Raíssa Azevedo

API com Python e Django REST Framework - Básico

O **Django REST Framework** utiliza toda a facilidade do framework web **Django** juntamente com todo o poder da linguagem **Python** fazendo disso uma combinação perfeita para criação de **APIs** modernas.

Desta forma a comunicação entre sua aplicação web e sua aplicação mobile ficará perfeita.

- O que são APIs?

- O que é REST?

- O que é o Django REST Framework?

- Criar APIs para os métodos HTTP GET, POST, PUT e DELETE;

- Utilizar permissionamento;

- Utilizar limitação de requests;

- Utilizar paginação de dados;

- Utilizar autenticação via Token;

- Testar as APIs;

API's REST:

Muitas APIs encontradas na internet utilizam do conceito de **REST** (Representational State Transfer), ou Transferência Representacional de Estado.
Trata-se da criação de uma interface de comunicação utilizando puramente HTTP.

API – Application Programming Interface
É uma interface de comunicação de aplicações de forma programática. Ou seja, uma interface é criada para que diferentes aplicações se comuniquem de forma simples e eficiente. São criadas através de padrões de design chamado **RESTful** que são conhecidas como **API Rest**.

REST – Representational State Transfer
O protocolo HTTP é por onde a internet "roda" é por design, sem estado. Isso significa que toda requisição feita a um servidos é única pos estas requisições não guardam dados (estados) entre uma requisição e outra.

Entendendo os EndPoints:
Para a criação de endpoinst é preciso utilizar os conceitos de gramática de substantivo e verbo.
Substantivos:
Numa API Rest há o conceito de **resources** (recursos). Que pode ser um **model** dentro de uma aplicação. São através de resources que utiliza-se as operações de CRUD (Create, Retrieve, update, Delete), que é feito através de **URI** específicas dentro da aplicação.
http://www.meusistema.com.br/**api/v1/produtos** (URI). As URI são os endpoints.
Podem representar uma **coleção de registos** (/produtos) ou um **registro individual** (/produtos/42).

O Design correto de uma API faz toda diferença na utilização da API.

Verbos:

Indicam uma ação. Ou seja, coisas que estamos/queremos fazer. Em API's RESTful, é feito o uso de verbos HTTP para indicar a ação que queremos realizar com nosso recurso.

GET – É usado tanto para acessar uma coleção de recursos quanto um recurso individual.

HTTP GET - /api/v1/produtos/42

POST – É utilizado apara adicionar um novo recurso na **coleção.**

HTTP POST - /api/v1/produtos

PUT – É utilizado para atualizar um recurso existente na coleção (individual).

HTTP PUT - /api/v1/produtos/42

DELETE – É utilizado para excluir um recurso ecistente na coleção.

HTTP DELETE - /api/v1/produtos/42

Entendento as Requests:

Trata-se de um dispositivo que faz uma requisição a um servidor.

As requisições (requests) possuem muito mais informações do que um simples verbo HTTP e uma URI para qual foi feita a requisição.

É possível mudar o aspecto da requisição ára qie seja ´pssível alteraro formato das respostas que serão enviadas pelo servidor HTTP.

/api/v1/produtos**?order=desc&limit=10**

Tudo que está após o simbolo de interrogação (“?”) são conjuntos de pares chave/valor que podem ser utilizados pela API para alterar os dados de acordo com esses parâmetros. Esta forma de passar os dados em uma requisição é chamada de **query string**.

Cabeçalho da Request – Accept

Específica o formato do arquivo que o requester (requisitante) quer.

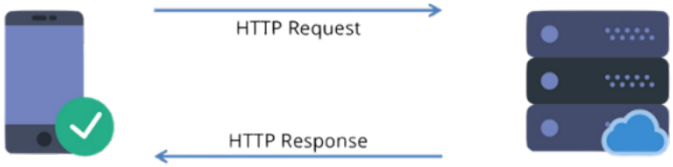
Accept: application/xml ou json ou pdf…

Há também o Accept language que especifica qual lingua o conteúdo será enviado.

