

# API COM PYTHON E DJANGO

Raíssa Azevedo

**PARA**  
Beginners BR

**APRESENTADO POR**  
Raíssa Azevedo

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 servido é única pois estas requisições não guardam dados (estados) entre uma requisição e outra.

### **Entendendo os EndPoints:**

Para a criação de endpoints 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.

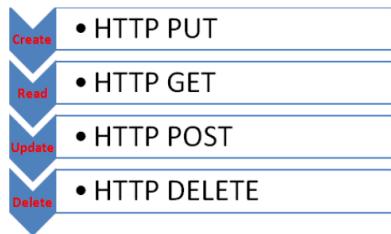
<http://www.meusistema.com.br/api/v1/produtos> (URI). As URI são os endpoints.

Podem representar uma **coleção de registo** (/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 para 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 existente na coleção.

HTTP DELETE - /api/v1/produtos/42

## Entendendo 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 para que seja possível alterar o formato das respostas que serão enviadas pelo servidor HTTP.

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

Tudo que está após o símbolo 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

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

Accept: application/xml ou json ou pdf...

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

### Entendendo 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.

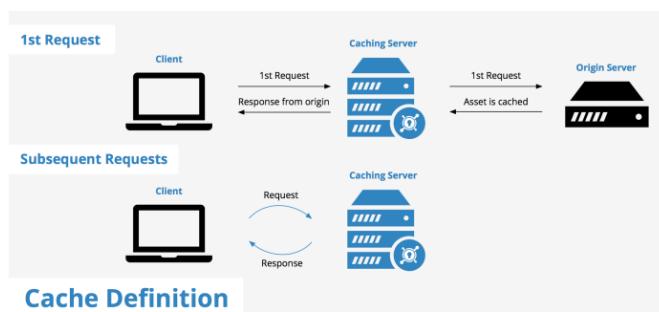
O código HTTP de 400 a 499 indica que a requisição foi realizada com algum erro, do lado do cliente (solicitante)

O código 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 1º passo para manter a API disponível é fazer o uso de cache. Fazendo o uso do mesmo os dados vêm direto do servidor que processou e enviou a resposta. Um segundo cliente faz uma nova requisição mas agora os dados podem ser pegos diretamente 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 requisições 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 sem utilizar o **DRF**, entretanto, por se tratar de uma ferramenta 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_length=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 título 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_framework.permissions.IsAuthenticatedOrReadOnly',
    )
}
```

Isso significa que se o cliente logou na API, ele pode fazer todas a manutenções (criar, deletar, update ou consultar 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 = Avaliacao  
        fields = (  
            'id',  
            'curso',  
            'nome',  
            'email',  
            'comentario',  
            'avaliacao',  
            'criacao',  
            'ativo'  
        )
```

A classe é criada com um nomeSerializer que herda serializers.modelSerializer. Onde 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_kwargs = {  
    '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 CursoSerializer(serializers.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\u00f3mero 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 1º passo para a construção da API.

No arquivo **views.py**

```
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.

