aufgabe (Tag 4 für die Schüler):

1. **"Rezept-Suche"-View implementieren:**
    - Öffnen Sie die `views.py` Ihrer `recipes`-App im Projekt `recipe_platform`.
    - Fügen Sie eine neue View-Funktion namens `search_recipes` hinzu.
    - Diese Funktion soll zwei optionale Query-Parameter entgegennehmen:
        - `q`: Für den Suchbegriff (Standardwert: "alle Rezepte").
        - `cuisine`: Für die Küchenart (Standardwert: "beliebige Küche").
    - Die View soll eine `HttpResponse` zurückgeben, die eine Nachricht wie "Suche nach: '[Suchbegriff]' in der Küche: '[Küchenart]'" enthält.
    - *Noch keine Datenbankinteraktion!* Es geht darum, die Query-Parameter korrekt auszulesen.
2. **URL-Routing für Rezept-Suche anlegen:**
    - Öffnen Sie die `urls.py` Ihrer `recipes`-App (`recipe_platform/recipes/urls.py`).

- Fügen Sie ein neues URL-Pattern hinzu, das den Pfad `'search/'` der `search_recipes`-View-Funktion zuordnet. Geben Sie dieser Route den Namen `'search'`.

3. **Funktionalität überprüfen:**

   - Stellen Sie sicher, dass der Django-Entwicklungsserver läuft.
   - Testen Sie die neue Funktionalität, indem Sie folgende URLs im Browser aufrufen und die Ausgabe überprüfen:
     - `http://127.0.0.1:8000/recipes/search/`
     - `http://127.0.0.1:8000/recipes/search/?q=pasta`
     - `http://127.0.0.1:8000/recipes/search/?q=pizza&cuisine=italienisch`
     - `http://127.0.0.1:8000/recipes/search/?cuisine=asiatisch`

4. **MVT im eigenen Projekt reflektieren:**

   - Überlegen Sie anhand Ihres `recipe_platform`-Projekts, wie die MVT-Komponenten (`models.py`, `views.py`, `urls.py` und die hypothetischen Templates) miteinander interagieren, wenn ein Benutzer eine Suchanfrage für Rezepte startet.
   - Skizzieren Sie den Datenfluss von der Browser-Anfrage bis zur Antwort, die Sie im Terminal sehen.