PYTHON ACADEMY

SERIALIZADORES (SERIALIZERS) NO DJANGO REST FRAMEWORK (DRF)

Aprenda a configurar e utilizar Serializadores (Serializers) no Django Rest Framework para criar APIs REST.

PYTHONACADEMY.COM.BR

Salve salve Pythonista!

Neste artigo, vamos explorar **Serializadores no Django Rest Framework (DRF)**.

Você aprenderá sobre os conceitos de serialização e deserialização, como utilizar serializadores (Serializers) no DRF e como desenvolvê-los em uma aplicação prática.

Entender os serializadores é crucial para criar APIs eficientes e robustas em Django, permitindo converter entre tipos de dados complexos e formatos nativos do Python.

Conceitos de Serialização e Deserialização

Em termos simples, **serialização** é o processo de converter instâncias complexas, como `QuerySets` do Django, em formatos de dados que podem ser facilmente renderizados, como JSON, XML ou outros.

Enquanto isso, **deserialização** é o processo inverso, de conversão de dados primitivos de entrada (como JSON e XML) em tipos de dados complexos (Models do Django, por exemplo).

> *Por que isso é importante?*

Em uma API, é essencial comunicar-se com diferentes sistemas, converter dados complexos em um formato simples é fundamental para transferência e armazenamento de informações.

Utilizando Serializadores no Django Rest Framework

Vamos criar uma aplicação simples para ilustrar como utilizar serializadores no Django Rest Framework.

Para ver todos os detalhes de como configurar um projeto utilizando Django REST Framework, acesse o artigo "Configurando um projeto Django Rest Framework" clicando aqui

Criando um Projeto Django

Primeiro, vamos criar um simples projeto Django:

```
django-admin startproject exemplo_serializers
cd exemplo_serializers
python manage.py startapp core
```

Configurando a App Django no `settings.py`

Adicione a App Django criada à variável `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'core',
]
```

Criando os Modelos (`models.py`)

Vamos criar um Model simples que modela um Livro em nosso sistema:

```python
from django.db import models

class Livro(models.Model):
    titulo = models.CharField(max_length=100)
    autor = models.CharField(max_length=100)
    publicado_em = models.DateField()

    def __str__(self):
        return self.titulo
```

Aqui definimos um Model Livro com: - `titulo`: campo de texto com máximo de 100 caracteres - `autor`: campo de texto com máximo de 100 caracteres - `publicado_em`: campo de data, para armazenar a data da postagem

Serializadores (`serializers.py`)

Agora, vamos criar a Classe responsável por serializar e deserializar objetos do tipo `Livro` em nossa API:

```python
from rest_framework import serializers
from .models import Livro

class LivroSerializer(serializers.ModelSerializer):
    class Meta:
        model = Livro
        fields = ['id', 'titulo', 'autor', 'publicado_em']
```

Agora vamos à explicação: - `LivroSerializer` herda de `serializers.ModelSerializer`, que é uma maneira simplificada de criar um serializador para modelos Django. - `class Meta` define o modelo de origem e os campos que serão serializados.

Com isso, podemos então testar localmente as funcionalidades da nossa API criando e serializando dados.

> 💡 *Estou construindo o **DevBook**, uma plataforma que usa IA para criar ebooks técnicos — com código formatado e exportação em PDF. Não deixe de conferir!*

Criando dados e Serializando

Para testar localmente o que desenvolvemos, abra o shell do Django executando o comando `python manage shell` na raíz do projeto (com seu ambiente virtual ativado) e digite o seguinte código:

```
from core.models import Livro
from core.serializers import LivroSerializer

livro = Livro(titulo="Aprendendo Python", autor="Autor Desconhecido",
        publicado_em="2023-01-01")
livro.save()

serializer = LivroSerializer(livro)
print(serializer.data)
```

Ao fazer isso, a saída esperado será:

```
{'id': 1, 'titulo': 'Aprendendo Python', 'autor': 'Autor Desconhecido',
'publicado_em': '2023-01-01'}
```

Explicação: - Primeiro, criamos uma instância de `Livro` e a salvamos no banco de dados com o método `save()`. - Em seguida, utilizamos o `LivroSerializer` para serializar o objeto `livro`. - Por fim, printamos o resultado da serialização presente no campo `serializer.data`

Deserializando Dados

No exemplo abaixo, veremos como podemos transformar dados de Json para um `dict` Python e vice-versa (novamente, execute os códigos no `shell` do Django):

```python
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
import io

# Serializando para JSON
json_livro = JSONRenderer().render(serializer.data)
print(json_livro)

# Deserializando de JSON
stream = io.BytesIO(json_livro)
data = JSONParser().parse(stream)
serializer = LivroSerializer(data=data)
if serializer.is_valid():
    livro = serializer.save()
print(livro)
```

A saída do código acima será:

```
b'{"id":1,"titulo":"Aprendendo Python","autor":"Autor Desconheci-
do","publicado_em":"2023-01-01"}'
Livro(id=1, titulo='Aprendendo Python', autor='Autor Desconhecido', pu-
blicado_em='2023-01-01')
```

Explicação: - **Serialização para JSON:** `JSONRende-rer().render(serializer.data)` converte os dados serializados no formato JSON. - **Deserialização de JSON:** Utilizamos `JSONParser()` para converter o JSON de volta para um formato que o Django entende, e `LivroSerializer` para criar e validar uma instância `Livro`.

Boas Práticas

Agora que você já entende como converter de tipos simples para complexos e vice-versa, vamos ver algumas boas práticas no desenvolvimento de APIs utilizando Django REST Framework.

Validações

Sempre que possível, utilize **validações personalizadas** em seus serializadores para garantir a integridade dos dados.

```
class LivroSerializer(serializers.ModelSerializer):
    class Meta:
        model = Livro
        fields = ['id', 'titulo', 'autor', 'publicado_em']

    def validate_titulo(self, value):
        if 'Python' not in value:
            raise serializers.ValidationError("O título deve conter a
        palavra 'Python'.")
        return value
```

No código acima, `validate_titulo` é um método de validação que garante que a palavra 'Python' esteja presente no título do livro.

Esse é um comportamento bastante peculiar do DRF: basta você criar um método no formato `validate_<nome do campo>` que - automagicamente - o DRF irá executar essa validação quando estiver serializando e desserializando dados!

> *Incrível!* 😍

Otimização

Utilize **serializadores parciais** para atualização de dados específicos:

```
livro = Livro.objects.get(id=1)
serializer = LivroSerializer(livro, data={'titulo': 'Python Avançado'},
        partial=True)
if serializer.is_valid():
    livro = serializer.save()
```

No código acima, `partial=True` permite atualizar apenas campos específicos sem a necessidade de enviar todos os dados novamente.

Conclusão

Neste artigo, exploramos o uso de **Serializadores no Django Rest Framework**, aprendemos sobre **serialização e deserialização**, e como implementá-los em uma aplicação Django.

Vimos exemplos práticos e dicas de boas práticas, como validações personalizadas e serializações parciais.

Fique ligado nos próximos pois ainda abordaremos muito conteúdo fera sobre Django REST Framework!

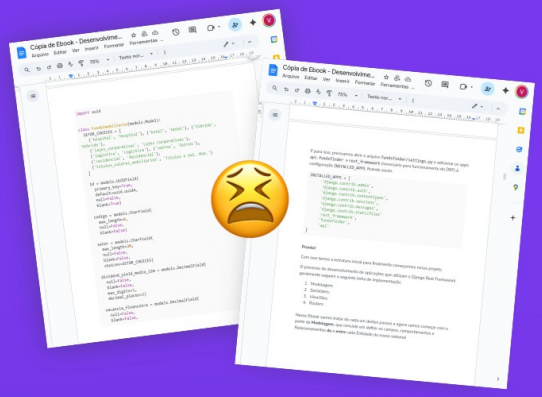
Te vejo lá!

Não se esqueça de conferir!

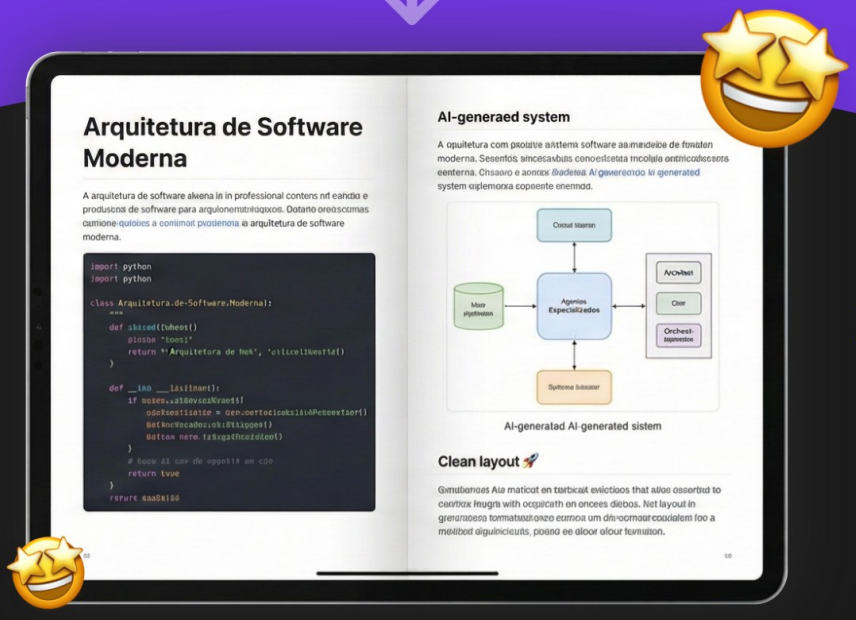
DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código**!



Chega de formatar código no Google Docs



Syntax Highlight

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS