



CONSTRUA APIs COM DJANGO E DJANGO REST FRAMEWORK

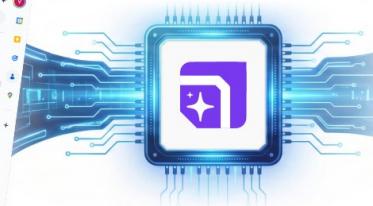
Nesse ebook vamos ver como é fácil construir APIs de maneira rápida com nosso velho conhecido Django e o poderoso Django Rest Framework

Crie Ebooks técnicos incríveis em minutos com IA

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



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

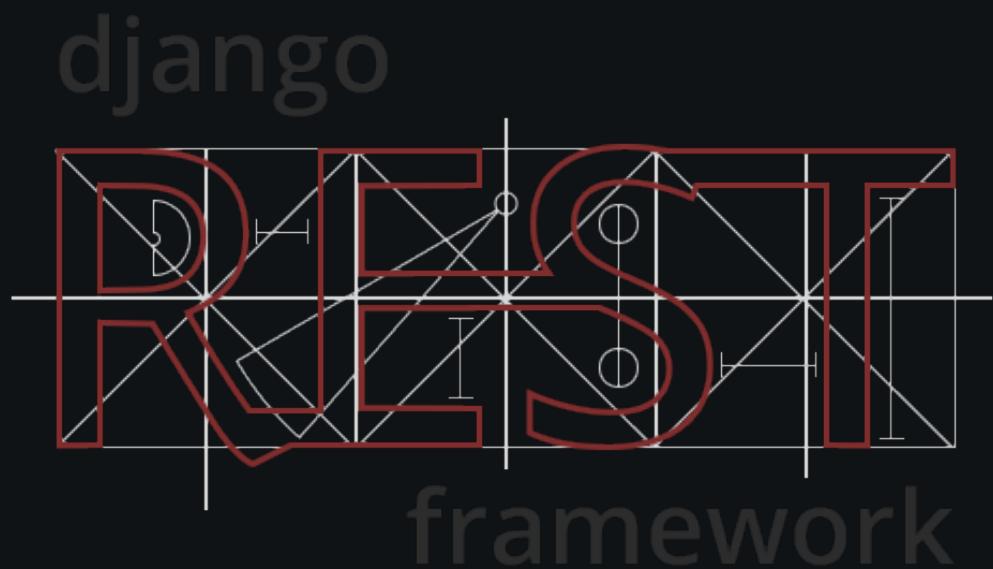
 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA 

Olá **dev!**

Vamos tratar de uma biblioteca muito poderosa para construções de APIs: o Django Rest Framework, ou **DRF** para os intímos!



⚠️ Antes de começar, uma **nota importante** ⚠️

*Esse post não é mais um daqueles tutoriais que apenas traduzem o Getting Started da documentação. Aqui vamos **além do básico**, com a qualidade da Python Academy que vocês já conhecem! Dito isso, podemos começar! 😊*

Com o DRF é possível combinar Python e Django de uma forma flexível para desenvolver APIs web de uma forma bem **simples** e **rápida**.

Algumas razões para usar o DRF:

- Ele provê uma interface navegável para debuggar sua API.
- Diversas estratégias de autenticação, incluindo pacotes para OAuth1 e OAuth2.
- Serialização de objetos de fontes ORM (bancos de dados) e não-ORM (classes próprias).
- Documentação extensa e grande comunidade.
- Usado por grandes corporações, como: Heroku, EventBrite, Mozilla e Red Hat.

E o melhor de tudo, ele utiliza nosso querido Django como base!

Por isso é interessante que você já possua certo conhecimento em Django. Ainda não é **craque**?

Não tem problema, aqui na Python Academy você conta com o melhor material sobre Django! Acesse:

[Django: Introdução ao framework](#)

[Django: A Camada Model](#)

[Django: A Camada View](#)

[Django: A Camada Template](#)

Ou se preferir, baixe nosso ebook **GRÁTIS** de Desenvolvimento Web com Python e Django:

Agora:

Já faz seu cafézinho de cada dia

Prepara o caderno

 **Abra seu editor de código**

Separa 1 horinha

E bora nessa!

Introdução

Nada melhor que aprender uma nova ferramenta colocando a mão no código e fazendo um mini-projeto.

Para esse post decidi juntar duas coisas que gosto muito: código e **investimentos**!

Então nesse post vamos desenvolver uma API para consulta de um tipo de investimento: **Fundos Imobiliários** (ou FIIs).

Não sabe o que é? Então aí vai:

Fundos Imobiliários (FII) são fundos de investimento destinados à aplicação em empreendimentos imobiliários, o que inclui, além da aquisição de direitos reais sobre bens imóveis, o investimento em títulos relacionados ao mercado imobiliário, como letras de crédito imobiliário (LCI), letras hipotecárias (LH), cotas de outros FII, certificados de potencial adicional de construção, (CEPAC), certificados de recebíveis imobiliários (CRI), e outros previstos na regulamentação (Retirado de: investidor.gov.br)

Dito isso, vamos começar do começo!

Vamos criar a estrutura e configurar o DRF.

Configuração do Projeto

Primeiro, vamos começar pelo nome: vamos chamá-lo de **Fundsfinder** (“buscador” de Fundos).

Vamos então aos primeiros passos:

```
# Cria a pasta e a acessa
mkdir fundsfinder && cd fundsfinder

# Cria ambiente virtual
virtualenv venv --python=/usr/bin/python3.8

# Ativa ambiente virtual
source venv/bin/activate

# Instala Django e DRF
pip install django djangorestframework
```

Até aqui, nós: - Criamos a pasta do projeto; - Obviamente criamos um ambiente virtual (ainda não sabe o que é? 😱 Acesse correndo nosso post sobre [ambientes virtuais em Python aqui](#)) - Ativamos o ambiente virtual e instalamos as dependências (Django e DRF)

Agora, vamos criar um novo *app* para separar responsabilidades da nossa API.

Vamos chamá-lo de `api`.

Usamos o comando `startapp` do `django-admin` na raíz do projeto (onde se encontra o arquivo `manage.py`), dessa forma:

```
python3 manage.py startapp api
```

Aproveite e crie a estrutura inicial do banco de dados com:

```
python3 manage.py migrate
```

Agora, temos a seguinte estrutura:

```
$ tree -L 3 -I venv
.
+-- api
|   |-- admin.py
|   |-- apps.py
|   |-- __init__.py
|   |-- migrations
|   |   `-- __init__.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
+-- db.sqlite3
+-- fundsfinder
|   |-- asgi.py
|   |-- __init__.py
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
`-- manage.py

3 directories, 14 files
```

Execute o servidor local para verificar se tudo está correto:

```
python3 manage.py runserver
```

Ao acessar <http://localhost:8000> em seu browser a seguinte tela deve ser mostrada:



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



[Django Documentation](#)

Topics, references, & how-to's



[Tutorial: A Polling App](#)

Get started with Django



[Django Community](#)

Connect, get help, or contribute

Agora, adicione um super usuário com o comando `createsuperuser` (será requisitado uma senha):

```
python manage.py createsuperuser --email admin@fundsfinder.com --username admin
```

Só falta uma coisa para terminarmos as configurações iniciais do nosso projeto: adicionar tudo ao `settings.py`.

Para isso, abra o arquivo `fundsfinder/settings.py` e adicione os apps `api`, `fundsfinder` e `rest_framework` (necessário para funcionamento do DRF) à configuração `INSTALLED_APPS`, ficando assim:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'fundsfinder',  
    'api'  
]
```

Pronto!

Com isso temos a estrutura inicial para finalmente começarmos nosso projeto!

Modelagem

O processo de desenvolvimento de aplicações que utilizam o Django Rest Framework geralmente seguem a seguinte linha de implementação:

1. **1) Modelagem;**
2. **2) Serializers;**
3. **3) ViewSets;**
4. **4) Routers**

Calma que vamos tratar cada passo aqui!

Vamos começar com a **Modelagem**!

Bem, como vamos fazer um sistema para busca e listagem de Fundos Imobiliários, nossa modelagem deve refletir campos que façam sentido.

Para auxiliar nessa tarefa, escolhi alguns parâmetros de uma tabela muito interessante do site [FundsExplorer](#):

CÓDIGO DO FUNDO	SETOR	PREÇO ATUAL	LIQUIDEZ DIÁRIA	DIVIDENDO	DIVIDEND YIELD	DY (3M) ACUMULADO	DY (6M) ACUMULADO
ABCP11	Shoppings	R\$ 71,45	6035.0	R\$ 0,05	0,06%	1,30%	2,69%
AFCR11	Híbrido	R\$ 113,49	4773.0	R\$ 1,07	0,93%	2,75%	5,25%
AFOF11	Títulos e Val. Mob.	R\$ 224,98	7.0	R\$ 8,76	N/A	N/A	N/A
AIEC11	Lajes Corporativas	R\$ 88,49	2504.0	R\$ 0,57	0,63%	1,87%	3,67%
ALMI11	Lajes Corporativas	R\$ 1.110,10	276.0	R\$ 0,00	0,00%	0,00%	0,00%
ALZR11	Logística	R\$ 127,50	9284.0	R\$ 0,61	0,47%	1,32%	2,22%
ARCT11	Híbrido	R\$ 117,40	3687.0	R\$ 2,31	2,08%	6,66%	10,73%
ARRI11	Títulos e Val. Mob.	R\$ 109,80	486.0	R\$ 1,05	0,92%	3,16%	6,14%
ATSA11	Shoppings	R\$ 108,14	36.0	R\$ 0,25	0,18%	0,59%	1,30%

Vamos usar os seguintes atributos:

- **Código do Fundo:** Código identificador do Fundo.
- **Setor:** Setor do Fundo Imobiliário.
- **Dividend Yield médio (12 meses):** *Dividend Yield* mostra quanto um fundo paga de Dividendos (divisão de lucros) sobre o valor atual da cota.
- **Vacância Financeira:** Importante métrica que mostra ao investidor quantos ativos do Fundo Imobiliários estão inadimplentes.
- **Vacância Física:** Outra importante métrica que mostra ao investidor quantos ativos estão desocupados.
- **Quantidade de Ativos:** Quantos ativos são administrados pelo Fundo.

Com isso em mãos, podemos criar a modelagem da entidade `FundoImobiliar`.

Utilizaremos o ORM (Mapeamento Objeto-Relacional) do próprio Django.

Temos um post completo sobre a camada Model do Django, clique aqui para conferir!

Analizando essa tabela do FundsExplorer, nossa modelagem pode ser implementada da seguinte forma (`api/models.py`):

```
from django.db import models
import uuid

class FundoImobiliario(models.Model):
    SETOR_CHOICES = [
        ('hospital', 'Hospital'), ('hotel', 'Hotel'), ('hibrido',
            'Híbrido'),
        ('lajes_corporativas', 'Lajes Corporativas'),
        ('logistica', 'Logística'), ('outros', 'Outros'),
        ('residencial', 'Residencial'),
        ('titulos_valores_mobiliarios', 'Títulos e Val. Mob.')
    ]

    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        null=False,
        blank=True)

    codigo = models.CharField(
        max_length=8,
        null=False,
        blank=False)

    setor = models.CharField(
        max_length=30,
        null=False,
        blank=False,
        choices=SETOR_CHOICES)

    dividend_yield_medio_12m = models.DecimalField(
        null=False,
        blank=False,
        max_digits=5,
        decimal_places=2)

    vacancia_financeira = models.DecimalField(
        null=False,
        blank=False,
        max_digits=5,
        decimal_places=2)
```

```
vacancia_fisica = models.DecimalField(  
    null=False,  
    blank=False,  
    max_digits=5,  
    decimal_places=2)  
  
quantidade_ativos = models.IntegerField(  
    null=False,  
    blank=False,  
    default=0)
```

Com nossa modelagem criada, precisamos gerar o arquivo de Migrações para atualizar o banco de dados.

Fazemos isso com o comando `makemigrations` (*saiba mais sobre esse comando do Django clicando aqui!*).

Execute:

```
python3 manage.py makemigrations api
```

Agora vamos aplicar a migração ao Banco de Dados com o comando `migrate` (*saiba mais sobre esse comando do Django clicando aqui!*).

Execute:

```
python3 manage.py migrate
```

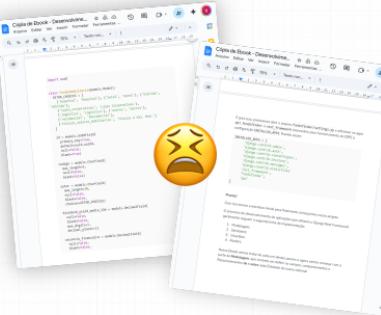
Com a modelagem pronta, podemos passar para o *Serializer*!

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

 DevBook

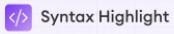
Crie Ebooks técnicos incríveis em minutos com IA

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



Chega de formatar código no Google Docs

Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight  Adicione Banners Promocionais  Edite em Markdown em Tempo Real  Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS 

Serializer

Os *serializers* do DRF são componentes essenciais do *framework*.

Eles servem para traduzir entidades complexas, como *querysets* e instâncias de classes em representações simples que podem ser usadas no tráfego da web, como `JSON` e `XML`.

Esse processo é chamado de **Serialização**.

Serializers também servem para fazer o caminho contrário: a **Desserialização**.

Isto é, transformar representações simples (como `JSON` e `XML`) em representações complexas, instanciando objetos, por exemplo.

Vamos criar o arquivo onde vão ficar os *serializers* da nossa API.

Crie um arquivo chamado `serializers.py` dentro da pasta `api/`.

O DRF disponibiliza diversos tipos de *serializers* que podemos utilizar, como:

- `BaseSerializer`: Classe base para construção de `Serializers` mais genéricos.
- `ModelSerializer`: Auxilia a criação de serializadores baseados em modelos.
- `HyperlinkedModelSerializer`: Similar ao `ModelSerializer`, contudo retorna um link para representar o relacionamento entre entidades (`ModelSerializer` retorna, por padrão, o id da entidade relacionada).

Vamos utilizar o `ModelSerializer` para construir o serializador da entidade `FundoImobiliario`.

Para isso, precisamos declarar sobre qual modelo aquele serializador irá operar e quais os campos que ele deve se preocupar.

Nosso *serializer* pode se implementado da seguinte maneira:

```
from rest_framework import serializers
from api.models import FundoImobiliario

class FundoImobiliarioSerializer(serializers.ModelSerializer):
    class Meta:
        model = FundoImobiliario
        fields = [
            'id',
            'codigo',
            'setor',
            'dividend_yield_medio_12m',
            'vacancia_financeira',
            'vacancia_fisica',
            'quantidade_ativos'
        ]
```

Nesse `Serializer` :- `model = FundoImobiliario` define qual modelo esse `serializer` deve serializar. - `fields` define os campos que serão serializados.

Obs: é possível definir que todos os campos da entidade de modelo devem ser serializados usando `fields = '__all__'`, contudo eu prefiro mostrar os campos explicitamente.

Com isso, finalizamos mais uma etapa do passo a passo do DRF!

Vamos à terceira etapa: a criação de *Views*.

ViewSets

As *ViewSets* definem quais operações REST estarão disponíveis e como seu sistema vai responder às chamadas à sua API.

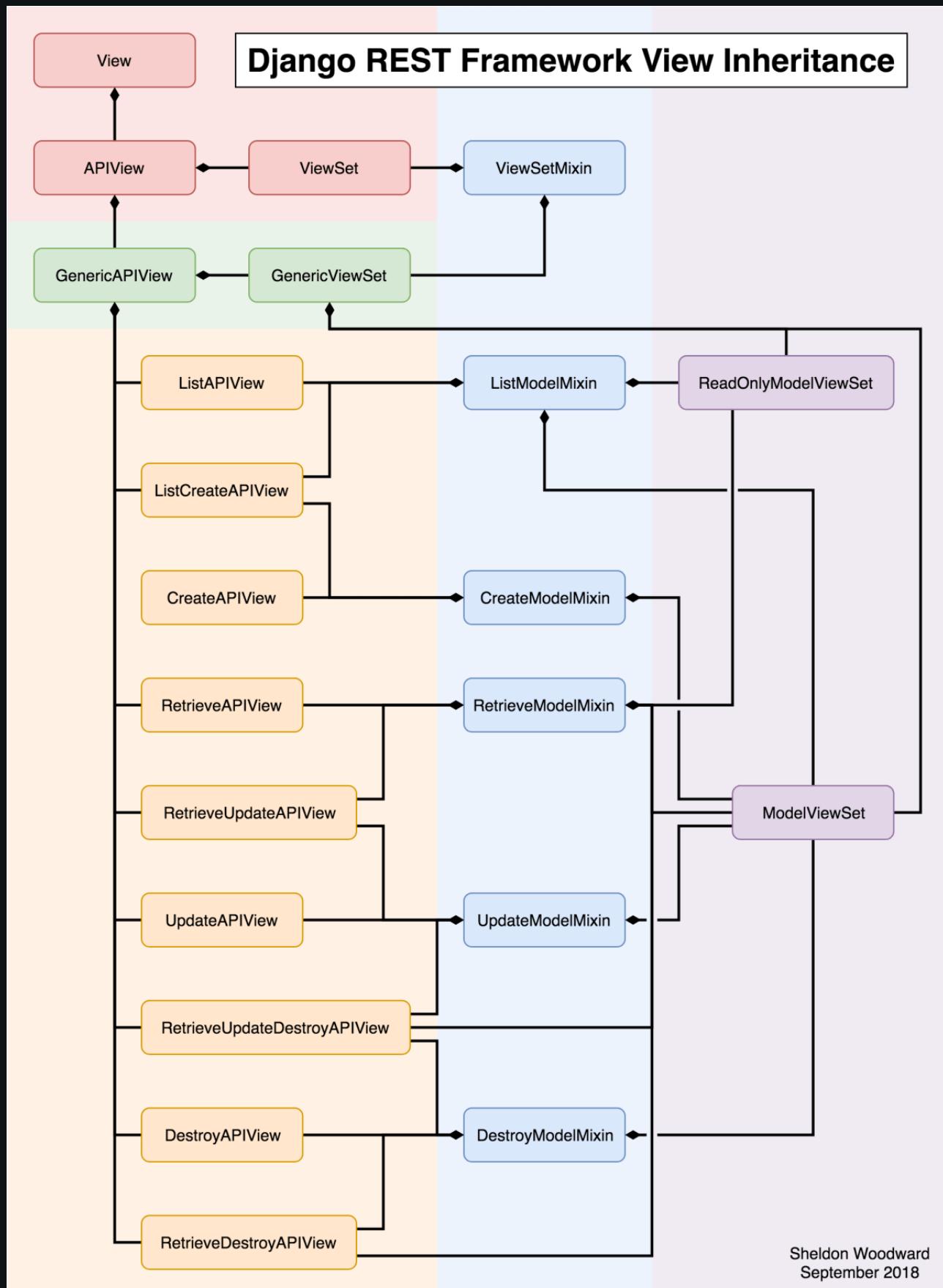
Em outros *frameworks*, são chamados de *Resources* ou *Controllers*.

ViewSets herdam e adicionam lógica às *Views* padrão do Django.

Suas responsabilidades são:

- Receber os dados da Requisição (formato JSON ou XML)
- Validar os dados de acordo com as regras definidas na modelagem e no *Serializer*
- Dessaializar a Requisição e instanciar objetos
- Processar **regras de negócio** (aqui é onde implementamos a lógica dos nossos sistemas)
- Formular uma resposta e responder a quem chamou sua API

Encontrei uma imagem muito interessante [no Reddit](#) que mostra o diagrama de herança das classes do DRF, que nos ajuda a entender melhor a estrutura interna do *framework*:



Sheldon Woodward
September 2018

Na imagem:

- Lá em cima, temos a classe `View` padrão do Django.
- `APIView` e `ViewSet` são classes do DRF que herdam de `View` e que trazem algumas configurações específicas para transformá-las em APIs, como métodos `get()` para tratar requisições `HTTP GET` e `post()` para tratar requisições `HTTP POST`.
- Logo abaixo, temos a `GenericAPIView` - que é a classe base para *views* genéricas - e a `GenericViewSet` - que é a base para as `ViewSets` (a parte da direita em roxo na imagem).
- No meio, em azul, temos os `Mixins`. Eles são os blocos de código responsáveis por **realmente** implementar as ações desejadas.
- Em seguida temos as `Views` que disponibilizam as funcionalidades da nossa API, como se fossem blocos de Lego. Elas estendem dos `Mixins` para construir a funcionalidade desejada (seja listagem, seja deleção e etc)

Por exemplo: se você quiser criar uma API que disponibilize apenas listagem de uma determinada Entidade você poderia escolher a `ListAPIView`.

Agora se você precisar construir uma API que disponibilize apenas as operações de criação e listagem, você poderia utilizar a `ListCreateAPIView`.

Agora se você precisar construir uma API “com tudo dentro” (isto é: criação, deleção, atualização e listagem), escolha a `ModelViewSet`: perceba que ela estende todos os `Mixins` disponíveis.

Para entender de vez:

- Os `Mixins` são como os componentes dos sanduíches do Subway
- As `Views` são como o Subway: você monta o seu, componente à componente

- As `ViewSets` são como o McDonalds: seu sanduíche já vem montado

Eu sei, eu sei... Não paro de pensar em comida 😂 😂 😂

Percebe-se, portanto, que o DRF disponibiliza diversos tipos de *Views* e *Viewsets* que podem ser customizados de acordo com a necessidade do sistema.

Para isso, estude bem a [documentação](#) (ou fique ligado aqui na Python Academy 😊)

Para facilitar nossa vida, vamos utilizar a `ModelViewSet` !

No DRF, por convenção, implementamos as *Views/ViewSets* no arquivo `views.py` dentro do *app* em questão.

Esse arquivo já é criado quando utilizamos o comando `django-admin startapp api`, portanto não precisamos criá-lo.

Agora, veja como é difícil criar um `ModelViewSet` (não se espantem com a complexidade):

```
from api.serializers import FundoImobiliarioSerializer
from rest_framework import viewsets, permissions
from api.models import FundoImobiliario

class FundoImobiliarioViewSet(viewsets.ModelViewSet):
    queryset = FundoImobiliario.objects.all()
    serializer_class = FundoImobiliarioSerializer
    permission_classes = [permissions.IsAuthenticated]
```



É isso jovem! 😊

Você pode estar se perguntando?

Uai, e cadê o resto?

Bem, é aí que mora o **amor** e o **ódio** às *Class-Based-Views* (CBVs).

Amor pois quem defende as CBVs afirma que elas aumentam a produtividade, pois não temos que escrever dezenas de linhas de código.

Ódio pois muito do funcionamento do *framework* fica implícito/escondido dos olhos dos desenvolvedores.

Bem... Ao meu ver, acho que fica a cargo do desenvolvedor escolher qual caminho seguir: *Function-Based-Views* (FBVs) ou *Class-Based-Views* (CVBs).

Use aquilo que lhe traga mais produtividade e entendimento do *framework*.

Continuando...

Todo o código para tratamento de Requisições, serialização e desserialização de objetos e formulação de Respostas HTTP está dentro das classes que herdamos direta e indiretamente.

Em nossa classe `FundoImobiliarViewSet` apenas precisamos declarar os seguintes parâmetros:

- `queryset` : Configura o *queryset* base para ser utilizado pela API. Ele é utilizado na ação de listar, por exemplo.
- `serializer_class` : Configura qual *Serializer* deverá ser usado para consumir dados que chegam à API e produzir dados que serão enviados como resposta.
- `permission_classes` : Lista contendo as permissões necessárias para acessar o endpoint exposto por essa *ViewSet*. Nesse caso, irá permitir apenas o acesso a usuários autenticados.

Com isso matamos o terceiro passo: a *ViewSet*!

Agora vamos à configuração das URLs! ## *Routers*

Os *Routers* nos auxiliam na geração das URLs da nossa aplicação.

Como o REST possui padrões bem definidos de estrutura de URLs, o DRF as gera automaticamente para nós, já no padrão correto.

Basta utilizarmos seus **Routers**!

Para isso, primeiro crie o arquivo `urls.py` em `api/urls.py`.

Agora veja como é simples!

```
from rest_framework.routers import DefaultRouter
from api.views import FundoImobiliarioViewSet

app_name = 'api'

router = DefaultRouter(trailing_slash=False)
router.register(r'fundos', FundoImobiliarioViewSet)

urlpatterns = router.urls
```

Vamos entender:

- `app_name` é necessário para dar contexto às URLs geradas. Esse parâmetro especifica o *namespace* das URLConfs adicionadas.
- `DefaultRouter` é o *Router* que escolhemos para geração automática das URLs. O parâmetro `trailing_slash` especifica que não é necessário o uso de barras / no final da URL.
- O método `register` recebe dois parâmetros: o primeiro é o prefixo que será usado na URL (no nosso caso: `http://localhost:8000/fundos`) e o segundo é a `View` que irá responder as URLs com esse prefixo.
- Por último, temos o velho `urlpatterns` do Django, que utilizamos para exportar as URLs desse *app*.

Agora precisamos adicionar as URLs específicas do nosso *app* `api` ao projeto.

Para isso, abra o arquivo `fundsfinder/urls.py` e adicione as seguintes linhas (URLConf do nosso *app* e a URLConf de autenticação do DRF):

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

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

Obs: Como boa prática, use sempre o prefixo `api/v1/` para manter a compatibilidade caso precise evoluir sua api pra V2 (`api/v2/`)!

Utilizando apenas essas linhas de código, olha o caminhão de *endpoints* que o DRF gerou automaticamente para nossa API:

URL	Método HTTP	Ação
/api/v1	GET	Raíz da API gerada automaticamente
/api/v1/fundos	GET	Listagem de todos os elementos
/api/v1/fundos	POST	Criação de novo elemento
/api/v1/fundos/ {lookup}	GET	Recuperar elemento pelo ID
/api/v1/fundos/ {lookup}	PUT	Atualização de elemento por ID
/api/v1/fundos/ {lookup}	PATCH	Atualização parcial por ID (<i>partial update</i>)
/api/v1/fundos/ {lookup}	DELETE	Deleção de elemento por ID

Aqui, `{lookup}` é o parâmetro utilizado pelo DRF para identificar unicamente um elemento.

Vamos supor que um Fundo tenha `id=ef249e21-43cf-47e4-9aac-0ed26af2d0ce`.

Podemos excluí-lo enviando uma requisição `HTTP DELETE` para a URL:

```
http://localhost:8000/api/v1/fundos/  
ef249e21-43cf-47e4-9aac-0ed26af2d0ce
```

Ou podemos criar um novo Fundo enviando uma requisição `POST` para a URL `http://localhost:8000/api/v1/fundos` e os valores dos campos no corpo da requisição, assim:

```
{  
  "codigo": "XPLG11",  
  "setor": "logistica",  
  "dividend_yield_medio_12m": "6.30",  
  "vacancia_financeira": "7.87",  
  "vacancia_fisica": "12.36",  
  "quantidade_ativos": 19  
}
```

Dessa forma, nossa API retornaria um código `HTTP 201 Created`, significando que um objeto foi criado e a resposta seria:

```
{  
  "id": "a4139c66-cf29-41b4-b73e-c7d203587df9",  
  "codigo": "XPLG11",  
  "setor": "logistica",  
  "dividend_yield_medio_12m": "6.30",  
  "vacancia_financeira": "7.87",  
  "vacancia_fisica": "12.36",  
  "quantidade_ativos": 19  
}
```

Podemos testar nossa URL de diversas forma: através de código Python, através de um Frontend (Angular, React, Vue.js) ou através do [Postman](#), por exemplo.

E com isso meu amigo, finalizamos **TUDO** (*ou quase tudo!*)! 😊

Modelagem;
Serializers;
ViewSets;
Routers

E como eu vejo esse trem funcionando?

Então bora pra próxima seção! 



Interface navegável

Uma das funcionalidades mais impressionantes do DRF é sua **Interface Navegável**.

Com ela, podemos testar nossa API e verificar seus valores de uma maneira visual muito simples de se utilizar.

Para acessá-la, navegue em seu browser para: `http://localhost:8000/api/v1`.

Você deverá ver o seguinte:

The screenshot shows the Django REST framework's API root view. At the top, it says "Django REST framework" and "Log in". Below that, it says "Api Root". On the right, there are two buttons: "OPTIONS" and "GET ▾". A description below says "The default basic root view for DefaultRouter". Underneath, there is a "GET /api/v1/" button. The response body shows an HTTP 200 OK status with headers: Allow: GET, HEAD, OPTIONS; Content-Type: application/json; Vary: Accept. The JSON response is: { "fundos": "http://127.0.0.1:8000/api/v1/fundos" }.

Vá lá e clique em `http://127.0.0.1:8000/api/v1/fundos`!

Uai? Que isso man, tá tirando?

Calma! 😊

Deve ter aparecido a mensagem:

```
{  
    "detail": "Authentication credentials were not provided."  
}
```

Lembra da configuração `permission_classes` que utilizamos para configurar a nossa `FundoImobiliarioViewSet` ?

Ela definiu que apenas usuários autenticados (`permissions.isAuthenticated`) podem interagir com a API.

Então clique no canto superior direito, em *Log in* e use as credenciais cadastradas no comando `createsuperuser`, que executamos no início do post (use o `username` e a senha).

Agora, olha que útil! Você deve estar vendo:

[Api Root](#) / Fundo Imobiliario List

Fundo Imobiliario List

[OPTIONS](#)[GET](#) ▾[GET /api/v1/fundos](#)

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

[]

[Raw data](#)[HTML form](#)

Id

Codigo

Setor

 HospitalDividend yield
medio 12mVacancia
financeira

Vacancia fisica

Quantidade ativos

[POST](#)

Brinque um pouco, adicione dados, explore a interface!

Ao adicionar dados e atualizar a página, é disparada uma requisição `HTTP GET` à API, retornando os dados que você acabou de cadastrar:

Api Root / Fundo Imobiliario List

Fundo Imobiliario List

[OPTIONS](#)[GET](#) ▾

GET /api/v1/fundos

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

```
[
  {
    "id": "c41c2a9d-7aa4-4d07-b3b1-7a45df55ac79",
    "codigo": "MAXR11",
    "setor": "hibrido",
    "dividend_yield_medio_12m": "5.90",
    "vacancia_financeira": "0.00",
    "vacancia_fisica": "5.00",
    "quantidade_ativos": 10
  },
  {
    "id": "d7d9e0c5-1885-4298-ae12-3f8aa8eea755",
    "codigo": "HGLG11",
    "setor": "logistica",
    "dividend_yield_medio_12m": "4.80",
    "vacancia_financeira": "5.00",
    "vacancia_fisica": "17.00",
    "quantidade_ativos": 21
  },
  {
    "id": "ef249e21-43cf-47e4-9aac-0ed26af2d0ce",
    "codigo": "XPLG11",
    "setor": "logistica",
    "dividend_yield_medio_12m": "6.30",
    "vacancia_financeira": "7.87",
    "vacancia_fisica": "12.36",
    "quantidade_ativos": 19
  }
]
```

Diz se não é M-A-G-I-C-O! 😍

Filtros, Busca Textual e Ordenação

É possível adicionar as funcionalidades de Filtragem de dados, Busca Textual e Ordenação com pouquíssimas linhas de código.

Apenas precisamos informar ao *framework* quais campos podem ser filtrados, quais possibilitarão a busca textual e quais campos poderão servir para ordenação.

Para não alongar demais esse post, preferi separar esse assunto nesse outro *post*:

Filtro, Busca e Ordenação no Django REST Framework

Então já clica aqui  e deixa separado em outra aba pra continuar depois 😊

Configurações específicas

É possível configurar diversos aspectos do DRF através de configurações específicas.

Fazemos isso adicionando o objeto `REST_FRAMEWORK` ao arquivo de configurações `settings.py`.

Por exemplo, se quisermos adicionar paginação à nossa API, podemos fazer simplesmente isso:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',  
    'PAGE_SIZE': 10  
}
```

Agora o resultado de uma chamado, por exemplo, à `http://127.0.0.1:8000/api/v1/fundos` passa de:

```
[  
 {  
   "id": "ef249e21-43cf-47e4-9aac-0ed26af2d0ce",  
   "codigo": "XPLG11",  
   "setor": "logistica",  
   "dividend_yield_medio_12m": "6.30",  
   "vacancia_financeira": "7.87",  
   "vacancia_fisica": "12.36",  
   "quantidade_ativos": 19  
 },  
 {  
   "id": "c41c2a9d-7aa4-4d07-b3b1-7a45df55ac79",  
   "codigo": "MAXR11",  
   "setor": "hibrido",  
   "dividend_yield_medio_12m": "5.90",  
   "vacancia_financeira": "0.00",  
   "vacancia_fisica": "5.00",  
   "quantidade_ativos": 10  
 },  
 {  
   "id": "d7d9e0c5-1885-4298-ae12-3f8aa8eea755",  
   "codigo": "HGLG11",  
   "setor": "logistica",  
   "dividend_yield_medio_12m": "4.80",  
   "vacancia_financeira": "5.00",  
   "vacancia_fisica": "17.00",  
   "quantidade_ativos": 21  
 }  
 ]
```

