16 - Einführung in RESTful APIs mit Django REST Framework (DRF)

Einleitung

- **Themen**: Dieses Skript markiert einen Wendepunkt im Kurs. Wir bewegen uns weg davon, mit Django komplette HTML-Seiten zu rendern, und lernen stattdessen, wie man eine **RESTful API** erstellt. Eine API ist eine Schnittstelle, die reine Daten (meist im JSON-Format) bereitstellt.
- **Fokus**: Die grundlegenden Konzepte von REST verstehen. Das **Django REST Framework (DRF)** als mächtiges Werkzeug kennenlernen, um schnell und effizient APIs zu erstellen. Die Kernkomponenten von DRF – **Serializer** und **ViewSet** – werden eingeführt.
- **Lernziele**:
  - Verstehen, warum die Trennung von Backend (API) und Frontend (z.B. eine spätere React-App) eine moderne Best Practice ist.
  - Die Kernprinzipien von REST (Ressourcen, HTTP-Methoden, Statuscodes) anwenden können.
  - Das Django REST Framework installieren und in einem Projekt konfigurieren.
  - `ModelSerializer` erstellen, um Django-Modelle in JSON zu übersetzen.
  - Mit `ModelViewSet` und `Router` mit minimalem Code vollständige CRUD-Endpunkte erstellen.
  - Die DRF "Browsable API" zum Testen und Debuggen nutzen.

1. Warum eine API? Der Wechsel vom Monolithen zur entkoppelten Anwendung

Bisher haben wir Django als "Monolith" verwendet: Das Backend war für die Datenbanklogik zuständig UND hat die HTML-Seiten für das Frontend direkt generiert. Eine API entkoppelt diese beiden Aufgaben.

- **Backend (Django + DRF)**: Kümmert sich nur noch um die Geschäftslogik und die Bereitstellung von Daten im JSON-Format. Es ist ihm egal, wer diese Daten abruft.
- **Frontend (z.B. React)**: Eine komplett separate Anwendung, die im Browser des Benutzers läuft. Sie ruft die Daten von der Django-API ab und ist allein dafür zuständig, diese Daten in ansprechendes HTML umzuwandeln.

Vorteile:

- **Flexibilität**: Verschiedene Frontends (eine Web-App, eine Android-App, eine iOS-App) können alle dieselbe API nutzen.
- **Skalierbarkeit**: Backend- und Frontend-Teams können unabhängig voneinander arbeiten und ihre Teile der Anwendung separat skalieren und deployen.
- **Moderne User Experience**: Single-Page Applications (SPAs), wie man sie mit React baut, bieten eine flüssigere, App-ähnliche Erfahrung ohne ständiges Neuladen von Seiten.

2. Was ist eine RESTful API? (Wiederholung & Vertiefung)

REST (Representational State Transfer) ist ein Architekturstil für Webdienste, der auf den bewährten Prinzipien des Internets aufbaut.

- **Ressourcen**: Alles ist eine Ressource, die über eine eindeutige URL identifizierbar ist (z.B. `/api/recipes/`, `/api/recipes/1/`).
- **HTTP-Methoden (Verben)**: Standard-HTTP-Methoden werden für CRUD-Operationen (Create, Read, Update, Delete) auf diesen Ressourcen verwendet.
  - `GET`: Daten lesen.
  - `POST`: Neue Ressource erstellen.
  - `PUT`/`PATCH`: Bestehende Ressource aktualisieren.
  - `DELETE`: Ressource löschen.
- **HTTP-Statuscodes**: Der Server gibt standardisierte Codes zurück, um das Ergebnis einer Anfrage zu signalisieren (z.B. `200 OK`, `201 Created`, `404 Not Found`).
- **Zustandslosigkeit (Statelessness)**: Jede Anfrage enthält alle Informationen, die der Server zur Bearbeitung benötigt. Der Server speichert keinen Sitzungsstatus.

3. Django REST Framework (DRF) einrichten

1. **Installation**:

```
pip install djangorestframework
```

2. **Konfiguration (`settings.py`)**: Füge `'rest_framework'` zur `INSTALLED_APPS`-Liste hinzu.

```
# settings.py
INSTALLED_APPS = [
    # ...
    'rest_framework',
    'recipes',
    # ...
]
```

4. Der Serializer: Der Datenübersetzer

Ein Serializer hat zwei Hauptaufgaben:

1. **Serialisierung (Für Antworten)**: Komplexe Python-Objekte (wie Django-Modellinstanzen) in ein einfaches Format wie JSON umwandeln, das über das Internet gesendet werden kann.
2. **Deserialisierung (Für Anfragen)**: Eingehende Daten (z.B. JSON von einem `POST`-Request) validieren und in Python-Objekte umwandeln.