```
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'),
]
```

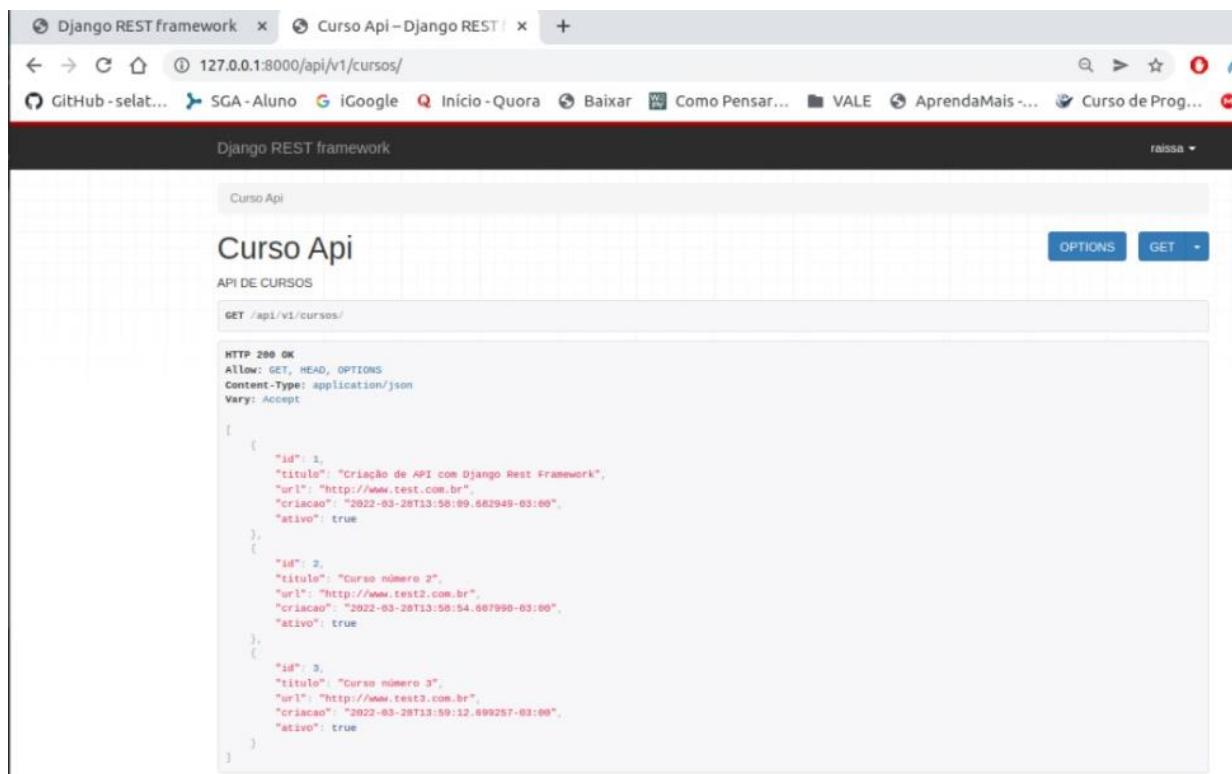
Em seguida informar as novas rotas criadas no **urls.py** do projeto.

```
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



The screenshot shows a browser window with the title "Django REST framework" and a sub-tab "Curso Api - Django REST". The URL in the address bar is "127.0.0.1:8000/api/v1/cursos/". The page content is titled "Curso Api" and "API DE CURSOS". It displays a "GET /api/v1/cursos/" operation with the following response:

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "titulo": "Criação de API com Django Rest Framework",
        "url": "http://www.test.com.br",
        "criacao": "2022-03-28T13:58:09.662949-03:00",
        "ativo": true
    },
    {
        "id": 2,
        "titulo": "Curso número 2",
        "url": "http://www.test2.com.br",
        "criacao": "2022-03-28T13:58:54.667990-03:00",
        "ativo": true
    },
    {
        "id": 3,
        "titulo": "Curso número 3",
        "url": "http://www.test3.com.br",
        "criacao": "2022-03-28T13:59:12.699257-03:00",
        "ativo": true
    }
]
```

At the top right of the API documentation, there are "OPTIONS" and "GET" buttons.

## API de Avaliações

The screenshot shows a browser window with the URL `127.0.0.1:8000/api/v1/avaliacoes/`. The page title is "Avaliacao Api". Below it, there's a section titled "API DE AVALIAÇÕES" with a "GET /api/v1/avaliacoes/" button. The main content area displays a JSON response for a GET request:

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "curso": 1,
        "nome": "Raissa Azevedo",
        "comentario": "Esse é um teste do tutorial de domínio de criação de API com Django RESTful Framework",
        "avaliacao": "9.8",
        "criacao": "2022-03-28T14:00:12.114334-03:00",
        "ativo": true
    },
    {
        "id": 2,
        "curso": 2,
        "nome": "Raissa Azevedo",
        "comentario": "Teste 2 de API",
        "avaliacao": "6.8",
        "criacao": "2022-03-28T14:00:50.222583-03:00",
        "ativo": true
    },
    {
        "id": 3,
        "curso": 3,
        "nome": "Rafael Boaventura",
        "comentario": "Esse curso é excelente",
        "avaliacao": "8.9",
        "criacao": "2022-03-28T14:01:16.382512-03:00",
        "ativo": true
    }
]
```