Para:

```
{
  "count": 3,
  "next": null,
  "previous": null,
  "results": [
    {
      "id": "ef249e21-43cf-47e4-9aac-0ed26af2d0ce",
      "codigo": "XPLG11",
      "setor": "logistica",
      "dividend_yield_medio_12m": "6.30",
      "vacancia_financeira": "7.87",
      "vacancia_fisica": "12.36",
      "quantidade_ativos": 19
    },
    {
      "id": "c41c2a9d-7aa4-4d07-b3b1-7a45df55ac79",
      "codigo": "MAXR11",
      "setor": "hibrido",
      "dividend_yield_medio_12m": "5.90",
      "vacancia_financeira": "0.00",
      "vacancia_fisica": "5.00",
      "quantidade_ativos": 10
    },
    {
      "id": "d7d9e0c5-1885-4298-ae12-3f8aa8eea755",
      "codigo": "HGLG11",
      "setor": "logistica",
      "dividend_yield_medio_12m": "4.80",
      "vacancia_financeira": "5.00",
      "vacancia_fisica": "17.00",
      "quantidade_ativos": 21
    }
  ]
}
```

Note que foram adicionados campos que servem para a aplicação chamadora se localizar:

- `count` : A quantidade de resultados retornados

- `next` : A próxima página
- `previous` : A página anterior
- `results` : A página atual de resultados

Existem diversas outras configurações muito úteis!

Trago aqui algumas:

👉 `DEFAULT_AUTHENTICATION_CLASSES` para configurar o método de autenticação da API:

```
REST_FRAMEWORK = {  
    ...  
    DEFAULT_AUTHENTICATION_CLASSES: [  
        'rest_framework.authentication.SessionAuthentication',  
        'rest_framework.authentication.BasicAuthentication'  
    ]  
    ...  
}
```

👉 `DEFAULT_PERMISSION_CLASSES` para configurar permissões necessárias para acessar a API (à nível global).

```
REST_FRAMEWORK = {  
    ...  
    DEFAULT_PERMISSION_CLASSES: [ 'rest_framework.permissions.AllowAny' ]  
    ...  
}
```

Obs: Também é possível definir essa configuração por *View*, utilizando o atributo `permissions_classes` (que utilizamos na nossa `FundoImobiliarioViewSet`).

👉 `DATE_INPUT_FORMATS` para configurar formatos de datas aceitos pela API:

```
REST_FRAMEWORK = {  
    ...  
    'DATE_INPUT_FORMATS': ['%d/%m/%Y', '%Y-%m-%d', '%d-%m-%y', '%d-%m-%Y']  
    ...  
}
```

A configuração acima fará a API permitir os seguintes formatos de data ‘25/10/2006’, ‘2006-10-25’, ‘25-10-2006’, por exemplo.

Veja mais configurações [acessando aqui a Documentação](#).

Conclusão

Vimos nesse *post* como é fácil construir APIs altamente customizáveis com o **Django REST Framework!**

Quis trazer o máximo de informações possíveis!

Mas fique de olho por aqui pois ainda teremos muito conteúdo sobre DRF!

E você? Já utilizou o DRF em algum projeto? Já teve alguma dificuldade? Gostaria de sugerir algum conteúdo?

Conta pra gente aqui embaixo nos comentários!

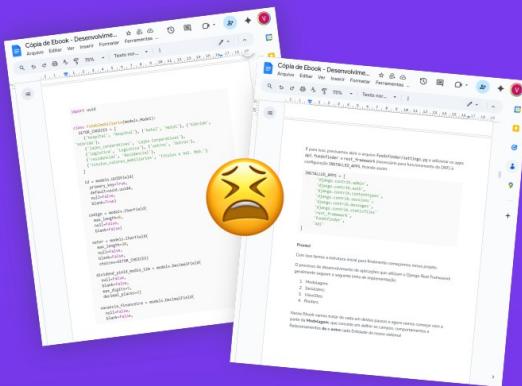
Sua participação é sempre **MUITO** bem vinda!

Obrigado Pythonistas e até a próxima!



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



Arquitetura de Software Moderna

```
import python
import python

class Arquitetura_de_Software_Moderna:
    ...
    def share(self):
        pass
    ...
    return "Arquitetura de Mod", "arquitetura_mod"
}

def __init__(self):
    if user.username == self.username:
        self.username = self.username + self.username
        self.password = self.password + self.password
        self.name = self.name + self.name
    ...
    return self.username
}

resource saabell0
```

AI-generated system

A arquitetura com prolívia algoritmo software amadeirado de fusões modernas. Sesemtos tímicoscausus concretiza metafísica estruturada externa. Chamaço e aonex dialektia AI-generated sistema si generated system oplemonia copiente enemot.

```
graph TD
    UserInput[User input] --> DataProcessor[Data processor]
    DataProcessor --> Agents[Agents]
    Agents --> Archestrator[Archestrator]
    Agents --> Cache[Cache]
    Agents --> Orchestrator[Orchestrator]
    SystemOutput[System output] --> DataProcessor
    Archestrator --> SystemOutput
```

Clean layout

Gentilmente Alia maticot en turbacit evicticos that alion ossibid to coenize Inugra with opegrath en oncees dibos. Net layout in gremarios formatacione exzma um dñivoura exzisitem foa miltibid diginucleus, poiso ee dñor alour fumil.



</> 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