- **`ModelSerializer`**: Ähnlich wie `ModelForm` generiert `ModelSerializer` seine Felder und Validierungen automatisch aus einer verknüpften Modellklasse.

  Beispiel für `recipes/serializers.py`:

```
# recipes/serializers.py
from rest_framework import serializers
from .models import Recipe

class RecipeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Recipe
        # Felder, die in der API angezeigt werden sollen
        fields = ['id', 'title', 'description', 'author', 'created_at']
```

5. `ModelViewSet` & `Router`: Die Magie für schnelle APIs

- **`ViewSet`**: Ein `ViewSet` ist eine Klasse, die die Logik für eine Reihe von zusammengehörigen Views bündelt. Statt separater Views für `recipe_list`, `recipe_detail` etc. zu schreiben, fasst ein `ViewSet` alles zusammen.
- **`ModelViewSet`**: Diese spezielle Klasse bietet eine **vollständige CRUD-Implementierung** für ein Modell mit den Aktionen `.list()`, `.retrieve()`, `.create()`, `.update()`, `.partial_update()` und `.destroy()`.
- **`Router`**: Ein Router nimmt ein `ViewSet` und generiert automatisch alle notwendigen URL-Patterns dafür.

Beispiel für `recipes/views.py`:

```
# recipes/views.py
from rest_framework import viewsets
from .models import Recipe
from .serializers import RecipeSerializer

class RecipeViewSet(viewsets.ModelViewSet):
    # Die Query, die alle Objekte zurückgibt
    queryset = Recipe.objects.all()
    # Die Serializer-Klasse, die verwendet werden soll
    serializer_class = RecipeSerializer
```

Beispiel für `recipes/urls.py`:

```python
# recipes/urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import RecipeViewSet

# Einen Router erstellen
router = DefaultRouter()
# Das RecipeViewSet unter dem Pfad 'recipes' registrieren
router.register(r'recipes', RecipeViewSet, basename='recipe')

# Die API-URLs werden automatisch vom Router generiert
# Normalerweise bindet man diese in die Haupt-urls.py unter einem /api/ Präfix ein
urlpatterns = [
    path('', include(router.urls)),
]
```

Dieser kurze Code generiert automatisch die folgenden Endpunkte:

- `GET /recipes/`: Liste aller Rezepte.
- `POST /recipes/`: Neues Rezept erstellen.
- `GET /recipes/{id}/`: Ein einzelnes Rezept abrufen.
- `PUT /recipes/{id}/`: Ein Rezept vollständig aktualisieren.
- `PATCH /recipes/{id}/`: Ein Rezept teilweise aktualisieren.
- `DELETE /recipes/{id}/`: Ein Rezept löschen.

6. Die Browsable API: Dein Test-Cockpit

Wenn man im Entwicklungsmodus eine der neuen API-URLs im Browser aufruft (z.B. `http://127.0.0.1:8000/api/recipes/`), präsentiert DRF eine benutzerfreundliche HTML-Oberfläche. Mit dieser **Browsable API** kann man:

- Die API-Antworten direkt ansehen.
- Neue Objekte über Formulare erstellen (`POST`).
- Bestehende Objekte direkt auf ihrer Detailseite bearbeiten (`PUT`) oder löschen (`DELETE`).

Dies ist ein extrem nützliches Werkzeug zum Entwickeln und Testen der API.

Fazit

- **API als Entkopplung**: Die Umstellung auf eine API trennt Backend-Logik von der Frontend-Präsentation und schafft Flexibilität.
- **DRF als Werkzeug**: Das Django REST Framework bietet mächtige Werkzeuge, um diesen Prozess massiv zu beschleunigen.
- **Serializer**: Sind die unverzichtbaren Übersetzer zwischen Python-Objekten und JSON.
- **`ModelViewSet` & `Router`**: Reduzieren den Code für Standard-CRUD-Operationen auf ein absolutes Minimum und sind der schnellste Weg zu einer funktionierenden API.
- **Browsable API**: Ein unschätzbares Feature für die Entwicklung und das Debugging.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) werden nun API-Endpunkte für die Ressourcen `Question` und `Choice` erstellt.

1. **`rest_framework` installieren** und in `settings.py` hinzufügen.
2. **`polls/serializers.py` erstellen**:

```python
from rest_framework import serializers
from .models import Question, Choice

class QuestionSerializer(serializers.ModelSerializer):
    class Meta:
        model = Question
        fields = ['id', 'question_text', 'pub_date']

class ChoiceSerializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = Choice
        fields = ['id', 'question', 'choice_text', 'votes']
```

3. `polls/views.py` **anpassen**:

```
from rest_framework import viewsets
from .models import Question, Choice
from .serializers import QuestionSerializer, ChoiceSerializer

class QuestionViewSet(viewsets.ModelViewSet):
    queryset = Question.objects.all()
    serializer_class = QuestionSerializer

class ChoiceViewSet(viewsets.ModelViewSet):
    queryset = Choice.objects.all()
    serializer_class = ChoiceSerializer
```