## Criando APIViews para o método HTTP POST:

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

```
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.**

Raissa Azevedo

# API com Python e Django Rest Framework

## - Intermediário



Como criar de forma mais simples um  
CRUD genérico.

### **Introdução**

- Criando CRUD genérico
- Sobrescrevendo métodos genéricos
- ViewSets e Routers
- Customizando ViewSets
- Relações com Django Rest Framework
- Paginação

Tudo de forma mais simples e clean  
que o apresentado no módulo básico.

## Criando CRUD genérico com Django Rest Framework:

A melhoria de inserção de códigos é feita para evitar a repetição da estrutura no momento de adicionar as funções CRUD.

Para substituir todas as funções criadas no **views.py** utiliza-se:

```
from rest_framework import generics
from .models import Curso, Avaliacao
from .serializers import CursoSerializer, AvaliacaoSerializer

class CursoAPIView(generics.ListCreateAPIView):
    queryset = Curso.objects.all()
    serializer_class = CursoSerializer

class AvaliacaAPIView(generics.ListCreateAPIView):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer
```

Assim como no Django avançado existe o **Class Based View** o Django Rest também conta com melhorias de otimização de código.

O **ListCreate** indicado em genérico indica que é possível listar e criar os métodos, ou seja, as funções GET e POST já estão ativas.

No **queryset** todos os objetos estão sendo passados, através do **objects.all()** e o **serializer\_class** informa qual a classe vai sofrer serialização.

The screenshot shows the Django REST framework's browsable API interface. At the top, there is a navigation bar with the title "Django REST framework" and a dropdown menu "raissa". Below the header, a "POST /api/v1/cursos/" button is visible. The main area displays the response to a previous POST request, showing the following JSON data:

```
HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Location: http://www.testeapi.com.br
Vary: Accept

{
    "id": 6,
    "titulo": "Curso Teste de API",
    "url": "http://www.testeapi.com.br",
    "criacao": "2022-03-31T10:37:50.000Z",
    "ativo": true
}
```

Below the response, there is a form for creating a new course. The form fields are:

- Titulo**: Campo com placeholder "Curso Teste de API".
- Url**: Campo com placeholder "http://www.testeapi.com.br".
- Ativo**: Um checkbox marcado com um checkmark.

At the bottom right of the form, there are two buttons: "Raw data" and "HTML form". To the right of the form, there is a "POST" button.

Agora além de manter o módulo **raw** de registro, também possui o formulário (HTML Form).

## Atualizando e Deletando os dados da API

A única mudança na estrutura do código é a adição de **RetrieveUpdateDestroyAPIView** essa indicação na construção da classe ativa os elementos de CRUD de Update e Delete dos dados de uma API.

```
from rest_framework import generics
from .models import Curso, Avaliacao
from .serializers import CursoSerializer, AvaliacaoSerializer

class CursosAPIView(generics.ListCreateAPIView):
    queryset = Curso.objects.all()
    serializer_class = CursoSerializer

class CursoAPIView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Curso.objects.all()
    serializer_class = CursoSerializer

class AvaliacoesAPIView(generics.ListCreateAPIView):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer

class AvaliacaoAPIView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer
```

**OBS.** O nome das classes de criação e de deleção precisa ser diferente também (sugestão: usar um no plural).

Dessa forma, o arquivo de rotas também precisa ser atualizado.