Entendento as Responses:

Existe um série de requisições verificadas na request, essas são:

- Na requisição existe query string?
- Qual foi o verbo HTTP que realizou a ação?
- Quais são os dados do cabeçalho?
- Qual o formato requisitado?
- E claro, o recurso requisitado é individual ou uma coleção?



O response tem que bater com tudo solicitado na request.

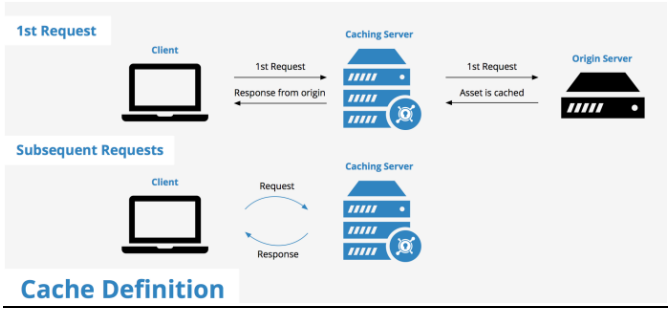
O Código HTTP possui vários níveis de erro de status:

O código HTTP de 200 a 299 indica que tudo está ok.

O código HTTP de 300 a 399 indica que a requisição foi entendida pelo servidor, porém o recurso está em outro local.
O código HTTP de 400 a 499 indica que a requisição foi realizada com algum erro, do lado do cliente (solicitante)
O códifo HTTP de 500 a 599 indica que a requisição foi realizada mas houve algum erro do lado do servidor.

Entendendo sobre a segurança da API Rest:
Uma API que não consegue suprir a demanda é tão ruim quanto não ter nenhuma.

O 1° passo para manter a API disponível é fazer o uso de cache. Fazendo o uso do mesmo os dados vem direto do servidor que processou e enviou a resposra. Um segundo cliente faz uma nova requisição mas agora os dados podem ser pegos direto do cache (mais rápido e eficiente).



Ferramentas como **Redis** ou **Memcache** podem ser utilizados pra isso.
Para evitar queda do servidor por N° maior de requsições é preciso adicionar um numero limite para acessar os serviços (é feito com contrato de número de requisições por mês).

Outro passo importante para acesso é garantir a **autenticação** e a **autorização** de acesso. A forma mais comum de autenticação é feita através do uso de **Token.**

O que é Django Rest Framework?
É uma ferramenta (biblioteca) que após instalada e configurada, é executada no topo de um projeto Django.

É possível fazer uma API dem utilizar o **DRF**, entretanto, por se tratar d euma feramenta especializada nessa tarefa, o uso é recomendado por ser feito de maneira mais prática, segura e rápida.

O que é o DRF?
É uma ferramenta que provê o **Model Serialization**, ou seja, o DRF mapeia os Django Models e facilita na serialização/deserialização para JSON, que é a base das API's.

Instalação e Configuração do Django Rest Framework:
Antes de tudo é necessário a criação de um projeto Django.

```
pip3 install django==2.2.9  ou 3.2.12
```

Essa versão foi escolhida por ser LTS, porque é Long Term Support. Ou seja, vai continuar tendo atualizações por mais tempo.

```
django-admin startproject escola .
django-admin startapp cursos
```

Em seguida é importante fazer as configurações necessárias no **settings.py**

```
INSTALLED_APPS = [
        "cursos",
]
LANGUAGE_CODE = "pt-br"
TIME_ZONE = "America/Sao_Paulo"
STATIC_ROOT= os.path.join(BASE_DIR, 'staticfiles')
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path,join(BASE_DIR, 'media')
```

Em seguida a configuração é realizada no arquivo **models.py**

```
from django.db import models

class Base (models.Model):
        criacao = models.DateTimeField (auto_now_add= True)
        atualizacao = models.DateTimeField(auto_now= True)
        ativo = models.BooleanField(default=True_

        class Meta:
                abstract = True
```