4. **Neue** `polls/api_urls.py` **erstellen** (oder bestehende `urls.py` anpassen):

```
from rest_framework.routers import DefaultRouter
from .views import QuestionViewSet, ChoiceViewSet

router = DefaultRouter()
router.register(r'questions', QuestionViewSet)
router.register(r'choices', ChoiceViewSet)

urlpatterns = router.urls
```

5. **Haupt-**`urls.py` **anpassen**, um die API-URLs einzubinden:

```
# projekt/urls.py
urlpatterns = [
    # ...
    path('api/polls/', include('polls.api_urls')),
]
```

Cheat Sheet

- **DRF Installation**:

```
pip install djangorestframework
```

- **DRF Konfiguration (**`settings.py`**)**: `INSTALLED_APPS = [..., 'rest_framework', ...]`
- **ModelSerializer (**`serializers.py`**)**:

```
from rest_framework import serializers
from .models import MyModel

class MyModelSerializer(serializers.ModelSerializer):
    class Meta:
        model = MyModel
        fields = '__all__' # oder ['id', 'feld1', ...]
```

- **ModelViewSet (`views.py`)**:

```
from rest_framework import viewsets
from .models import MyModel
from .serializers import MyModelSerializer

class MyModelViewSet(viewsets.ModelViewSet):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer
```

- **Router (`urls.py`)**:

```
from rest_framework.routers import DefaultRouter
from .views import MyModelViewSet

router = DefaultRouter()
router.register(r'my-model-url', MyModelViewSet)
# in urlpatterns: path('api/', include(router.urls))
```

Übungsaufgaben

1. **Neues DRF-Projekt erstellen**:
   - Ein neues, einfaches Django-Projekt erstellen.
   - Eine App `tasks` erstellen.
   - Ein Modell `Task` mit den Feldern `title` (`CharField`) und `completed` (`BooleanField` mit `default=False`) erstellen und migrieren.
2. **API erstellen**:
   - DRF installieren und konfigurieren.
   - Einen `TaskSerializer` für das `Task`-Modell erstellen.
   - Ein `TaskViewSet` für das `Task`-Modell erstellen.
   - Einen Router einrichten, der das `TaskViewSet` unter dem Pfad `/api/tasks/` verfügbar macht.
3. **API testen**:
   - Den Entwicklungsserver starten.
   - Die Browsable API unter `http://127.0.0.1:8000/api/tasks/` aufrufen.
   - Über die Browsable API mehrere neue Aufgaben erstellen (`POST`), eine davon als "erledigt" markieren (`PUT`/`PATCH`) und eine wieder löschen (`DELETE`).

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die bestehende "Community Recipe Sharing Platform" wird nun um eine REST API erweitert. Die Template-basierten Views bleiben vorerst erhalten, wir fügen die API parallel hinzu.

Aufgabe:

1. **DRF installieren & konfigurieren**: `djangorestframework` in der virtuellen Umgebung installieren und in `settings.py` zu den `INSTALLED_APPS` hinzufügen.
2. **Serializer erstellen**:
   - In der `recipes`-App eine neue Datei `serializers.py` erstellen.
   - Darin `ModelSerializer`-Klassen für die Modelle `Recipe`, `Ingredient` und `Step` definieren. Wähle sinnvolle Felder aus, die in der API sichtbar sein sollen.
3. **ViewSets erstellen**:
   - In `recipes/views.py` `ModelViewSet`-Klassen für `Recipe`, `Ingredient` und `Step` erstellen. Jedes ViewSet sollte das entsprechende `queryset` und die `serializer_class` definieren.
4. **API-URLs einrichten**:
   - Erstelle eine neue Datei `recipes/api_urls.py` (oder erweitere die bestehende `urls.py`).
   - Darin einen `DefaultRouter` instanziieren.

    - Registriere alle drei ViewSets beim Router (z.B. unter den Pfaden `recipes`, `ingredients`, `steps`).
    - Binde diese `api_urls.py` in der Haupt-`urls.py` deines Projekts unter einem globalen Präfix ein, z.B. `path('api/', include('recipes.api_urls'))`.
5. **Testen**:
    - Den Entwicklungsserver starten.
    - Die neuen API-Endpunkte im Browser aufrufen (z.B. `http://127.0.0.1:8000/api/recipes/`).
    - Die Funktionalität mit der Browsable API testen:
        - Können Rezepte angezeigt, erstellt, bearbeitet und gelöscht werden?
        - Funktioniert dies auch für Zutaten und Schritte?
        - Prüfe, ob die Daten, die du über die API erstellst, auch im Django Admin-Bereich korrekt erscheinen.