```
from django.urls import path
from .views import CursoAPIView, CursosAPIView, AvaliacaoAPIView, AvaliacoesAPIView

urlpatterns = [
    path('cursos/', CursosAPIView.as_view(), name='cursos'),
    path('avaliacoes/', AvaliacoesAPIView.as_view(), name='avaliacoes'),
    path('cursos/<int:pk>', CursoAPIView.as_view(), name='curso'),
    path('avaliacoes/<int:pk>', AvaliacaoAPIView.as_view(), name='avaliacao')
]
```

Na rota do elemento CRUD de deleção é adicionado o **primary key**, que é uma ID, através de <int:pk>.

The screenshot shows the Django REST framework's browsable API interface. At the top, it says "Django REST framework" and "raissa ▾". Below that, there's a title "Curso Api" and three buttons: "DELETE", "OPTIONS", and "GET".  
The "GET" button is highlighted. Below it, the URL "GET /api/v1/cursos/6/" is shown. The response is a JSON object:

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 6,
    "titulo": "Curso Teste de API",
    "url": "http://www.testeapi.com.br",
    "criacao": "2022-03-31T10:37:50.9629-03:00",
    "ativo": true
}
```

  
Below the response, there's a form titled "HTML form". It has fields for "Titulo" (with value "Curso Teste de API"), "Url" (with value "http://www.testeapi.com.br"), and "Ativo" (with checked checkbox). At the bottom right of the form is a "PUT" button.

## Sobrescrevendo os métodos genéricos

Trata-se de formas de sobrescrever os métodos para mudar o comportamento das funções da API.

O objetivo é criar uma rota para associar duas APIs que são complementares... Ou seja, o curso e suas respectivas avaliações.

```
from django.urls import path
from .views import CursoAPIView, CursosAPIView, AvaliacaoAPIView, AvaliacoesAPIView

urlpatterns = [
    path('cursos/', CursosAPIView.as_view(), name='cursos'),
    path('cursos/<int:pk>', CursoAPIView.as_view(), name='curso'),
    path('cursos/<int:curso_pk>/avaliacoes/', AvaliacoesAPIView.as_view(), name='curso_avaliacoes'),
    path('cursos/<int:curso_pk>/avaliacoes/<int:avaliacao_pk>', AvaliacaoAPIView.as_view(), name='curso_avaliacao'),
    path('avaliacoes/', AvaliacoesAPIView.as_view(), name='avaliacoes'),
    path('avaliacoes/<int:pk>', AvaliacaoAPIView.as_view(), name='avaliacao')
]
```

O elemento <int:curso\_pk> indica que a ID extraída é do curso. Assim como, o elemento <int:avaliacao\_pk> indica que a ID extraída é da avaliação.

Dessa forma a url gerada vai numerar a ID do curso seguida da ID da avaliação.

EX. **api/v1/cursos/3/2** Isso significa que estamos observando a 2º avaliação do curso de ID 3.

Com as rotas alteradas o arquivo **views.py** também precisa sofrer alterações:

```
class AvaliacoesAPIView(generics.ListCreateAPIView):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer

    def get_queryset(self):
        if self.kwargs.get('curso_pk'):
            return self.queryset.filter(curso_id=self.kwargs.get('curso_id'))
        return self.queryset.all()
```

Correção:

kwargs.get('curso\_pk')

O **self.kwargs.get** vai pegar os valores das pk (urls.py), e se tiver esse valor, retorna o queryset das avaliações filtradas para aquele curso. Caso não tenha, retorna todas as avaliações.

Cursos representa uma coleção, mas a avaliação é individual. Portanto, a função para sobrescrever tem que ser o **get\_object()** para tratar do objeto (avaliação) de forma individual.

```
class AvaliacaoAPIView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer

    def get_object(self):
        if self.kwargs.get('curso_pk'):
            return get_object_or_404(self.get_queryset(), curso_id=self.kwargs.get('curso_id'), pk=self.kwargs.get('avaliacao_pk'))
        return get_object_or_404(self.get_queryset(), pk=self.kwargs.get('avaliacao_pk'))
```

No tratamento do objeto individual a função **get\_object\_or\_404** vai buscar o objeto através do (**'curso\_pk'**) se positivo vai buscar as ID das avaliações. Ou automaticamente vai exibir o erro de **404 error** caso não encontre ou haja erro de solicitação de dados.

## Utilizando ViewSets e Routers:

O objetivo dos ViewSets e Router é diminuir a poluição visual dos endpoints, assim como facilitar na construção da estrutura da API.

É muito utilizado nas melhorias e no versionamento das API, ou seja, manter a antiga funcionando e criar uma atualizada contendo melhorias que não estão presentes na versão anterior.

**OBS.** Por isso a importância de indicar a versão (**v1**) na url de acesso da API.

O ViewSets e Routers é uma melhoria que serve para automatizar a criação de endpoints nas API's.

OViewSet consiste em agrupar toda a lógica de um determinado recurso em apenas uma classe,

## Contextualizando:

**from rest\_framework import viewsets** no arquivo **views.py**

```
'''API VERSÃO 2'''

class CursoViewSet(viewsets.ModelViewSet):
    queryset = Curso.objects.all()
    serializer_class = CursoSerializer

class AvaliacaoViewSet(viewsets.ModelViewSet):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer
```

Essa estrutura de dados faz a mesma coisa que aquela sequencia de dados separadas de curso e avaliação construídas no CRUD anteriormente.

No arquivo **url.py** da aplicação cursos:

**from rest\_framework.routers import SimpleRouter**

```
from django.urls import path
from rest_framework.routers import SimpleRouter
from .views import CursoAPIView, CursosAPIView, AvaliacaoAPIView, AvaliacoesAPIView

from .views import (
    CursoAPIView,
    CursosAPIView,
    AvaliacaoAPIView,
    AvaliacoesAPIView,
    CursoViewSet,
    AvaliacaoViewSet)

router = SimpleRouter()
router.register('cursos', CursoViewSet)
router.register('avaliacoes', AvaliacaoViewSet)
```

No arquivo urls.py do projeto:

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

from cursos.urls import router

urlpatterns = [
    path('api/v1/', include('cursos.urls')),
    path('api/v2/', include(router.urls)),
    path('admin/', admin.site.urls),
    path('auth/', include('rest_framework.urls')),
]
```

Ou seja, não há necessidade de construção manual das rotas. O **router** cria essas rotas de forma automatizada.

**OBS.** Entretanto usando esse método a rota conjunta não irá funcionar. Porque o **SimpleRouter** só gera o CRUD para um único model.

## Corrigindo o erro de CRUD para um único model

Para fazer com que a rota aceite o uso de vários models na url... É feito:

No arquivo **views.py**

```
from rest_framework.decorators import action
from rest_framework.response import Response
```

Agora executar alterações em **CursoViewSet**

```
class CursoViewSet(viewsets.ModelViewSet):
    queryset = Curso.objects.all()
    serializer_class = CursoSerializer

    @action(detail=True, methods=['get'])
    def avaliacoes(self, request, pk=None):
        curso = self.get_object()
        serializer = AvaliacaoSerializer(curso.avaliacoes.all(), many=True)
        return Response(serializer.data)

class AvaliacaoViewSet(viewsets.ModelViewSet):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer
```

O decorador de **action** está ativado para criar uma nova rota. Onde vai trazer todas as avaliações, definidas na função **def** desse curso.

## Customizando as ViewSets:

Quando não há necessidade da API performar todos os métodos a alternativa utilizada é importar os mixins, dessa forma o código fica:

```
from rest_framework import mixins
```

```
class AvaliacaoViewSet(viewsets.ModelViewSet):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer


class AvaliacaoViewSet(
    mixins.ListModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.UpdateModelMixin,
    mixins.DestroyModelMixin,
    viewsets.GenericViewSet
):
    queryset = Avaliacao.objects.all()
    serializer_class = AvaliacaoSerializer
```

As duas classes desempenham a mesma função, a diferença é que o **ViewSet** é um compilado geral, e com o uso dos **mixins** é possível atribuir somente as funções desejadas.

## Utilizando relações com Django REST Framework:

As estratégias para apresentação de dados, são as mesmas para qualquer caso, precisando apenas definir primariamente qual o tipo de relação tratada, sendo elas: Um para um, um para muitos ou muitos para muitos, entre muitos.

O tipo de exibição das relações é feito no arquivo **serializers.py** adicionando o atributo extra na classe desejada:

### Nested Relationship