Ou seja, quando for criado irá pegar a data automática do sistema, e sempre que for atualizado também irá pegar a data automática do sistema.

```
class Curso(Base):
        titulo = models.Charfield(max_lenght= 200)
        url = models.URLField(unique= True)

        class Meta:
        verbose_name = “Curso”
        verbose_name_plural = “Cursos”

        def __str__(self):
                return self.titulo

class Avaliacao(Base):
        curso = models.ForeignKey(Curso, related_name='avaliacoes', on_delete=models.CASCADE)
            nome = models.CharField(max_length=255)
            email = models.EmailField()
            comentario = models.TextField(blank=True, default='')
            avaliacao = models.DecimalField(max_digits=2, decimal_places=1)

            class Meta:
                verbose_name = 'Avaliação'
                verbose_name_plural = 'Avaliações'
                unique_together = ['email', 'curso']

            def _str_(self):
                return f'{self.nome} avaliou o curso {self.curso} com nota = {self.avaliacao}'
```

Dando sequência no mesmo arquivo... As configurações seguem o titulo e uma URL que é definida como única.

O próximo passo é registrar o model criado no **admin.py**

```
from django.contrib import admin
from .models import Curso, Avaliacao

@admin.register(Curso)
class CursoAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'url', 'criacao', 'atualizacao', 'ativo')

@admin.register(Avaliacao)
class AvaliacaoAdmin(admin.ModelAdmin):
    list_display = ('curso', 'nome', 'email', 'avaliacao', 'criacao', 'avaliacao', 'ativo')
```

Depois dos models registrados no admin. Por via do terminal, executar as migrations:

python3 manage.py makemigrations

python3 manage.py migrate

Para a manipulação dos dados, criar um super usuário.

python3 manage.py createsuperuser

Para testar as funcionalidades, executar o servidor.

python3 manage.py runserver

Realizar o cadastro de alguns personagens para tratar os dados na API

Após o cadastro a aplicação Django está pronta... Agora é feita de fato a criação da API

Criando a API com Django Rest Framework

```
pip3 install djangorestframework markdown django-filter
```

O **markdown** é utilizado para fazer páginas de documentação. O django-filter é utilizado para facilitar a utilização de filtros nos projetos.

pip3 freeze > requirements.txt

Vai criar um documento txt mapeando todas as bibliotecas utilizadas no projeto.

```
INSTALLED_APPS = [
        "cursos",
        "django_filters",
        "rest_framework"
]
```

No final do mesmo arquivo de **settings.py**

```
# DRF
REST_FRAMEWORK = {
        'DEFAULT_AUTHENTICATION_CLASSES': (
                'rest_framework.authentication.SessionAuthentication',
        )
        'DEFAULT_PERMISSION_CLASSES': (
                'rest_framwork.permissions.IsAuthenticatedOrReadOnly',
        )
}
```

Isso significa que se o cliente logou na API, ele pode fazer todas a manutenções (criar, deletar, update ou consltar dados). Caso ele não esteja autenticado (anônimo) ele pode fazer apenas a leitura dos dados.

No arquivo **url.py**

```
from django.urls import path, include

urlpatterns = [
        path('admin/', admin.site.urls),
        path('auth/', include('rest_framework.urls')),
]
```

python3 manage.py runserver

/auth/login

Após fazer o login vai cair na página **/account/profile/** que é o redirect padrão do django.

OBS: Instalar o app **insomnia rest**

Usando o Model Serializers

É um recurso do DRF capaz de transformar os models em estruturas JSON e também o contrario. Ou seja, ele faz a transformação de objetos python.

Isso se dá porque o JSON é o formato ideal para troca de dados entre APIs na internet.

Criar um novo arquivo python denominado **serializers**

```
from rest_framework import serializers
from .models import Curso, Avaliacao

class AvaliacaoSerializer(serializers.ModelSerializer):
    class Meta:
        extra_kwargs = {
            'email': {'write_only': True}
        }
        model = Avaliacaofields = (
            'id',
            'curso',
            'nome',
            'email',
            'comentario',
            'avaliacao',
            'criacao',
            'ativo'
        )
```

A classe é criada com um nomeSerializer que herda serializers.modelSerializer. Omde são feitas as configurações do model. Que recebe o nome do model importado. Nesse caso é **Avaliacao**.

