

Checkliste Django/DRF-Projekte

Bitte erfülle alle Punkte auf dieser Liste, bevor du das Projekt einreichst. (**Definition of Done** - DoD)

1. Allgemeines

Endpoints

- Alle Endpoints sind nach Dokumentation erstellt
- Das Projekt muss durch unsere PM-Tests bei Abgabe eine Test-Coverage von mind. 95% erreichen.

Clean Code/Dokumentation

- Eine Funktion/Methode hat max. eine Aufgabe und maximal 14 Zeilen
- Kein auskommentierter Code oder `print()` Befehle verbleiben im Projekt
- Code ist [PEP8](#) konform.
- Der Code ist Dokumentiert/Kommentiert

GitHub Repository

- Es existiert eine aussagekräftige README.MD, die mindestens alles beinhaltet zum starten des Projektes! Sämtliche Besonderheiten sind hier aufzuführen!
- Die README.MD sollte zwingend auf Englisch verfasst sein.
- Das Backend ist in einem eigenen Repository hochgeladen ohne Frontend
- Es existiert eine vollständige requirements.txt
- Die Datenbank sollte niemals auf Github geladen werden!

2. Conventions

Projekt- & App-Struktur

- Das Projekt wird beim Starten `core` genannt.
(Dadurch heißt der Ordner mit `settings.py`, `urls.py`, `wsgi.py` usw. `core` – klar von den Apps unterscheidbar.)

- Alle Apps erhalten ein sprechendes Präfix oder Suffix, z. B. `auth_app`, `kanban_app`
- Jede App enthält zusätzlich einen `api/`-Ordner indem sich die `serializers.py`, `views.py`, `urls.py`, `permissions.py` usw. befinden.
- Die Admin Umgebung soll nutzbar sein.

Models

- sprechende Klassennamen im `PascalCase`, z. B. `UserProfile`
- Felder im `snake_case`, z. B. `first_name`, `is_active`
- Verwende für eine sinnvolle Darstellung die `__str__` Methode und ggf. in den Meta Optionen `verbose_name`, `verbose_name_plural`, `ordering`
- keine Logik in Modellen
- Definiere Model-Beziehungen sauber mit `related_name` und `on_delete` bspw:
`user = models.ForeignKey(User, on_delete=models.CASCADE, related_name="projects")`

Serializer

- Nutze `ModelSerializer` für CRUD-Serialisierungen.
- Gib Felder explizit an bspw. `fields = ["id", "title"]` und nicht mit `__all__`
- Benenne Felder in der gewünschten Reihenfolge.
- Wenn eine extra Validierung eines Feldes nötig ist, dann nutze bspw.
`def validate_title(self, value): ...`
oder bei Zusammenhängen die `def validate(self, attrs): ...`

Views

- Verwende `ModelViewSet` für CRUD, `APIView` für individuelle Endpunkte als auch `GenericAPIView`
- `queryset` und `serializer_class` gehören als Properties in die Klasse
- `get_queryset()` für dynamische Querysets verwenden, z. B. User-spezifisch
- Permissions klar deklarieren mit `permission_classes = [...]`

URLs

- API-Routen sind ressourcenorientiert, nicht aktionsbasiert:
`/api/boards/42/` statt `/api/getProjectById/`
- Jede App hat ihre eigene URL-Datei
- Hauptprojekt (core) hat zentrales Routing in dem alle urls included werden

Permissions & Auth

- Jede App hat ihre eigene `permissions.py`, sofern nötig
- Kombiniere Permissions logisch (`IsAuthenticated` & `IsOwner`)
- Keine offenen Endpunkte ohne expliziten Grund/Vorgabe

3. Best Practices

Best Practices für Imports

Importe gruppieren und sortieren

Bsp.:

1. Standardbibliothek

`import os`

`from datetime import datetime`

2. Drittanbieter (Third-party)

`from django.db import models`

`from rest_framework import serializers`

3. Lokale Importe (eigene Module)

`from .models import Project`

`from .services.project_logic import create_project_with_tasks`

Klares Verantwortlichkeitsprinzip

Als Nutzer möchte ich mich registrieren können, damit ich ein persönliches Konto erstellen kann.

Models: Datenstruktur

Serializers: Validierung & Transformation

Views: API-Logik & Routing

Permissions: Zugriffskontrolle

HTTP-Statuscodes korrekt verwenden

DRF erledigt das oft automatisch – überschreibe nicht unnötig das Verhalten.
Dennoch ist dies zu beachten/testen. (siehe hier: [link](#))

Bspw.:

Zweck	Statuscode
Objekt erfolgreich erstellt	201 CREATED
Kein Inhalt zurückgegeben	204 NO CONTENT
Validierungsfehler	400 BAD REQUEST
Berechtigung fehlt	403 FORBIDDEN
Objekt nicht gefunden	404 NOT FOUND