```
class CursoSerializer(serializers.ModelSerializer):
    # Nested Relationship
    avaliacoes = AvaliacaoSerializer(many=True, read_only=True)

    class Meta:
        model = Curso
        fields = (
            'id',
            'titulo',
            'url',
            'criacao',
            'ativo',
            'avaliacoes'
        )
```

O Nested Relationship serve para listar todas as avaliações de cada determinado curso.

```
Django REST framework
    }
],
{
  "id": 2,
  "titulo": "Curso número 2",
  "url": "http://www.test2.com.br",
  "criacao": "2022-03-28T13:58:54.607990-03:00",
  "ativo": true,
  "avaliacoes": [
    {
      "id": 2,
      "curso": 2,
      "nome": "Raissa Azevedo",
      "comentario": "Teste 2 de API",
      "avaliacao": "6.8",
      "criacao": "2022-03-28T14:00:50.222503-03:00",
      "ativo": true
    },
    {
      "id": 4,
      "curso": 2,
      "nome": "Maria Silva",
      "comentario": "Teste de avaliação de curso 2 da API com Django Framework.",
      "avaliacao": "8.7",
      "criacao": "2022-03-31T13:19:06.826407-03:00",
      "ativo": true
    }
  ]
}
```

Porém, com a suposição que haja diversos cursos, com centenas de avaliações, a lista de API ficaria gigante e não seria favorável a visualização desse arquivo. Causando problema de performance. Ou seja, é uma relação mais adequada ao relacionamento **um para um**.

## HyperLinked Related Field

Favorável quando se trata de muitos dados, ao invés de mostrar as avaliações em lista... Ele vai colocar um HyperLink para acesso.

```
class CursoSerializer(serializers.ModelSerializer):
    # HyperLinked Related Field
    avaliacoes = serializers.HyperlinkedIdentityField(many=True, read_only=True, view_name='avaliacao-detail')

    class Meta:
        model = Curso
        fields = (
            'id',
            'titulo',
            'url',
            'criacao',
            'ativo',
            'avaliacoes'
        )
```

A chave está em **view\_name='avaliacao-detail'** que vai prover o link para abrir os detalhes da avaliação.

```
"id": 2,
"titulo": "Curso número 2",
"url": "http://www.test2.com.br",
"criacao": "2022-03-28T13:58:54.607990-03:00",
"ativo": true,
"avaliacoes": [
  "http://127.0.0.1:8000/api/v2/avaliacoes/2/",
  "http://127.0.0.1:8000/api/v2/avaliacoes/4/"
]
```

Assim a performance da API vai ser mais otimizada quando se tratar de uma sequencia de dados extensa.

## Primary Key Related Field

Vai exibir os dados linkando através da chave primária de identificação.

The screenshot shows two parts: a code editor on the left and a terminal or JSON viewer on the right. The code editor contains a Python class named `CursoSerializer` which inherits from `serializers.ModelSerializer`. It includes a `avaliacoes` field annotated with `many=True, read_only=True`, indicating it's a Primary Key Related Field. The `Meta` class specifies the model is `Curso` and the fields are `'id', 'titulo', 'url', 'criacao', 'ativo', 'avaliacoes'`. The terminal on the right shows the resulting JSON output for a course with ID 2, titled "Curso número 2", created on "2022-03-28T13:58:54.607990-03:00", and active. The `avaliacoes` field is represented as a list containing the IDs 2 and 4.