Dentro dos campo **fields** são indicados tudo que é necessário apresentar, puxando o atributo do model indicado.

```
extra_kargs = {
        'email': {'write_only': True}
    }
```

Esse parâmetro extra adicionado impede que os emails sejam coletados para envios de spam. São feitos visando a privacidade do usuário da API.

O **WriteOnly** significa que o e-mail será exigido apenas quando for feito o cadastro, ou seja, na hora de consultar o e-mail ficará oculto.

```
Class CursoSerializer9serializers.ModelSerializer):
    Class Meta:
        model = Curso
        fields = (
        'id',
        'titulo',
        'url',
        'criacao',
        'ativo'
    )
```

Como não há nenhuma informação que prejudique o usuário não há necessidade de criar um parâmetro kwarg extra.

Observando o funcionamento dos serializers:

python3 manage.py shell

```
>>> from rest_framework.renderers import JSONRenderer
>>> from cursos.models import Curso
>>> from cursos.serializers import CursoSerializer
>>> curso = Curso.objects.latest('id')
>>> curso
<Curso: Curso número 3>
>>> curso.titulo
'Curso número 3'
>>> serializer = CursoSerializer(curso)
>>> serializer
CursoSerializer(<Curso: Curso número 3>):
    id = IntegerField(label='ID', read_only=True)
    titulo = CharField(max_length=255)
    url = URLField(max_length=200, validators=[<UniqueValidator(queryset=Curso.objects.all())>])
    criacao = DateTimeField(read_only=True)
    ativo = BooleanField(required=False)
>>> type(serializer)
<class 'cursos.serializers.CursoSerializer'>
>>> serializer.data
{'id': 3, 'titulo': 'Curso número 3', 'url': 'http://www.test3.com.br', 'criacao': '2022-03-28T13:59:12.699257-03:00', 'ativo': True}
>>>
```

O serializer.data retorna um dicionário Python com todas as informações registradas na criação do curso.

Para converter esse formato de dicionário python para JSON é utilizado o **JsonRenderer.**

```
>>> serializer.data
{'id': 3, 'titulo': 'Curso número 3', 'url': 'http://www.test3.com.br', 'criacao': '2022-03-28T13:59:12.699257-03:00', 'ativo': True}
>>> JSONRenderer().render(serializer.data)
b'{"id":3,"titulo":"Curso n\xc3\xbamero 3","url":"http://www.test3.com.br","criacao":"2022-03-28T13:59:12.699257-03:00","ativo":true}'
>>>
```

Esse **b** indica que é uma string binária que é muito mais rápida de otimizar na transformação do JSON.

Criando APIViews para o método HTTP GET:

A criação do **serializer.py** foi de fato o 1° passo para a construção da API.

No arquivo **views.py**

```python
from rest_framework.views import APIView
from rest_framework.response import Response

from .models import Curso, Avaliacao
from serializers import CursoSerializer, AvaliacaoSerializer


class CursoAPIView(APIView):
    ''' API DE CURSOS '''
    def get(self, request):
        cursos = Curso.objects.all()
        serializer = CursoSerializer(cursos, many=True)
        return Response(serializer.data)


class AvaliacaoAPIView(APIView):
    '''API DE AVALIAÇÕES'''
    def get(self, request):
        avaliacoes = Avaliacao.objects.all()
        serializer = AvaliacaoSerializer(avaliacoes, many=True)
        return Response(serializer.data)
```

Após a criação das views, é necessária a criação das rotas.

Criar um arquivo **urls.py** na aplicação cursos.

