15 - Django: Projekt-Konsolidierung & Best Practices

Einleitung

- **Themen**: Dieses Skript dient als Abschluss des reinen Django-Frontend-Moduls. Es geht darum, das bisher Gelernte zu wiederholen, zu festigen und das eigene Projekt zu einer vollständigen, funktionierenden Anwendung auszubauen.
- **Fokus**: Zusammenführung aller Konzepte: Models, Views, Templates, Forms, Authentifizierung und Dateiverwaltung. Ein besonderer Schwerpunkt liegt auf der Überprüfung des eigenen Codes anhand von Best Practices für sauberen, lesbaren und wartbaren Code.
- **Lernziele**:
  - Alle bisherigen Django-Konzepte in einem einzigen, kohärenten Projekt anwenden können.
  - Die eigene Code-Struktur kritisch überprüfen und verbessern.
  - Best Practices für Django-Projekte kennen und anwenden.
  - Ein voll funktionsfähiges, Template-basiertes Django-Projekt abschließen.
  - Übung im Code-Review und Debugging erhalten.

Checkliste für ein sauberes Django-Projekt

Dieses Kapitel dient als Checkliste, mit der das eigene Projekt überprüft und verbessert werden kann.

1. Code-Struktur & Clean Code

- **Logische App-Struktur**: Ist das Projekt sinnvoll in Apps unterteilt? Jede App sollte eine klare, abgegrenzte Funktionalität haben (z.B. `recipes`, `accounts`).
- **"Fat Models, Thin Views"**: Dieses Prinzip ist eine wichtige Leitlinie.
  - **Views (`views.py`)** sollten "dünn" sein, d.h. sie kümmern sich um die Verarbeitung von HTTP-Requests und -Responses, die Formularvalidierung und die Authentifizierung.
  - **Models (`models.py`)** können "fett" sein. Geschäftslogik, die sich direkt auf die Daten eines Modells bezieht, sollte als Methode im Modell selbst implementiert werden.
    - **Schlechtes Beispiel (in der View):** `is_recent = mein_rezept.pub_date > (timezone.now() - timedelta(days=1))`
    - **Gutes Beispiel (im Modell):**

```
# recipes/models.py
class Recipe(models.Model):
    # ... felder ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

# In der View/Template:
# if mein_rezept.was_published_recently(): ...
```

- **Sinnvolle Benennung**: Sind Variablen, Funktionen, Klassen und URL-Namen klar und verständlich benannt? Ein Name sollte seine Funktion beschreiben.
- **`__str__`-Methode**: Hat jedes Modell eine `__str__`-Methode? Dies ist unerlässlich für eine lesbare Darstellung von Objekten, besonders im Admin-Bereich und beim Debugging.

2. URL-Design & Navigation

- **Logische URL-Pfade**: Sind die URLs einheitlich und intuitiv? (z.B. `recipes/`, `recipes/1/`, `recipes/1/bearbeiten/`, `recipes/neu/`).
- **Konsequente Nutzung von `{% url %}`**: Sind alle Links in den Templates dynamisch mit dem `{% url 'app_name:url_name' %}`-Tag generiert? Hartcodierte Pfade wie `<a href="/recipes/1/">` sind fehleranfällig und sollten vermieden werden.

3. Template-Struktur & Wiederverwendung

- **Effektive Vererbung (`{% extends %}`)**: Wird eine zentrale `base.html`-Datei genutzt, um redundanten HTML-Code (wie `<head>`, Header, Footer) zu vermeiden?

- **Sinnvolle Includes (`{% include %}`)**: Werden wiederkehrende Komponenten wie Navigationsleisten oder Sidebars in eigene Dateien ausgelagert und bei Bedarf inkludiert?
- **Statische Dateien (`{% static %}`):** Werden alle Pfade zu CSS, JavaScript und Bildern korrekt und dynamisch mit dem `{% static %}`-Tag geladen?

4. Formular-Verarbeitung & Sicherheit

- **GET vs. POST**: Ist die Logik in den Views, die Formulare verarbeiten, klar zwischen dem Anzeigen des leeren Formulars (`GET`) und der Verarbeitung der gesendeten Daten (`POST`) getrennt?
- **CSRF-Schutz**: Enthält jedes `<form>`-Tag, das Daten per `POST` sendet, den `{% csrf_token %}`-Tag?
- **Datei-Uploads**: Haben Formulare, die Dateien hochladen, das Attribut `enctype="multipart/form-data"`?

5. Authentifizierung & Autorisierung

- **Geschützte Views**: Sind alle Views, die eine Anmeldung erfordern (z.B. zum Erstellen oder Bearbeiten von Inhalten), mit dem `@login_required`-Decorator geschützt?
- **Besitz-Prüfung (Ownership)**: Ist in den Bearbeitungs- und Lösch-Views sichergestellt, dass ein Benutzer nur seine eigenen Inhalte verändern kann? (z.B. mit einer Prüfung wie `if recipe.author != request.user:`).
- **Benutzer-Feedback (Messages Framework)**: Erhält der Benutzer nach wichtigen Aktionen (Login, Logout, Speichern eines Formulars) klares Feedback durch Nachrichten?

Fazit

- **Funktionalität ist nicht alles**: Ein funktionierendes Projekt ist der erste Schritt. Ein sauberes, wartbares und gut strukturiertes Projekt ist das Ziel professioneller Entwicklung.
- **Best Practices als Leitfaden**: Die oben genannten Punkte sind keine starren Regeln, sondern bewährte Praktiken, die helfen, die Code-Qualität nachhaltig zu verbessern.
- **Refactoring ist Teil des Prozesses**: Es ist normal und wichtig, den eigenen Code regelmäßig zu überprüfen und zu verbessern ("Refactoring"). Dieser Tag bietet die perfekte Gelegenheit dazu.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) wird eine abschließende Code-Review durchgeführt.

1. **Code-Überprüfung**: Der Dozent geht durch die `views.py`, `models.py` und Templates des Projekts und wendet die Checkliste an.
2. **Demonstration**: Es wird gezeigt, wie z.B. eine Logik aus einer View in eine Modell-Methode ausgelagert wird oder wie hartcodierte URLs durch `{% url %}`-Tags ersetzt werden.
3. **Abschluss**: Das Ergebnis ist eine saubere, voll funktionsfähige, Template-basierte Webanwendung, die als Referenz für die Schülerprojekte dient.

Cheat Sheet

Checkliste für Code-Qualität

- **Models**:
  - ☐ Jedes Modell hat eine `__str__`-Methode.
  - ☐ Geschäftslogik ist in Modell-Methoden gekapselt ("Fat Models").
- **Views**:
  - ☐ Views sind kurz und auf eine Aufgabe fokussiert ("Thin Views").
  - ☐ `get_object_or_404()` wird anstelle von manuellen `try...except`-Blöcken verwendet.
  - ☐ `@login_required` schützt alle notwendigen Views.
  - ☐ Ownership wird vor Bearbeitungs-/Löschvorgängen geprüft.
- **Templates**:
  - ☐ `{% extends %}` wird für ein Basis-Layout genutzt.
  - ☐ `{% include %}` wird für wiederkehrende Komponenten genutzt.
  - ☐ Alle URLs werden dynamisch mit `{% url 'app_name:url_name' %}` generiert.
  - ☐ Alle `POST`-Formulare enthalten `{% csrf_token %}`.
  - ☐ Alle statischen Dateien werden mit `{% static 'path/...' %}` geladen.

- **Allgemein**:
  - [ ] `accounts`-App ist für Benutzerverwaltung zuständig.
  - [ ] `.env`-Datei wird für sensible Daten verwendet und ist in `.gitignore`.

Übungsaufgaben

Die Hauptübung für diesen Tag ist die Arbeit am eigenen Projekt.

1. **Peer Code-Review**:
   - In Kleingruppen (2-3 Personen) stellen sich die Teilnehmenden gegenseitig ihre Projekte vor.
   - Anhand der oben genannten Checkliste gibt jeder Feedback zum Code des anderen.
     - Was ist gut gelöst?
     - Wo gibt es Verbesserungspotenzial?
     - Gibt es unklare Stellen im Code?
2. **Individuelles Refactoring**:
   - Basierend auf dem erhaltenen Feedback und der Checkliste überarbeitet jeder Teilnehmende den eigenen Code.
3. **Fragen & Debugging-Session**:
   - Offene Fragen und verbleibende Fehler werden im Plenum oder mit Unterstützung des Dozenten geklärt.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Das Ziel des heutigen Tages ist es, die "Community Recipe Sharing Platform" zu einer voll funktionsfähigen, sauberen Webanwendung auszubauen.

Aufgabe:

1. **Funktionalität vervollständigen**: Sicherstellen, dass alle Features aus den vorherigen Skripten korrekt implementiert sind und zusammenspielen:
   - Benutzer können sich registrieren, an- und abmelden.
   - Angemeldete Benutzer können neue Rezepte (inkl. Bild) erstellen.
   - Angemeldete Benutzer können **nur ihre eigenen** Rezepte bearbeiten und löschen.
   - Alle Rezepte werden in einer Listenansicht angezeigt.
   - Ein Klick auf ein Rezept führt zu einer Detailansicht.
2. **Code-Review mit der Checkliste**:
   - Das eigene Projekt anhand der "Checkliste für ein sauberes Django-Projekt" kritisch überprüfen.
   - Insbesondere folgende Punkte prüfen:
     - Sind alle URLs in den Templates dynamisch?
     - Wird Template-Vererbung konsequent genutzt?
     - Ist die `add_recipe`-View mit `form.save(commit=False)` korrekt implementiert, um den `author` zu setzen?
     - Ist die `edit_recipe`-View gegen den Zugriff von falschen Benutzern abgesichert?
     - Gibt es für jedes Modell eine `__str__`-Methode?
3. **Refactoring**:
   - Alle identifizierten Schwachstellen oder unsauberen Code-Stellen verbessern.
4. **Abschluss**: Am Ende des Tages sollte eine robuste, Template-basierte Django-Anwendung stehen, die bereit ist für den nächsten großen Schritt: die Entkopplung des Frontends mit einer REST API.