```
class CursoSerializer(serializers.ModelSerializer):
    # Primary Key Related Field
    avaliacoes = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
    class Meta:
        model = Curso
        fields = (
            'id',
            'titulo',
            'url',
            'criacao',
            'ativo',
            'avaliacoes'
        )
```

```
{
  "id": 2,
  "titulo": "Curso número 2",
  "url": "http://www.test2.com.br",
  "criacao": "2022-03-28T13:58:54.607990-03:00",
  "ativo": true,
  "avaliacoes": [
    2,
    4
  ]
}
```

Ou seja, ao invés de exibir o link com os dados extras, a API agora via apenas exibir a lista de ID. È a forma mais leve para a performance. Ideal, para centenas de milhares de dados.

## Utilizando Paginação com Django REST Framework

A paginação é um técnica usada para garantir a performance a API tendo em vista uam quantidade grande de dados a serem acessados.

### No `settings.py`

The screenshot shows a portion of a `settings.py` file. It defines the `#DRF` section under `REST_FRAMEWORK`. This section contains configurations for authentication, permissions, and pagination. The `DEFAULT_PAGINATION_CLASS` is set to `'rest_framework.pagination.PageNumberPagination'` and the `PAGE_SIZE` is set to 2.

```
#DRF
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ),
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 2
}
```

Isso indica que vai haver 2 elementos por página, por se tratar de poucos dados. Por exemplo se houvesse 100, poderia colocar 10 elementos por página e assim sucessivamente. O `PAGE_SIZE` vai depender da quantidade de dados que a API estará comportando.

Com a técnica de paginação configurada:

The screenshot shows the Django REST framework's built-in browsable API interface. At the top, it says "Django REST framework" and "raissa". Below that, the title "Avaliacoes Api" is displayed, along with "OPTIONS" and "GET" buttons. A navigation bar at the bottom right shows page numbers 1, 2, and "»".

The main content area shows a successful HTTP 200 OK response. The response headers include:

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
```

The JSON response body is as follows:

```
{
  "count": 4,
  "next": "http://127.0.0.1:8000/api/v1/avaliacoes/?page=2",
  "previous": null,
  "results": [
    {
      "id": 1,
      "curso": 1,
      "nome": "Raissa Azevedo",
      "comentario": "Esse é um teste do tutorial de domínio de criação de API com Django RESTful Framework",
      "avaliacao": "9.6",
      "criacao": "2022-03-28T14:00:12.114334-03:00",
      "ativo": true
    },
    {
      "id": 2,
      "curso": 2,
      "nome": "Raissa Azevedo",
      "comentario": "Teste 2 de API",
      "avaliacao": "6.8",
      "criacao": "2022-03-28T14:00:50.222503-03:00",
      "ativo": true
    }
}
```

É possível notar que agora há um menu de paginação no canto superior direito. Assim como, um elemento chamado **count** que não havia anteriormente. Que vai indicar o número de cursos ou avaliação cadastrados.

Para garantir que os dados sejam ordenados de forma correta na paginação da API, adicionar as seguintes configurações no **models.py**

```
class Curso(Base):
    titulo = models.CharField(max_length=255)
    url = models.URLField(unique=True)

    class Meta:
        verbose_name = 'Curso'
        verbose_name_plural = 'Cursos'
        ordering = ['id']

    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']
        ordering = ['id']
```

O **ordering** indica que a ordem de exibição dos elementos nas páginas da API vai ser ditada pelo número da ID.

## No arquivo **views.py**

```
class CursoViewSet(viewsets.ModelViewSet):
    queryset = Curso.objects.all()
    serializer_class = CursoSerializer

    @action(detail=True, methods=['get'])
    def avaliacoes(self, request, pk=None):
        self.pagination_class.page_size = 2
        avaliacoes = Avaliacao.objects.filter(curso_id=pk)
        page = self.paginate_queryset(avaliacoes)

        if page is not None:
            serializer = AvaliacaoSerializer(page, many=True)
            return self.get_paginated_response(serializer.data)

        serializer = AvaliacaoSerializer(avaliacoes, many=True)
        return Response(serializer.data)
```

As configurações são alteradas para garantir a paginação correta, ou seja, há a verificação se há paginas. E passa por serializer apenas as paginas. Caso não haja, ai passa todos os elementos pelo serializer final. Esse método garante a boa performance da API.