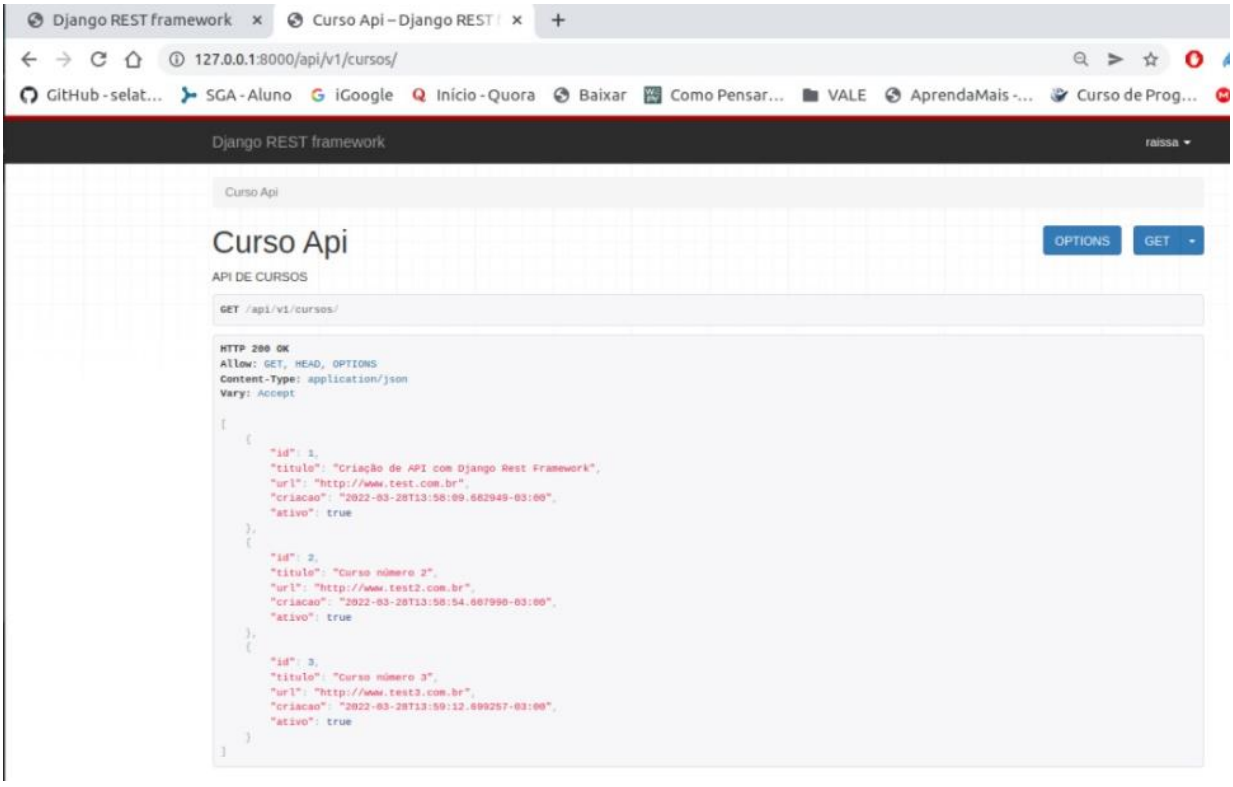
```python
from django.urls import path
from .views import CursoAPIView, AvaliacaoAPIView

urlpatterns = [
    path('cursos/', CursoAPIView.as_view(), name='cursos'),
    path('avaliacoes/', AvaliacaoAPIView.as_view(), name='avaliacoes'),
]
```

Em seguida informar as novas rotas criadas no **urls.py** do projeto.

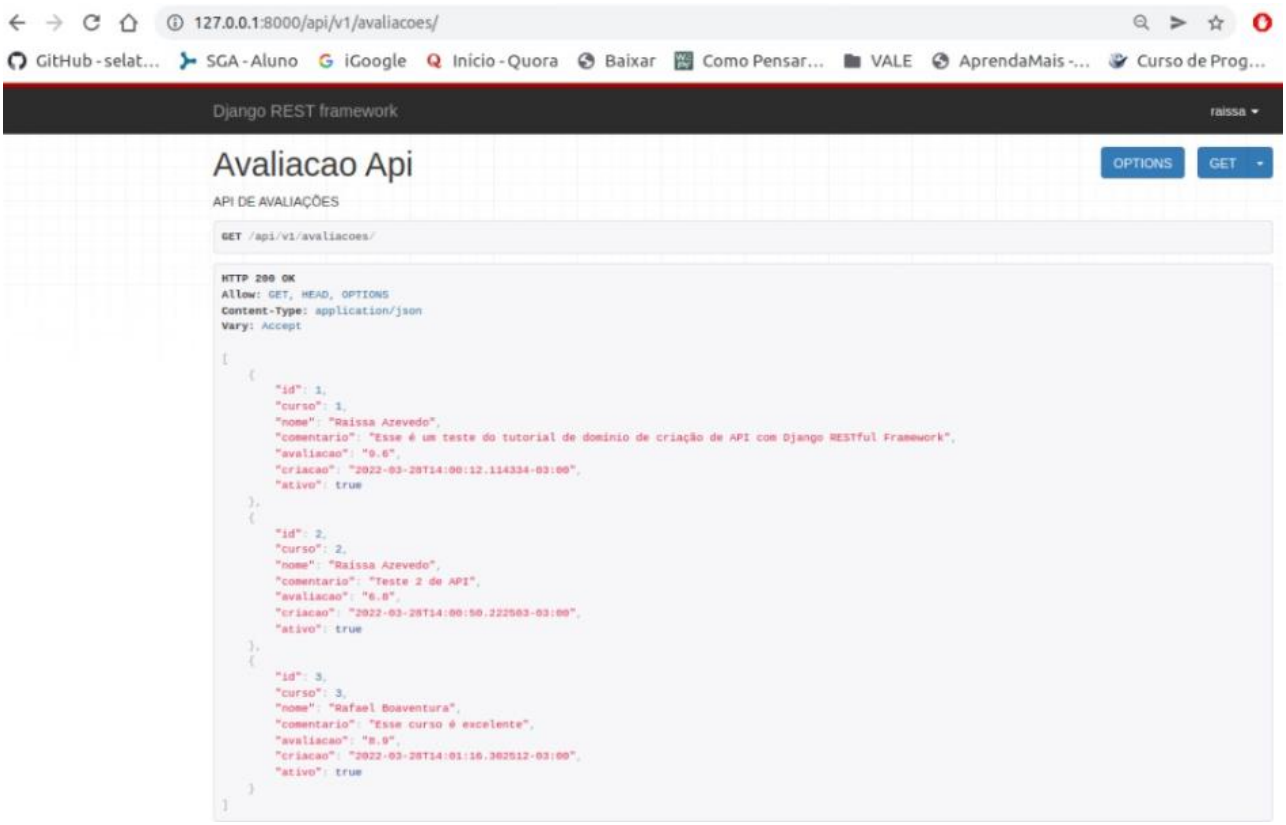
```python
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('api/v1/', include(('cursos.urls'))),
    path('admin/', admin.site.urls),
    path('auth/', include('rest_framework.urls')),
]
```

É importante indicar a versão da api (**v1**) para facilitar nas atualizações e melhorias da API e para mapear quais clientes ainda não atualizaram para a versão nova.

API de Cursos



API de Avaliações



Criando APIViews para o método HTTP POST:

O método post é implementado para permitir criar novos recursos na API

```python
from rest_framework import status

from .models import Curso, Avaliacao
from .serializers import CursoSerializer, AvaliacaoSerializer


class CursoAPIView(APIView):
    ''' API DE CURSOS '''
    def get(self, request):
        cursos = Curso.objects.all()
        serializer = CursoSerializer(cursos, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = CursoSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
```

O método **Post** também recebe um request. Que irá criar os dados, serializar para criar um novo recurso. Irá verificar se os dados são válidos e salvar os dados. E preparar uma resposta enviando os dados salvos e uma resposta HTTP.

OBS. Existe uma forma mais simples de criar a API sem utilizar o tempo inteiro a função **def** para separar, o CRUD.

Isso é feito com a parte de Django RESTful Framework intermediário.